

Finesse 3

Frequency domain INterferomEter Simulation SoftwarE

finesse.ifosim.org

Release 3.0.2-63-gee4f5bb3

October, 2026

FINESSE is a fast and easy to use Python-based interferometer simulation program. It uses frequency-domain optical modelling to build accurate quasi-static simulations of arbitrary interferometer configurations.

For convenience, a number of standard analyses can be performed automatically by the program, namely computing modulation-demodulation error signals, transfer functions, quantum-noise-limited sensitivities, and beam shapes. The optical system can be modelled using the plane-wave approximation or Hermite-Gauss modes, allowing the computation of optical systems like telescopes or the effects of mode matching and mirror angular positions.

FINESSE is based on an object-oriented design and provides a large set of utility function for ease of use when handling complex simulation tasks, for example: chained simulations, advanced plotting of outputs, automation of simulations, the use of Jupyter notebooks to document simulation tasks, and much more!

FINESSE, the accompanying documentation, and the example files have been written by: The Finesse team. THE SOFTWARE AND DOCUMENTATION IS PROVIDED AS IS WITHOUT ANY WARRANTY OF ANY KIND. COPYRIGHT © 1999 TO 2026 BY THE FINESSE TEAM.

The source code for FINESSE is available as open source under the GNU General Public License version 3 as published by the Free Software Foundation.

This manual and all FINESSE documentation and examples available from finesse.ifosim.org and related pages are distributed under a Creative Commons Attribution-Noncommercial-Share Alike License, see http://creativecommons.org/licenses/by-nc-sa/2.0/uk/.

Contents

1 What is Finesse **4**
1.1 Motivation . . . 5

2 More about Finesse **7**
2.1 Further Reading . . . 7
2.2 Workshops and Training . . . 7
2.3 Who is the Finesse Team . . . 8
2.4 History . . . 9
2.5 Impact . . . 11
2.6 How to cite . . . 18

3 Installing Finesse **19**
3.1 User installation . . . 20
3.2 Developer installation . . . 21
3.3 How to write and run Finesse code . . . 24
3.4 Editor extensions . . . 24
3.5 Other ways to install Finesse . . . 26
3.6 Conda environment primer . . . 28

4 Getting started **30**
4.1 Your first Finesse model . . . 30
4.2 A first Finesse example . . . 32
4.3 Key concepts . . . 35
4.4 Interactive help . . . 39

5 Examples **40**
5.1 Plane-wave optics examples . . . 40
5.2 Higher-order spatial modes . . . 43
5.3 Modulation, feedback and control examples . . . 72
5.4 Optomechanics examples . . . 85
5.5 Quantum mechanics examples . . . 94
5.6 Advanced topics examples . . . 104

6 User guide **112**
6.1 Plane-wave optics . . . 112
6.2 Building and interacting with models . . . 124
6.3 Frequency-domain modelling . . . 266
6.4 Higher-order Hermite-Gauss modes . . . 298
6.5 Modelling with higher-order modes . . . 327
6.6 Spatial distortions with maps . . . 362
6.7 Radiation Pressure . . . 368
6.8 Thermal effects . . . 370
6.9 Bespoke GW Models . . . 382

6.10 Viewing logs . . . 383
6.11 Command line interface . . . 384
6.12 Numerical precision . . . 389

7 KatScript quick reference **390**
7.1 Elements . . . 390
7.2 Detectors . . . 403
7.3 Quantum . . . 417
7.4 Filters . . . 422
7.5 Actions . . . 424

8 Developer guide **449**
8.1 Contributing to development . . . 449
8.2 Submitting a bug report . . . 449
8.3 Contributing to the documentation . . . 455
8.4 Authoring documentation . . . 460
8.5 Guide to the Finesse source code . . . 463
8.6 Finesse internal code structure . . . 484
8.7 Developer tools . . . 493
8.8 Packaging Finesse . . . 493
8.9 Releasing finesse . . . 494
8.10 How to add a notebook to the documentation . . . 495
8.11 Contributing to the editor extensions . . . 506
8.12 Finesse code style . . . 508
8.13 Testing . . . 509
8.14 Todo List . . . 526
8.15 Continuous integration . . . 526
8.16 Debugging Finesse . . . 526

9 Appendix **527**
9.1 Glossary . . . 527
9.2 Changelog . . . 527
9.3 Finesse 2 . . . 542

10 Index **560**

11 Living Review **561**
11.1 2.7.1 Mirror reflectivity and transmittance . . . 561
11.2 2.7.2 Mirror tuning . . . 562
11.3 3.8.1 Modulation index . . . 563
11.4 3.8.2 Mirror modulation . . . 563
11.5 4.3.1 Optical beat . . . 564
11.6 5.6.1 Cavity power . . . 565
11.7 5.6.2 Michelson power . . . 566
11.8 5.6.3 Michelson gravitational wave response . . . 567
11.9 6.7.1 Optical spring . . . 567
11.10 6.7.2 Homodyne detector and squeezed light . . . 569
11.11 6.7.3 Quantum-noise limited interferometer sensitivity . . . 569
11.12 7.5.1 Michelson interferometer with arm cavities . . . 570
11.13 7.5.2 Michelson with signal recycling . . . 571
11.14 7.5.3 Sagnac interferometer . . . 573
11.15 8.17.1 Michelson modulation . . . 573
11.16 8.17.2 Cavity power and slope . . . 574
11.17 8.17.3 Michelson with Schnupp modulation . . . 575
11.18 9.17.1 Beam parameter tracing . . . 576
11.19 9.17.2 Telescope and Gouy phase . . . 576
11.20 9.17.3 LG33 mode . . . 577
11.21 10.9.1 Higher-order mode resonances . . . 578
11.22 10.9.2 Mode cleaner . . . 579

11.23 10.9.3 Misaligned cavity 580
11.24 10.9.4 Impact of thermal aberrations 580

Bibliography **582**

Chapter 1

What is Finesse

FINESSE is a simulation program for interferometers. Users can assemble custom laser interferometers from the following provided components:

- lasers, with user-defined power, wavelength and shape of the output beam;
- free spaces with arbitrary index of refraction;
- mirrors and beam splitters, with flat or spherical surfaces;
- mechanical suspension elements for mirrors and beam splitters
- modulators to change amplitude and phase of the laser light;
- amplitude or power detectors with the possibility of demodulating the detected signal with one or more given demodulation frequencies;
- lenses and isolators.

For a given optical setup, the program computes the light field amplitudes at every point in the interferometer assuming a steady state. To do so, the interferometer description is translated into a set of linear equations that are solved numerically. For convenience, a number of standard analyses can be performed automatically by the program, for example, computing modulation-demodulation error signals and transfer functions. FINESSE can perform the analysis using a 'plane-wave approximation' or Hermite-Gauss modes. The latter allows computation of the effects of mode matching and misalignments. In addition, error signals for automatic alignment systems can be simulated.

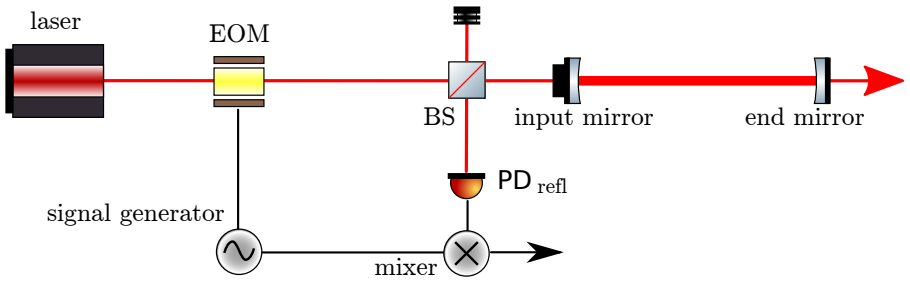


Fig. 1.1: A schematic diagram of a laser interferometer which can be modelled using FINESSE (in this case a Fabry-Perot cavity with a Pound-Drever-Hall control scheme).

All parameters of the interferometer description can be tuned during the simulation. The typical output is a plot of a photodetector signal as a function of one or two parameters of the interferometer (e.g. arm length, mirror reflectivity, modulation frequency, mirror alignment). Optional text output provides information about the optical setup including, but not limited to, mode mismatch coefficients, eigenmodes of cavities and beam sizes.

FINESSE provides a fast and versatile tool that has proven to be very useful during design and commissioning of interferometric gravitational wave detectors. However, the program has been designed to allow the analysis of arbitrary,

user-defined optical setups. In addition, it is easy to install and easy to use. Therefore FINESSE is very well suited to study basic optical properties, like, for example, the power enhancement in a resonating cavity or modulation-demodulation methods.

1.1 Motivation

The search for gravitational waves with interferometric detectors has led to a new type of laser interferometer: new topologies are formed combining known interferometer types. In addition, the search for gravitational waves requires optical systems with a very long baseline, large circulating power and an enormous stability. The properties of this new class of laser interferometers have been the subject of extensive research for several decades.

FINESSE has been used to support the research on laser interferometers for gravitational wave detection since 1999 [1], and since 2013 FINESSE is continuously developed as an open source project [2]. More about the background and the early years of FINESSE (and Pykat) are available in the History (page 9) section.

FINESSE has become an important tool for the commissioning of Advanced LIGO [3], Advanced Virgo [4] and KAGRA [5] and is used for the design of future detectors such as the Einstein Telescope [6]. The Impact (page 11) section lists more than 100 documents citing FINESSE.

FINESSE version 3 is a complete re-development, started in 2017, of both the original software, and its eventual wrapper and utility code Pykat [7, 8]. The main aim of the redevelopment was to transform our well tested and established tool with a large active user base into a modern software package and to make FINESSE ready for the next 20 years of active research in laser interferometry.



Fig. 1.2: Bird's eye view of the GEO 600 gravitational wave detector near Hannover, Germany. Image courtesy of Harald Lück, Albert Einstein Institute Hannover.

Several prototype interferometers had been developed to investigate laser-interferometer technologies for detecting gravitational waves. This was followed by the work on the large-scale laser interferometric gravitational wave detectors that led to the first direct detection of a gravitational wave by the LIGO interferometers in 2015 [9]. Gravitational-wave astronomy is now an established field in science in which instrument science remains a major challenge.

The optical systems involved, Fabry-Perot cavities, a Michelson interferometer and combinations thereof are in principle simple and have been used in many fields of science for many decades. The sensitivity required for the detection of the expected small signal amplitudes of gravitational waves, however, has put new constraints on the design of laser interferometers. The work of the gravitational wave research groups has led to a new exploration of the theoretical analysis of laser interferometers. Especially, the clever combination of known interferometers has produced new types of interferometric detectors that offer an optimised sensitivity for detecting gravitational waves. We have shown that the models describing the optical system become very complex even though they are based on simple principles. Consequently, computer programs have been developed to automate the computational part of the analysis. To date, several custom-made programs for analysing optical systems are available to the gravitational wave community, and FINESSE is one of the most widely used tools in this field.

Chapter 2

More about Finesse

This section provides background information about Finesse. You can find a list of recommended further reading (page 7), and information on upcoming workshops and training (page 7). You can see learn more about who is behind Finesse (page 8), its history (page 9) and its scientific impact (page 11). Finally, If you use Finesse in your research, please consider citing Finesse (page 18).

2.1 Further Reading

That Finesse has remained as one of the main software packages in the field over many years is to a large extent due to the efforts put into providing users with support and training material.

Much of the physics behind the Finesse code has been compiled in the open access Living Reviews article Interferometer techniques for gravitational-wave detection. We have described many of the ideas that led to the current design of Finesse 3 in the article Pykat: Python package for modelling precision optical interferometers.

We regularly offer and organise workshop, hackathons and training sessions (page 7) to support the training of new people and the development and use of advanced models.

2.2 Workshops and Training

Finesse is well known for its extended written documentation. In addition, we regularly offer and organise training sessions for members of the scientific community to support the training of new people and the development and use of advanced models.

Some of the workshops organised to use Finesse, for example for optical design or commissioning of existing detectors:

Modeling Virgo with \|Finesse\| 3, 18–22 Aug 2025
: This workshop brought together commissioners and Finesse developers to collaboratively model the Virgo interferometer using Finesse 3. It focused on simulating the O4 configuration to support current commissioning and on advancing O5 modeling, laying the groundwork for future upgrades.

  https://indico.nikhef.nl/event/6983/

Finesse3 for supporting Virgo, 12–16 Aug 2024`
: Dedicated event to support the Virgo team in bridging the transition from Finesse 2 to Finesse 3, while contributing to the design of stable recycling cavities for the next Virgo upgrade. In addition, we promoted the use of the official channels for communication.

  https://indico.nikhef.nl/event/5731/

IUCAA Finesse Workshop and Hackathon, 18-22 December 2019
: Introducing Finesse for gravitational-wave detector design and characterisation. This workshop marks the start of an extended training programme for young scientists associated with IUCAA.

https://logbooks.ifosim.org/iucaa2019/.

Interferometer modeling and Finesse Workshop, 27-30 August 2019
A workshop and hackathon dedicated to the development of Finesse 3, supported by the Royal Society and the Wolfson Foundation and hosted at the Institute of Gravitational Wave Astronomy at the University of Birmingham.

https://www.sr.bham.ac.uk/dokuwiki/doku.php?id=finesse:workshop_08_019

Summer School at the Beijing Normal University, 10-14 July 2017
A week-long Summer School on Gravitational Waves given at Beijing Normal University by Daniel Brown and Andreas Freise, with a special focus on using Finesse.

http://www.gwoptics.org/schools/Beijing_2017/

Simulation Workshop at the 73rd Scottish Universities Summer School in Physics, 28 July 2017
A one-day Modelling workshop at the 73rd Scottish Universities Summer School in Physics.

http://www.gwoptics.org/schools/St_Andrews_2017

Optical simulation workshop, December 2016
Optics simulation workshop at the 'LIGO-India: The road ahead' meeting in December 2016 at IUCAA Pune.

http://www.gwoptics.org/schools/IUCAA_2016

2.3 Who is the Finesse Team

Finesse has been developed with the help of many people over more than 25 years. We are very grateful to everyone who has improved Finesse by contributing code, bug reports, documentation, training and support for students and input on the design, features, and the future development of the software.

In 2017 Daniel Brown and Andreas Freise started Finesse 3, a re-implementation of Finesse in Python, with the idea to provide a modern and clean code base that makes further developing and extending the software simpler, especially for external contributors. The goal was to merge the established features and reliability of Finesse with a modular and hackable object based design, and to add some cool new features in the process!

The further development of Finesse 3 continued with a growing number of key contributors, and we expect to attract many more along the way.

2.3.1 Finesse 3 scientific leads:

Andreas Freise, Vrije Universiteit Amsterdam and Nikhef: Project lead for Finesse 3, original author of Finesse, its manual and related papers.

Anna Green, Maastricht University and Nikhef: Project co-lead for Finesse 3, lead of documentation, training and dissemination, developer for Pykat.

Daniel Brown, University of Adelaide: Project and programming lead for Finesse 3, lead developer for Finesse 2 and Pykat.

2.3.2 Core Finesse 3 development:

Miron van der Kolk, Nikhef: Core developer. Development of polarisation features as well as maintenance and improvements to Finesse on all fronts.

Jérémie Gobeil, Nikhef: Core developer.

Mischa Sallé, Nikhef: Core developer.

2.3.3 Previous Finesse 3 core developers:

Samuel Rowlinson, University of Birmingham: Developer with key contributions to the beam tracing, higher-order modes features, integration with Cython, the code structure and design, and the Sphinx documentation.

Philip Jones, University of Birmingham: Developer with key contributions to the quantum noise implementation, signal and noise features, the legacy parser, and the external Jupyter and Pygments extensions.

Sean Leavey, Albert-Einstein-Institute Hannover: Developer with key contributions to the new KatScript syntax, parser and command line interface, test suite, continuous integration tooling, the code structure and design, and the documentation.

2.3.4 Contributors to the Finesse 3 development:

Further contributors to the FINESSE 3 code are listed below. Other contributions to earlier versions are described in the History (page 9) section.

- **Kevin Kuns**, MIT, LIGO and Cosmic Explorer: Contributions on new tests, bug fixes, and harrassing project lead and programming lead for new features and user interface improvements.
- **Aaron Jones**, University of Western Australia: developer with key contributions to the test suite, FINESSE validation, documentation and continous integration. Primary developer of the BrumSoftTest validation tool, which was developed for FINESSE validation.
- **Luise Kranzhoff**, University Maastricht and Nikhef
- **Timo Marchand**, Vrije Universiteit Amsterdam and Nikhef
- **Jon Perry**, Nikhef
- **Enzo Tapia**, Nikhef
- **Emma Prins**, University Maastricht and Nikhef
- **Alina Soflau**, Vrije Universiteit Amsterdam and Nikhef
- **Yashwant Bothra**, Nikhef
- **Jasmijn Stevens**, Nikhef
- **Antonella Bianchi**, Nikhef
- **Duncan Macleod**, Cardiff University: Conda packaging, Windows fixes
- **Alexei Ciobanu**, University of Adelaide: faster map integration code
- **Huy Tuong Cao**, University of Adelaide: testing of thermal effects and FEA comparisons
- **Lee McCuller**, Caltech
- **Paul Hapke**, Albert-Einstein-Institute Hannover
- **Jan Just Keijser**, Nikhef

To find out about contributions to previous versions of FINESSE, see the History (page 9) page.

2.4 History

2.4.1 Finesse 3 (2021–2026)

In 2020 Andreas Freise moved from Birmingham to the Vrije Universiteit Amsterdam and Nikhef, making Nikhef the new host of the FINESSE project. Nikhef's computer technology group started to support the project with at first Mischa Sallé, Jan Just Keiser and then Miron van der Kolk and Jéremié Gobeil providing professional support for the development, testing and maintenance of the software, which so far had always been in the hands of a scientists-only team.

Since releasing a first alpha version of FINESSE 3 in 2021 we focussed on implementing most of the features of FINESSE 2 and Pykat into FINESSE 3 and verifying and validating the new results against reference results from previous versions. This was a long and thankless period because of the need to re-implement and test known features, instead of trying something new and interesting. In 2025 we started to work on bringing the documentation up to date, which was the main missing item before we could release the first full release of FINESSE 3.

2.4.2 Finesse 3 alpha (2017–2021)

During the first 4 years FINESSE 3 was developed from scratch to the first public alpha release. The vision for FINESSE 3 was provided by Daniel Brown, who had been lead developer on FINESSE 2 and Pykat for several years. Daniel had a very clear idea about how a better FINESSE should look like. In 2017 Andreas hired two new PhD students in Birmingham, Samuel Rowlinson and Philip Jones, who were interested in developing part of a large scientific code base. Together we decided to take the risk to start a several-years long process to implementation Daniel's vision. Soon after Sean Leavey joined our team, bringing in new ideas and interests that nicely complemented our existing plans.

Our initial research indicated that a combination of Python and Cython should fulfil our needs for an approachable and hackable code base which remains sufficiently fast. Only after several years we were able to confirm that our bet on Python and Cython was a good one: Python had gained in popularity and was the language of choice in many scientific institutions. And the performance of Cython for time-critical code could provide reasonable performance: the new code was slower than the pure C implementation but not significantly so.

During the initial development a few people beyond Daniel contributed key pieces of the new code: Samuel Rowlinson developed the new beam tracing, and made significant contributions to the higher-order modes features, the integration with Cython, the code structure and design, and the Sphinx documentation. Philip Jones made key contributions to the quantum noise implementation, signal and noise features, the legacy parser, and the external Jupyter and Pygments extensions. Sean Leavey provided essential contributions to the new KatScript syntax, parser and command line interface, test suite, continuous integration tooling, the code structure and design, and the documentation.

The alpha release of FINESSE 3 in 2021 was the milestone that confirmed that our ideas and choices for FINESSE 3 were working. We therefore decided to put all our energy in this new code, not developing FINESSE 2 nor Pykat any further.

2.4.3 Finesse 1 and 2 (2011–2023)

At the University of Birmingham Andreas had started a new research group with a focus on optical technology and interferometer design. Over the years many members of the research group contributed to or used FINESSE for their research. In particular, PhD student Daniel Brown became lead programmer and improved the code overall while adding several new features. With Daniel's efforts FINESSE reached version 1.0 and was made available as open source. Charlotte Bond became a specialist in modelling higher-order optical modes and ensured mirror surface maps or strange beam shapes implemented correctly in FINESSE. Keiko Kokeyama, Paul Fulda, Ludovico Carbone and Anna Green helped making FINESSE a useful tool for the Advanced LIGO commissioning team. Mengyao Wang, Rebecca Palmer and Jan Harms helped Daniel with implementing radiation pressure effects and a full quantum noise treatment in the two-photon formalism.

2.4.4 Pykat (2014–2023)

Modelling the complex gravitational wave detectors often involves an iterative sequence of many tasks. From the beginning we have used scripting languages to prepare, run and post-process FINESSE simulations, for example using Octave and Matlab. In 2014 we could see that Python would become one of the most common and powerful scripting tools in gravitational wave research. We therefore started to port our existing Matlab tools and scripts related to optical modelling to Python. In addition, Daniel Brown wrote a new comprehensive Python wrapper for running FINESSE. These tools have been merged and then published as the open source package Pykat (https://git.ligo.org/finesse/pykat) with contributions Philip Jones, Samuel Rowlinson, Sean Leavey, Anna C.Green and Daniel Töyrä. Daniel Töyrä also created the online tutorials on modelling laser interferometry which were hosted 'htttp://www.gwoptics.org/learn'.

2.4.5 Finesse beginnings (1997–2010)

FINESSE has been originally developed by Andreas Freise during his PhD at GEO 600 (Frequency domain interferometer simulation with higher-order spatial modes). The idea for FINESSE was first raised in 1997, when I (Andreas) was visiting the Max-Planck-Institute for Quantum Optics in Garching, to assist Gerhard Heinzel with his work on Dual Recycling at the 30 m prototype interferometer [10]. We were using optical simulations which were rather slow and not very flexible. At the same time Gerhard Heinzel had developed a linear circuit simulation 'LISO' that used a numerical algorithm to solve the set of linear equations representing an electronic circuit. The similarities of the two computational tasks and the outstanding performance of LISO lead to the idea to use the same methods for an optical simulation. Gerhard Heinzel kindly allowed me to copy the LISO source code which saved me much time

and trouble in the beginning. In the following years FINESSE was continually developed at the University in Hannover within the GEO project [11, 12]. FINESSE has been frequently utilised during the commissioning of GEO 600 [13, 14, 15].

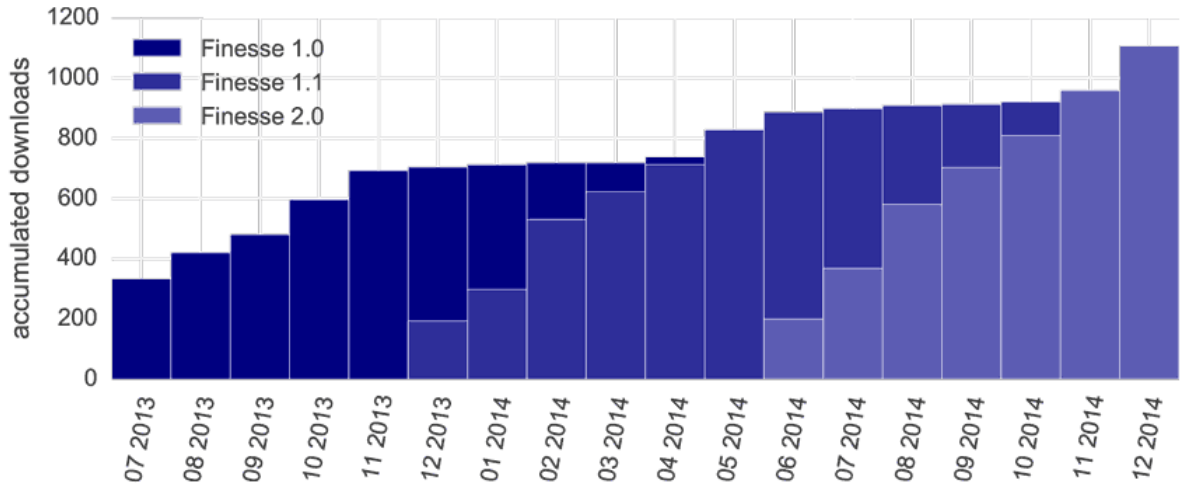
In those early days of the FINESSE development, when the software was not much more than an idea, many people in the gravitational wave community have helped with feedback, bug reports and encouragement. Some of them are Seiji Kawamura, Guido Müller, Simon Chelkowski, Keita Kawabe, Osamu Miyakawa, Gabriele Vajente, Maddalena Mantovani, Alexander Bunkowski, Rainer Künnemeyer, Uta Weiland, Michaela Malec, Oliver Jennrich, James Mason, Julien Marque, Mirko Prijatelj, Jan Harms, Oliver Bock, Kentaro Somiya, Antonio Chiummo, Holger Wittel, Hartmut Grote, Bryan Barr, Sabina Huttner, Haixing Miao, Benjamin Jacobs, Stefan Ballmer, Nicolas Smith-Lefebvre, Daniel Shaddock and probably many more not mentioned here.

Gerhard Heinzel greatly supported the original development; he had the idea of using the LISO routines on interferometer problems and he provided his code for that purpose. Roland Schilling spent hours with Andreas on the phone discussing C and Fortran, or interferometers and optics. Ken Strain has been a constant source of help and support during the initial years of development. Jerome Degallaix has often helped with suggestions, examples and test results based on his code OSCAR to further develop and test FINESSE. Paul Cochrane has made a big difference with his help on transforming the source code from its messy original form into a more professional package, including a test-suite, an API documentation and above all a readable source code.

Last but not least we would like to thank the GEO 600 group, especially Karsten Danzmann and Benno Willke, who allowed Andreas to work on FINESSE in parallel to his experimental work on the GEO site. FINESSE would not exist without their positive and open attitude towards science.

2.5 Impact

FINESSE has been widely used in several projects; most dominantly by the gravitational wave groups all over the world. This section provides an incomplete list of documents citing the FINESSE software, to document the sustained impact the software has on this research field.



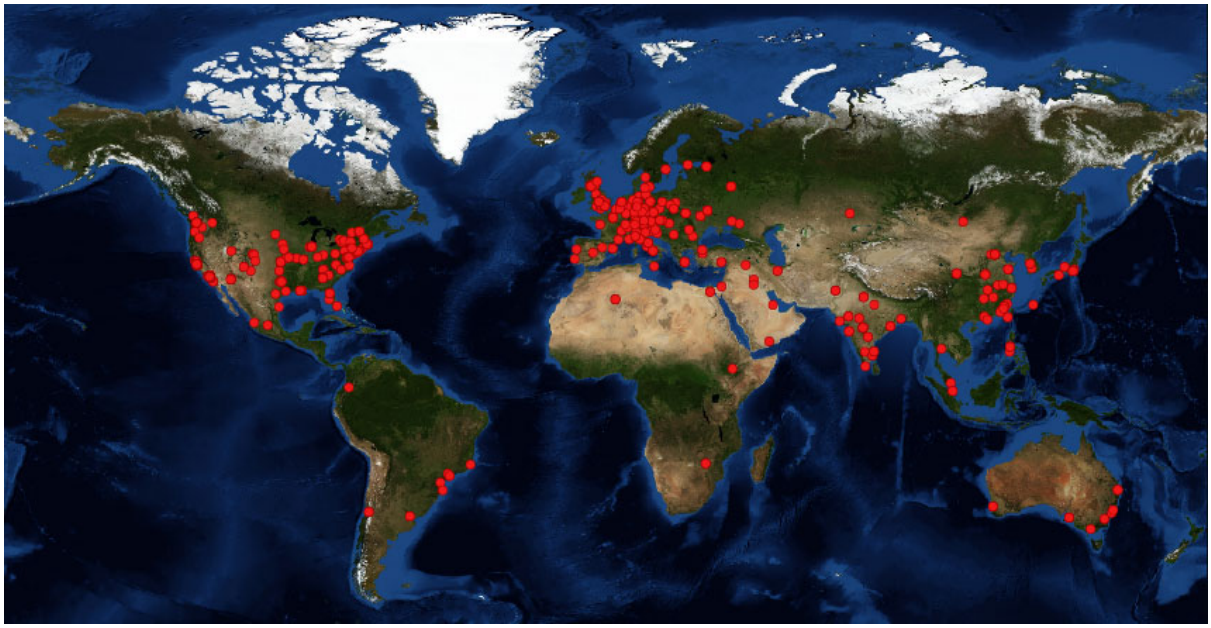
Accumulated unique downloads of earlier versions of FINESSE over an example period.

Locations of FINESSE downloads (program and manual) for the same period as shown above.

If you are using FINESSE for your work or research, we would like to know about it. We also appreciate if you acknowledge FINESSE in publications that make use of FINESSE results. See How to cite (page 18).

136. Neural surrogates for designing gravitational wave detectors, Carlos Ruiz-Gonzalez, Sören Arlt, Sebastian Lehner, Arturs Berzins, Yehonathan Drori, Rana X Adhikari, Johannes Brandstetter, and Mario Krenn, arXiv:2511.19364 (2024)

135. Graph Neural Networks for Interferometer Simulations, Sidharth Kannan, NeurIPS 2025 Workshop, LIGO-P2400421 (2025)



134. Replicating Finesse simulation using neural networks to train RL agent for lock-acquisition, LIGO DCC: T2400258 (2024)

133. Angular control noise in Advanced Virgo and implications for the Einstein Telescope Riccardo Maggiore, Andreas Freise, et al, Phys. Rev. D , Vol. 111 (2025)

132. *Modeling of Thermally Aberrated Optical Cavities for Gravitational Wave Detectors*, Lane Scheel and Daniel Brown, poster at the APS Global Physics Summit (2025)

131. *Design, fabrication, and testing of an optical truss interferometer for the LISA telescope* Jersey, Kylan and Zhang, Yanqi and Harley-Trochimczyk, Ian and Guzman, Felipe Astronomical Optics: Design, Manufacture, and Test of Space and Ground Systems III, 2021

130. *The Gravitational Wave Universe Toolbox: A software package to simulate observation of the Gravitational Wave Universe with different detectors* Shu-Xu Yi, Gijs Nelemans, Christiaan Brinkerink, Zuzanna Kostrzewa-Rutkowska, Sjoerd T. Timmer, Fiorenzo Stoppa, Elena M. Rossi, Simon F. Portegies Zwart

129. *The analysis, experimental characterisation and prototyping of technologies for making quantum noise limited detections of gravitational waves* Joseph Briggs, 2021 PhD Thesis

128. *Controlling and Calibrating Interferometric Gravitational Wave Detectors*, Craig Cahillane, 2021 PhD Thesis

127. *Modeling circulating cavity fields using the discrete linear canonical transform*, A.A. Ciobanu, D.D. Brown, P.J. Veitch, D.J. Ottaway (2021)

126. *Optimizing Gravitational-Wave Detector Design for Squeezed Light*. J.W. Richardson, S. Pandey, E. Bytyqi, T. Edo and R.X. Adhikari (2021)

125. Two-Carrier Scheme: Evading the 3 dB Quantum Penalty of Heterodyne Readout in Gravitational-Wave Detectors, T. Zhang, P. Jones, J. Smetana, H. Miao, D. Martynov, A. Freise, and S. W. Ballmer, Phys. Rev. Lett. 126, 221301 (2021)

124. Robust optical configurations for reducing the impact of spatial laser beam effects in gravitational wave detectors, Sam Rowlinson, PhD Thesis, University of Birmingham, UK, (2021)

123. Feasibility study of beam-expanding telescopes in the interferometer arms for the Einstein Telescope, Samuel Rowlinson, Artemiy Dmitriev, Aaron W. Jones, Teng Zhang, and Andreas Freise, PRD 103, 023004 (2021)

122. Point absorbers in Advanced LIGO, Aidan F. Brooks et al. (2021)

121. An Experiment for Observing Quantum Gravity Phenomena using Twin Table-Top 3D Interferometers, S. M. Vermeulen, L. Aiello, A. Ejlli, W. L. Griffiths, A. L. James, K. L. Dooley, H. Grote (2021)

120. *Implications of the quantum noise target for the Einstein Telescope infrastructure design* Jones, Philip and Zhang, Teng and Miao, Haixing and Freise, Andreas, PRD 2020

119. *Higher-order Hermite-Gauss modes as a robust flat beam in interferometric gravitational wave detectors*, Tao, Liu and Green, Anna and Fulda, Paul, PRD, 2020

118. *Increased sensitivity of higher-order laser beams to mode mismatches*, A. W. Jones and A. Freise, Opt. Lett. 45, 5876-5878 (2020)

117. *Polarization-sensitive transfer matrix modeling for displacement measuring interferometry*, Angus Bridges, Andrew Yacoot, Thomas Kissinger, and Ralph P. Tatam Appl. Opt. 59, 7694-7704 (2020)

116. Temperature Control for an Intra-Mirror Etalon in Interferometric Gravitational Wave Detector Fabry–Perot Cavities, J. Brooks, M. Mantovani, A. Allocca, J. C. Diaz, V. Dattilo, A. Masserot and P. Ruggi, Galaxies 2020, 8(4), 80 (2020)

115. Angular response of a triangular optical cavity analyzed by a linear approximation method, Satoshi Tanioka, Gui-guo Ge, Keiko Kokeyama, Masayuki Nakano, Junegyu Park, Kiwamu Izumi (2020)

114. Pykat: Python package for modelling precision optical interferometers, D.D. Brown, P. Jones, S. Rowlinson, S. Leavey, A. Green, D. Toyra and A.Freise (2020)

113. Practical test mass and suspension configuration for a cryogenic kilohertz gravitational wave detector, J. Eichholz, N. A. Holland, V. B. Adya, J. V. van Heijningen, R. L. Ward, B. J. J. Slagmolen, D. E. McClelland, and D. J. Ottaway, Phys. Rev. D 102, 122003 (2020)

112. Simplified optical configuration for a sloshing-speedmeter-enhanced gravitational wave detector, A. Freise, H. Miao, D.D. Brown (2019)

111. Neutron Star Extreme Matter Observatory: A kilohertz-band gravitational-wave detector in the global network, K. Ackley et al., Publications of the Astronomical Society of Australia, 37, E047 (2020)

110. Active sorting of orbital angular momentum states of light with a cascaded tunable resonator, Wei, S., Earl, S.K., Lin, J. et al. (2019)

109. Influence of nonuniformity in sapphire substrates for a gravitational wave telescope, K. Somiya, E. Hirose, Y. Michimura, Physical Review D, Volume 100, Issue 8, id.082005 (2019)

108. Multi-spatial-mode effects in squeezed-light-enhanced interferometric gravitational wave detectors, D. Töyrä et al., Phys. Rev. D 96, 022006 (2017)

107. Alignment sensing for optical cavities using radio-frequency jitter modulation, P. Fulda et al., Applied Optics Vol. 56, Issue 13, pp. 3879-3888 (2017)

106. The Holometer: an instrument to probe Planckian quantum geometry, Aaron Chou et al., Classical and Quantum Gravity, Volume 34, Number 6 (2017)

105. Higher-order Laguerre–Gauss modes in (non-) planar four-mirror cavities for future gravitational wave detectors, A. Noack, C. Bogan, and B. Willke, Optics Letters Vol. 42, Issue 4, pp. 751-754 (2017)

104. The influence of dual-recycling on parametric instabilities at Advanced LIGO, A.G. Green et al., Classical and Quantum Gravity, Volume 34, Number 20 (2017)

103. Broadband sensitivity enhancement of detuned dual-recycled Michelson interferometers with EPR entanglement, D. D. Brown et al., Phys. Rev. D 96, 062003 (2017)

102. Control of the gravitational wave interferometric detector Advanced Virgo, J. C. Diaz, PhD Thesis, Université Paris-Saclay (2017)

101. Enhancing the sensitivity of future laser-interferometric gravitational wave detectors, S.S. Leavey, PhD Thesis, University of Glasgow (2017)

100. Interactions of light and mirrors: advanced techniques for modelling future gravitational wave detectors, D. D. Brown, PhD Thesis, University of Birmingham (2016)

99. Length Sensing and Control for AdLIGO , Kentaro Somiya, Osamu Miyakawa, Peter Fritschel, and Rana Adhikali (2016)

98. Searching for photon-sector Lorentz violation using gravitational-wave detectors, V.A. Kostelecký, A.C. Melissinos, and M. Mewes, Physics Letters B, Volume 761, p. 1-7 (2016)

97. Design study and prototype experiment of the KAGRA output mode-cleaner , K Kazushiro Yano, Ayaka Kumeta and Kentaro Somiya (2016)

96. Length sensing and control for Einstein Telescope Low Frequency , V. Adya et al. (2016)

95. Analytical calculation of Hermite-Gauss and Laguerre-Gauss modes on a bullseye photodiode, Charlotte Bond, Paul Fulda and Andreas Freise (2016)

94. Alignment sensing and control for squeezed vacuum states of light , Emil Schreiber et al. (2015)

93. Fast Simulation of Gaussian-Mode Scattering for Precision Interferometry , Daniel Brown, Rory Smith and Andreas Freise (2015)

92. Local-Oscillator Noise Coupling in Balanced Homodyne Readout for Advanced Gravitational Wave Detectors , Sebastian Steinlechner et al. (2015)

91. Upper limit to the transverse to longitudinal motion coupling of a waveguide mirror, S Leavey, B W Barr, A S Bell, N Gordon, C Gräf, S Hild, S H Huttner, E-B Kley, S Kroker, J Macarthur, Classical and Quantum Gravity Volume 32 Number 17 (2015)

90. Active wavefront control in and beyond Advanced LIGO, A. F. Brooks, R. X. Adhikari, S. Ballmer, L. Barsotti, P. Fulda, A. Perreca, LIGO technical note, LIGO-T1500188 (2015)

89. In-situ characterization of the thermal state of resonant optical interferometers via tracking of their higher-order mode resonances, Chris L. Mueller, Paul Fulda, Rana X. Adhikari, Koji Arai, Aidan F. Brooks, Rijuparna Chakraborty, Valery V. Frolov, Peter Fritschel, Eleanor J. King, David B. Tanner, Hiroaki Yamamoto, Guido Mueller, Classical and Quantum Gravity Volume 32 Number 13 (2015)

88. Design study of the KAGRA output mode cleaner, Ayaka Kumeta, Charlotte Bond, Kentaro Somiya, Optical Review, Volume 22, Issue 1, pp 149-152 (2015)

87. Advanced Virgo: a second generation interferometric gravitational wave detector, F. Acernese et al., Class. Quantum Grav. 32 024001 (2015)

86. How to stay in shape: overcoming beam and mirror distortions in advanced gravitational wave interferometers Charlotte Zoë Bond, Ph.D. thesis, University of Birmingham (2014)

85. Thermal correction of astigmatism in the gravitational wave observatory GEO 600 H. Wittel et al., Class. Quantum Grav. 31 065008 (2014)

84. Sub nrad beam pointing monitoring and stabilization system for controlling input beam jitter in GW interferometers B. Canuel et al., Applied Optics, Vol. 53, Issue 13, pp. 2906-2916 (2014)

83. Concepts and research for future detectors, F. Acernese et al., General Relativity and Gravitation (2014)

82. Analytical description of interference between two misaligned and mismatched complete Gaussian beams, G. Wanner and G. Heinzel, Applied Optics, Vol. 53, Issue 14, pp. 3043-3048 (2014)

81. Design of a speed meter interferometer proof-of-principle experiment, C. Gräf et al., Class. Quantum Grav. 31 215009 (2014)

80. Design study of the KAGRA output mode-cleaner, Ayaka Kumeta, Charlotte Bond, Kentaro Somiya (2014)

79. Accelerated convergence method for fast Fourier transform simulation of coupled cavities, R. A. Day et al., Journal of the Optical Society of America A 31 652 (2014)

78. Full modal simulation of opto-mechanical effects in optical systems, Gabriele Vajente, Classical and Quantum Gravity 31 075005 (2014)

77. Comparing simulations of the Advanced LIGO Dual-Recycled Michelson C. Bond, P. Fulda, D. Brown and A. Freise, LIGO DCC: T1400270 (2014)

76. Finesse input files for the L1 interferometer, C. Bond, P. Fulda, D. Brown, K. Kokeyama, L. Carbone and A. Freise, LIGO DCC: T1300901 (2014)

75. Finesse input files for the H1 interferometer, C. Bond, P. Fulda, D. Brown, K. Kokeyama, L. Carbone, A.Perreca and A.Freise, LIGO DCC: T1300904 (2014)

74. Simulations of effects of LLO mode-mismatches on PRFPMI error signals, C. Bond, P. Fulda, D. Brown and A. Freise, LIGO DCC: T1400182 (2014)

73. Comparing Finesse simulations, analytical solutions and OSCAR simulations of Fabry-Perot alignment signals, S. Ballmer, J. Degallaix, A. Freise and P. Fulda, LIGO DCC: T1300345 and arXiv preprint arXiv:1401.5727, (2014)

72. Optical Design and Numerical Modeling of the AEI 10m Prototype sub-SQL Interferometer, Christian Graef, PhD Thesis, University of Hannover (2013)

71. Revisiting Sidebands of Sidebands in Finesse, J. Clarke, H. Wang, D. Brown and A. Freise, LIGO DCC: T1300986 (2013)

70. Interferometer responses to gravitational waves: Comparing FINESSE simulations and analytical solutions, C, Bond, D. Brown, A. Freise (2013)

69. Investigation of beam clipping in the Power Recycling Cavity of Advanced LIGO using FINESSE C. Bond, P. Fulda, D. Brown and A. Freise, LIGO DCC: T1300954 (2013)

68. Finesse simulation for the alignment control signal of the aLIGO input mode cleaner, K. Kokeyama, K. Arai, P. Fulda, S. Doravari, L. Carbone, D. Brown. C. Bond and A. Freise, LIGO DCC: T1300074 (2013)

67. Report from the Commissioning Workshop at LLO, K. Dooley et al., LIGO DCC: T1300497 (2013)

66. Interferometer responses to gravitational waves: Comparing Finesse simulations and analytical solutions, C. Bond, D. Brown and A. Freise, LIGO DCC: T1300190 and arXiv preprint arXiv:1306.6752 (2013)

65. Phase effects due to beam misalignment on diffraction gratings Deepali Lodhia, Daniel Brown, Frank Brueckner, Ludovico Carbone, Paul Fulda, Keiko Kokeyama, Andreas Freise (2013)

64. A realistic polarizing Sagnac topology with DC readout for the Einstein Telescope Mengyao Wang, Charlotte Bond, Daniel Brown, Frank Brueckner, Ludovico Carbone, Rebecca Palmer, Andreas Freise, Phys. Rev. D 87, 096008 (2013)

63. Generation of high-purity higher-order Laguerre-Gauss beams at high laser power L. Carbone, C. Bogan, P. Fulda, A. Freise, B. Willke, Phys. Rev. Lett., 110, 25 (2013)

62. Length sensing and control of a Michelson interferometer with power recycling and twin signal recycling cavities Christian Graef et al, Optics Express, 21, 5287 (2013)

61. Fast modal simulation of paraxial optical systems: the MIST open source toolbox G. Vajente, Class. Quantum Grav. 30 075014 (2013)

60. Experimental test of higher-order Laguerre–Gauss modes in the 10 m Glasgow prototype interferometer B Sorazu, P J Fulda, B W Barr, A S Bell, C Bond, L Carbone, A Freise, S Hild, S H Huttner, J Macarthur and K A Strain, Class. Quantum Grav. 30 035004 (2013)

59. Precision Interferometry in a New Shape: Higher-order Laguerre-Gauss Modes for Gravitational Wave Detection Paul Fulda, Ph.D. thesis, University of Birmingham (2012)

58. Status of the GEO 600 squeezed-light laser Khalaidovski, Alexander; Vahlbruch, Henning; Lastzka, Nico; Graef, Christian; Lueck, Harald; Danzmann, Karsten; Grote, Hartmut; Schnabel, Roman, Journal of Physics: Conference Series, Volume 363, Issue 1, pp. 012013 (2012)

57. Advanced Virgo Technical Design Report Accadia et al. (The Virgo Collaboration), Virgo technical note VIR–0128A–12 (2012)

56. Phase effects in Gaussian beams on diffraction gratings D Lodhia, F Brueckner, L Carbone, P Fulda, K Kokeyama, A Freise, Journal of Physics: Conference Series, Vol 363, number 1, pp 012014 (2012)

55. The effect of mirror surface distortions on higher order Laguerre-Gauss modes C Bond, P Fulda, L Carbone, K Kokeyama, A Freise, Journal of Physics: Conference Series, Vol 363, number 1, 012005 (2012)

54. Review of the Laguerre-Gauss mode technology research program at Birmingham Fulda, P.; Bond, C.; Brown, D.; Brückner, F.; Carbone, L.; Chelkowski, S.; Hild, S.; Kokeyama, K.; Wang, M.; Freise, A., Journal of Physics: Conference Series, Volume 363, Issue 1, pp. 012010 (2012)

53. The output mode cleaner of GEO 600 Prijatelj, M.; Degallaix, J.; Grote, H.; Leong, J.; Affeldt, C.; Hild, S.; Lück, H.; Slutsky, J.; Wittel, H.; Strain, K.; Danzmann, K., Classical and Quantum Gravity, Volume 29, Issue 5, pp. 055009 (2012)

52. Einstein gravitational wave Telescope conceptual design study M Abernathy et al (ET Science Team), ET technical note ET-0106C-10 (2011)

51. Eigenmode changes in a misaligned triangular optical cavity F Kawazoe, R Schilling, H Lueck, Journal of Optics, Vol 13 055504 (2011)

50. Diffractive gratings in high-precision interferometry for gravitational wave detection Jonathan Hallam, Ph.D. thesis, University of Birmingham (2011)

49. Higher order Laguerre-Gauss mode degeneracy in realistic, high finesse cavities C Bond, P Fulda, L Carbone, K Kokeyama, A Freise, Physical Review D, Vol 84, number 10, id 102002 (2011)

48. Optical properties of 3-port-grating coupled cavities Oliver Burmeister, Ph. D. thesis, university of Hannover (2010)

47. Experimental demonstration of higher-order Laguerre-Gauss mode interferometry Fulda, Paul; Kokeyama, Keiko; Chelkowski, Simon; Freise, Andreas, Physical Review D, vol. 82, Issue 1, id. 012002 (2010)

46. Experimental demonstration of displacement noise free interferometry Antonio Perreca, Ph. D. Thesis, University of Birmingham (2010)

45. Gravitational-wave detector-derived error signals for the LIGO thermal compensation system R S Amin, J A Giaime, Classical and Quantum Gravity, Vol 27, pp 215002 (2010)

44. Automatic Alignment for the first science run of the Virgo interferometer F Acernese et al., Astroparticle Physics, Vol 33, number 3, pp 131 (2010)

43. Interferometer Techniques for Gravitational-Wave Detection A.Freise and K.Strain, Living Reviews in Relativity, 13 (2010)

42. Modeling Thermal Phenomena and Searching for New Thermally Induced Monitor Signals in Large Scale Gravitational Wave Detectors R Amin, Ph. D. thesis, University of Florida (2010)

41. Broadband squeezing of quantum noise in a Michelson interferometer with Twin-Signal-Recycling A Thuering, C Graef, H Vahlbruch, M Mehmet, K Danzmann, R Schnabel, Optics letters, Vol 34, number 6, pp 824 (2009)

40. Virgo Input Mirrors thermal effects characterization, R. Day, V. Fafone, J. Marque, M. Pichot, M. Punturo, A. Rocchi, Virgo technical note VIR-0191A-10 (2010)

39. DC-readout of a signal-recycled gravitational wave detector S Hild, H Grote, J Degallaix, S Chelkowski, K Danzmann, A Freise, M Hewitson, J Hough, H Lueck, M Prijatelj, K A Strain, J R Smith, B Willke, Classical and Quantum Gravity, Vol 26, 055012 (2009)

38. Prospects of higher-order Laguerre-Gauss modes in future gravitational wave detectors S Chelkowski, S Hild, A Freise, Physical Review D, Vol 79, issue 12, id 122002 (2009)

37. Using the etalon effect for in situ balancing of the Advanced Virgo arm Cavities Hild, S.; Freise, A.; Mantovani, M.; Chelkowski, S.; Degallaix, J.; Schilling, R., Classical and Quantum Gravity, Volume 26, Issue 2, pp. 025005 (2009)

36. On Special Optical Modes and Thermal Issues in Advanced Gravitational Wave Interferometric Detectors Jean-Yves Vinet, Living Rev. Relativity 12, (2009)

35. Coupling of lateral grating displacement to the output ports of a diffractive Fabry-Perot cavity J Hallam, S Chelkowski, A Freise, S Hild, B Barr, K A Strain, O Burmeister, R Schnabel, Journal of Optics A: Pure and Applied Optics, Volume 11, Issue 8, pp. 085502 (2009)

34. Quantum noise and radiation pressure effects in high power optical interferometers T R Corbitt, Ph. D. Thesis, Massachusetts Institute of Technology (2008)

33. Measurement and simulation of laser power noise in GEO 600 J R Smith, J Degallaix, A Freise, H Grote, M Hewitson, S Hild, H Lueck, K A Strain, B Willke, Classical and Quantum Gravity, Vol 25, pp 035003 (2008)

32. Development of a signal-extraction scheme for resonant sideband extraction K.Kokeyama, K.Somiya, F.Kawazoe, S.Sato, S.Kawamura, and A.Sugamoto, Classical Quantum Gravity, Vol 25, pp 235013 (2008)

31. Experimental investigation of a control scheme for a zero-detuning resonant sideband extraction interferometer for next-generation gravitational-wave detectors F.Kawazoe, A.Sugamoto, V.Leonhardt, S.Sato, T.Yamazaki, M.Fukushima, S. Kawamura, O.Miyakawa, K.Somiya, T.Morioka, and A.Nishizawa, Classical Quantum Gravity, Vol 25, pp 195008 (2008)

30. Demonstration of Displacement- and Frequency-Noise-Free Laser Interferometry Using Bidirectional Mach-Zehnder Interferometers S.Sato, K.Kokeyama, R.L.Ward, S.Kawamura, Y.Chen, A.Pai, and K.Somiya, Physics Review Letters, Vol 98, id 141101 (2007)

29. Beyond the first Generation: Extending the Science Range of the Gravitational Wave Detector GEO 600. Stefan Hild, Ph. D. thesis, University of Hannover (2007)

28. Squeezed Light and Laser Interferometric Gravitational Wave Detectors Simon Chelkowski, Ph. D. thesis, University of Hannover (2007)

27. The GEO 600 core optics W Winkler et al., Optics Communications, Vol 280, number 2, pp 492 (2007)

26. Measurement of the optical parameters of the Virgo interferometer F Acernese et al., Applied Optics, Vol 46, pp 3466 (2007)

25. Demonstration and comparison of tuned and detuned signal recycling in a large-scale gravitational wave detector S Hild, H Grote, M Hewitson, H Lueck, J R Smith, K A Strain, B Willke, K Danzmann, Classical and Quantum Gravity, Vol 24, pp 1513 (2007)

24. A novel concept for increasing the peak sensitivity of LIGO by detuning the arm cavities S Hild, A Freise, Classical and Quantum Gravity, Vol 24, pp 5453 (2007)

23. Optical modulation techniques for length sensing and control of optical cavities B W Barr, S H Huttner, J R Taylor, B Sorazu, M Plissi, K A Strain, Applied optics, Vol 46, number 31, pp 7739 (2007)

22. The Virgo automatic alignment system F Acernese et al., Classical and Quantum Gravity, Vol 23, pp 91 (2006)

21. Measurement of a low-absorption sample of OH-reduced fused silica S Hild, H Lueck, W Winkler, K A Strain, H Grote, J Smith, M and Malec, M Hewitson, B Willke, J Hough, K Danzmann, Applied optics, Vol 45, number 28, pp. 7269 (2006)

20. Des tests du Modele Standard a la recherche d'ondes gravitationnelles E Tournefier, habilitation (2006)

19. Lock Acquisition Scheme for the Advanced LIGO Optical Configuration O Miyakawa et al., Journal of Physics: Conference Series, 32 265–269 (206)

18. Control sideband generation for dual-recycled laser interferometric gravitational wave detectors B W Barr, O Miyakawa, S Kawamura, A J Weinstein, R Ward, S Vass, K A Strain, Classical and Quantum Gravity, Vol 23, pp 5661 (2006)

17. Compensation of strong thermal lensing in advanced interferometric gravitational waves detectors J. Degallaix, Ph. D. Thesis University of Western Australia (2006)

16. ` <http://deposit.ddb.de/cgi-bin/dokserv?idn=982491557&dok_var=d1&dok_ext=pdf&filename=982491557.pdf%0A>`__ Formulation of instrument noise analysis techniques and their use in the commissioning of the gravitational wave observatory, GEO 600 JR Smith Ph. D. thesis, University of Hannover (2006)

15. Diagonalizing sensing matrix of broadband RSE S.Sato, K.Kokeyama, F.Kawazoe, K.Somiya, and S.Kawamura, Journal of Physics Conference Series, Vol 32, pp 470 (2006)

14. Downselect of the signal extraction scheme for LCGT K.Kokeyama, S.Sato, F.Kawazoe, K.Somiya, M.Fukushima, S.Kawamura, and A. Sugamoto, Journal of Physics Conference Series, Vol 32, pp 424 (2006)

13. Commissioning of advanced, dual-recycled gravitational-wave detectors: simulations of complex optical systems guided by the phasor picture M Malec, Ph. D. thesis, University of Hannover (2006)

12. Mathematical framework for simulation of quantum fields in complex interferometers using the two-photon formalism T Corbitt, Y Chen, N Mavalvala, Physical Review A, vol. 72, Issue 1, id. 013818 (2005)

11. Development of a frequency-detuned interferometer as a prototype experiment for next-generation gravitational-wave detectors K.Somiya, P.Beyersdorf, K.Arai, S.Sato, S.Kawamura, O.Miyakawa, F. Kawazoe, S.Sakata, A.Sekido, and N.Mio, Applied Optics, Vol 44, pp 3179 (2005)

10. Feedforward correction of mirror misalignment fluctuations for the GEO 600 gravitational wave detector J R Smith, H Grote, M Hewitson, S Hild, H Lueck, M Parsons, K A Strain, B Willke, Classical and Quantum Gravity, Vol 22, pp 3093 (2005)

9. The status of GEO 600 H Grote et al., Classical and Quantum Gravity, Vol 22, pp. 193 (2005)

8. Towards dual recycling with the aid of time and frequency domain simulations M Malec, H Grote, A Freise, G Heinzel, K A Strain, J Hough, K Danzmann, Classical and Quantum Gravity, Vol 21, pp 991 (2004)

7. The status of GEO 600 Strain, K.A. et al. In: Proceeding of SPIE: Gravitational Wave and Particle Astrophysics Detectors (2004)

6. Dual recycling for GEO 600 H Grote, A Freise, M Malec, G Heinzel, B Willke, H Lueck, K A Strain, J Hough, K Danzmann, Classical and Quantum Gravity, Volume 21, pp 473 (2004)

5. Characterization of the LIGO 4 km Fabry–Perot cavities via their high-frequency dynamic responses to length and laser frequency variations M Rakhmanov, F Bondu, O Debieu, R L Savage Jr, Classical and Quantum Gravity, Vol 21, pp 487 (2004)

4. Thermal lensing compensation for AIGO high optical power test facility J Degallaix, C Zhao, L Ju, D Blair, Classical and Quantum Gravity, Vol 21, pp 903 (2004)

3. Thermal correction of the radii of curvature of mirrors for GEO 600 Lueck, H. Freise, A. Gossler, S. Hild, S. Kawabe, K. Danzmann, K., Classical and Quantum Gravity, Vol 21, pp 985 (2004)

2. The Next Generation of Interferometry: Multi-Frequency Optical Modelling, Control Concepts and Implementation Andreas Freise, Ph. D. thesis, University of Hannover (2003)

1. Positions-und Orientierungsregelung von als Pendel aufgehaengten Resonatorspiegeln U Weiland, Diploma Thesis, University of Hannover (2000)

2.6 How to cite

FINESSE can be cited in publications, technical notes, or any other work by referencing the project-level Zenodo DOI https://doi.org/10.5281/zenodo.821363. This link will always take you to the latest version of the DOI. Version-specific ones can also be found on Zenodo.

2.6.1 BibTeX

For those using BibTeX to manage their references, the following BibTeX entry can be used to cite the project:

```
@software{brown_2025_12662017,
  author       = {Brown, Daniel David and
                  Freise, Andreas and
                  Cao, Huy Tuong and
                  Ciobanu, Alexei and
                  Gobeil, Jeremie and
                  Green, Anna and
                  Hapke, Paul and
                  Jones, Philip and
                  van der Kolk, Miron and
                  Kuns, Kevin and
                  Leavey, Sean and
                  Perry, Jonathan Warren and
                  Rowlinson, Samuel and
                  Sallé, Mischa},
  title        = {FINESSE},
  month        = mar,
  year         = 2025,
  publisher    = {Gitlab},
  version      = {3.0a32},
  doi          = {10.5281/zenodo.12662017},
  url          = {https://doi.org/10.5281/zenodo.12662017},
}
```

Chapter 3

Installing Finesse

There are multiple ways to install and use FINESSE. See the table below to choose the best option for you.

Finesse installation	Description
Cloud-based (No installation)	• Run FINESSE within cloud-based Jupyter notebooks. • No installation required, only a browser and internet access. • Use the latest stable release. • Direct link to the "Getting started with Finesse 3" on two free platforms:  • Open-source, community driven, free service • Limited in resources  • Free to use, with paid options for more resources. • Require a Google account.
User (page 20)	• Standard FINESSE user installation. • Run code on your own computer. • Use the latest stable release. • Basic command-line knowledge is needed. • Requirements: conda (page 20)
Developer (page 21)	• Finesse installation for developers and contributors. • Compile and run code on your own computer. • Use stable releases, development branches or your own branches. • Some knowledge of command line and git is needed. • Requirements: conda (page 21), git (page 21), make (page 22), VS Build Tools (Windows only) (page 22)

3.1 User installation

3.1.1 Requirements

Conda

The supported way to install FINESSE is by installing it in a conda environment. You will need `conda` or a similar software (e.g. `mamba`) to manage the conda environment(s).

How to install `conda` with Miniforge

To install `conda`, we recommend using Miniforge, which is a minimal conda distribution that is configured to use conda-forge as the default channel, the channel where FINESSE is published. `mamba` is also installed with miniforge.

Linux/macOS

1. Follow the instructions on download miniforge or use the following command to download and execute the installer:

   ```
   $ curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)-$(u
   $ bash Miniforge3-$(uname)-$(uname -m).sh
   ```

2. Follow the instructions in the installer:
   - Accept the license by typing *yes* and press enter.
   - Press enter to accept the default installation location.
   - Type *yes* and press enter to initialize miniforge.
3. Close and reopen a terminal to start using `conda`.

Windows

1. Download the Miniforge Windows installer
2. Run the installer and follow the instructions (default options are ok).
3. To open a prompt with `conda` available, search for *Miniforge prompt* in the start menu.

For more information about installing miniforge, see the miniforge install.

3.1.2 Installation procedure

To create a new conda environment named *finesse3-env* with FINESSE installed, execute in a terminal

```
$ conda create --name finesse3-env finesse
```

If you didn't install `conda` *via Miniforge, you may need to add the* `--channel conda-forge` *option to the command above.*

After the installation is complete, you can activate the environment with:

```
$ conda activate finesse3-env
```

You now have a working FINESSE installation. See How to write and run Finesse code (page 24) for some useful tools and methods to run FINESSE code.

A good starting point to learn FINESSE are the notebooks from the Finesse 3: Getting started repository, or the Getting started (page 30) section of this documentation. For more info on how to manage conda environments see Conda environment primer (page 28).

3.2 Developer installation

The developer install uses the source code to compile and install FINESSE in a conda environment. This is the recommended way to install FINESSE if you want to contribute to the project, use the latest development branch, or your own branches.

3.2.1 Requirements

Conda

The supported way to install FINESSE is by installing it in a conda environment. You will need `conda` or a similar software (e.g. `mamba`) to manage the conda environment(s).

How to install `conda` with Miniforge

To install `conda`, we recommend using Miniforge, which is a minimal conda distribution that is configured to use conda-forge as the default channel, the channel where FINESSE is published. `mamba` is also installed with miniforge.

Linux/macOS

1. Follow the instructions on download miniforge or use the following command to download and execute the installer:

   ```
   $ curl -L -O "https://github.com/conda-forge/miniforge/releases/latest/download/Miniforge3-$(uname)-$(u
   $ bash Miniforge3-$(uname)-$(uname -m).sh
   ```

2. Follow the instructions in the installer:
   - Accept the license by typing *yes* and press enter.
   - Press enter to accept the default installation location.
   - Type *yes* and press enter to initialize miniforge.
3. Close and reopen a terminal to start using `conda`.

Windows

1. Download the Miniforge Windows installer
2. Run the installer and follow the instructions (default options are ok).
3. To open a prompt with `conda` available, search for *Miniforge prompt* in the start menu.

For more information about installing miniforge, see the miniforge install.

Git

FINESSE uses git to manage its source code. You will need it to clone the repository, change branches, update your local copy of the code, and contribute to the project.

How to install `git`

Follow the instructions at git-scm.com/download for your system.

or

Install `git` in your conda environment(s). To install it in your active environment, within a terminal/prompt, execute the following command

```
$ conda install git
```

Make

For compiling code, installing FINESSE, and managing the conda environment via the *Makefile*.

How to install `make`

If your system doesn't have `make` available (*e.g.* Windows) or you want an up-to-date version (*e.g.* macOS), you can install `make` in your environment. To add `make` to your active environment, execute

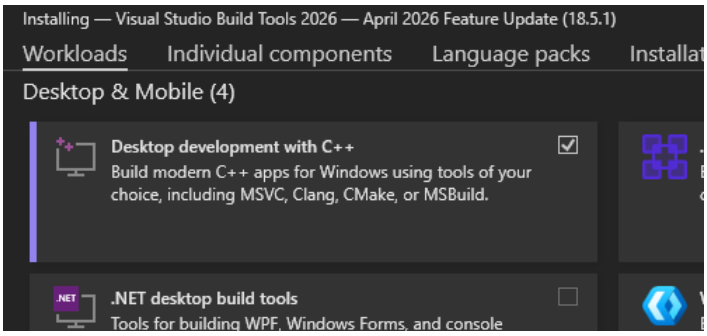
```
$ conda install make
```

VS Build Tools (Windows only)

On Windows systems, the `VS Build Tools` are needed to compile FINESSE.

How to install the `VS Build Tools`

1. Download and execute the VS Build Tools installer.
2. Select the *Desktop development with C++* workload.



3. Click on *Install* and wait for the installation to complete.

3.2.2 Installation procedure

Cloning the repository

The source code is hosted on GitLab at https://gitlab.com/ifosim/finesse/finesse3. To clone the repository on your system, open a terminal and navigate where you want your copy of FINESSE source to be. During the cloning process, a directory named `finesse3` will be created in your current directory.

Cloning with SSH

Using SSH to clone the repository allows a seamless interaction with GitLab. To be able to use SSH, you will need to setup your SSH keys on GitLab.

To clone FINESSE using SSH, execute

```
$ git clone git@gitlab.com:ifosim/finesse/finesse3.git
```

Cloning with HTTPS

You can clone the repository without GitLab authentication by using HTTPS. However, if you want to push changes to the repository, you will need an account on GitLab and your GitLab credentials will be needed for certain interaction with GitLab.

To clone FINESSE using HTTPS, execute

```
$ git clone https://gitlab.com/ifosim/finesse/finesse3.git
```

If you initially setup your repository with HTTPS, but wish to use SSH instead, you can change the remote URL to use SSH by executing:

```
$ git remote set-url origin git@gitlab.com:ifosim/finesse/finesse3.git
```

Creating the environment

You can create a FINESSE developer environment by using the *finesse3-dev* target of the Makefile. To do so, open a terminal and navigate to the FINESSE source directory. Then execute

```
$ make finesse3-dev
```

This will automatically create a new environment called *finesse3-dev*, install the required packages, some additional packages, compile FINESSE and install it in the environment as an editable package.

How to create the environment manually

If for any reason, you prefer to not use the *finesse3-dev* make target, you can create the environment by executing the following commands from the FINESSE source root directory.

1. Create the environment, by executing from the FINESSE root directory

   ##### Linux/macOS

   ```
   $ conda env create -n finesse3-dev -f environment.yml
   ```

   ##### Windows

   ```
   C:  Users  user  finesse3> conda env create -n finesse3-dev -f environment-win.yml
   ```

2. Activate the environment and install the additional packages

   ```
   $ conda activate finesse3-dev
   ```

3. Install FINESSE as an editable package

   ```
   $ pip install --editable .
   ```

4. Install the pre-commit hooks

   ```
   $ pre-commit install
   ```

After the installation is complete, you can activate the environment created by executing

```
$ conda activate finesse3-dev
```

You can also run the test suite to make sure that everything is working properly

```
(finesse3-dev) $ pytest tests
```

3.2.3 Installation notes

Updating after changes in the code

Because FINESSE was installed in the environment as an editable package (*i.e.* the environment is linked to the FINESSE source directory), any modification of the Python source code, including when changing branch with `git`, will be reflected in the conda environment. This is not the case for the Cython source code as what Python sees is the compiled package. To recompile the package, execute the following in the source root directory, with the *finesse3-dev* environment (or your own custom environ environment) activated:

```
(finesse3-dev) $ make
```

For more info on how to manage conda environments see Conda environment primer (page 28).

Using `ccache` for faster cython compilation

If you are going to re-compile the Cython extensions often during development, it might be worth it to set up ccache. Ccache is a compiler cache. It speeds up recompilation by caching the result of previous compilations and detecting when the same compilation is being done again.

3.3 How to write and run Finesse code

FINESSE is a Python package and can be used inside any Python code. Usually, that will be as a stand-alone Jupyter Notebook, Python script, or inside a broader project.

3.3.1 Jupyter notebooks

A Jupyter Notebook combines blocks of text, code and output (including plotting) to create a single document (a.k.a. literate programming). This approach to writing code is especially useful in a scientific context where the live kernel and block-level execution allows fast code iterations, and the literate programming allows documented calculation that can be easily shared.

This is a common way to write and run FINESSE code and probably where you should start. Most examples in the documentation are written as Jupyter Notebooks.

Jupyter

For stand-alone notebooks and small projects you can use Jupyter Lab to easily create, edit and execute notebooks in your browser. Jupyter Lab can be installed in your conda environment with:

```
(finesse3-env) $ conda install jupyterlab
```

Then you can start a notebook server in your project directory with:

```
(finesse3-env) $ jupyter lab
```

This will open a browser window with the Jupyter Lab interface where you can create, browse, edit, and run notebooks.

Integrated Development Environments

More complex projects may benefit from using a modern Integrated Development Environment (IDE) such as VS Code with the Jupyter extension or PyCharm that support notebooks directly.

This allows you to use the full power of an IDE for editing, managing, debugging and running your code and notebooks.

3.3.2 Scripts

You can write Python scripts that use the FINESSE Python API and functionalities. These scripts can be executed from the command line by calling the Python interpreter with the script as argument:

```
(finesse3-env) $ python my_script.py
```

This can be a versatile way to execute some calculation especially if you need to run them on a cluster, a remote server, or via other scripting tools.

3.3.3 kat3

FINESSE include a KatScript interpreter called `kat3`. It exists mostly for historical reason and it is not recommended to use it for new projects. Instead, using the Python scripts or notebooks.

3.4 Editor extensions

A number of extensions to text editors and IDEs have been written to ease working with FINESSE and KatScript code. This page contains information and installation instructions for each of them.

Table of Contents

- JupyterLab (page 25)
- Visual Studio Code (page 25)

- Vim (page 26)
- Pygments (page 26)

3.4.1 JupyterLab

The JupyterLab extension provides syntax highlighting of KatScript within Python triple-quoted strings that start with a special marker: `#kat` for Finesse 3 style KatScript, or `#kat2` for Finesse 2 style KatScript.

```
import finesse

model = finesse.Model()
model.parse("""
#kat
l l1 P=1
s s1 l1.p1 m1.p1 L=5
m m1 R=0.9 L=0

pd pdo m1.p2.o
qnoised qd m1.p2.o nsr=true

fsig(1)
xaxis(l1.P, lin, 1, 100, 100)
""")
```

```
import finesse

model = finesse.Model()
model.parse_legacy("""
#kat2
l l1 1 0 n1
s s1 5 n1 n2
m2 m1 0.9 0 0 n2 n3

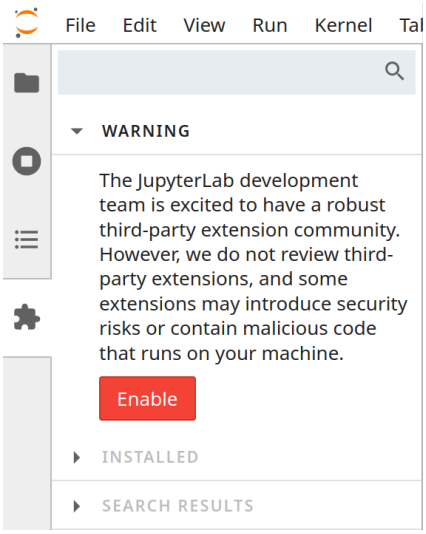
pd pdo n3
qnoisedS qd 1 $fs n3

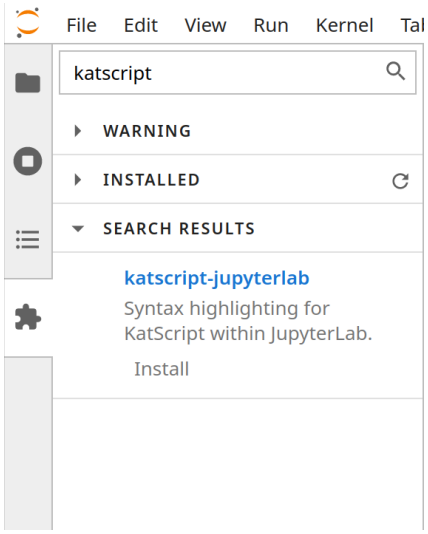
fsig sig 1
xaxis l1 P lin 1 100 100
""")
```

You can install the JupyterLab extension either via a Python packaging tool such as pip or the JupyterLab extension manager. To install via pip, run:

```
$ pip install katscript-jupyterlab
```

To install via the JupyterLab extension manager, enable it and search for KatScript:





3.4.2 Visual Studio Code

The Visual Studio Code extension provides syntax highlighting of KatScript in `.kat` files. It defaults to Finesse 3 style KatScript; to highlight Finesse 2 style KatScript, start the file with the comment `#kat2`.

You can install the Visual Studio Code extension from the built in extension manager:

```
l l1 P=1
s s1 l1.p1 m1.p1 L=5
m m1 R=0.9 L=0

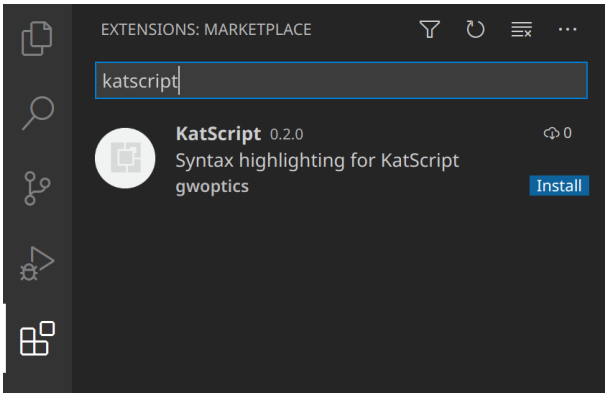
pd pdo m1.p2.o
qnoised qd m1.p2.o nsr=true

fsig(1)
xaxis(l1.P, lin, 1, 100, 100)
```

```
#kat2
l l1 1 0 n1
s s1 5 n1 n2
m2 m1 0.9 0 0 n2 n3

pd pdo n3
qnoisedS qd 1 $fs n3

fsig sig 1
xaxis l1 P lin 1 100 100
```



3.4.3 Vim

The Vim extension provides syntax highlighting of KatScript in `.kat` files. It defaults to Finesse 3 style KatScript; to highlight Finesse 2 style KatScript, start the file with the comment `#kat2`.

```
l l1 P=1
s s1 l1.p1 m1.p1 L=5
m m1 R=0.9 L=0

pd pdo m1.p2.o
qnoised qd m1.p2.o nsr=true

fsig(1)
xaxis(l1.P, lin, 1, 100, 100)
```

```
#kat2
l l1 1 0 n1
s s1 5 n1 n2
m2 m1 0.9 0 0 n2 n3

pd pdo n3
qnoisedS qd 1 $fs n3

fsig sig 1
xaxis l1 P lin 1 100 100
```

You can install the Vim extension using your preferred method of managing extensions from the git repository.

3.4.4 Pygments

The Pygments extension is installed along with Finesse, and contains two highlighters. The first applies to `.kat` files, and requires no further setup - Pygments will automatically use the right highlighter. The second applies to KatScript within Python triple-quoted strings that start with `#kat`, and must be specified manually when using Pygments:

3.5 Other ways to install Finesse

The supported way to install Finesse is by using the conda-forge package (page 20). In addition of conda-forge, Finesse is also distributed on pypi (the Python Package Index) and can installed using many package managers. If you choose to install Finesse using those alternative package managers, it is assumed that you are knowledgeable enough about the tool to adapt the installation process to your needs and manage it yourself. The commands are given here mostly as references. The list of alternatives presented here is not exhaustive.

Note that Finesse is currently not available for Windows on pypi. See Issue #518.

```
$ pygmentize model.kat -f terminal
l l1 P=1
s s1 l1.p1 m1.p1 L=5
m m1 R=0.9 L=0

pd pdo m1.p2.o
qnoised qd m1.p2.o nsr=true

fsig(1)
xaxis(l1.P, lin, 1, 100, 100)
```

```
$ pygmentize -l katscriptinpython model.py -f terminal
import finesse

model = finesse.Model()
model.parse("""
#kat
l l1 P=1
s s1 l1.p1 m1.p1 L=5
m m1 R=0.9 L=0

pd pdo m1.p2.o
qnoised qd m1.p2.o nsr=true

fsig(1)
xaxis(l1.P, lin, 1, 100, 100)
""")
```

3.5.1 Pip

pip is the package installer for Python. You can install package from PyPI with it.

```
$ # Create a new virtual environment
$ python -m venv finesse_env
$ source finesse_env/bin/activate
$
$ # Install finesse
$ pip install finesse
```

To install FINESSE as an editable package for development, change the last lines for:

```
$ # Install Finesse as editable
$ pip install --editable /path/to/finesse3
```

After activation of the environment, subsequent command will be executed in the environment.

3.5.2 UV

uv is an fast Python package and project manager, written in Rust. It manages virtual environments automatically. To create a new environment and install FINESSE:

```
$ # Initialize a new project
$ uv init finesse-project
$ cd finesse-project
$
$ # Add finesse to the project
$ uv add finesse
```

To install FINESSE as an editable package for development, change the last lines for:

```
$ # Add finesse to the project as editable
$ uv add --editable /path/to/finesse3
```

To run a Python script using the environment, when inside the project directory tree use:

```
$ uv run python script.py
```

3.5.3 Poetry

Poetry is a tool for dependency management and packaging in Python. It automatically creates and manages a virtual environment for your projects.

```
$ # Initialize a new project
$ poetry new finesse-project
$ cd finesse-project
$
$ # Add finesse to the project
$ poetry add finesse
```

To install FINESSE as an editable package for development, change the last lines for:

```
$ # Add finesse to the project as editable
$ poetry add --editable /path/to/finesse3
```

To run a Python script using the environment, when inside the project directory tree use:

```
$ poetry run python script.py
```

3.6 Conda environment primer

3.6.1 conda vs mamba

`mamba` was created as a drop-in replacement for `conda`. It is a reimplementation of the package manager in C++. An environment created with `conda` can be modified and activated by `mamba`, and vice versa. Some slight differences exist on some commands, but all the commands displayed here can be used by both.

Use either one you prefer.

3.6.2 Creating a new environment

By listing the packages

```
$ conda create --name <env_name> <pkg1> [<pkg2> [...]]
```

By using an environment file

```
$ conda create --name <env_name> -f <environment.yml>
```

What is the environment file?

The environment file is a YAML file that contains the list of packages to install in the environment. It can be used to create a new environment, or to update an existing one.

The environment file is named `environment.yml` by convention, but it can be named anything you want.

A basic environment file looks like this:

```yaml
name: finesse3-env  # name of the environment
channels:           # channel to use
    - conda-forge
dependencies:       # What to install
    - python=3.14
    - finesse
    - numpy
    - pandas
    - matplotlib
    - jupyterlab
    - pip
    - pip:          # installed by pip, not conda
        - tqdm
```

3.6.3 Activating an environment

To be able to use an environment, it must first be activated. By activating it, `conda` will modify the current terminal variables so that the executables and libraries from the environment are available, and used instead of any system ones. For example, if *git* is available on your system but that you also installed it in your environment, the one from the environment will be used after activation of the environment.

```
$ conda activate <env_name>
```

Note

Commands targeting the active environment can also be executed without activating if by adding *–name <env_name>* to the command.

3.6.4 Deactivating an environment

```
$ conda deactivate
```

3.6.5 Deleting/removing an environment

```
$ conda env remove --name <env_name>
```

3.6.6 Executing a command

After activating an environment, all following commands executed inside this terminal will be executed within the environment.

You can also execute a command in an environment without activating it by using `conda run`:

```
$ conda run --name <env_name> <cmd>
```

3.6.7 Adding packages to an environment

```
$ conda add <pkg1> [<pkg2> [...]]
```

3.6.8 Updating an environment

Note

Updating an environment can lead to conflicts between packages (notably with pip packages). It is often simpler and cleaner to delete the environment and recreate it.

Update all packages

```
$ conda update --all
```

Update specific packages

```
$ conda update <pkg1> [<pkg2> [...]]
```

Install a specific version of packages

```
$ conda update "<pkg1>=<version>" [<pkg2>[=<version>] [...]]
```

Chapter 4

Getting started

If you are new to FINESSE, here is a good place to start. You can first learn how to design your first Finesse model (page 30). You can examine a first notebook example (page 32) and run it on your system. Finally, you can familiarize yourself with some key concepts (page 35) of FINESSE and the interactive help system (page 39).

If you have general or technical questions on FINESSE, you can ask them on our Finesse Matrix channel. This public channel is the easiest way to get help from us or the community. You can also use the mailing list at finesse-users AT nikhef.nl.

For users within the gravitational wave community we have a Mattermost channel on Ligo servers. This will require you to log in with your Albert.Einstein credentials.

If you encounter any issue using FINESSE, please file a bug report (page 449).

4.1 Your first Finesse model

4.1.1 1. Mapping the Optical System

To run a simulation, you need to create a simple text describing your optical setup, simulation commands and optional comments. This can be a multi-line string in a Jupyter notebook cell, or a text file ("kat file") that you load. Before writing any code, start by visualising and mapping the system you want to model. Sketch the optical setup and identify all major components such as lasers, mirrors, beamsplitters, free spaces, and others. Then, you must break the system down into basic components and the nodes that connect them. Nodes represent the points of connection or separation between two components. Finally, assign a short and unique name to every component. This name is how you'll refer to the element in the input file commands and how Finesse will reference it in any output or internal messages.

4.1.2 2. Describing Components and Connections

The input file consists of a series of component descriptions and commands. We call the Finesse syntax to describe these components KatScript. A component description is typically entered on a single line following the following format:

```
keyword name parameters
```

The keyword specifies the component type (for example *m* for mirror, *s* for space, *l* for laser). Following the name are the parameters, which are numerical values defining the component's properties, such as power (*P*), frequency (*f*) for a laser, or power reflectivity (*R*), power transmissivity (*T*), and tuning (*phi*) for a mirror.

For example, a laser named *laser1* with *P=1* W, zero frequency offset from the default frequency (*ref*), is described as:

```
l laser1 P=1 f=0
```

A mirror named *mirror1* with *R=0.9* and *T=0.1* is described as:

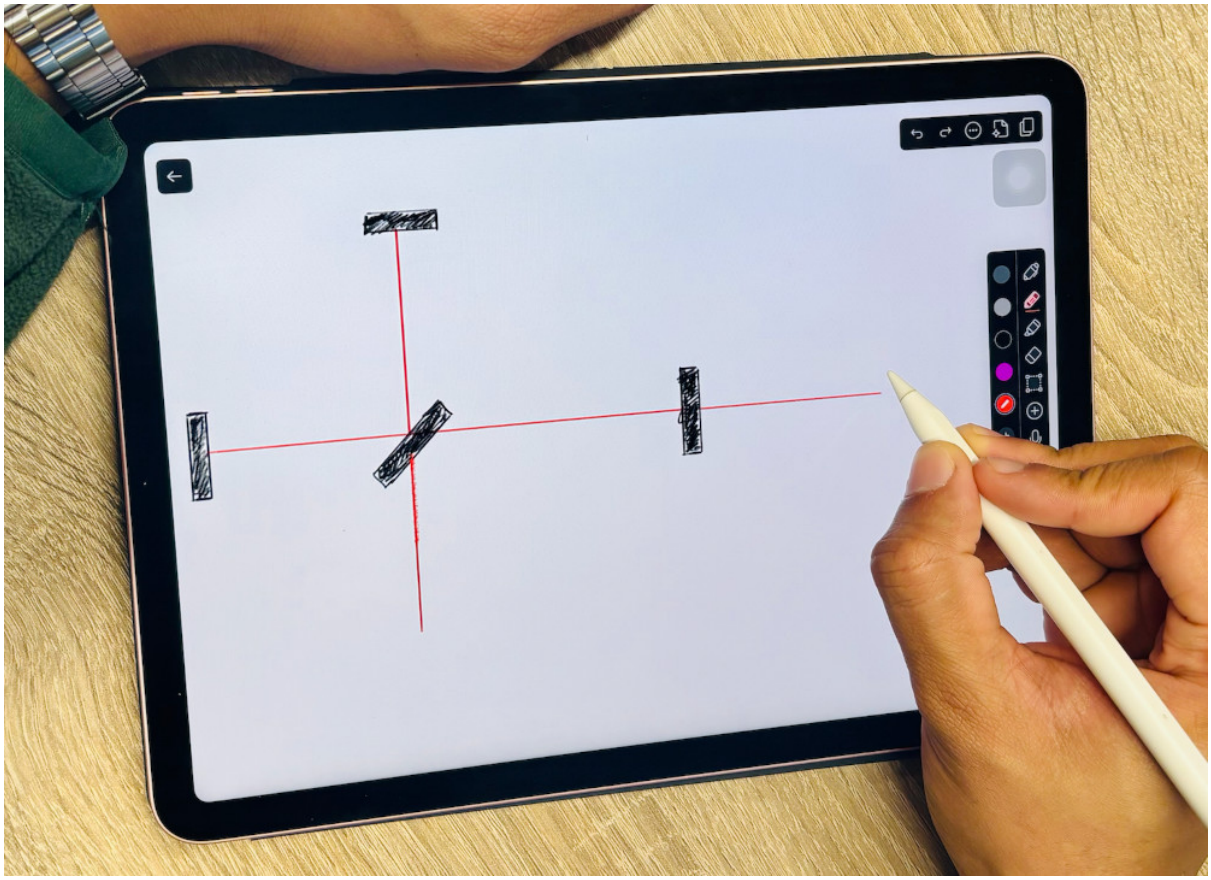


Fig. 4.1: Sketching a model by hand

```
m mirror1 R=0.9 T=0.1
```

In Finesse, connectivity is defined using 'Ports' and 'Nodes'. This mechanism allows the model to handle not only the flow of light but also complex feedback loops involving electrical and mechanical signals.

Most components feature various Ports (e.g., *p1*, *p2*) which are its connection points. Each port then contains one or more 'Nodes' that represent the actual light or signal information flow. You access these specific states directly using the standard convention: `component.port.node`.

- **Optical Ports** always contain an Input Node (*.i*) and an Output Node (*.o*), describing the field entering and exiting the component. Optical connections between elements must be made using a *Space* component (or alternatively using the *Link* command).
- **Electrical and Mechanical Ports** contain Signal Nodes. These nodes represent small, linearised states (such as mirror motion or voltage) and are vital for modeling active control systems, as they can be connected using *Wires* to form feedback loops.

Most components connect to exactly two nodes. Sources, like lasers connect to one, and beam splitters connect to four. Crucially, every node must connect to at least one component but never more than two, or the system description will be inconsistent.

Detectors are special components. They can be placed at any node in the interferometer (without blocking the existing light or signal flow at that node), and each detector specifies one output variable to be computed. Specifying multiple detectors allows for the computation of several signals simultaneously.

4.1.3 3. Defining the Simulation Task

Commands in Finesse are primarily used to define model properties, enabling advanced analysis features and higher-order modes (HOMs), and these are not yet necessary for your first simulations.

All simulations in Finesse are run via the Python Application Programming Interface (API). An API is essentially a set of clearly defined tools and protocols that allows your Python code to interact with and control the Finesse simulation engine. This allows for adaptable, computationally efficient analysis directly within your scripts.

All simulations are executed using Actions, which are single or multiple atomic simulation tasks that form a complete Analysis to be performed on the model. This system is efficient because the model is built only once, allowing complex analyses to be defined and run sequentially or concurrently.

The most basic simulation task is performed by the *Xaxis* Action. To use it, you must import the *Xaxis* class from *finesse.analysis.actions* and pass the instantiated object directly to the model.run() method. This Action sweeps a parameter, computes all detector outputs, and returns the result as a Solution object.

For example, to sweep the power of the laser (*l1.P*) linearly from 0 to 1 over 10 steps, you write:

```
from finesse.analysis.actions import Xaxis

# ... model definition here ...
sol = model.run(Xaxis(model.l1.P, 'lin', 0, 1, 10))
```

Finesse includes a wide range of specialized Actions and Analysis tools designed for different computational tasks, such as finding a system's operating point, calculating noise curves, or modeling complex locking schemes. [link to finesse analysis actions]()

The next section contains a first FINESSE example.

This page is a notebook. You can download it: `a_first_finesse_example.ipynb`

4.2 A first Finesse example

In this example notebook, we will look at a so-called dual-recycled Michelson Fabry–Perot interferometer, similar to the configuration used in Advanced LIGO. The key learning goals of this notebook should be:

1. Understanding how to build a Finesse model and define the components in Finesse.
2. Learning how to analyse/study the model using detectors.

3. Learning how to perform analyses and actions using Finesse.

The purpose of this notebook is to introduce a recommended workflow structure for building your models and to give an overview of the basic functionalities of Finesse. The structure we follow here is the one we recommend adopting for your future Finesse models. By the end of this notebook, you should be able to construct your own Finesse model.

```
import finesse
# if you want to do make custom plots you would import matplotlib
#import matplotlib.pyplot as plt
from finesse.analysis.actions import Xaxis
finesse.init_plotting()
```

4.2.1 Building a Finesse model with KatScript

The Finesse model and the various components/optics (building blocks) are defined below in a string with KatScript syntax.

```
# Model
# ----------------------------------------------------------------
# Creating the basic Finesse model object
base = finesse.Model()

# Building an optical layout into the model
base.parse("""
l LI P=125
s s0 LI.p1 eom1.p1 L=0.1
mod eom1 f=6e6 midx=0.2
s s1 eom1.p2 PRM.p1 L=0.1

# Power recycling mirror
m PRM T=0.03 L=37.5u phi=90
s prc PRM.p2 bs1.p1 L=53

# Central beamsplitter
bs bs1 R=0.5 T=0.5 phi=0 alpha=45

# X-arm
s sx bs1.p3 ITMX.p1 L=4.5
m ITMX T=0.014 L=37.5u phi=89.997
s Lx ITMX.p2 ETMX.p1 L=3995
m ETMX T=0.01 L=37.5u phi=89.997

# Y-arm
s sy bs1.p2 ITMY.p1 L=4.45
m ITMY T=0.014 L=37.5u phi=3m
s Ly ITMY.p2 ETMY.p1 L=3995
m ETMY T=0.01 L=37.5u phi=3m

# Signal recycling mirror
s src bs1.p4 SRM.p1 L=50.525
m SRM T=0.2 L=37.5u phi=-90

""")
```

```
<finesse.model.Model at 0x7fdbc2a56900>
```

4.2.2 Adding detectors in Finesse

Now we need to think what we want to do with the interferometer. It can be helpful to ask this question in the form of: 'what plot should I make?'. Then you can define what detector to add to the simulation (what is on the y-axis) and which parameter to tune (what is on the x-axis).

```
# creating a 'deep' copy of the base model.
# This is recommended as changes made to this model does not effect the base model. This
# is a good practice as it helps in debugging and studying different aspects of your model.

base2 = base.deepcopy()

# Adding detectors to the model
base2.parse(
"""
# photodiodes (DC)
pd b1 ETMX.p2.o
pd b2 ETMY.p2.o
# amplitude detector measuring the amplitude of light at a given frequency
ad b4ad PRM.p1.i f=eom1.f
"""
)

# We run the model to test the outputs of the detector for the current configuration
sol = base2.run()
print(f"Power transmitted through the end mirrors, ETMX: {sol['b1']:.4} W, ETMY: {sol['b2']:.
↪4} W")
print(f"Complex amplitude of one RF field component entering the interferometer:  {sol['b4ad
↪']:.4}")
```

```
Power transmitted through the end mirrors, ETMX: 2.525 W, ETMY: 2.525 W
Complex amplitude of one RF field component entering the interferometer:  (0.01399+1.112j)
```

4.2.3 Actions, Analyses, & Solutions

In order to make a plot we rely on syntax that we call 'actions'. To change a parameter we use an 'Xaxis' action.

```
# making a copy of the model with detectors
base3 = base2.deepcopy()

# scanning the laser power and plotting the output of the detectors
sol1 = base3.run(Xaxis(base3.LI.P, 'lin', 100, 150, 50))
sol1.plot();
```

```
<Figure size 576x355.968 with 1 Axes>
```

```
<Figure size 576x355.968 with 2 Axes>
```

As expected, the power and amplitude on all detectors increase proportionally with the laser power. The built-in plot command provides easy access to a basic plot. You can also access the data in the sol1 object to create a customised plot.

As a second example, we move an end mirror of the Michelson interferometer. The movement (scan) is microscopic — we change what is called the tuning of the mirror, which is specified in degrees, with 360° corresponding to a change in mirror position of one wavelength of light.

```
# making a copy of the model with detectors
base4 = base2.deepcopy()

# scanning the microscopic position (phi) of end test mass (ETMX)
sol2 = base4.run(Xaxis(base4.ETMX.phi, 'lin', -200, 200, 400, relative=True))
# only plotting the detector on transmission for ETMX
sol2.plot("b1", logy=True);
```

```
<Figure size 576x355.968 with 1 Axes>
```

In this case, we plot only the data from one photodiode. This shows the change in power in the interferometer, which is determined by a combination of the resonance condition in the so-called power-recycled cavity and the interference condition in the Michelson interferometer. The triple peaks arise because the EOM generates two RF sidebands with a frequency offset from the laser's input field.

See also

Migrating from Pykat + Finesse 2 (page 557)

4.3 Key concepts

There are three distinct layers in the FINESSE interaction that will be referred to throughout this documentation:

The model
: a particular optical configuration or layout (see `finesse.model`)

The simulation
: when a FINESSE model is executed to generate a particular output (see `finesse.simulations`)

The solution
: data generated by a particular simulation (see `finesse.solutions`)

4.3.1 Detectors and probes

A *detector* is defined as a component that converts an optical input into an electrical output, this can be a simple photo diode measuring a light power, or a complex device with multiple (optical) inputs and signal processing such as demodulation. A detector will have input and output nodes, will be computed as part of the matrix representing the model. It can only perform linear operations (for signal processing) and therefor applies only to of AC signals. For example, a detector cannot measure the light power at DC. However, detectors can be useful for modelling signal propagation in mixed systems (optical, electrical, mechanical).

A *probe* is a user-defined output of any simulation result, for example a specific light field amplitude or an electrical signal. A probe can include specific post-processing of the simulation output, for example, summing up powers in optical field components, or demodulation. A probe is not part of the model itself and does not block the light or signal path and it can therefor include non-linear post-processing. Many probes can be connected to each node without interfering with the model or each other.

4.3.2 The model graph

Any FINESSE model is defined as a nodal network, which is stored in a graph. See model graph (page 35) for an extensive explanation.

The model graph

Simulations in Finesse are always done using a Model. A Model is a mathematical description of a (hypothetical) optical experiment.

In the end, the model will be 'solved' by the means of inverting a matrix that represents a system of linear equations. The result of this inversion is a complex description of the light field (homs/freqs) at every 'point' in the model. What the 'points' in the model are, is more formally defined in the model 'graph'.

When defining your Finesse model, you are not supplying the three-dimensional coordinates of your components in some reference frame. So not:

```
m m1 x=5.6 y=2.5 z=1.2
```

but:

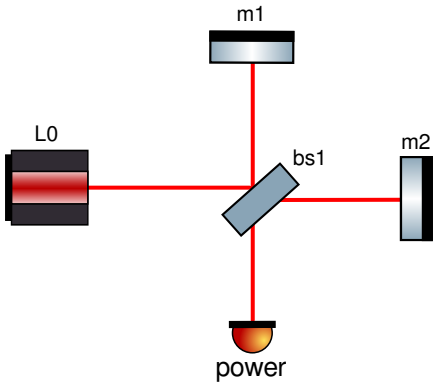
```
m m1 R=0.9 T=0.1
s s1 m1.p2 m2.p1
m m2 R=0.9 T=0.1
```

We want to describe *how* components are connected, not *where* components are (some caveats regarding tilt/yaw of mirrors/beamsplitters).

This graph of connectivity can be described on three levels:

- Components
- Ports
- Nodes

We will be using the following Michelson model.




```
import networkx as nx

import finesse
from finesse.plotting.graph import show_graphviz

finesse.configure(plotting=True)

kat = finesse.Model()
kat.parse(
    """

    l L0 P=1  # Add a Laser named L0 with a power of 1 W.
    s s0 L0.p1 bs1.p1  # Space attaching L0 <-> m1 with length of 0 m (default).

    bs bs1 R=0.9 T=0.1  # central beamsplitter

    # top mirror
    s s1 bs1.p2 m1.p1 L=1
    m m1 R=0.99 T=0.01

    # right mirror.
    s s2 bs1.p3 m2.p1 L=1
    m m2 R=0.991 T=0.009

    pd power bs1.p4.o
    """
);
```

We can visualize the model graph on the component level. This gives us a high-level overview, but does not contain all the information in the katscript. We can see for example that both mirrors are connected to the central beamsplitter, but we don't see *how* they are connected.

```
kat.plot_graph(network_type="component", layout="dot")
```

```
<IPython.core.display.SVG object>
```

Since our beamsplitter is not symmetric (it will reflect most of the incoming light towards `m1`), we need a graph with more information. We can see this on the port level:

```
kat.plot_graph(network_type="port", format="svg", path=None, layout="dot", show=True);
```

```
<IPython.core.display.SVG object>
```

We see that each component actually consists of one or multiple 'ports', which are numbered `p1` `p2` etcetera. They are represented with the ovals. The lines represent connections between different ports. Besides connections between ports of *different* components, there are also connections between ports of the *same* component. Both mirrors (page 142) and beamsplitters (page 148) have multiple ports representing internal reflections and transmissions.

With this view we can see that `m1` is connected to `p2` of the beamsplitter, which contains the reflected light from `bs.p1` and the transmitted light from `bs1.p4`.

Still there is information missing from view of the model. At `m1.p1`, there is both light coming *into* the mirror and *going out* of the mirror (reflection). Actually all the optical connections in Finesse are bidirectional. This we can see more clearly in the node overview

```
kat.plot_dcfields_graph(add_fields=False);
```

```
<IPython.core.display.SVG object>
```

This overview can be confusing, especially for larger models. It show that every port actually has two nodes:

- `i` for incoming
- `o` for outgoing

On the right side of the image, we can see the full mirror, with `m2.p1.i` *reflecting* into `m2.p1.o` and `transmitting` into `m2.p2.o`.

This model graph corresponds directly to the system of linear equations that Finesse solves. Every single node here is associated with a complex amplitude of the light field at that location. Initially, only `L0.p1.o` has a value of `1+0j`. After inverting the interferometer matrix and multiplying it with the RHS vector (containing all these nodes), we get a *steady-state* solution of all complex amplitudes at every node in the model graph.

Every arrow in the graph represents a linear equation that determines the relationship between two nodes. Some of these arrows are `Space` components (between nodes of different components), while other are internal equations. The *parameters* used in these equations are specified in the katscript. A very simplified example:

$$\text{m2.p1.o} = \sqrt{m2.R} \cdot \text{m2.p1.i}$$

Below we visualize the steady state solution and the couplings using the `plot_dc_fields_graph` method:

```
kat = finesse.Model(
    """
l l1
s s0 l1.p1 m1.p1
m m1 R=0.9 T=0.1
"""
)

kat.plot_dcfields_graph(add_fields=True, add_operators=True, operator_labels=True);
```

```
<IPython.core.display.SVG object>
```

Using some Finesse internals, we can directly relate this to the interferometer matrix. See an overview of Finesse internals (page 484) for more details.

```
with kat.built() as sim:
    carrier_solver = sim.carrier
    klu_mat = carrier_solver.M()
    klu_mat.print_matrix(round_values=True)
    klu_mat.print_rhs(round_values=True)
```

```
Matrix carrier: nnz=12 neqs=6
    (col, row) = value
    (0, 0) = 1.00+0.00j  : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0 -> I,node=l1.p1.i,f=0,
↪fidx=0,Neq=1 mode=0
    (1, 1) = 1.00+0.00j  : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=l1.p1.o,f=0,
↪fidx=1,Neq=1 mode=0
    (1, 2) = -1.00-0.00j : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=m1.p1.i,f=0,
↪fidx=2,Neq=1 mode=0
    (2, 2) = 1.00+0.00j  : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.p1.i,f=0,
↪fidx=2,Neq=1 mode=0
    (2, 3) = -0.95-0.00j : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.p1.o,f=0,
↪fidx=3,Neq=1 mode=0
    (2, 5) = 0.00-0.32j  : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.p2.o,f=0,
↪fidx=5,Neq=1 mode=0
    (3, 0) = -1.00-0.00j : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=l1.p1.i,f=0,
↪fidx=0,Neq=1 mode=0
    (3, 3) = 1.00+0.00j  : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=m1.p1.o,f=0,
↪fidx=3,Neq=1 mode=0
    (4, 3) = 0.00-0.32j  : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.p1.o,f=0,
↪fidx=3,Neq=1 mode=0
    (4, 4) = 1.00+0.00j  : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.p2.i,f=0,
↪fidx=4,Neq=1 mode=0
    (4, 5) = -0.95+0.00j : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.p2.o,f=0,
↪fidx=5,Neq=1 mode=0
    (5, 5) = 1.00+0.00j  : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0 -> I,node=m1.p2.o,f=0,
↪fidx=5,Neq=1 mode=0


Vector carrier: neqs=6
    (row) = value
    (0) = 1.34+0.00j : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0
    (1) = 1.41+0.00j : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0
    (2) = 1.41+0.00j : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0
    (3) = 1.34+0.00j : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0
    (4) = 0.00+0.00j : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0
    (5) = 0.00+0.45j : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0
```

The couplings between different nodes are listed in the interferometer matrix. We only list the non-zero values of this sparse matrix. The steady-state solution is listed in the right-hand side vector. Here we see a single value for every node in the model graph. The values represent the complex field amplitudes of the electrical field.

Note

This page only demonstrates a single-frequency plane-wave simulation. Normally, Finesse stores an array of complex amplitudes for every higher-order mode for every frequency that is being modelled. In the list above, the `f=0` and `mode=0` refer to indices in this array.

Output & Detectors

To get any useful data out of your simulation, you usually use Actions and Detectors. Actions are (very roughly said) ways to run your model (solving the matrix) one or multiple times, usually changing some component parameters in between solves.

You will notice that detectors are not a part of the model graph. Detectors only work on the output of simulation,

i.e. on the complex light amplitudes at every node. Adding detectors does not change the result of your simulation, it only changes which outputs are easily available. To create detectors which have an impact on the simulation, like a sensor in a control loop, see the readout component. (page 183)

That is why you have to specify a full node for a Detector (`m1.p2.o`). The detector simply takes the complex amplitude and either returns it directly (`AmplitudeDetector` (page 172)) or calculates a quantity based on the complex amplitude (`PowerDetector` (page 180)).

4.3.3 The simulation matrix

FINESSE uses sparse matrices to perform the main simulation task.

4.4 Interactive help

The `finesse.help()` function accepts either a string or object as input and will show corresponding help. If a string is provided, it is assumed to be a KatScript path (page 128) like `m`, `beamsplitter`, or `bs.T`. Command names such as `xaxis` are also supported.

This operates similarly to the Python built-in function `help()`: either a pager is opened (if using a console; use `PgUp` and `PgDn` to navigate, and `q` to escape) or the help text is printed (if using a notebook).

Chapter 5

Examples

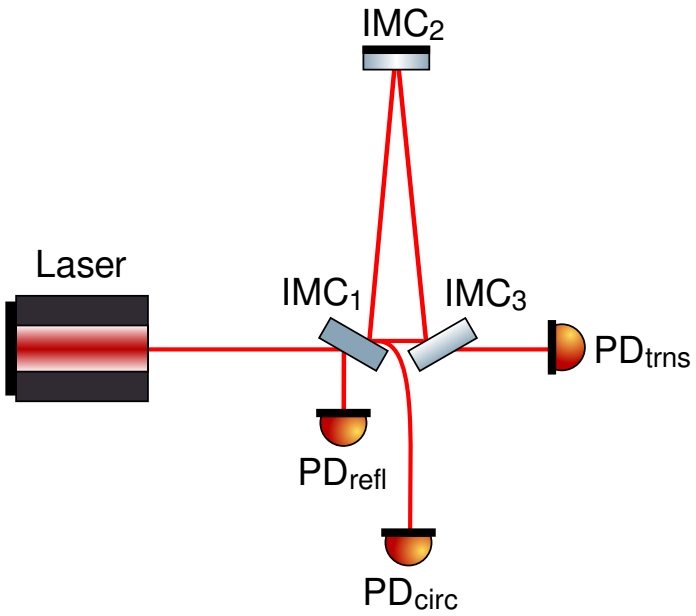
Here you will find a selection of examples for performing various simulation tasks using FINESSE.

5.1 Plane-wave optics examples

Here you will find a selection of examples for performing various simulation tasks about plane wave optics using FINESSE.

5.1.1 Triangular cavity

Below, we construct a triangular cavity formed by three mirrors (denoted IMC_1, IMC_2, and IMC_3) and three optical paths between them.



In LIGO, a triangular cavity of this type is placed immediately after the laser, and is known as the input mode cleaner (IMC). Its purpose is to clean and stabilize the laser beam before it enters the main interferometer. For example, the laser's optical frequency is not perfectly constant in time, but oscillates slightly around its nominal, central value. The IMC acts as a frequency filter reducing the laser's noise before it enters the main interferometer.

In this example we vary both the microscopic tuning of mirror IMC_3, and the laser frequency. At resonance all power is transmitted and the reflected power drops to zero. When the tuning of IMC_3 is varied, the cavity is at resonance when the extra path length is a whole number of wavelengths, that is when the phase picked up is a multiple of $360°$.

Since light reflected from a tuned mirror or beamsplitter picks up a phase shift of twice the tuning, the cavity is at resonance when the tuning of IMC_3 is a multiple of $180°$. When instead the laser frequency is varied, the cavity is at resonance when a whole numer of extra wavelengths fits inside the cavity. This occurs when the frequency offset is a multiple of $\Delta\omega = c/L$, where c is the speed of light and L is the total length of the cavity (for the LIGO parameters used below $\Delta\omega = 9.1$MHz). This example demonstrates the use of the triangular cavity as a frequency filter: specific resonance frequencies are transmitted, while other frequencies are heavily suppressed.

Note

Even though we say 'mirrors' in the text, we have to use the `beamsplitter` (page 149) component in Finesse to create any cavity with more than two mirrors. This is because the Finesse `mirror` (page 142) component assumes normal incidence.

```
import finesse
from finesse.analysis.actions import Xaxis
import matplotlib.pyplot as plt
finesse.init_plotting()

# parameters from https://finesse.docs.ligo.org/finesse-ligo/parameter_files/llo.html

IMC_model = finesse.Model()
IMC_model.parse(
    """
    l l1 P=1                    # laser with P=1W
    s s1 l1.p1 IMC_1.p1 L=0     # space of 1m length connecting laser to IMC

    # Input Mode Cleaner (IMC) mirrors
    # Light transmitted through IMC_1 (from p1 to p3) enters IMC_3 (in p1 and is
    # reflected in p2). Then it continues into IMC_2 (enters in p1, reflected in p2).
    # Finally it returns from IMC_2.p2 into IMC_1.p4, which reflects into MC1.p3,
    # closing the triangular cavity.
    bs IMC_1    T=6150u     L=20u   Rc=101609
    bs IMC_2    T=3.5u      L=20u   Rc=27.178
    bs IMC_3    T=6130u     L=20u   Rc=171747

    # Connect spaces to form a triangle
    s L13   IMC_1.p3    IMC_3.p1    L=0.465
    s L23   IMC_3.p2    IMC_2.p1    L=16.24057
    s L12   IMC_2.p2    IMC_1.p4    L=16.24057

    # Photodiodes measuring the reflected, circulating, and transmitted light
    pd power_reflected      IMC_1.p2.o
    pd power_circulating    IMC_1.p3.o
    pd power_transmitted    IMC_3.p3.o
    """
)

# Change the tuning of mirror IMC_3 up to second resonance
out_tuning = IMC_model.run(Xaxis('IMC_3.phi', 'lin', 0, 360, 1000))

# Change the frequency of the laser up to second resonance
cavity_length = IMC_model.L13.L + IMC_model.L23.L + IMC_model.L12.L
resonance_freq = 2.99792e8 / cavity_length
out_laser = IMC_model.run(Xaxis('l1.f', 'lin', 0, 2*resonance_freq, 1000))

fig, ax = plt.subplots(1, 2, figsize=(10, 4))
ax[0].semilogy(out_tuning.x1, out_tuning['power_reflected'], label='reflected')
ax[0].semilogy(out_tuning.x1, out_tuning['power_circulating'], label='circulating')
ax[0].semilogy(out_tuning.x1, out_tuning['power_transmitted'], label='transmitted')
ax[0].set_xlabel(r'IMC3 tuning [$\degree$]')
```

(continues on next page)

(continued from previous page)

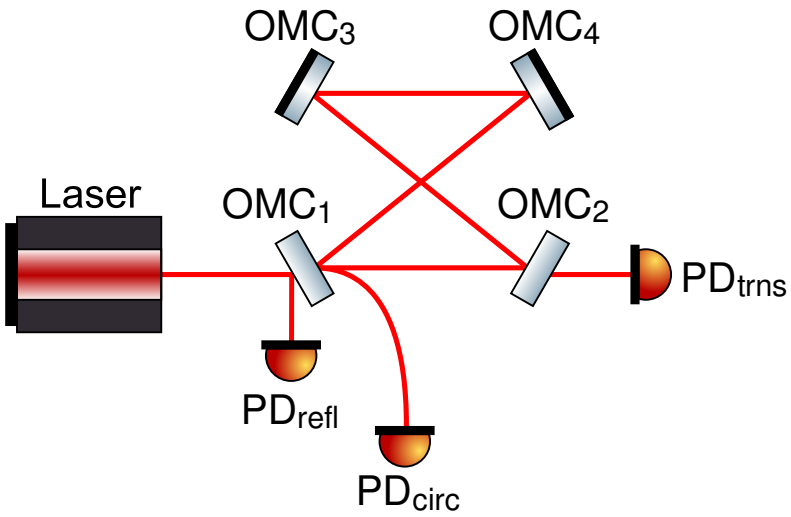
```
ax[0].set_ylabel('Power [W]')
ax[0].legend()

ax[1].semilogy(out_laser.x1, out_laser['power_reflected'], label='reflected')
ax[1].semilogy(out_laser.x1, out_laser['power_circulating'], label='circulating')
ax[1].semilogy(out_laser.x1, out_laser['power_transmitted'], label='transmitted')
ax[1].set_xlabel('frequency offset [Hz]')
ax[1].set_ylabel('Power [W]')
ax[1].legend()
plt.show()
```

```
<Figure size 960x384 with 2 Axes>
```

5.1.2 Bow tie cavity

Below, we model the output mode cleaner (OMC) of LIGO, which is a type of ring optical cavity made of four mirrors (denoted OMC_1 to OMC_4) arranged so that the optical path crosses itself, forming a shape similar to a bow tie.



In optics, any laser beam can be described as a sum of transverse Gaussian modes (see Higher-order Hermite-Gauss modes (page 298)), which form a complete basis of functions for the transverse field profile. The ideal beam in an interferometer is the fundamental TEM00 mode, and imperfections in the interferometer optics, as mirror surface defects or small mirror tilts, couple part of the light into higher-order modes of this basis.

The input mode cleaner (see Triangular cavity (page 40)) is placed before the main interferometer to ensure that a clean TEM00 beam enters the instrument. But even if the input beam is perfectly clean, additional higher-order modes can be generated inside the interferometer by optical imperfections and misalignments. That's why an output mode cleaner is placed at the detection port. The OMC transmits the fundamental mode that carries the gravitational-wave signal and suppresses higher-order modes produced by the interferometer itself.

```
import finesse
from finesse.analysis.actions import Xaxis
finesse.init_plotting()

# parameters from https://finesse.docs.ligo.org/finesse-ligo/parameter_files/llo.html

OMC_model = finesse.Model()
OMC_model.parse(
    """
    l l1 P=125                  # laser with P=125W
    s s1 l1.p1 OMC_1.p1 L=1     # space of 1m length connectint laser to OMC
```

(continues on next page)

(continued from previous page)

```
    # Output Mode Cleaner (OMC) mirrors
    # Light transmitted through OMC_1 (from p1 to p3) enters OMC_2 (in p1 and is
    # reflected in p2). It continues into OMC_3 (enters in p1, reflected in p2).
    # Then it continues into OMC_4 (enters in p1, reflected in p2).
    # Finally it returns from OMC_4.p2 into OMC_1.p4, which reflects back into OMC_1.p3,
    # closing the bow tie cavity.
    bs OMC_1    T=0.0076    L=0     alpha=2.7609
    bs OMC_2    T=0.0075    L=0     alpha=2.7609
    bs OMC_3    T=36.0u     L=0     alpha=4.004     Rc=2.57321
    bs OMC_4    T=35.9u     L=0     alpha=4.004     Rc=2.57369

    # Connect spaces to form a bow tie
    s sOMC_1    OMC_1.p3    OMC_2.p1    L=0.2815
    s sOMC_2    OMC_2.p2    OMC_3.p1    L=0.2842
    s sOMC_3    OMC_3.p2    OMC_4.p1    L=0.2815
    s sOMC_4    OMC_4.p2    OMC_1.p4    L=0.2842

    # Photodiodes measuring the reflected, circulating, and transmitted light
    pd power_reflected      OMC_1.p2.o
    pd power_circulating    OMC_1.p3.o
    pd power_transmitted    OMC_2.p3.o
    """
)

# Change the tuning of mirror OMC_2 from 0 to 360 degrees
out = OMC_model.run(Xaxis('OMC_2.phi', 'lin', 0, 360, 1000))
out.plot(logy=True);
```

```
<Figure size 576x355.968 with 1 Axes>
```

5.2 Higher-order spatial modes

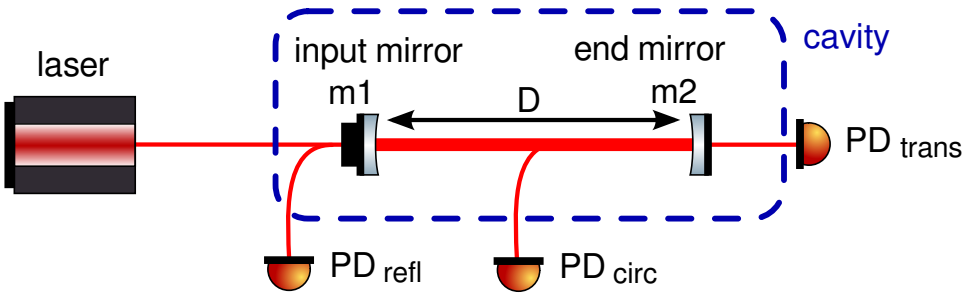
Here you will find a selection of examples for performing various simulation tasks using FINESSE.

5.2.1 An optical cavity

The best way to familiarise yourself with Finesse is with example models. In this section we will cover the simple case of modelling resonances in a Fabry-Perot cavity as a starting point.

The optical configuration

In the figure below we show the layout that we want to model. This consists of an input laser incident upon a two-mirror optical cavity. Measuring the reflected, transmitted and circulating power requires the use of three photodiodes - $\mathrm{PD}_{\mathrm{refl}}$, $\mathrm{PD}_{\mathrm{trans}}$ and $\mathrm{PD}_{\mathrm{circ}}$, respectively.



To find the resonances of this cavity we can either change the tuning(s) (see Tuning (page 114) for explanation on tuning) or scan over the frequency of the input laser. In this case we will alter the tuning of the input mirror `m1` in the range $\phi_{\mathrm{m1}} \in [-180°, 180°]$ to show multiple (repeated) resonances.

The Finesse model

```
import finesse

finesse.configure(plotting=True)

kat = finesse.Model()
kat.parse(
    """
    # Add a Laser named L0 with a power of 1 W.
    l L0 P=1

    # Space attaching L0 <-> m1 with length of 0 m (default).
    s s0 L0.p1 m1.p1

    # Input mirror of cavity.
    m m1 R=0.99 T=0.01

    # Intra-cavity space with length of 1 m.
    s CAV m1.p2 m2.p1 L=1

    # End mirror of cavity.
    m m2 R=0.991 T=0.009

    # Power detectors on reflection, circulation and transmission.
    pd refl m1.p1.o
    pd circ m2.p1.i
    pd trns m2.p2.o
    """
);
```

Firstly, we call `configure()` with `plotting=True`. This sets up some defaults for plotting styles, such as line colours and font size. Next, we create a new `Model`, and then parse our KatScript (see KatScript (page 125) for more details on the syntax used). This creates a model of a Fabry-Perot cavity with a length of 1 metre and nearly equal reflectivities on both the input mirror and end mirror. The input laser power is set to 1 W.

Plotting the outputs

To run this model and plot the outputs from the power-detectors we can simply write:

```
# Scan over the detuning DOF of m1 from -180 deg to +180 deg with 400 points.
out = kat.run("xaxis(m1.phi, lin, -180, 180, 400)")
out.plot(logy=True);
```

```
<Figure size 576x355.968 with 1 Axes>
```

The call to `run()` is passed an `Action` that will operate on the model. In this case it is an `Xaxis`, which sweeps some parameter of the model between two limits. This returns an `ArraySolution`, containing the output of each detector at each point on the axis. We then plot the detector outputs, via `plot()`. The `logy` flag tells the plotting routines to use a log-scale on the y-axis of any produced figures.

For more details on actions and analyses, see the analyses introduction (page 198).

Plotting the beam profile

To plot the beam profile, we need to add some curvatures to our mirrors and define the cavity. This will implicitly switch the simulation performed by Finesse from plane wave (see Plane-wave optics (page 112)) to Hermite-Gaussian (see Hermite-Gauss modes and coupling (page 306)).

```
# Add some curvatures to our mirrors.
kat.m1.Rc = -0.7
```

(continues on next page)

(continued from previous page)

```
kat.m2.Rc = 0.7

# Define the cavity.
kat.parse("cavity cavity1 source=m1.p2.o via=m2.p1.i priority=1")

# Plot the beam trace, starting from cavity the eigenmode.
tsy = finesse.tracing.tools.propagate_beam(
    to_node=kat.L0.p1.i, from_node=kat.m2.p1.o, direction="y"
)
tsy.plot();
```

```
<Figure size 576x355.968 with 2 Axes>
```

The resulting plot shows the cavity eigenmode (see Cavity eigenmodes (page 304) section)

To make the plot more interesting, we could add a lens to the input coupler. To do this, we build a compound mirror:

```
# Remove s0 so we can add in a lens.
# FIXME kat.remove is not yet working, so we use a list
# comprehension to remove s0 from the model and
# reparse it into a new model. (See unparse below)
kat2 = finesse.Model()
kat2.parse("\n".join([l for l in kat.unparse().splitlines() if "s0" not in l]))

# Add a silicon lens with an AR surface to the input coupler.
kat2.parse(
    """
    space s0 L0.p1 m1_AR.p1 L=0.5
    mirror m1_AR R=100e-6 L=10e-6 Rc=-0.3
    space s_M1_sub portA=m1_AR.p2 portB=m1.p1 L=10e-3 nr=3.5
    """
)

# Plot beam trace, starting from the cavity eigenmode.
tsy = finesse.tracing.tools.propagate_beam(
    to_node=kat2.L0.p1.i, from_node=kat2.m2.p1.o, direction="y"
)
tsy.plot();
```

```
<Figure size 576x355.968 with 2 Axes>
```

Printing model information

There are three options for outputting a textual summary of the model. Firstly, printing the model directly gives the usual Python object information:

```
print(kat)
```

```
<finesse.model.Model object at 0x7a4e2ff06900>
```

Secondly, you can output a tree view of the nodal network with components as nodes and connected components as branches. Such a tree can be printed in the following way:

```
print(kat.component_tree(kat.L0))
```

```
○ L0
└──○ m1
   └──○ m2
```

The argument to `component_tree()` is the component to start from (the tree's `root`). Any component in the model can be the tree root.

> **Note**
>
> Models that contain cyclic connections (such as Sagnac interferometers) will not show one of the connections that would create the cycle, to prevent infinite recursion.

Lastly, the most verbose option is to `unparse()` the model. This returns KatScript which can be used to recreate the model.

```
print(kat.unparse())
```

```
    # Add a Laser named L0 with a power of 1 W.
    l L0 P=1

    # Space attaching L0 <-> m1 with length of 0 m (default).
    s s0 L0.p1 m1.p1

    # Input mirror of cavity.
    m m1 R=0.99 T=0.01 L=(1-(m1.R+m1.T)) Rc=[-0.7, -0.7]

    # Intra-cavity space with length of 1 m.
    s CAV m1.p2 m2.p1 L=1

    # End mirror of cavity.
    m m2 R=0.991 T=0.009 L=(1-(m2.R+m2.T)) Rc=[0.7, 0.7]

    # Power detectors on reflection, circulation and transmission.
    pd refl m1.p1.o
    pd circ m2.p1.i
    pd trns m2.p2.o
cavity cavity1 source=m1.p2.o via=m2.p1.i priority=1

# Items below could not be matched to original script, or were not present when the model␣
↪was originally parsed.


modes(maxtem=0)
```

5.2.2 Cavity eigenmodes

Optical cavities, such as Fabry-Perot cavities, typically have a complex mix of both longitudinal and transverse resonant modes (page 304). The properties of such modes is usually characterised by the free spectral range (page 124) and the accumulated roundtrip gouy phase (page 308) cavity parameters. Using these parameters you can determine how each transverse mode will resonate relative to other optical modes and frequencies.

This simple analysis isn't however easily applicable to more complex scenarios where optical defects are present in the cavity. In such cases the usual Hermite-Gaussian modal basis no longer describes the system and each higher order mode can experience different losses.

FINESSE contains tools to analyse such effects. One of these is the `eigenmodes` (page 233) analysis. This analysis calculates the round trip matrix operator in the Hermite-Gaussian (page 306) basis and performs an eigen-decomposition on it to determine the eigenvalues and eigenvectors. The eigenvalues provide information about the relative phases and amplitudes of each resonant mode of the cavity. The eigenvectors tell us what mix of HG modes make up a particular resonance. This type of analysis is a much faster way to study the resonances of a cavity compared to scanning the cavity length or laser frequency whilst injecting higher order modes, as is typically done with other tools and with older versions of FINESSE.

We start with a simple optical cavity based on an Advanced LIGO arm cavity:

```
import numpy as np
import matplotlib.pyplot as plt
import finesse
from finesse.knm import Map
from finesse.utilities.maps import circular_aperture

finesse.init_plotting()

model = finesse.Model()
model.parse(
    """
    l l1
    mod mod1 f=9.1M midx=0.1
    m m1 R=0.984 T=0.014 Rc=-1940
    m m2 R=1 T=0 Rc=2245
    link(l1, mod1, m1, 3994, m2)
    cav cavity m2.p1.o
    modes(maxtem=2)

    fd E_arm m2.p1.i l1.f  # Field detector
    """
);
```

We have included HG modes up to order 2 and also a field detector to output the HG mode content within the arm.

The `eigenmodes` (page 233) analysis is simple to call as shown below. For this simulation we are running multiple eigenmode analyses at the frequency of the carrier (page 120) and those of two radio-frequency sidebands. Lastly we perform a typical cavity scan using `xaxis` (page 209).

```
sol = model.run(
    """
    series(
        eigenmodes(cavity, -mod1.f, name="l9"),
        eigenmodes(cavity, 0,       name="c0"),
        eigenmodes(cavity, mod1.f,  name="u9"),
        xaxis(m2.phi, lin, -10, 190, 500, name="scan")
    )
    """
)

print(sol)
```

```
- Solution Tree
o series - SeriesSolution
├──o l9 - EigenmodesSolution
├──o c0 - EigenmodesSolution
├──o u9 - EigenmodesSolution
└──● scan - ArraySolution
```

The overall `series` solution then contains the solutions for each of the analyses it contains. As we have given names to the analyses we can access each one by name using e.g. `sol["scan"]`, `sol["c0"]`, etc.

```
sol_c0 = sol["c0"]
```

Note

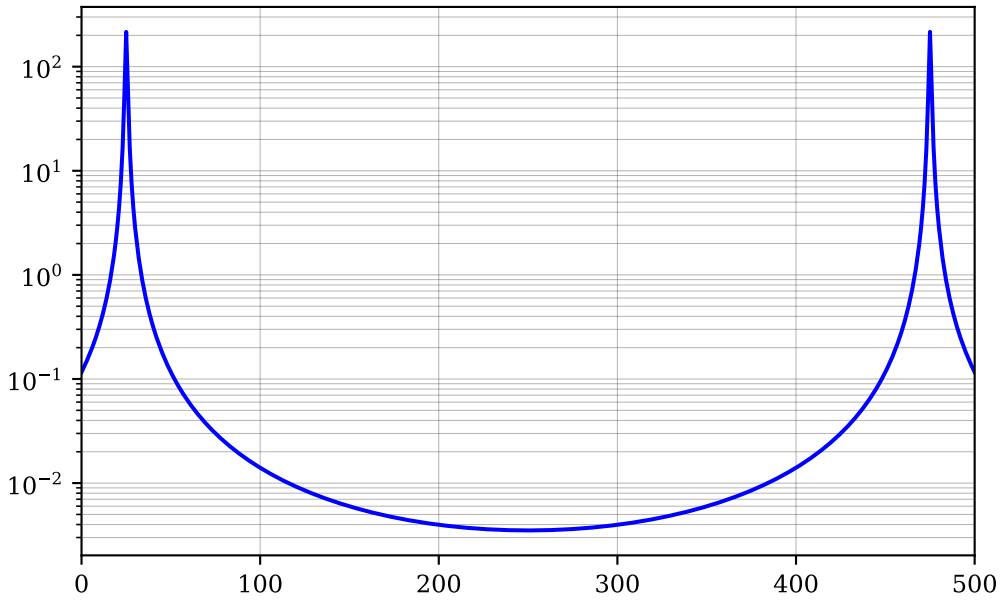
We can access documentation for the `EigenmodesSolution` using `help(sol_c0)` or, if in an interactive terminal such as IPython, `sol_c0??`.

We can see this has returned multiple outputs, such as the round-trip matrix operator which we perform the eigen-

decomposition on along with various other outputs.

The `xaxis` cavity scan can be plotted using:

```
plt.semilogy(abs(sol["scan"]["E_arm"])**2);
```



where we can see that only a single mode is present. This is to be expected as we have a perfect optical system; likewise, if we print the eigenvalues and eigenvectors, we see we have an identity matrix of vectors,

```
sol["c0"].eigvectors
```

```
array([[ 0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ],
       [ 0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ],
       [ 0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ],
       [ 0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ],
       [ 0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ,
         0.        +0.j        ,  0.        +0.j        ],
       [-0.76019981-0.64968934j, -0.96232591-0.2718986j ,
        -0.96232591-0.2718986j ,  1.        +0.j        ,
         1.        +0.j        ,  1.        +0.j        ]])
```

and each eigenvalue is identical in magnitude with a differing phase.

```
print(abs(sol["c0"].eigvalues))
print(np.angle(sol["c0"].eigvalues))
```

```
[0.99196774 0.99196774 0.99196774 0.99196774 0.99196774 0.99196774]
[ 2.22044605e-16 -8.63620612e-01 -8.63620612e-01 -1.72724122e+00
 -1.72724122e+00 -1.72724122e+00]
```

The phase difference between each mode is just the round-trip gouy phase difference between each mode order.

The eigenvalue amplitude tells us about the roundtrip loss of the eigenmode. The power loss in the i-th eigenmode is simply:

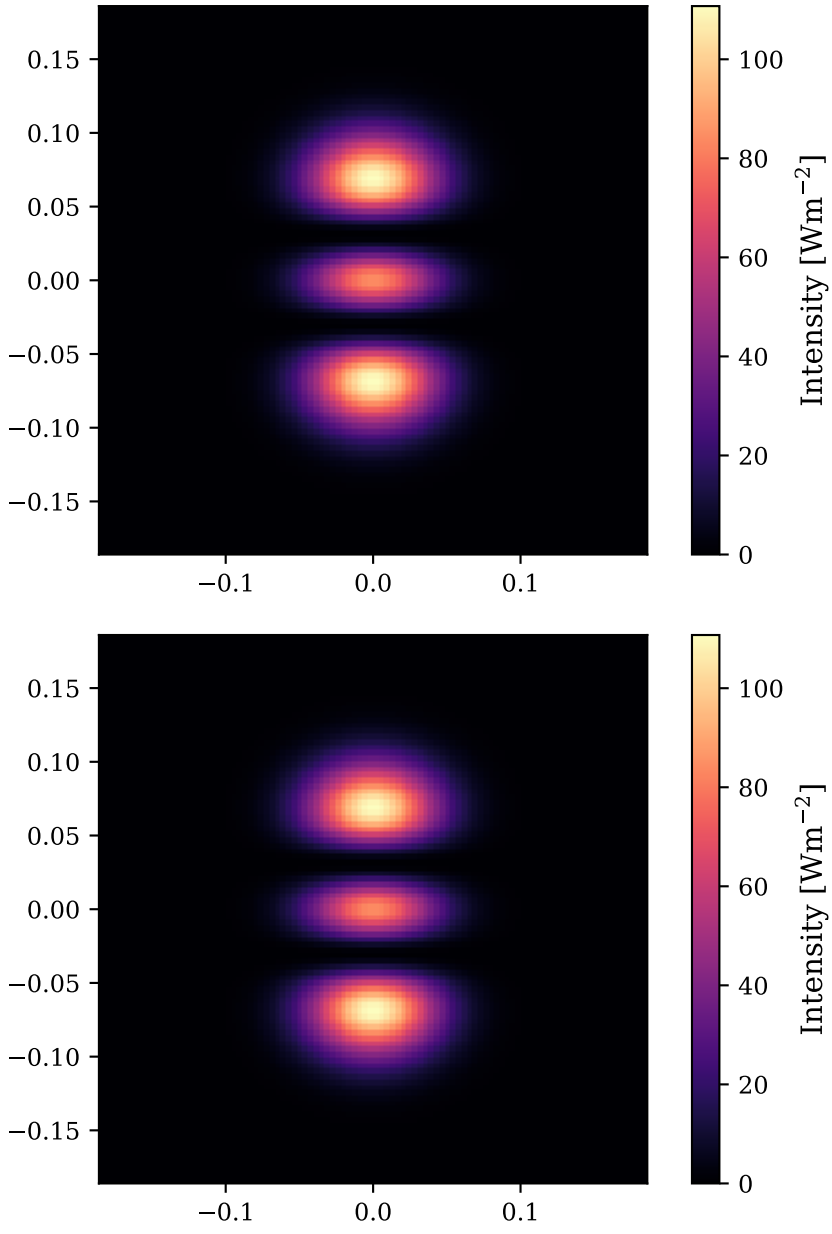
$$L_{i,rt} = 1 - |\gamma_i|^2$$

which in this case is just equal to the transmission of our input mirror as that is the only source of loss.

When you have an array of HG mode amplitudes the optical field can be plotted easily using the utility function `plot_field()`. This can be used to visually inspect each mode. The `EigenmodesSolution` itself also has a `plot_field()` function which plots a given eigenmode using the beam parameter used in the calculation. We should see the usual HG eigenmodes.

```
from finesse.plotting import plot_field

plot_field(
    model.homs,
    sol["c0"].eigvectors[:, 0],
    model.cavity.source.q
)
sol["c0"].plot_field(mode_idx=1)
```



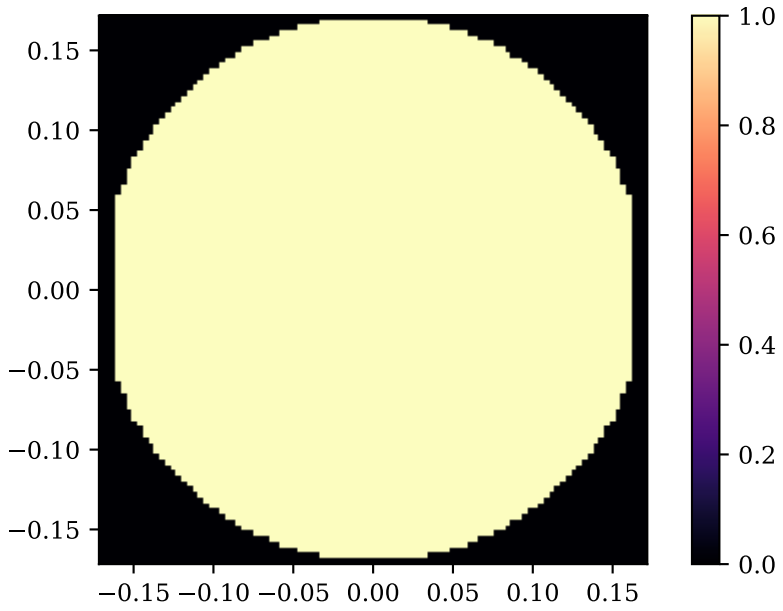
Adding defects

The most dominant effect that determines the eigenmodes of an optical cavity is the shape of mirror apertures.

This type of complex features is possible to be simulated with FINESSE using *maps*. These describe the amplitude and phase offsets that an optical field experiences when interacting with an element. We can create an amplitude aperture that is similar to the Advanced LIGO mirror shape. A value of `1` means the beam is left unchanged and `0` attenuates it.

```
x = y = np.linspace(-0.17, 0.17, 100)
ligo_ap = circular_aperture(x, y, 0.17)
ligo_ap[:, abs(y) >= 0.163] = 0
model.m1.surface_map = Map(x, y, amplitude=ligo_ap)
model.m2.surface_map = Map(x, y, amplitude=ligo_ap)

# Plot the aperture we are applying.
plt.pcolormesh(x, y, ligo_ap, rasterized=True, shading="auto")
plt.gca().set_aspect("equal")
plt.colorbar();
```

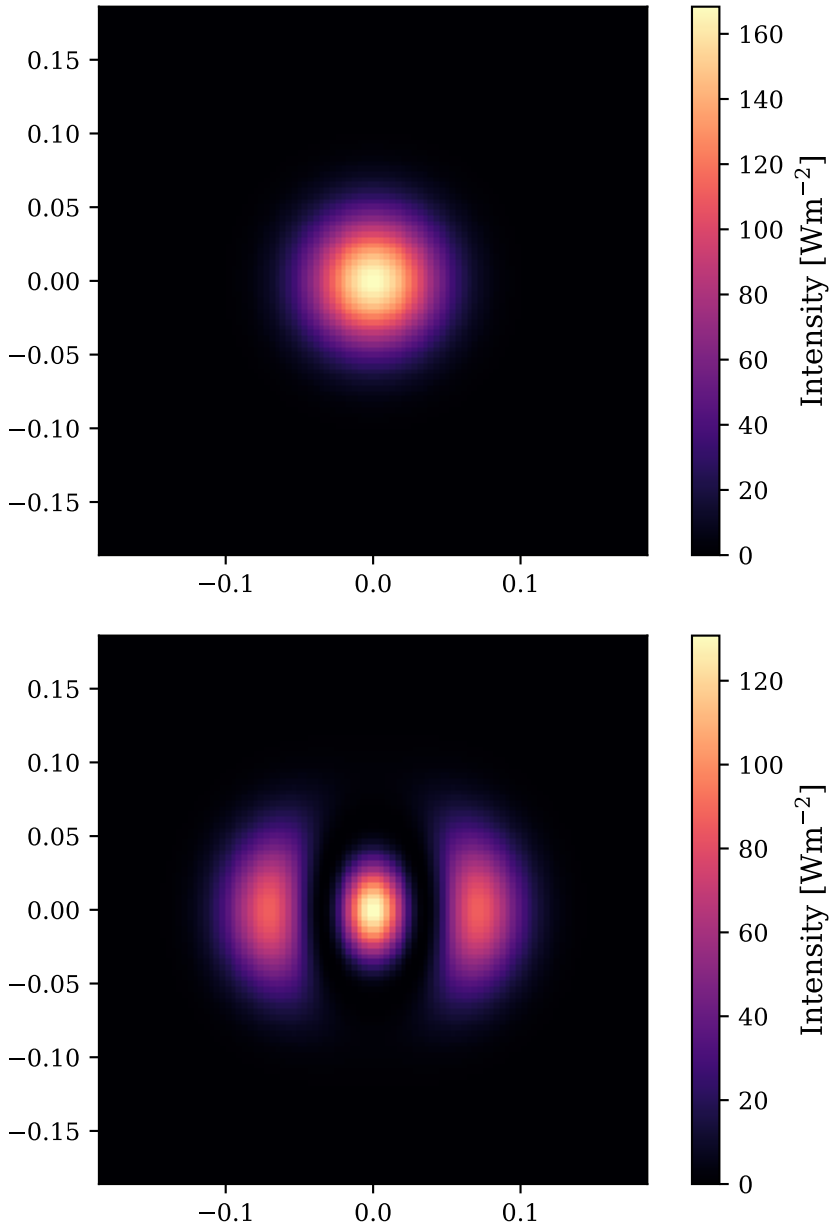


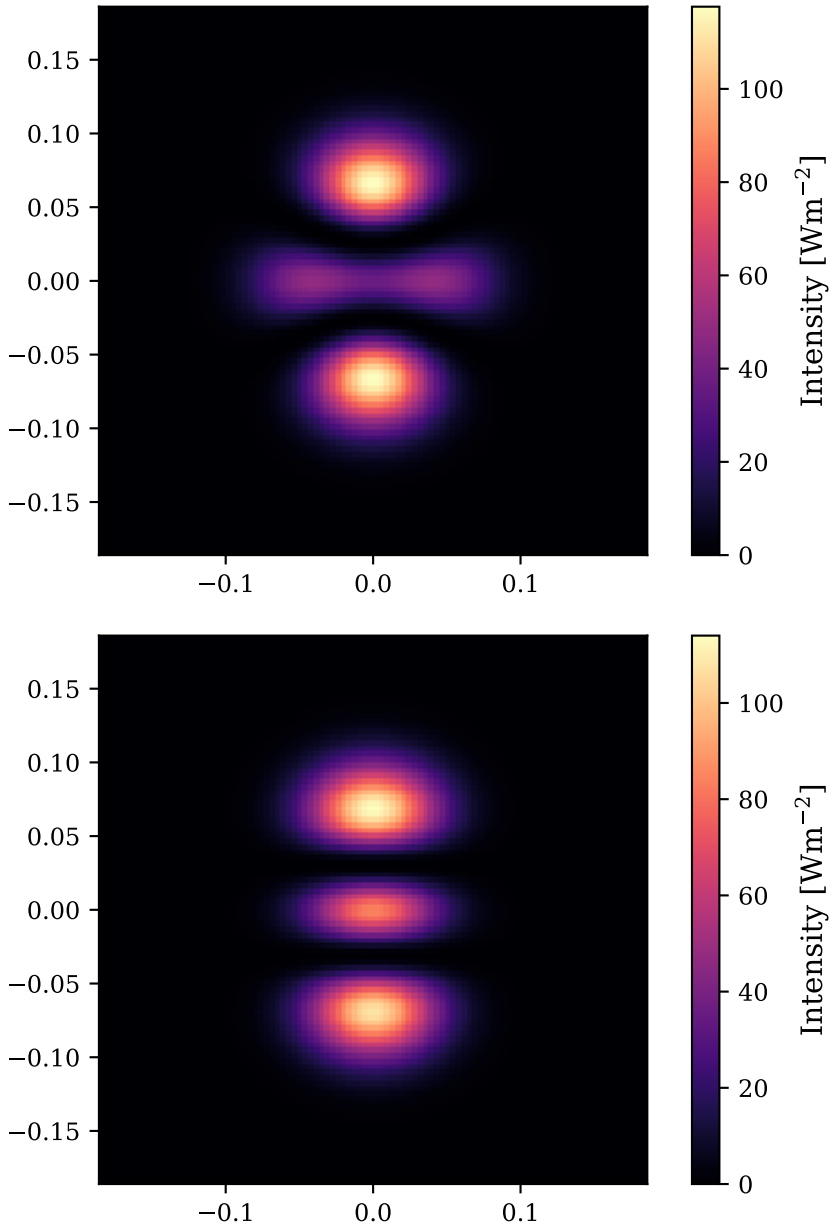
Note that the Advanced LIGO mirrors are not perfectly round as they have flat edges on which the suspension wires are attached.

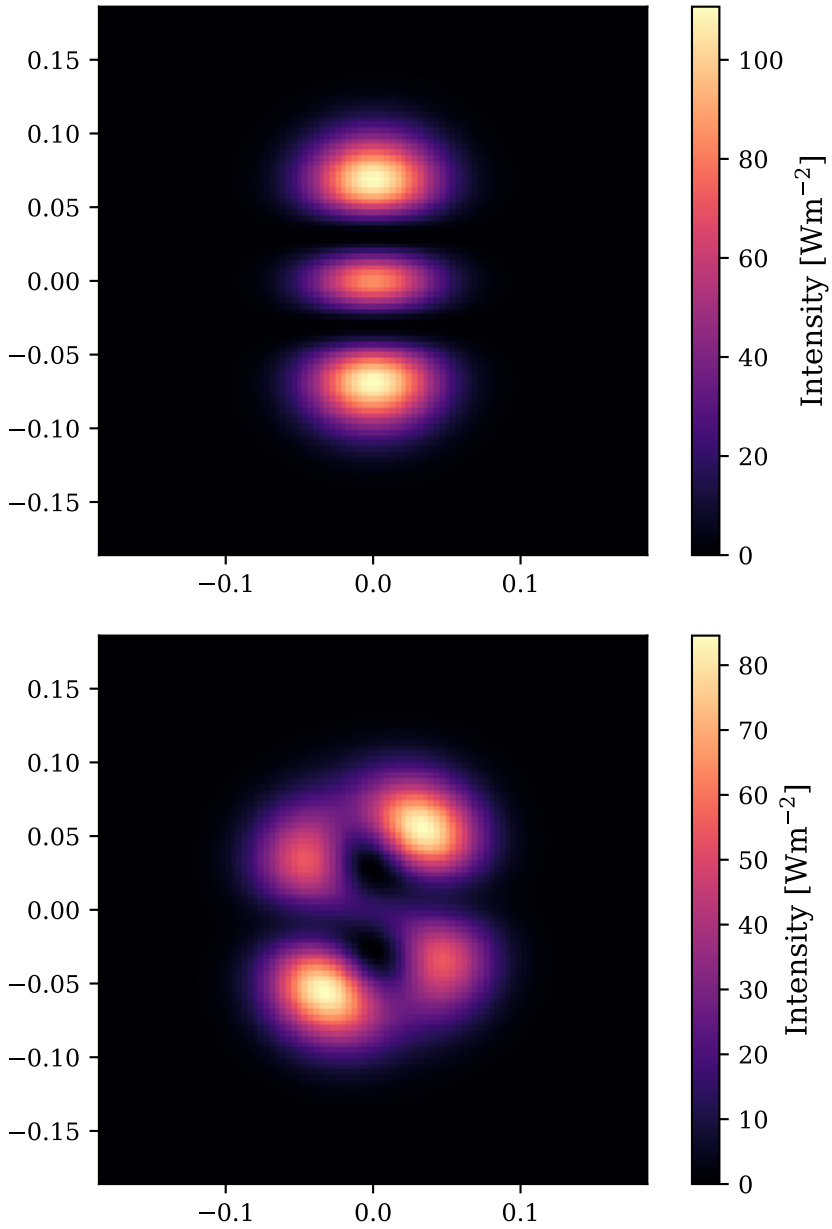
With the finite apertures added we can then re-run the eigenmode analysis and plot each eigenmode. We should see that we do not have perfect HG modes anymore. They are actually a more complex mix of Laguerre (page 324) and Hermite Gaussian modes, due to the combination of circular aperture with the flat edges.

```
sol_aperture = model.run("eigenmodes(cavity, 0, name='c0')")

for i in range(len(model.homs)):
    plot_field(
        model.homs,
        sol_aperture.eigvectors[:, i],
        model.cavity.source.q
    )
```


0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.1
0.0
0.1
160
140
120
100
80
60
40
20
0
Intensity [Wm^{-2}]
0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.1
0.0
0.1
120
100
80
60
40
20
0
Intensity [Wm^{-2}]


0.15
0.10
0.05
0.00
−0.05
−0.10
−0.15
−0.1
0.0
0.1
100
80
60
40
20
0
Intensity [Wm^{-2}]



This type of analysis can obviously be expanded to much higher numbers of modes using `maxtem`. In such cases you will find there are essentially an infinite number of optical modes, but larger modes will quickly become clipped and have large roundtrip losses.

Note

It should be noted that the order of the eigenmodes is not guaranteed in any way. They may appear to be sensibly organised in simple cases, but when more complex systems are studied they will jump around. This makes analysing the eigenmode information somewhat challenging as you must sort them by some appropriate metric for analysing them.

5.2.3 Defining cavity eigenmodes with the cavity command

When modelling higher order mode effects in optical cavities you (the user) must decide which optical bases to use. This means you must choose which gaussian beam parameters (page 303) ($q = w_0 + iz_r$) are used at each node in the model. Finesse will then model the transverse shape of the optical modes using the Hermite-Gaussian decomposition, with the q value defining the shape of the beam at the node. Technically, any choice of q should work, assuming you

can use infinite HG modes. However, computationally this is not ideal, nor possible. What you must do is try to optimally choose which q to use.

For a single cavity the choice is clear, we should use the standard eigenmode calculation that is retrieved from the roundtrip ABCD matrix calculations (page 310). This defines the q values used at all the intra-cavity nodes. Next, the beam tracing (page 306) algorithm will take this cavity eigenmode and trace it to all other nodes in the model that have not be defined, ensuring every node has a q. If you change the cavity geometry, therefore the eigenmode, the beam trace will be repeated and all other nodes will follow.

Coupled cavities

Models become more complex with coupled cavities as we no longer have a well defined theoretical description of what the correct eigenmode should be. This is because there is no single roundtrip ABCD matrix that can be written and then solved.

In such cases we can only aim to include enough higher order modes such that we converge to a solution. The actual cavity eigenmode is then no longer purely described by the q values in the coupled cavity, but a combination of q and a set of complex-valued HG mode amplitudes in the q basis.

In FINESSE we have the `cavity` (page 195) command which allows you to quickly compute the simple cavity roundtrip eigenmodes. This command takes a start node and tries to find the simplest circular path in the model back to itself. You can also define a `via` node to guide it in more complicated models. For coupled cavities we have multiple ways in which we can use this command to help build our models. However, we must be careful on how we use it, as demonstrated below.

Let us study a power-recycled Fabry-Perot. In particular this is an aLIGO arm cavity and the power recyling cavity. This forms a simplified model of the common arm (CARM). Near the ITM we model any substrate thermal changes as with a simple lens object, which has a focal length.

```
import finesse
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

model = finesse.script.parse("""
################################################################################
###   length definitions
################################################################################
variable Larm 3994.47
variable LPR23 16.164  # distance between PR2 and PR3
variable LPR3BS 19.538 # distance between PR3 and BS
variable lmich 5.342   # average length of MICH
variable lschnupp 0.08
variable lPRC (3+0.5)*c0/(2*f1) # T1000298 Eq2.1, N=3

variable f1 9099471
variable f2 5*f1
variable nsilica 1.45
variable Mloss 0

laser L0 P=1
mod mod1 f=f1 midx=0.2
link(L0, mod1, PRM)
m PRM T=0.03 L=0 Rc=11.009 phi=0
s lp1 PRM.p2 PR2.p1 L=lPRC-LPR3BS-LPR23-lmich
bs PR2 T=250u L=Mloss alpha=-0.79 Rc=-4.545
s lp2 PR2.p2 PR3.p1 L=LPR23
bs PR3 T=0 L=Mloss alpha=0.615 Rc=36.027
s lp3 PR3.p2 BS.p1 L=LPR3BS

################################################################################
###   BS
################################################################################
```

(continues on next page)

(continued from previous page)

```
bs BS R=1 L=0 alpha=45

################################################################################
###   Yarm
################################################################################
# Distance from beam splitter to Y arm input mirror
s ly1 BS.p2 ITMYlens.p1 L=lmich-lschnupp/2-ITMYsub.L*ITMYsub.nr
lens ITMYlens f=34000
s ly2 ITMYlens.p2 ITMYAR.p1
m ITMYAR R=0 L=20u
s ITMYsub ITMYAR.p2 ITMY.p1 L=0.2 nr=nsilica
m ITMY T=0.014 L=Mloss Rc=-1934
s LY ITMY.p2 ETMY.p1 L=Larm
m ETMY T=5u L=Mloss Rc=2245

pd Parm ETMY.p1.i
pd Prc PRM.p2.o
modes(even, maxtem=12)

ad pu9 PRM.p2.i f=+f1
ad pl9 PRM.p2.i f=-f1
mathd prc_cost log10(Parm/280)+log10(abs(pu9))+log10(abs(pl9))
""")
```

So far in this model we have defined curvatures and lengths but not what q values should be used. A common attempt now is to apply a cavity command to the arm mirrors. This can be done adding the command below:

```
# make a copy so that we can add different cav commands
_model = model.deepcopy()
_model.parse("cav cavYARM ETMY.p1.o")
# nodes that will be defined by the ITM and ETM geometry
print(_model.cavYARM.path.nodes)
```

```
[<OpticalNode ETMY.p1.o @ 0x7b923f3e5440>, <OpticalNode ITMY.p2.i @ 0x7b923f3e5230>,
↪<OpticalNode ITMY.p2.o @ 0x7b923f3e52e0>, <OpticalNode ETMY.p1.i @ 0x7b923f3e5390>]
```

All other nodes, not in the arm cavity, will be traced from the closest node listed above.

This some has important consequences:

- If the power-recycling and arm cavities are perfectly mode-matched, then the arm mode traced around the power-recycling will be the ideal mode to use for both cavities.
- The input laser q is not defined, it will in this case be dependent on the arm cavity mode traced back to the laser. If you change the arm geometry, the laser q will change accordingly, something that is not typically what you will want to simulate.

Mode-mismatched coupled cavities

Perfectly matched cavities is not what we have in reality and is a typical case users want to model in FINESSE. Here are some suggestions for making better models.

Ensure that the laser q is defined. In realistic models the laser mode is independant of the interferometer geometry. In gravitational wave detectors, the input mode is likely defined by the geometry of the input mode-cleaner cavity.

If you do not know the input mode, then another good option is to calculate the mode that perfectly matches some state you are interested in. In FINESSE 3.0 we can use some optimizers to maximize the arm powers

```
_model = model.deepcopy()
_model.parse("cav cavYARM ETMY.p1.o")
# Maximize the arm power with respect to the PRM curvature and ITMY lens
# There are multiple optimisations that could be done here, depends on what
```

(continues on next page)

(continued from previous page)

```
# parameters you want to vary. This model is naturally astigmatic though due
# to the off-axis telescope.
opt_sol = _model.run("maximize(Parm, [ITMYlens.f, PRM.Rcx, PRM.Rcy, PRM.phi, ETMY.phi])")
print("HOMs Parm =       ", _model.run("noxaxis()")['Parm'])
_model.modes("off")
print("Plane-wave Parm = ", _model.run("noxaxis()")['Parm'])

# the plane-wave and HOM powers should be nearly identical, a sign that there
# is no mode-mismatch. If so, we then update the original model to use these
# new values
model.ITMYlens.f = opt_sol.x[0]
model.PRM.Rcx = opt_sol.x[1]
model.PRM.Rcy = opt_sol.x[2]

# Given the model is now in a perfectly matched state we can also take the
# beam parameter at the input of the PRM and set the original model to use that
model.PRM.p1.i.q = _model.PRM.p1.i.q
```

```
HOMs Parm =        32239.430040392363
Plane-wave Parm =  32239.430039709478
```

Setting a node's q value in python like this will internally generate a *gauss* command. This command sets the gaussian beam parameter to be used at that node. You can see what gauss commands are being currently used with:

```
model.gausses
```

```
{<OpticalNode PRM.p1.i @ 0x7b923f7557b0>: <'gPRM_p1_i' @ 0x7b923f6417f0 (Gauss)>}
```

It is also important to note that the optimizer used above is also varying the tuning of the cavities. This is of particular importance in doing these studies as we want to optimise the geometry but also keeping the coupled cavity on a sensible operating point, i.e. on resonance. A simple metric to use is maximizing the arm power. Note, this may not find a true operating point that would be found using radio-frequency readout schemes such as Pound-Drever-Hall (page 78).

Let us introduce a mismatch using the lens:

```
model.ITMYlens.f = np.inf # opt_sol.x[0] # use opt_sol for perfect matching
model.mismatches_table()
```

```
<finesse.utilities.tables.NumberTable at 0x7b923f6423c0>
```

We have roughly 5% mismatch at the ITM and 2% at the ETM. This is how the beam is mismatched to the input mode we calculated previously.

Now we want to see how applying the cavity command in different ways affects our model. We have different options: we can just specify the arm cavity, or just specify the power-recycling cavity, or both. Another option is to specify a cavity between the PRM and the ETM.

Here we simulate these options, with increasing maximum mode order used in the simulation. You should note that we must keep the arm cavities on resonance again, which we do using a *maximize* action again.

```
def run(model, orders):
    model.modes("off")
    plane_wave = model.run()
    homs = []
    for n in orders:
        model.modes("even", maxtem=n)
        with model.temporary_parameters():
            homs.append(model.run("""
                series(
                    maximize(Parm, [PRM.phi, ETMY.phi]),
```

(continues on next page)

(continued from previous page)

```
                    noxaxis() # get detector outputs
                )
            """)['noxaxis'])
    return plane_wave, homs

orders = range(0, 10, 2)

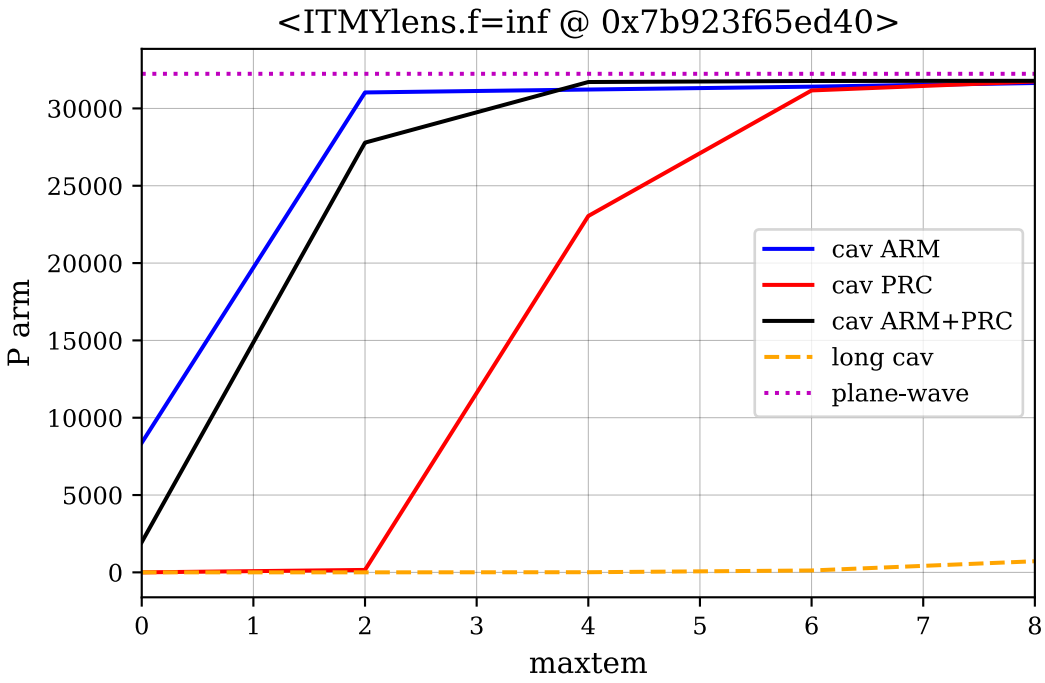
_model = model.deepcopy()
_model.parse("cav cavYARM ETMY.p1.o")
plane_wave, homs = run(_model, orders)

_model = model.deepcopy()
_model.parse("cav cavPRC PRM.p2.o via=ITMY.p1.i")
plane_wave2, homs2 = run(_model, orders)

_model = model.deepcopy()
_model.parse("cav cavYARM ETMY.p1.o")
_model.parse("cav cavPRC PRM.p2.o via=ITMY.p1.i")
plane_wave3, homs3 = run(_model, orders)

_model = model.deepcopy()
_model.parse("cav cavYARM ETMY.p1.o via=PRM.p2.i")
plane_wave4, homs4 = run(_model, orders)

plt.plot(orders, tuple(_['Parm'] for _ in homs), label='cav ARM')
plt.plot(orders, tuple(_['Parm'] for _ in homs2), label='cav PRC')
plt.plot(orders, tuple(_['Parm'] for _ in homs3), label='cav ARM+PRC')
plt.plot(orders, tuple(_['Parm'] for _ in homs4), label='long cav', ls='--')
plt.hlines(plane_wave['Parm'], 0, 8, ls=':', color='m', label='plane-wave')
plt.legend()
plt.xlabel("maxtem")
plt.ylabel("P arm")
plt.title(repr(model.ITMYlens.f));
```



This final plot shows us that, depending on how we set the interferometer cavity eigenmodes, we need different amounts of orders to describe it. Ideally we do not want to have large mismatches in singular places, as that will require many orders to converge. The best option in this case is defining both an arm and power-recycling cavity

command. This may not be the case for your model, so it is always worth checking, using a method like described above.

5.2.4 Extracting geometrical parameters

LIGO document DCC-T1900708 Cavity Tilts & Shifts [16] discusses the differences between the geometrical model and the modal model used by Finesse 2. This example replicates this work in Finesse 3 and discusses how to extract these geometrical parameters.

Geometrical parameters

Chapter 19 of [17] provides the following relations for the cavity axis tilt and shift as a function of the mirror alignments.

$$\Delta x_1 = \frac{g_2}{1 - g_1 \cdot g_2} L \cdot \theta_1 - \frac{1}{1 - g_1 \cdot g_2} L \cdot \theta_2$$

$$\Delta x_2 = \frac{1}{1 - g_1 \cdot g_2} L \cdot \theta_1 - \frac{g_1}{1 - g_1 \cdot g_2} L \cdot \theta_2$$

where the nomenclature follows [16] and [18]. As in [16], we consider the case of a hemispherical cavity, where the input coupler is the flat mirror and the output coupler is the curved mirror. A graphical representation of the possible combinations is shown in [16].

When the curved mirror rotates, the cavity axis shifts, but remains parallel to the input beam. When the flat mirror rotates, the cavity axis rotates.

The cavity axis rotation can be expressed in terms of the difference in spot positions,

$$\Delta\theta = \frac{\Delta x_1 - \Delta x_1}{L}$$

where L is the length of the cavity. These geometric relations can be expressed using the following python code,

```
import numpy as np

# Cavity parameters
L = 0.1                          # Cavity Length
RcM2 = 0.5                       # Radius of curvature of M2 mirror = 0.5 m
g1 = 1                           # g1 = 1 - L/RcM1 = 1 since RcM1 = infinity
g2 = 1 - L/RcM2                  # g2 for M2 mirror
print("g1g2 =", g1*g2)           # Stability check

# Mirror Tilt:
npoints = 200
theta_low = -1e-6
theta_high = 1e-6

def geometric(mirror, theta_low, theta_high, npoints):
    if mirror == 1:
        # M1 tilt
        theta_1 = np.linspace(theta_low, theta_high, npoints+1, endpoint=True)
        # M2 tilt
        theta_2 = 0
    elif mirror == 2:
        # M1 tilt
        theta_1 = 0
        # M2 tilt
        theta_2 = np.linspace(theta_low, theta_high, npoints+1, endpoint=True)
    else:
        raise NotImplementedError("Mirror should be 1 or 2 and an integer")

    # Resultant translation
    ## Displacement on M1
```

(continues on next page)

(continued from previous page)

```
    delta_x1 = (g2/(1-g1*g2))*L*theta_1 - (1/(1- g1*g2))*L*theta_2
    ## Displacement on M2
    delta_x2 = (1/(1-g1*g2))*L*theta_1 - (g1/(1-g1*g2))*L*theta_2
    ## Angular displacement of optical axis
    delta_theta = (delta_x2 - delta_x1)/L

    print("Results for analytical solutions:")
    print("Δx1: ", max(delta_x1), min(delta_x1), " m")
    print("Δx2: ", max(delta_x2), min(delta_x2), " m")
    print("Δθ: ", max(delta_theta), min(delta_theta), " m")

    return delta_x1, delta_x2, delta_theta
```

```
g1g2 = 0.8
```

where we have functionalised the geometric comparison for later analysis.

Finesse model

The system can be modelled in FINESSE using the following code,

```
import matplotlib.pyplot as plt
import finesse
from finesse import init_plotting
init_plotting()

T = 1e-6

# Build Finesse model
kat = finesse.Model()
kat.parse(
    f"""
    laser laser P=1.0 f=0 phase=0

    # Mirrors
    mirror M1 R=1-M1.T T={T} L=0.0 phi=0 Rc=[inf, inf] xbeta=0 ybeta=0
    mirror M2 R=1-M1.T T=M1.T L=0.0 phi=0 Rc=[{RcM2}, {RcM2}] xbeta=0 ybeta=0

    # Spaces
    space s0 portA=laser.p1 portB=M1.p1 L=1 nr=1
    space s_cav portA=M1.p2 portB=M2.p1 L=0.1 nr=1

    # Amplitude Detectors
    amplitude_detector det00_1 node=M1.p2.i f=0 n=0 m=0
    amplitude_detector det00_2 node=M2.p1.o f=0 n=0 m=0
    amplitude_detector det10_1 node=M1.p2.i f=0 n=1 m=0
    amplitude_detector det10_2 node=M2.p1.o f=0 n=1 m=0

    # Beam Property Detectors
    beam_property_detector bp_n2 prop=0 node=M1.p2.i direction=x q_as_bp=false
    beam_property_detector bp_n3 prop=0 node=M2.p1.o direction=x q_as_bp=false
    beam_property_detector cavwaist prop=1 node=M1.p2.i direction=x q_as_bp=false

    # Photodiodes
    power_detector_dc pcirc node=M1.p2.o pdtype=none

    # Config
    cavity cavity1 source=M1.p2.o via=M2.p1.i priority=1
    lambda(1.064e-06)
    modes(maxtem=3)
    """
```

(continues on next page)

(continued from previous page)

```
)

# Print beam tracing information
tsy = finesse.tracing.tools.propagate_beam(
    to_node=kat.M2.p1.o, from_node=kat.M1.p2.o, direction="y"
)
print(tsy.table())
```

```
┌─────────┬─────────┬──────────┬─────────┬──────────┬──────────┬─────────┬────────────┬──────────┐
│         ‖    z    │    w0    │   zr    │    w     │   RoC    │    S    │ Acc. Gouy │        q ␣
↪       │
╞═════════╪═════════╪══════════╪═════════╪══════════╪══════════╪═════════╪════════════╪══════════╡
│ M1.p2.o ‖    0 m  │ 260.26 um │ 200 mm  │ 260.26 um │   inf m  │  0 D    │        0°  │  0.000 +␣
↪0.200j  │
├─────────╫─────────┼──────────┼─────────┼──────────┼──────────┼─────────┼────────────┼──────────┤
│ M2.p1.i ‖ 100 mm  │ 260.26 um │ 200 mm  │ 290.98 um │  500 mm  │  2 D    │   26.565°  │  0.100 +␣
↪0.200j  │
├─────────╫─────────┼──────────┼─────────┼──────────┼──────────┼─────────┼────────────┼──────────┤
│ M2.p1.o ‖ 100 mm  │ 260.26 um │ 200 mm  │ 290.98 um │ -500 mm  │ -2 D    │   26.565°  │ -0.100 +␣
↪0.200j  │
└─────────╫─────────┴──────────┴─────────┴──────────┴──────────┴─────────┴────────────┴──────────┘
```

Note

Notice that only the transmissivity of M1 is specified. The reflectivity of M1 and the parameters of M2 are specified by reference, so that we can later vary M1 transmission and all other parameters will update. See KatScript (page 125) for more information.

Here Python f-strings are used to substitute parameters into the KatScript.

Relating the models

We can relate the spot position shifts and tilts derived from the geometric optics model, to the mode amplitudes in the modal model. Starting from the relations in Table 1 of [19] we can rearrange to obtain (See section 4.4.2 of [20] for a recent derivation),

$$\Delta x(z) = \mathcal{R}_e\left(a_{10}(z)\right) w(z)$$

$$\Delta y(z) = \mathcal{R}_e\left(a_{01}(z)\right) w(z)$$

$$\theta^x(z) = \mathcal{I}_m\left(a_{10}(z)\right) \theta^x_{\text{div}}$$

$$\theta^y(z) = \mathcal{I}_m\left(a_{01}(z)\right) \theta^y_{\text{div}}$$

We now define a function *run_simulation* which handles executing the the simulation for us. Additionally we can define a function *process_sim* which takes the `ArraySolution` object and extracts the geometric parameters, using the above relations, from our `amplitude_detector` (page 172) and `beam_property_detector` (page 173). See [16] for a precise derivation.

```
def run_simulation(mirror, theta_low, theta_high, npoints):
    return process_sim(
        kat.run(f"""
            xaxis(
                parameter=M{mirror:.0f}.xbeta,
                mode=lin,
                start={theta_low},
                stop={theta_high},
```

(continues on next page)

(continued from previous page)

```
                steps={npoints},
                pre_step=none,
                post_step=none
            )
            """)
        )

def process_sim(out):
    # Comparison
    acirc = np.sqrt(out["pcirc"].real) # circulating amplitude

    # intermodal phase HG10 -> HG00
    # at mirror 1
    intermodal_phase_1 = np.angle(out["det10_1"]) - np.angle(out["det00_1"])
    # at mirror 2
    intermodal_phase_2 = np.angle(out["det10_2"]) - np.angle(out["det00_2"])

    # Real part of \power normalised HG10 mode ampltiude
    # at M1
    a_shift_1 = (np.abs(out["det10_1"])/acirc)*np.cos(intermodal_phase_1)
    # at M2
    a_shift_2 = (np.abs(out["det10_2"])/acirc)*np.cos(intermodal_phase_2)

    # Imaginary part of power normalised HG10 mode ampltiude
    # at M1
    a_tilt_1 = (np.abs(out["det10_1"])/acirc)*np.sin(intermodal_phase_1)
    # at M2
    a_tilt_2 = (np.abs(out["det10_2"])/acirc)*np.sin(intermodal_phase_2)

    # Spot position shift
    # at M1
    dx_finesse_1 = out["bp_n2"].real*a_shift_1
    # at M2
    dx_finesse_2 = out["bp_n3"].real*a_shift_2

    # Wavefront tilt at
    # M1
    dtheta_1 = a_tilt_1*kat.lambda0/(np.pi*out["cavwaist"].real)
    # M2
    dtheta_2 = a_tilt_2*kat.lambda0/(np.pi*out["cavwaist"].real)

    print("Results for finesse solutions:")
    print("Intermodal_1: ", max(intermodal_phase_1/np.pi), min(dx_finesse_1/np.pi), " pi")
    print("Intermodal_2: ", max(intermodal_phase_2/np.pi), min(dx_finesse_2/np.pi), " pi")

    print("Δx1: ", max(dx_finesse_1), min(dx_finesse_1), " m")
    print("Δx2: ", max(dx_finesse_2), min(dx_finesse_2), " m")
    print("Δθ1: ", max(dtheta_1), min(dtheta_1), " m")
    print("Δθ2: ", max(dtheta_2), min(dtheta_2), " m")

    return (
        intermodal_phase_1,
        intermodal_phase_2,
        dx_finesse_1,
        dx_finesse_2,
        dtheta_1,
        dtheta_2,
        out.x[0]
    )
```

Discussion

Lastly, we can define a function to process these results and plot the output.

```
def comparison_plot(mirror, theta_low, theta_high, npoints, ax=None):

    delta_x1, delta_x2, delta_theta = geometric(
        mirror, theta_low, theta_high, npoints
    )

    (
        intermodal_phase_1,
        intermodal_phase_2,
        dx_finesse_1,
        dx_finesse_2,
        dtheta_1,
        dtheta_2,
        xaxis
    ) = run_simulation(mirror, theta_low, theta_high, npoints)

    if ax is None:
        fig, ax = plt.subplots(nrows=3, sharex=True, figsize=(4, 6/3*4))

    xaxis = 1e6 * xaxis

    ax2col = "darkgreen"

    # Plot dx1
    l3, = ax[0].plot(xaxis, 1e6 * delta_x1, "-", label="Geometric")
    l4, = ax[0].plot(xaxis, 1e6 * dx_finesse_1, "--",label="Modal")
    ax[0].plot([], [], "--", color=ax2col, label="Difference")
    ax[0].set_ylabel(r"$\Delta \mathrm{x}_1$ [µm]")

    # Plot dx1 err
    ax1_2 = ax[0].twinx()
    l5, = ax1_2.plot(xaxis, 1e12*(delta_x1 - dx_finesse_1), "--", color=ax2col)
    ax1_2.tick_params(axis="y", colors=ax2col)
    ax1_2.set_ylabel("Diff. [pm]", color=ax2col)

    # plot dx2
    ax[1].plot(xaxis, 1e6 * delta_x2, "-", label="Geometric")
    ax[1].plot(xaxis, 1e6 * dx_finesse_2, "--", label="Modal")
    ax[1].plot([], [], "--", color=ax2col, label="Difference")
    ax[1].set_ylabel(r"$\Delta \mathrm{x}_2$ [µm]")

    # plot dx2 err
    ax2_2 = ax[1].twinx()
    ax2_2.plot(xaxis, 1e12 * (delta_x2 - dx_finesse_2), "--", color=ax2col)
    ax2_2.tick_params(axis="y", colors=ax2col)
    ax2_2.set_ylabel("Diff. [pm]", color=ax2col)

    # plot dtheta1
    ax[2].plot(xaxis, 1e6 * delta_theta, "-", label="Geometric")
    ax[2].plot(xaxis, 1e6 * dtheta_1, "--", label="Modal")
    ax[2].plot([], [], "--", color=ax2col, label="Difference")
    ax[2].set_ylabel(r"$\theta^\mathrm{x}_0$ at M1 [µrad]")

    # plot dtheta1 err
    ax3_2 = ax[2].twinx()
    ax3_2.plot(xaxis, 1e12 * (dtheta_1 - delta_theta), "--", color=ax2col)
    ax3_2.tick_params(axis="y", colors=ax2col)
    ax3_2.set_ylabel("Diff. [prad]",color=ax2col)
```

(continues on next page)

(continued from previous page)

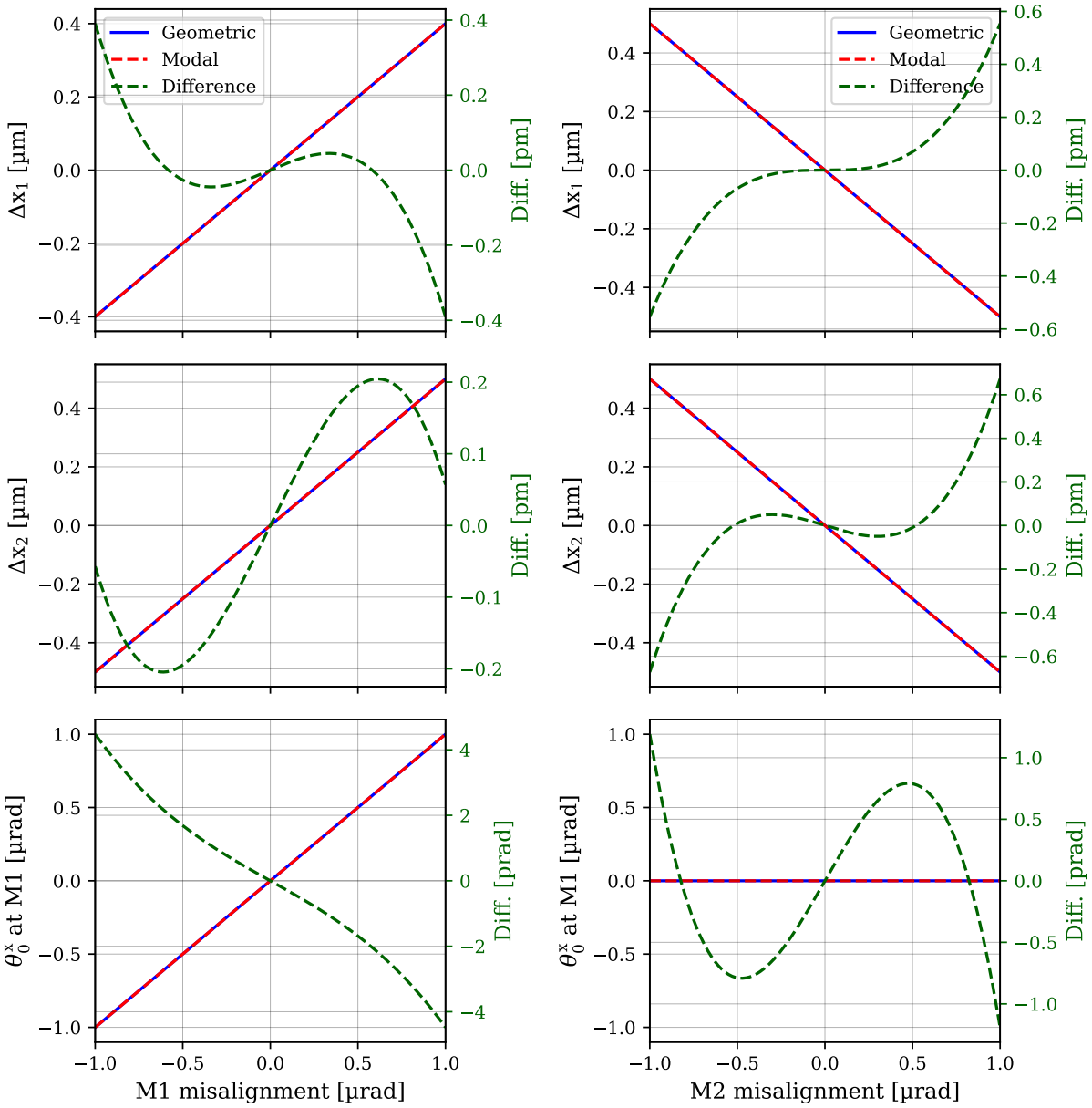
```
    ax[0].legend()
    ax[2].set_xlabel(f"M{mirror} misalignment [µrad]")

fig, ax = plt.subplots(nrows=3, ncols=2, sharex=True, figsize=(8, 6/3*4))
comparison_plot(1, theta_low, theta_high, npoints, ax=ax[:,0])
comparison_plot(2, theta_low, theta_high, npoints, ax=ax[:,1])
ax[2,1].set_ylim([-1.1, +1.1])
fig.tight_layout()
```

```
Results for analytical solutions:
Δx1:  4.0000000000000014e-07 -4.0000000000000014e-07  m
Δx2:  5.000000000000001e-07 -5.000000000000001e-07  m
Δθ:  9.999999999999993e-07 -9.999999999999993e-07  m
Results for finesse solutions:
Intermodal_1:  0.14758329930279382 -1.273240787814041e-07  pi
Intermodal_2:  -3.1834763941235237e-07 -1.5915492491370067e-07  pi
Δx1:  4.000003905247471e-07 -4.0000039052474715e-07  m
Δx2:  4.999999428915172e-07 -4.999999428915172e-07  m
Δθ1:  9.999955236676984e-07 -9.999955236676976e-07  m
Δθ2:  4.876993207838182e-12 -4.8769932079878946e-12  m
```

```
Results for analytical solutions:
Δx1:  5.000000000000001e-07 -5.000000000000001e-07  m
Δx2:  5.000000000000001e-07 -5.000000000000001e-07  m
Δθ:  0.0 0.0  m
```

```
Results for finesse solutions:
Intermodal_1:  1.516148857617145e-07 -1.591551193134631e-07  pi
Intermodal_2:  0.8524165339644525 -1.5915515721722656e-07  pi
Δx1:  5.000005536163827e-07 -5.000005536163827e-07  m
Δx2:  5.000006726945675e-07 -5.000006726945675e-07  m
Δθ1:  1.1907818466809523e-12 -1.1907818462932243e-12  m
Δθ2:  1.1180341616060985e-06 -1.1180341616060985e-06  m
```



In each plot the mirror angle is varied by from -1 µrad to +1 µrad, the left columns correspond to the M1 being rotated and the right columns correspond to M2 being rotated.

Note

Notice the wavefront tilt is only equal to the geometric cavity axis tilt at the beam waist and so these are only compared at M1. If the cavity does not have a beam waist at one of the Finesse nodes, it is not possible to directly compare these parameters.

As shown in the plots, the modal and geometric models agree up to pico-meter precision. The remaining error arises because the geometric model does not take into account the finite finesse of the optical resonator—which is fully modeled in the Finesse modal model. We can see the effect of this by reducing the mirror reflectivity.

```
kat.M2.xbeta = theta_high
kat.M1.xbeta = 0

(
    intermodal_phase_1,
```

(continues on next page)

(continued from previous page)

```
    intermodal_phase_2,
    dx_finesse_1,
    dx_finesse_2,
    dtheta_1,
    dtheta_2,
    xaxis
) = process_sim(
    kat.run(f"""
        xaxis(
            parameter=M1.T,
            mode=log,
            start=1e-7,
            stop=1,
            steps={npoints},
            pre_step=none,
            post_step=none)
        """)
    )

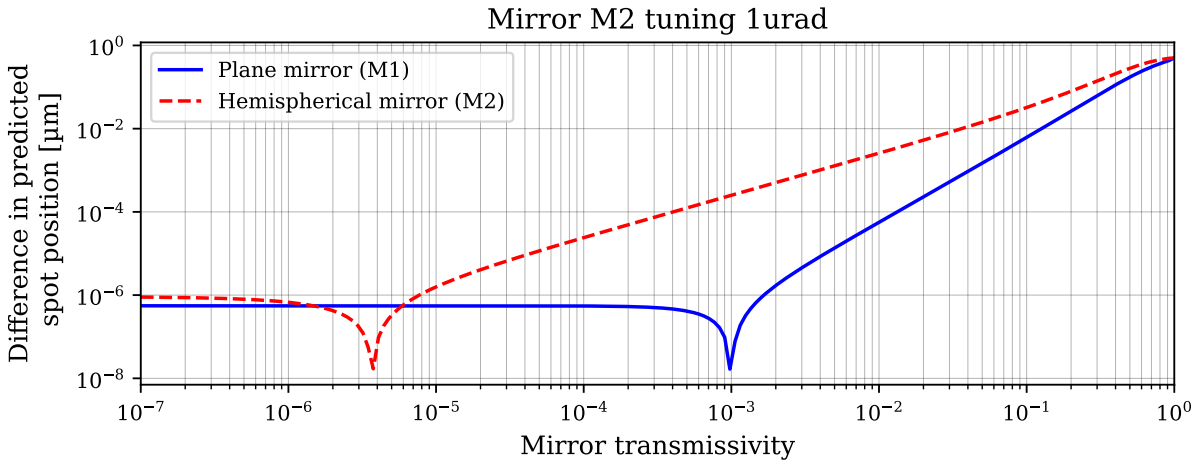
delta_x1, delta_x2, delta_theta = geometric(2, 0, kat.M2.xbeta, npoints)

fig, ax = plt.subplots(figsize=(8,8/3))

## Plot phases
ax.loglog(xaxis, 1e6 * np.abs(delta_x1[-1] - dx_finesse_1), "-", label="Plane mirror (M1)")
ax.loglog(xaxis, 1e6 * np.abs(delta_x2[-1] - dx_finesse_2), "--", label="Hemispherical␣
↪mirror (M2)")
ax.set_ylabel("Difference in predicted\n spot position [μm]")
ax.set_xlabel("Mirror transmissivity")
ax.set_title(f"Mirror M2 tuning {1e6 * kat.M2.xbeta:.0f}urad")
ax.legend()
```

```
Results for finesse solutions:
Intermodal_1:  0.0 -1.5915511931348423e-07  pi
Intermodal_2:  0.8524168204449615 -1.591552288373988e-07  pi
Δx1:  0.0 -5.000005536164491e-07  m
Δx2:  0.0 -5.000008976959744e-07  m
Δθ1:  1.1174658828196761e-06 -3.4407969937731483e-12  m
Δθ2:  1.7335501779427744e-06 0.0  m
Results for analytical solutions:
Δx1:  0.0 -5.000000000000001e-07  m
Δx2:  0.0 -5.000000000000001e-07  m
Δθ:  0.0 0.0  m
```

```
<matplotlib.legend.Legend at 0x78add8405090>
```



As shown, when the mirror transmission is very low, the spot position is accurately predicted up to a noise floor. This noise floor could either be numerical noise, or, the limitations of the approximations used in the modal model. When the mirrors are infinitely transmissive the geometric model fails completely, predicting a spot shift where none should occur.

5.2.5 Near unstable cavities

This example will simulate what happens as a well designed cavity is pushed close to instability. The example show recipes for the following activities and link to more detailed documentation:

- Designing an input telescope
- Updating model parameters
- Producing a cavity scan
- Using a CCD
- Locking a cavity

See also

Modelling with higher-order modes (page 327)

The model

We will consider the LIGO Pre-Mode-Cleaner design from [21]. This design is a ring resonator consisting of two flat mirrors and one curved mirror.

Designing an input telescope

In this section we will design an input telescope from a selection of standard lab optics.

Plotting a cavity scan

In this section we will show how the cavity control signals change as the cavity is pushed closer to instability.

5.2.6 Aligning and translating a beam with steering mirrors

This example works out how a beam can be shifted off-axis in FINESSE using a telescope such as depicted in Fig. 5.1, and verifies the analytical results derived in Simulating off-axis beams: calculating the shift (page 314) with simulations in FINESSE.

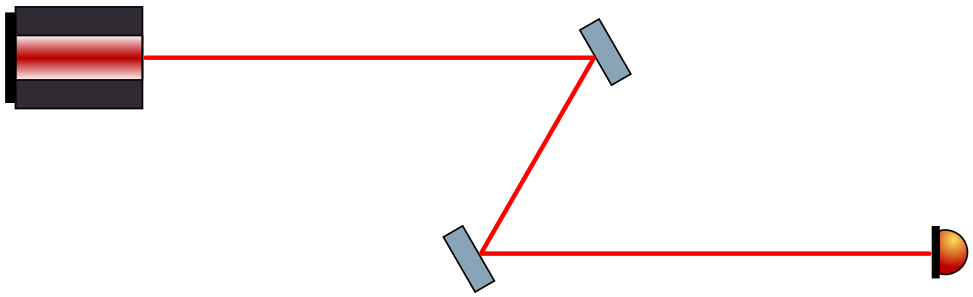


Fig. 5.1: Basic setup

Introduction

Given a Z-shaped telescope setup with two beamsplitters such as depicted in Fig. 5.1, we apply a small additional tilt β to the angle of incidence α of both beamsplitters. This will cause the outgoing beam to shift over a small distance Δ with respect to the untilted setup.

Such a small extra tilt is applied in FINESSE via the `xbeta` parameter of a `bs` (page 149) component. With respect to the "untilted beam" (i.e. $\beta = 0$) we derive in section Simulating off-axis beams: calculating the shift (page 314) both the shift in its position Δ and the change of its phase $\delta\varphi$, each as a function of the angle β and the distance s between the beamsplitters. For the shift Δ we find there Eq. (6.9):

$$\Delta = s \cdot \sin 2\beta \tag{5.1}$$

while for the phase change $\delta\varphi$ we derive Eq. (6.17):

$$\delta\varphi = \frac{2s \cdot \sin^2 \beta}{\lambda} \cdot 360° \tag{5.2}$$

Basic setup

Our basic setup consists of:

- a `laser` (page 165) with power 1.5 Watt
- two `beamsplitter` (page 149) components, bs1 and bs2:
  - the first 1.0 km right of the laser, the second 400 meters away from the first.
  - each having an angle of incidence $\alpha = 30°$.
  - each tilted over a small angle $\beta = 10^{-5}$ radians.
- a Gaussian beam with waist-size $w_0 = 10$ mm, at a distance 1.2 km right of the laser, i.e. precisely in the middle between the two beamsplitters.
- we do most calculations using maxtem 7 (i.e. $n + m$ of the higher order modes is ≤ 7)
- furthermore, the reference wavelength of FINESSE is $\lambda = 1064$ nm.

For these parameters Eq. (5.1) gives the following value for dimensionless Δ

$$\frac{\Delta}{w_0} = \frac{400 \cdot \sin(2 \cdot 10^{-5})}{10^{-2}} \approx 0.8$$

while Eq. (5.2) gives for $\delta\varphi$

$$\delta\varphi = \frac{2 \cdot 400 \cdot \sin^2 10^{-5}}{1.064 \cdot 10^{-6}} \cdot 360° \approx 27.1°$$

Our basic FINESSE setup looks as follows

```
import numpy as np
import matplotlib.pyplot as plt
import finesse
finesse.configure(plotting=True)

power = 1.5  # default laser power in Watt
w0 = 10e-3   # default gaussian waist size in meter3
alpha = 30   # angle of incidence for both BS in degrees0
xbeta = 1e-5 # default tilt of both beam splitters in radians
s1 = 1000    # distance till 1st beamsplitter
s2 = 400     # distance between beamsplitters
s3 = 600     # default distance between 2nd beamsplitter and detector
z0 = -1200   # default waist position, 1200 meter right of laser
maxtem = 7   # default maxtem

basescript = f"""
laser l1 P={power}
gauss g1 l1.p1.o w0={w0} z={z0}
modes(maxtem={maxtem})

space s1 l1.p1 bs1.p1 L={s1}
beamsplitter bs1 R=1 T=0 alpha={alpha} xbeta={xbeta}
space s2 bs1.p2 bs2.p1 L={s2}
beamsplitter bs2 R=1 T=0 alpha=bs1.alpha xbeta=bs1.xbeta
space s3 bs2.p2 n1.p1 L={s3}

# Add a 'nothing' component at the location of the detector
nothing n1
"""

basekat = finesse.Model()
basekat.parse(basescript);
```

Measuring a parallel beam

We will use our experimental setup in combination with a 1-dimensional `ccdline` (page 174) to measure the beam shape after 600 and 1000 meters respectively.

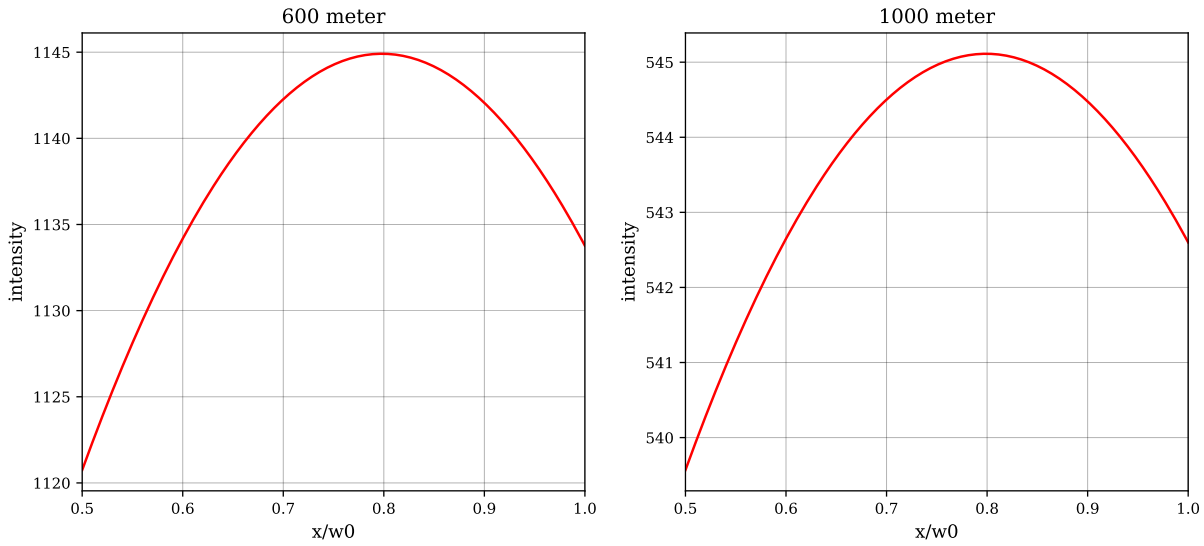
```
# Simulations: measure beam 600 and 1000 meter after second BS
s3a = 600
s3b = 1000

kat1 = finesse.Model()
kat1.parse(basescript)
kat1.parse("""
# Set a 1-D CCD at n1, measure around peak at 0.8 at 600 and 1000m
ccdline ccd1 node=n1.p1.i xlim=[0.5,1.0] npts=200
# Also put a full 2-D CCD at the same position
ccd ccd2 node=n1.p1.i xlim=[-3,3] ylim=[-3,3] npts=200
""")

out1 = kat1.run(f"""
    series(
        noxaxis(name="S600"),
        change(s3.L={s3b}),
        noxaxis(name="S1000")
    )
""")
```

We have also added a `ccd` (page 174) of which we will use the output below to plot the 2-dimensional beam cross section. Plotting the output from the two 1-dimensional `ccdline` (page 174) detectors

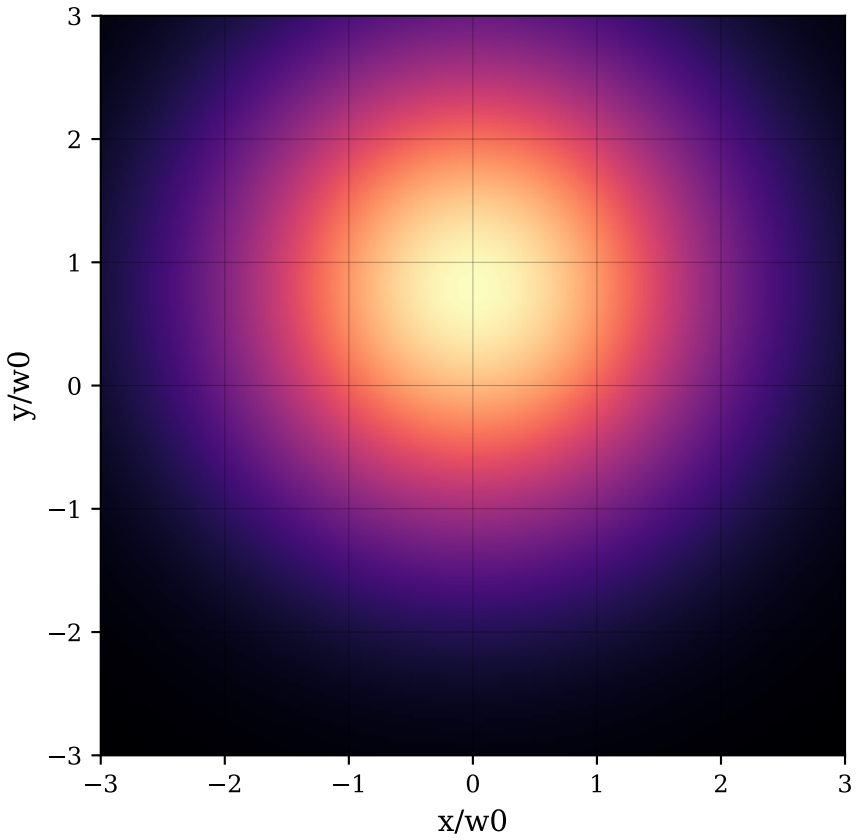
```
# Plot the results
f,ax = plt.subplots(ncols=2, figsize=(12, 5))
for (i, name, s3val) in ([0, 'S600', s3a], [1, 'S1000', s3b]):
    ax[i].plot(kat1.ccd1.xdata, out1[name]['ccd1'],'r')
    ax[i].set_title(f"{s3val:.0f} meter")
    ax[i].set_xlabel("x/w0")
    ax[i].set_ylabel("intensity")
```



We see that the Gaussian profile over distance becomes lower and wider, but retains its peak at the expected position $\frac{x}{w0} = 0.8$, confirming we are simulating a parallel but shifted beam.

For reference we also plot the 2-dimensional beam cross section at 600 meters as measured with the ccd (page 174):

```
f,ax = plt.subplots(ncols=1, figsize=(5, 5))
pxy_extent = (kat1.ccd2.xdata.min(), kat1.ccd2.xdata.max(),
              kat1.ccd2.ydata.min(), kat1.ccd2.ydata.max())
ax.imshow(out1['S600']['ccd2'].T, aspect='auto', extent = pxy_extent)
ax.set_xlabel("x/w0")
ax.set_ylabel("y/w0");
```



Measuring phase shift across the beam

We next measure the phase shift of the outgoing beam resulting from the tilting of the beamsplitters. For this we use the following simulation script (again in addition to the basic experimental setup (page 67)), using a `fline` (page 176) detector measuring the phase along the beam cross section at a distance 600 meter after the second beamsplitter. For reference, we also run the same simulation for $\beta = 0$. We expect the full result to be shifted over $0.8w_0$ and to have a phase that is larger by about $27.1°$.

```
# Plotting range for x: [peak-dx...peak+dx]
dx = 0.8

# Calculate expected Delta and delta phi
Delta = s2*np.sin(2*xbeta)/w0
dphi = 2*s2*np.sin(xbeta)**2/basekat.lambda0*360

kat2 = finesse.Model()
kat2.parse(basescript)
kat2.parse(f"""
fline fl1 node=n1.p1.i xlim=[{Delta}-{dx},{Delta}+{dx}] npts=200
fline fl2 node=n1.p1.i xlim=[-{dx},{dx}] npts=40
""")

out2 = kat2.run("""
    series(
        noxaxis(name='full'),
        change(bs1.xbeta=0),
        noxaxis(name='ref')
    )
""")
```

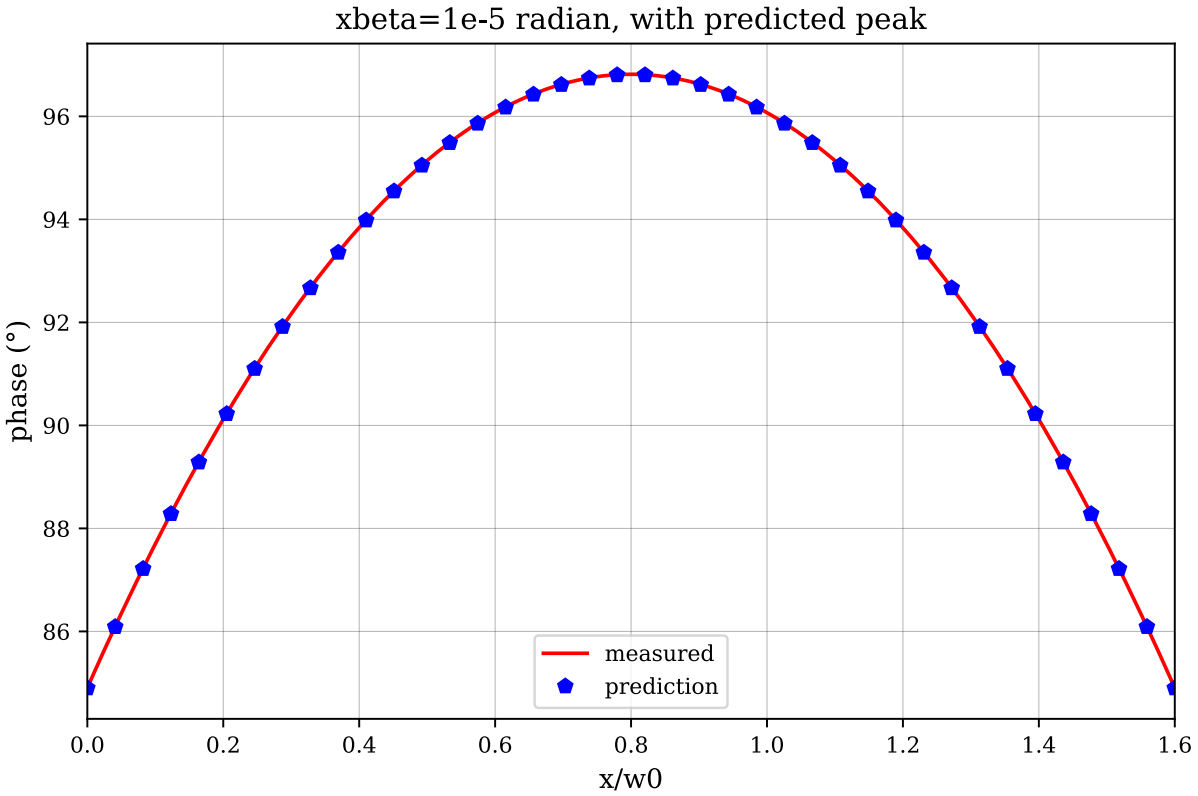
We expect the phase for the shifted beam to satisfy

$$\varphi_{\beta\neq 0}(x + \Delta) = \varphi_{\beta=0}(x) + \delta\varphi$$

In the plot below we combine the result for the full simulation (red drawn line) with this expected result (blue points), obtained from translating the $\beta = 0$ curve. It shows that the simulation matches the prediction perfectly.

```
f,ax = plt.subplots(ncols=1, figsize=(8, 5))

ax.plot(kat2.fl1.xdata,
        np.angle(out2['full']['fl1'], deg=True),
        'r', label="measured")
ax.plot(kat2.fl2.xdata + Delta,
        np.angle(out2['ref']['fl2'], deg=True) + dphi,
        'bp', label="prediction")
ax.set_title("xbeta=1e-5 radian, with predicted peak")
ax.set_xlabel("x/w0")
ax.set_ylabel("phase (°)")
ax.legend(loc='lower center');
```



Measuring phase shift versus tilt angle

The expected phase dependence as a function of the tilt angle β is given by Eq. (5.2), which we can verify using an `ad` (page 172) detector. We will use the following script (again in addition to the basic experimental setup (page 67))

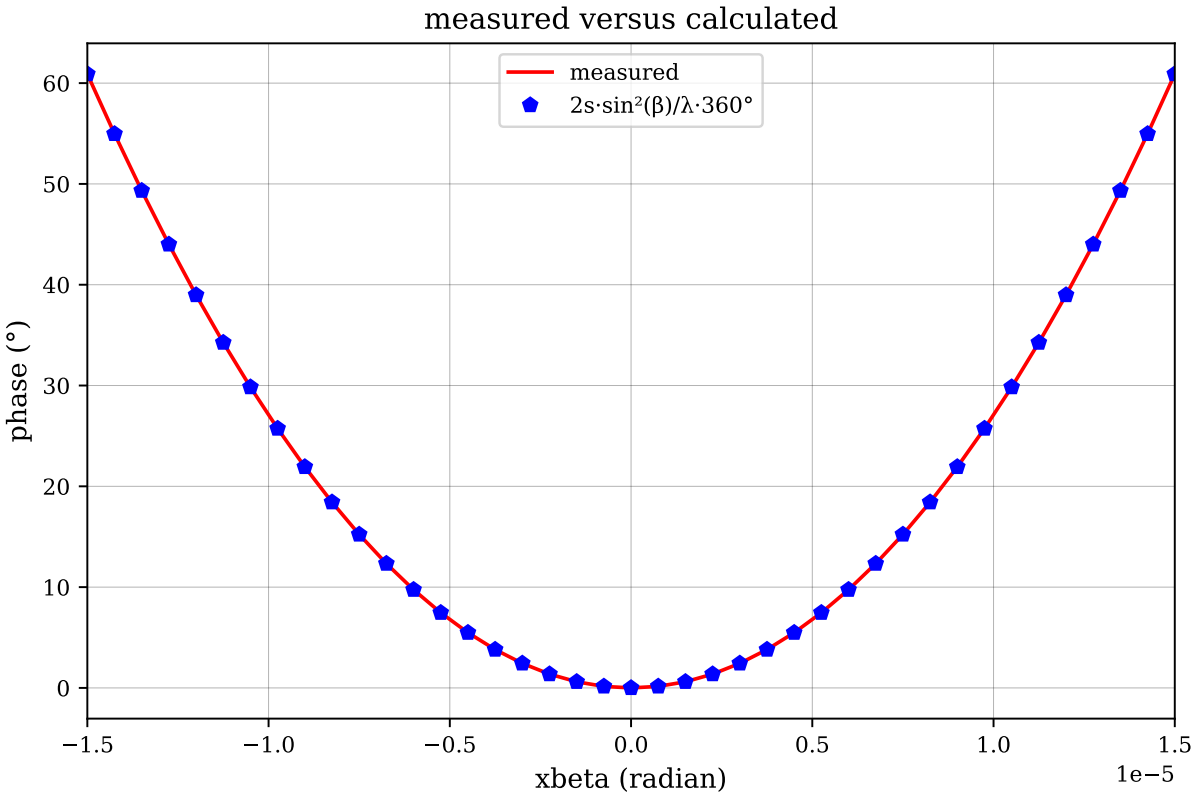
```
kat3 = finesse.Model()
kat3.parse(basescript)

# Note: important to specify n,m
kat3.parse("amplitude_detector ad1 node=n1.p1.i f=0 n=0 m=0")
out3 = kat3.run("xaxis(bs1.xbeta, lin, -1.5e-5, 1.5e-5, 40)")
```

and plot the result together with the expected behaviour Eq. (5.2). Note that we now have to grab the horizontal data from the `ArraySolution` object (*out3*) instead of from the `Model` object (*kat3*) since for this `ad` (page 172) detector we use an `xaxis` (page 209) instead of a `noxaxis` (page 208) action. Again we see that the simulation matches perfectly with the prediction.

```
f,ax = plt.subplots(ncols=1, figsize=(8, 5))

ax.plot(out3.x[0], np.angle(out3['ad1'], deg=True),
        'r', label="measured")
ax.plot(out3.x[0], 2*s2*np.sin(out3.x[0])**2/basekat.lambda0*360,
        'bp', label="2s·sin²(β)/λ·360°")
ax.set_title("measured versus calculated")
ax.set_xlabel("xbeta (radian)")
ax.set_ylabel("phase (°)")
ax.legend(loc="upper center");
```



5.3 Modulation, feedback and control examples

Here you will find a selection of examples for performing various simulation about modulation, feedback and control using FINESSE.

5.3.1 Locking actions

This example demonstrates the use of various locking-related actions. See also the Actions, Analyses, and Solutions introduction (page 198) for an introduction into actions, and Tuning (page 229) for detailed documentation of locking actions.

The FINESSE model

Defining a simple interferometer.

```
import finesse
from finesse.analysis.actions import GetErrorSignals, CheckLinearity, RunLocks, DragLocks
from finesse.analysis.actions import SensingMatrixDC, OptimiseRFReadoutPhaseDC
from finesse.analysis.actions.locks import SetLockGains
finesse.configure(plotting=True)
```

(continues on next page)

(continued from previous page)

```
kat = finesse.Model()
kat.parse(
    """
    # Adding a laser and modulator for use in
    # Pound-Drever-Hall-like signals.
    l L0 P=1
    s l_mod1 L0.p1 eo1.p1
    mod eo1 10M 0.1

    s s0 eo1.p2 BS.p1
    bs BS R=0.5 T=0.5

    # North Arm
    s s1 BS.p2 NI.p1
    m NI R=0.99 T=0.01 Rc=1429 phi=90
    s CAV NI.p2 NE.p1 L=10
    m NE R=0.991 T=0.009 Rc=1430 phi=90

    #East Arm
    s s2 BS.p3 EI.p1
    m EI R=0.99 T=0.01 Rc=1429 phi=0
    s CAV2 EI.p2 EE.p1 L=10
    m EE R=0.991 T=0.009 Rc=1430 phi=0

    dof NEz NE.dofs.z +1
    dof EEz EE.dofs.z +1
    dof NIz NI.dofs.z +1

    readout_rf rd_pdh1 NI.p1.o f=10M
    readout_rf rd_pdh2 EI.p1.o f=10M
    readout_rf rd_DF BS.p4.o f=10M

    # Two locks to put the cavities in resonance, and
    # one to move to the dark fringe.
    lock cav1_lock rd_pdh1.outputs.I NEz.DC 1 1e-9
    lock cav2_lock rd_pdh2.outputs.I EEz.DC 1 1e-9
    lock DF_lock rd_DF.outputs.I NIz.DC 1 1e-9

    cav cav1 NI.p2.o
    cav cav2 EI.p2.o
    modes(maxtem=5)
    """
)
```

```
<finesse.model.Model at 0x7dbaf66be900>
```

Using different Actions

As currently set up, the interferometer is in its locked condition. Let's kick it slightly away from this lock to see how the locking actions work.

```
kat.NEz.DC += 0.01
kat.EEz.DC += 0.001
```

GetErrorSignals

Use `GetErrorSignals` (page 230) to quickly see the error signals of the locks in a model at its current state. If particular locks are not specified, the error signals for all of them will be given.

```
sol_errs = kat.run(GetErrorSignals("cav1_lock", "DF_lock"))
print("Initial Error Signals:", sol_errs.results, end="\n\n")
```

```
Initial Error Signals: [-1.91529891e-03 -3.68269613e-11]
```

OptimiseRFReadoutPhaseDC

Use `OptimiseRFReadoutPhaseDC` (page 229) to determine and set the optimal demodulation phase for particular pairs of DOFs and RF readouts. If no pairs are specified, the demodulation phases for all locks will be optimised. Initial and optimal readout phases can be extracted from the solution object or by setting *verbose=True*.

```
sol_phases = kat.run(
    OptimiseRFReadoutPhaseDC(
        "NEz", "rd_pdh1_I", "EEz", "rd_pdh2_I", "NIz", "rd_DF_I", verbose=True
    )
)
```

```
┌──────────────────┬──────────┬──────────┬────────┐
│                  ‖ rd_pdh1  ‖ rd_pdh2  ‖ rd_DF  │
╞══════════════════╪══════════╪══════════╪════════╡
‖ Previous phases  ‖        0 ‖        0 ‖      0 ‖
├──────────────────┼──────────┼──────────┼────────┤
‖ Optimised phases ‖    179.8 ‖    179.8 ‖  179.3 ‖
└──────────────────┴──────────┴──────────┴────────┘
```

Note that each readout phase can only be optimised to a single `DegreeOfFreedom` (page 190). If a readout is paired with multiple DOFs, its phase will be optimised to the first `DegreeOfFreedom` (page 190) given and others are ignored during this step:

```
kat.run(OptimiseRFReadoutPhaseDC("NEz", "rd_pdh1_I", "EEz", "rd_pdh1_I"));
```

```
/usr/local/lib/python3.13/site-packages/finesse/analysis/actions/sensing.py:241:␣
↪OptimiseRFReadoutPhaseDCMultipleDofWarning: Readout 'rd_pdh1' is paired with␣
↪multiple drives. Demodulation phase will be optimised using the first drive␣
↪('NEz.DC'), ignoring drive(s) 'EEz.DC'.
  warn(
```

SetLockGains

Use `SetLockGains` (page 230) to determine and set the gains for particular locks. If no locks are specified, all enabled locks in the model are used. Gains are automatically calculated in `RunLocks` (page 231) when using the Newton method.

```
kat.run(SetLockGains())
```

```
<SeriesSolution of series @ 0x7dbaf3819550 children=0>
```

SensingMatrixDC

Use `SensingMatrixDC` (page 230) to calculate the entire sensing matrix. This action is also called within `RunLocks` (page 231) when using the Newton method.

```
sol_matrix = kat.run(SensingMatrixDC(["NEz", "EEz", "NIz"],
                                     ["rd_pdh1", "rd_pdh2", "rd_DF"]))
print("Sensing Matrix:", sol_matrix, end="\n\n", sep="\n")
```

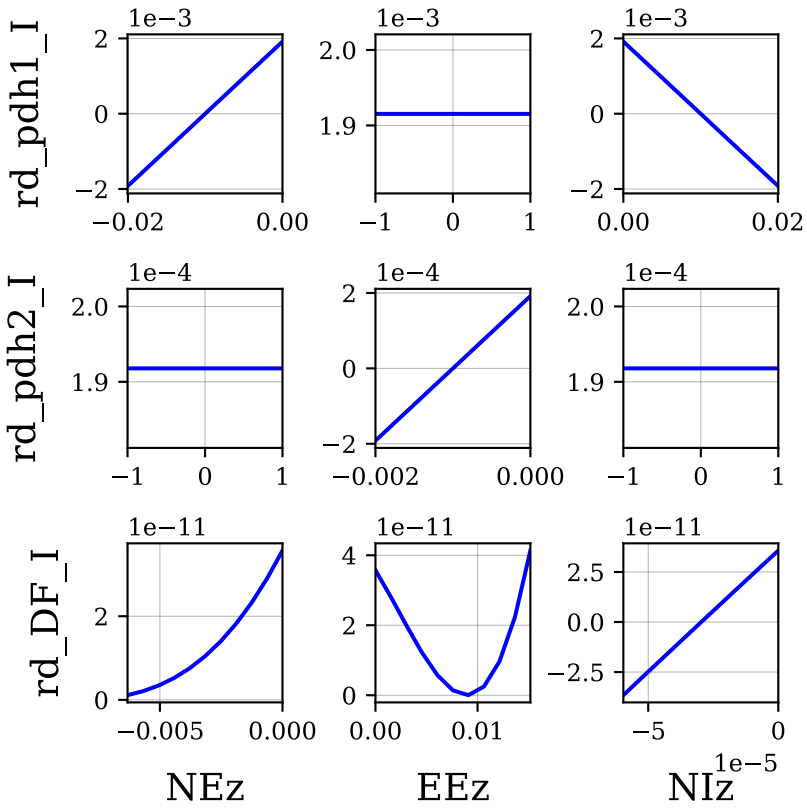
```
Sensing Matrix:
```

	rd_pdh1_I	rd_pdh1_Q	rd_pdh2_I	rd_pdh2_Q	rd_DF_I	rd_DF_Q
NEz	0.19	1.5E-09	0	0	1.1E-08	1.1E-08
EEz	0	0	0.19	-8.1E-10	-4.7E-09	-1.1E-08
NIz	-0.19	-1.5E-09	0	0	1.2E-06	-5.6E-12

CheckLinearity

Use `CheckLinearity` to plot the error signals as a function of each DOF independently. When locking, we assume that these plots are either linear or of very small slope (like, in this case, the bottom middle plot).

```
kat.run(CheckLinearity(plot_results=True))
```

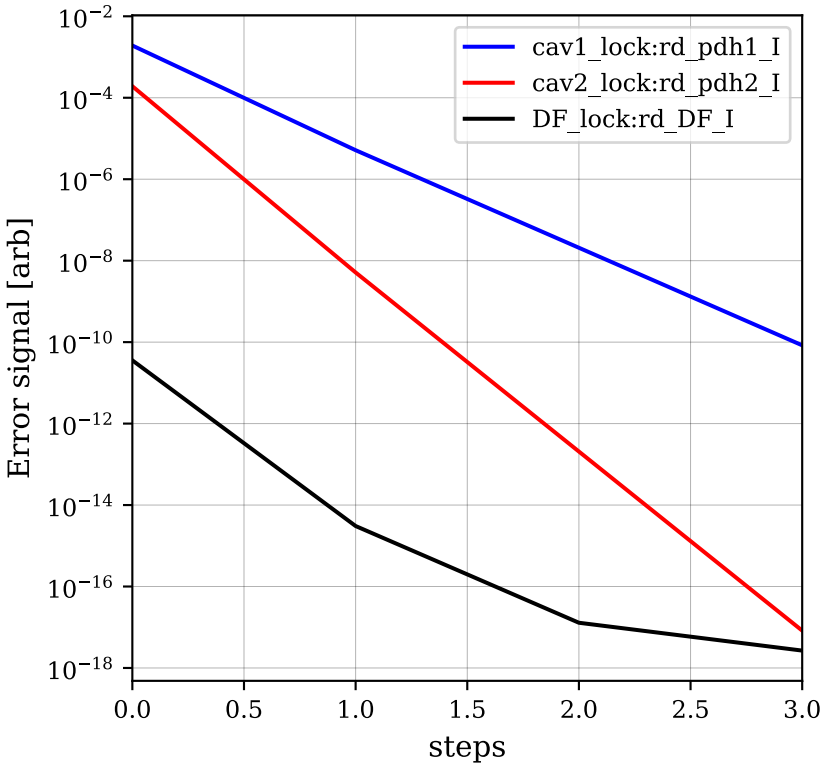



```
<CheckLinearitySolution of run locks @ 0x7dbaf38fa120 children=0>
```

RunLocks

Use `RunLocks` (page 231) to adjust the lock DOFs until the error signals are within tolerances or until max iterations are reached. This can be done with either the *proportional* method or the more advanced *newton* method (see `RunLocks` (page 231) for more information).

```
kat.run(
    RunLocks(
        method="newton",
        display_progress=False,
        max_iterations=100,
        scale_factor=1
    )
).plot()
```

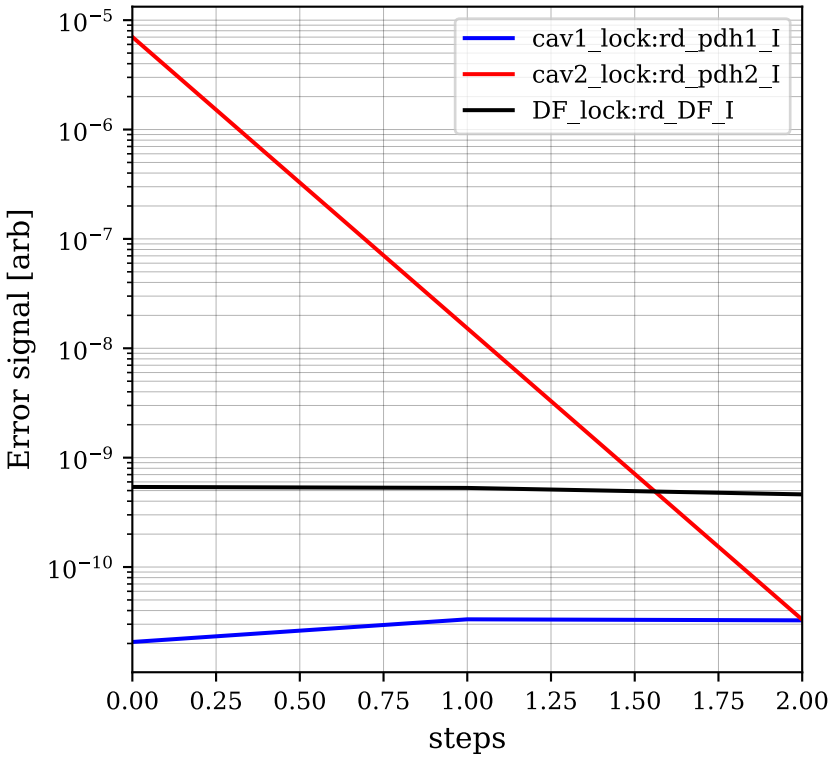


DragLocks

Use `DragLocks` to gradually introduce a substantial defect in your model, which you might not be able to lock to immediately.

```
print("Results of dragging locks:")
kat.run(OptimiseRFReadoutPhaseDC())
kat.run(
    DragLocks(
        method="newton",
        parameters=["NE.Rcx"],
        stop_points=[2],
        relative=True,
        max_iterations=100,
    )
).plot()
```

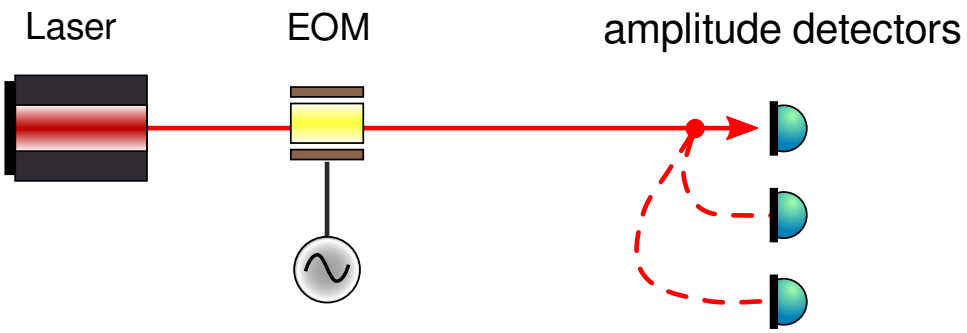
```
Results of dragging locks:
```



5.3.2 Radio-frequency modulation and Bessel functions

This example demonstrates the basic use of modulators (page 117) and `amplitude_detector` (page 172) for detecting specific frequency components of the light field.

The optical configuration



The optical layout simply consists of a laser, an electro-optical `Modulator` (page 159) (EOM) and three `AmplitudeDetector` (page 172)s. Note that all three detectors are connected to the same node and thus probe the same field amplitude.

The FINESSE model

```
import finesse
finesse.configure(plotting=True)

kat = finesse.Model()
kat.parse(
    """
    # Laser with power 1W
    laser L0 P=1
    s s0 L0.p1 eom1.p1
```

(continues on next page)

(continued from previous page)

```
    # Phase modulator at 40kHz, with a modulation index of 0.05
    mod eom1 f=40k midx=.05 order=5 mod_type=pm

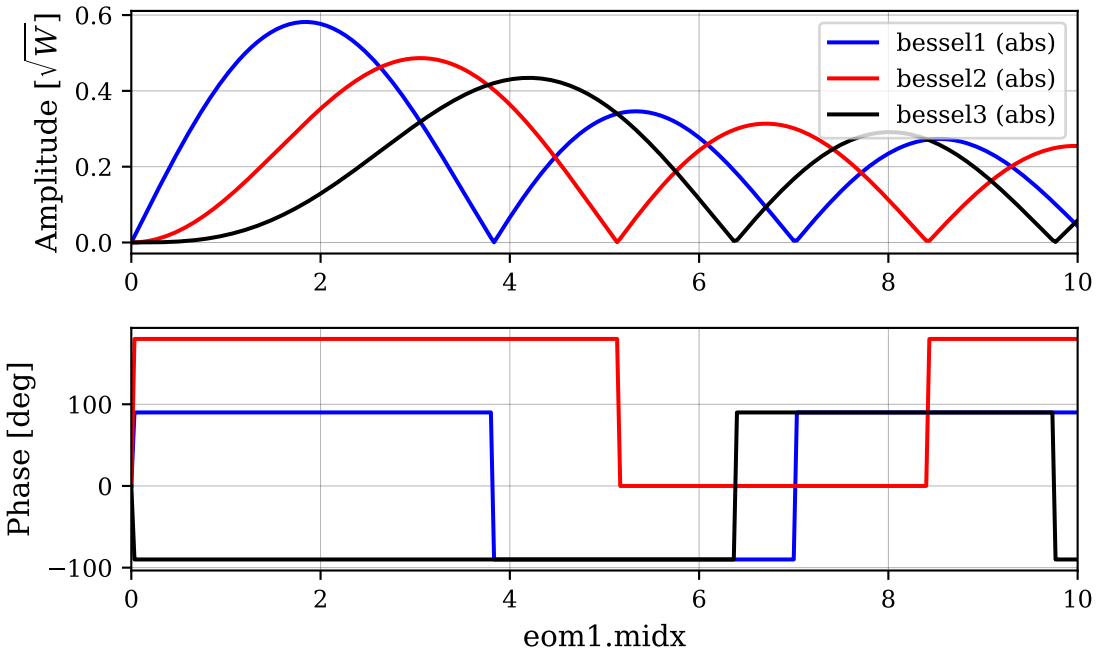
    # Three amplitude detectors, looking at the first three positive modulation
    # sidebands
    ad bessel1 eom1.p2.o f=40k
    ad bessel2 eom1.p2.o f=80k
    ad bessel3 eom1.p2.o f=120k
    """
)
```

```
<finesse.model.Model at 0x78ce00302900>
```

A laser beam is passed through an electro-optical modulator (EOM) which applies a phase modulation (page 117) with a frequency of 40 kHz and a modulation index of 0.05. The modulator is set such that up to five orders or sidebands will be created (at plus/minus 40, 80, 120, 160 and 200 kHz respectively). The modulated beam is then detected with three amplitude detectors, each of which is set to a particular frequency. This means, for example, that the detector `bessel2` can only 'see' the sideband at +80 kHz and no other part of the beam.

Output plots

```
# Sweep the modulation index of the modulator from 0 to 10
out = kat.run("xaxis(eom1.midx, lin, 0, 10, 300)")
out.plot();
```



Phase modulation (with up to five higher harmonics) is applied to a laser beam and amplitude detectors are used to measure the field at the first three harmonics. The three traces show the amplitude of the single sidebands at 40kHz, 80 kHz and 120 kHz as a function of the modulation index at the modulator. The sideband amplitude are given as Bessel functions of the first kind $J_n(m)$ with n being the sideband order and m the modulation index.

5.3.3 Pound-Drever-Hall locking

In order to maintain the resonance condition, optical resonators are often locked. Optical resonators do not always need to be locked in FINESSE, as the mirrors can be placed at the operating point exactly. However, some effects such as beam shape distortions drag the operating point of the resonator, and in these instances it is valuable to simulate the locking loop.

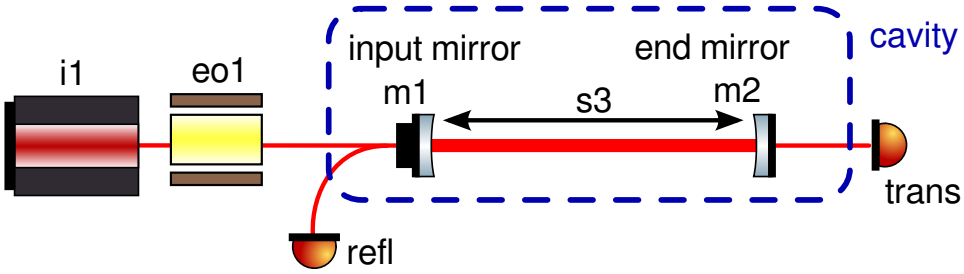
In this simple example, a two mirror cavity will be locked using a simple loop, using the Pound-Drever-Hall technique [22]. We will move the end mirror and watch the simulation adjust the position of the input mirror to maintain the operating point.

The optical setup

The optical setup and kat code are shown below.

See also

Section 8.10 of [18] for background information on resonator length sensing and control.




```
import finesse
finesse.configure(plotting=True)

model = finesse.Model()
model.parse(
    """
    laser i1 P=1 f=0
    space s1 portA=i1.p1 portB=eo1.p1
    modulator eo1 15M 0.7 order=3
    space s2 portA=eo1.p2 portB=m1.p1
    mirror m1 R=0.9 T=0.1
    space s3 portA=m1.p2 portB=m2.p1 L=1
    mirror m2 R=0.9 T=0.1

    power_detector_dc refl node=m1.p1.o
    power_detector_dc trans node=m2.p2.o
    """
)
```

```
<finesse.model.Model at 0x7cf84d14a900>
```

See also

An optical cavity (page 43) for an introduction to modeling a simple cavity.

Optimising the demodulation phase

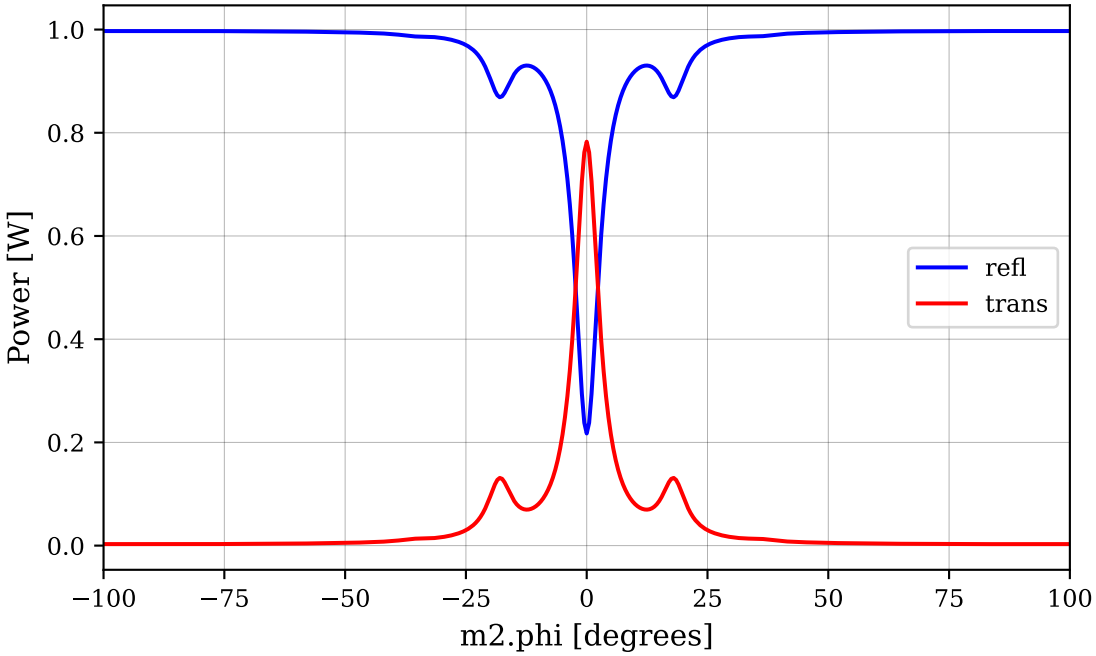
We can then create a number of power detectors with different demodulation (page 117) phases and inspect to see which one is optimal.

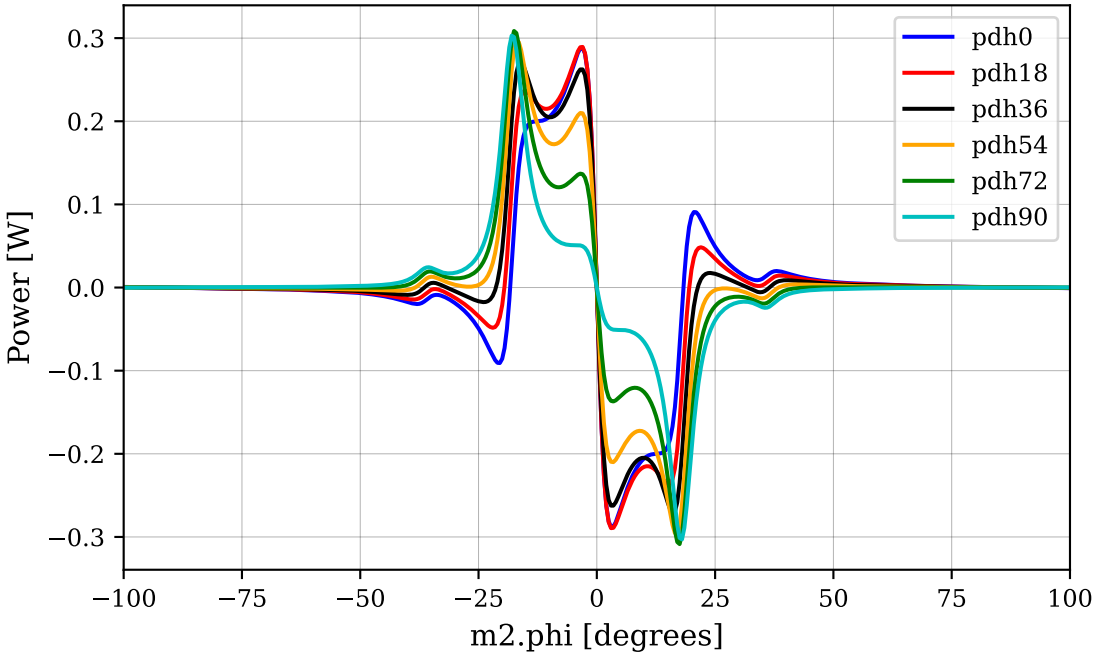
```
import numpy as np

for phase in np.linspace(0, 90, endpoint=True, num=6):
    model.parse(f"pd1 pdh{phase:.0f} node=m1.p1.o f=eo1.f phase={phase:.0f}")
```

The demodulation frequency of the photodiode is linked to the modulation frequency of the electro-optic modulator using a reference (see KatScript (page 125)). As you can see below, if we scan the cavity we see error signal cross zero at the operating point. We also see the resonance peaks for the sidebands created by the EOM.

```
sol = model.run("xaxis(m2.phi, lin, -100, 100, 400)")
sol.plot();
```





We can see that a demodulation phase of 18 degrees produces a reasonable error signal.

If required, we can compute the error signal by producing many detectors at different phases and computing the gradient of the error signal, as shown below.

```
def remove_demod_pds(verbose=True):
    for el in model.elements:
        if el.startswith('pdh'):
            model.remove(el)
```

(continues on next page)

(continued from previous page)

```
            if verbose: print(f'Removed {el}')

# Remove the old detectors.
remove_demod_pds()

# Create another array of phases.
phases = np.arange(0,90,1)
for phase in phases:
    model.parse(f"pd1 pdh{phase:.0f} node=m1.p1.o f=eo1.f phase={phase:.0f}")

# Sweep the mirror tuning across 3 points: ±1° and 0.
out = model.run("xaxis(m2.phi, lin, -1, 1, 2)")

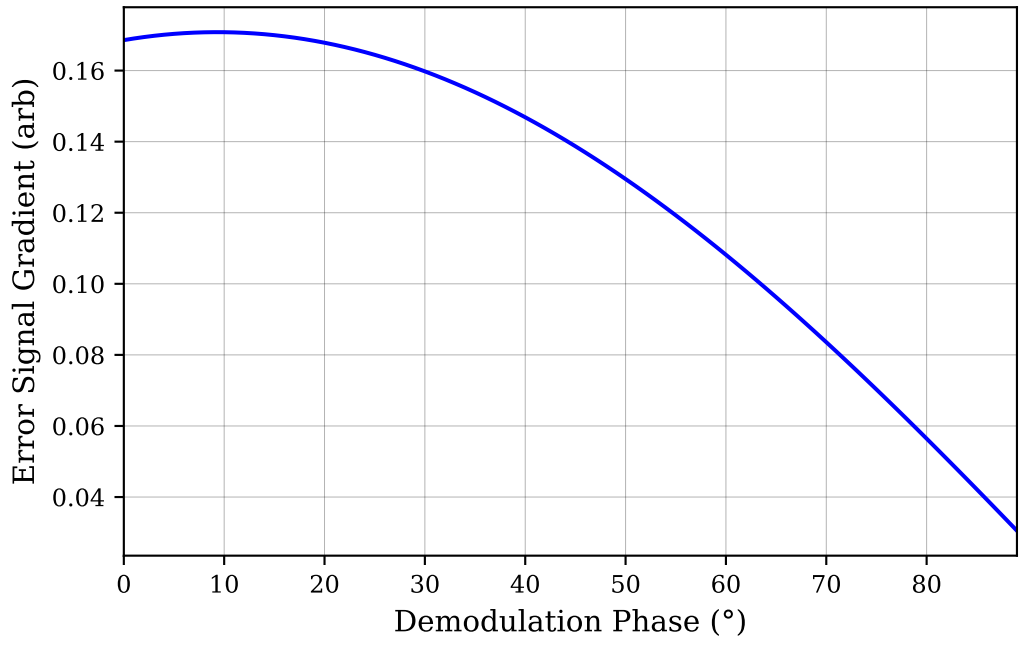
# Calculate the gradient of the error signal.
grad = []
for phase in phases:
    var = out[f'pdh{phase}']
    grad.append((var[0] - var[-1])/2)

# Find the maximum gradient.
grad = np.array(grad)
idxmax = np.argmax(np.abs(grad))
print(f'Maximum error signal gradient occurs at {phases[idxmax]} degrees')

# Plot.
from matplotlib import pyplot as plt
plt.plot(phases,grad)
plt.xlabel('Demodulation Phase (°)')
plt.ylabel('Error Signal Gradient (arb)');
```

```
Removed pdh0
Removed pdh18
Removed pdh36
Removed pdh54
Removed pdh72
Removed pdh90
```

```
Maximum error signal gradient occurs at 9 degrees
```



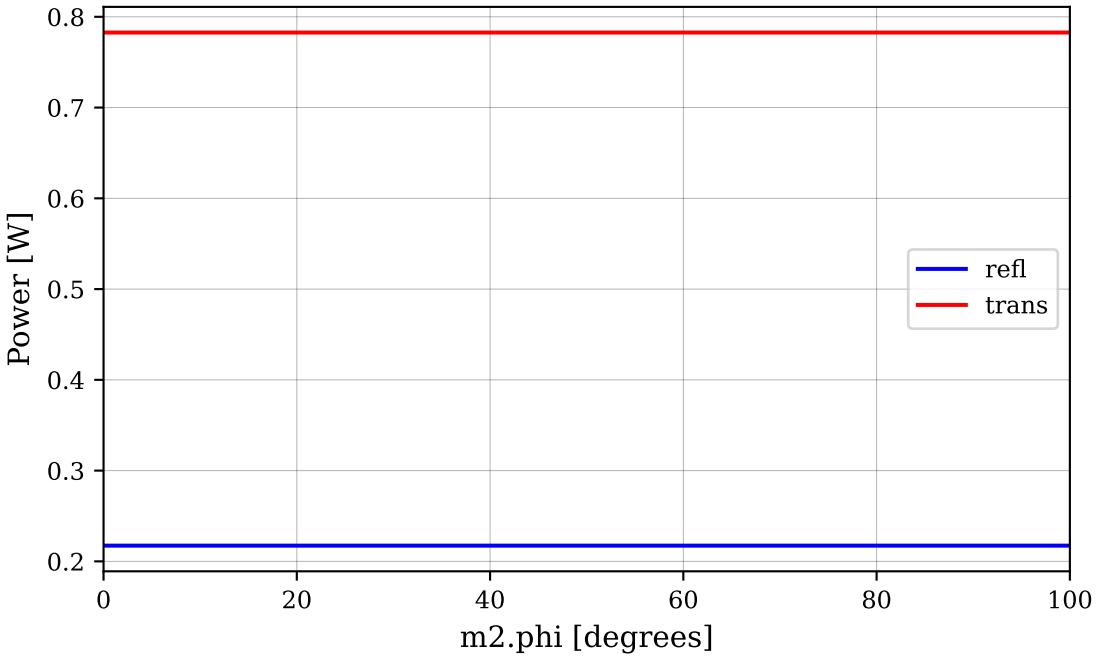
Closing the loop

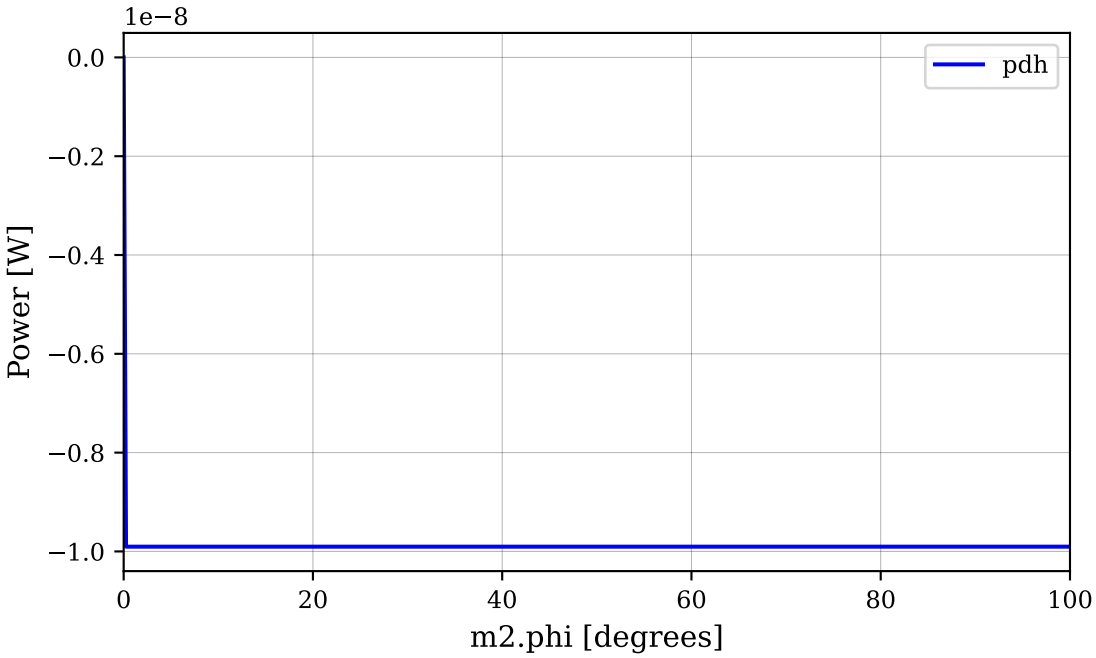
The cavity can then be locked to the operating point using the `lock` (page 190) command. The simulation must be instructed to run the locks before each step, otherwise it will not do this.

Note

The lock command only accepts real inputs and therefore you must specify a demodulation phase for the power detector, so that it outputs real numbers, if you wish to use it in a locking loop.

```
remove_demod_pds(verbose=False)
model.parse("""
power_detector_demod_1 pdh node=m1.p1.o f=eo1.f phase=9
lock lock_length pdh m1.phi -10 1e-8
""")
sol = model.run("xaxis(m2.phi, lin, 0, 100, 400, pre_step=run_locks())")
sol.plot();
```





As you can see, the position of M1 is now controlled to maintain the cavity resonance.

Using complex power demodulation for demodulation phase calculation

The optimal demodulation phase can be computed in multiple ways. As above we can do this manually. Another alternative is to use the power detector with its demodulation phase set to `None`. In this case the detector outputs the complex-value of the optical beat and a local oscillator before a demodulation phase is applied. The real and imaginary parts of this output are the in-phase and quadrature signals of the error signal. By computing the gradient of the real and imaginary part of this output with respect to the cavity length change we can calculate what the optimal demodulation phase is in much fewer steps.

See also

For more mathematical details of this, see Equation 4.13 in *Optical readout* of [18].

Note

Those used to FINESSE 2 will notice that there is no longer a `yaxis` command to specify where the output should be real and imaginary, or magnitude and phase. Detectors in FINESSE 3 are smarter, and return an output in the correct datatype. So detectors measuring DC powers always return real-valued outputs, and those computing complex-values return them appropriately.

Here we build the same cavity model as before but with the complex-valued power demodulator.

```
import finesse
import numpy as np
finesse.init_plotting()

model = finesse.Model()
model.parse(
    """
    laser i1 P=1 f=0
    space s1 portA=i1.p1 portB=eo1.p1
    modulator eo1 15M 0.7 order=3
```

(continues on next page)

(continued from previous page)

```
    space s2 portA=eo1.p2 portB=m1.p1
    mirror m1 R=0.9 T=0.1
    space s3 portA=m1.p2 portB=m2.p1 L=1
    mirror m2 R=0.9 T=0.1

    pd1 pdh_complex m1.p1.o f=eo1.f
    """
)
```

```
<finesse.model.Model at 0x7cf84cdc02d0>
```

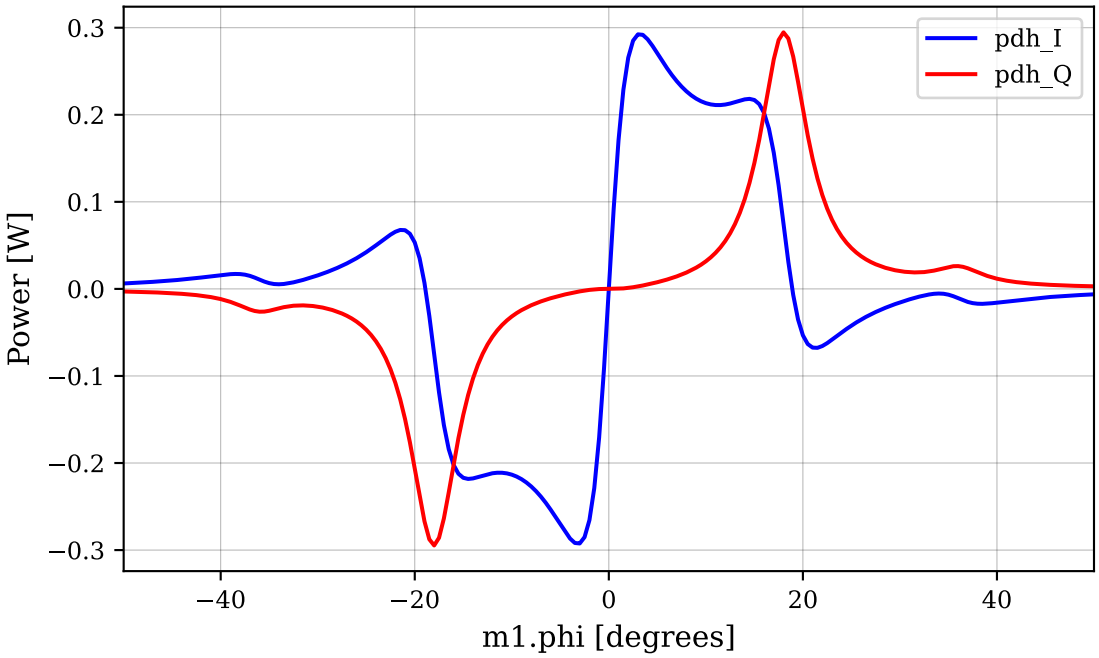
Next we use the `np.gradient` to perform a finite central difference calculation of the gradient. This requires computing two steps either side of our initial state, we do this with a single step `xaxis`. The optimal demodulation phase is then the complex argument of the differentiated value.

```
eps = 1e-9 # finite difference step size for gradient calculation
sol = model.run(f"xaxis(m1.phi, lin, -{eps / 2}, {eps / 2}, 1, relative=True)")
# compute the real and imaginary gradient with respect to the
# cavity length change, then compute the complex argument (angle)
# to get the optimial demodulation phase
opt_demod = np.angle(
    np.gradient(
        sol['pdh_complex'], eps
    ).mean(), # take mean as we need
    deg=True # make sure we compute angle in degrees
)
print(opt_demod)
```

```
9.232551164450188
```

Lastly, we make two real-valued demodulated power detectors for the in-phase and quadrature signals and plot the error signal.

```
model.parse(f"""
pd1 pdh_I m1.p1.o f=eo1.f phase={opt_demod}
pd1 pdh_Q m1.p1.o f=eo1.f phase={opt_demod + 90}
""")
sol = model.run("xaxis(m1.phi, lin, -50, 50, 200)")
sol.plot('pdh_I', 'pdh_Q');
```



5.4 Optomechanics examples

Here you will find a selection of examples for performing various simulation tasks using FINESSE.

5.4.1 Angular radiation pressure

Here we will create a model of a Fabry-Perot cavity with its two mirrors suspended from a simple pendulum. We will then drive some angular motions and see how the dynamics of the mirror are altered due to radiation pressure (page 369) effects. This is commonly referred to in the gravitational wave community as the HARD and SOFT rotational modes. This refers to the fact that the optical torques modify the two suspension modes making one mode harder (higher in resonant frequency) and the other softer (lower in resonant frequency).

More can be read up on regarding angular instabilities shown here in [23], [24], and more on Advanced LIGO angular control design in [25].

First let us import the various features we will need. For this example we will use the Python API rather than KatScript as a demonstration.

```
import cmath
import numpy as np
import finesse
import finesse.components as fc
import finesse.detectors as det
from finesse.analysis.actions import FrequencyResponse, Noxaxis

finesse.init_plotting()
```

Next we build out Fabry-Perot model and attach some simple `Pendulum` (page 193) dynamics. The pendulum has a resonance at 0.6 Hz and a moment of inertia of 0.757 kg·m^2.

```
L = 3994.5 # [m]
I = 0.757  # [kg·m2]
c = 299792458 # [m·s^-1]
f_sus = 0.6 # [Hz]

model = finesse.Model()
model.fsig.f = 1 # set some initial signal frequency
```

(continues on next page)

(continued from previous page)

```
model.modes(maxtem=1) # first order modes for modelling alignment signals

LASER = model.add(fc.Laser("LASER", P=1000))
# Add two mirrors for the cavity and attach a pendulum mechanics
M1 = model.add(fc.Mirror("M1", R=0.986, T=0.014, Rc=1934))
model.add(
    fc.mechanical.Pendulum(
        "M1_sus", model.M1.mech, mass=np.inf, I_yaw=np.inf, I_pitch=I, fpitch=f_sus
    )
)
M2 = model.add(fc.Mirror("M2", R=1, T=0, Rc=2245))
model.add(
    fc.mechanical.Pendulum(
        "M2_sus", model.M2.mech, mass=np.inf, I_yaw=np.inf, I_pitch=I, fpitch=f_sus
    )
)
model.connect(M1.p1, M2.p1, L=L)
model.connect(LASER.p1, M1.p2)
model.add(fc.Cavity('cavARM', M2.p1.o))
model.add(det.PowerDetector('P', M1.p1.o)); # cavity power
```

Next we need to analyse the geometry of our cavity and determine how much of each mirror motion makes up HARD and SOFT, this requires computing the *r* factor [25]

```
# Now we compute the decomposition of HARD and SOFT modes into motions of M1 and M2
g_1 = 1 - float(L / np.abs(M1.Rcx.value)) # input mirror g-factor
g_2 = 1 - float(L / np.abs(M2.Rcx.value)) # end mirror g-factor

r = model.add_parameter(
    'r',
    2 / ((g_1 - g_2) + np.sqrt((g_2 - g_1) ** 2 + 4)),
    description='r term in eq.3.5 in T0900511',
)
```

Above we have added a new parameter to the model using `model.add_parameter` so that later we can change its value and any other element that uses `model.r` will automatically update itself. This also means we can vary `r` with an action such as `Xaxis` (page 209) to sweep through different values easily.

Now we can define a new global degree of freedom in the model that combines the local motions of each cavity mirror. Here we use `r.ref` to make a reference to the new parameter we added.

```
HARD = model.add(fc.DegreeOfFreedom("HARD", M1.dofs.F_pitch, -1, M2.dofs.F_pitch, +r.ref))
SOFT = model.add(fc.DegreeOfFreedom("SOFT", M1.dofs.F_pitch, +r.ref, M2.dofs.F_pitch, +1))

print(HARD.amplitudes)
print(SOFT.amplitudes)
```

```
[-1 <Symbolic='+r' @ 0x7b072ad22eb0>]
[<Symbolic='+r' @ 0x7b072ad231c0> 1]
```

We can see our amplitudes now depend on a symbolic reference to *r*. You can see what the current numerical value is by simply casting this array to a float type which forces an evaluation of the

```
print("model.r = ", model.r.eval())
print("Current HARD amplutudes:", HARD.amplitudes.astype(float))
```

```
model.r =  1.1532419233817137
Current HARD amplutudes: [-1.          1.15324192]
```

It is worth stopping here to note that we are defining the HARD and SOFT mode to drive a pitch torque by using `M1.dofs.F_pitch`, a local Degree of Freedom Finesse component (DOF). A local degree of freedom collects together

the static (DC) and small signal (AC) states of an element. In this case it's a degree of freedom that drives a pitch force (or torque, all forces/torque nodes are prepended with *F_*).

We can see what this local degree of freedom is by outputing its *AC_IN*, *AC_OUT*, *DC* attributes.

```
print(repr(model.M1.dofs.F_pitch.AC_IN))
print(repr(model.M1.dofs.F_pitch.AC_OUT))
print(repr(model.M1.dofs.F_pitch.DC))
```

```
<SignalNode M1.mech.F_pitch @ 0x7b072af55e50>
<SignalNode M1.mech.pitch @ 0x7b072af55d90>
<M1.ybeta=((-HARD.DC)+(SOFT.DC*r)) @ 0x7b072afe1f30>
```

The *AC_IN* defines which signal node will be excited by this DOF, for example *HARD.AC.i*. The *AC_OUT* defines which nodes is used to generate the output of the DOF, *HARD.AC.o*. The *DC* attribute sets the static state of the DOF, in this case it is *M1.ybeta*, the static pitch misalignment of the the mirror *M1*. By setting the *HARD.DC* value, you can set what static DOF value is, relative to any *DC* parameter there is on the component itself.

Using the Python API it is possible to define your own `finesse.components.general.LocalDegreeOfFreedom` and mix and match any DC and AC inputs and outputs as you need. Commonly used ones are typically defined in an elements *.dofs* attribute.

Now to continue with the original example, we will set the power such that we get around 600kW in the cavity and perform a multiple-input-multiple-output (MIMO) frequency response analysis exciting the HARD and SOFT mode inputs to their outputs.

```
LASER.P = 1410 * 3/2 # get to roughly 600kW
freq_resp = FrequencyResponse(
    np.geomspace(0.1, 10, 2000),
    [HARD.AC.i, SOFT.AC.i],
    [HARD.AC.o, SOFT.AC.o],
)
sol = model.run(freq_resp)
# Also compute a single data point of the detectors to compute the power
# circulating in the cavity
out = model.run(Noxaxis())
print(sol.out.shape)
```

```
(2000, 2, 2)
```

The output of the frequqency response analysis is a (2000, 2, 2) array, as we have 2000 frequencies and a 2x2 MIMO matrix at each.

We can now plot the results and compare this against equation 2 in [24] which states how much the HARD and SOFT modes are altered by radiation pressure.

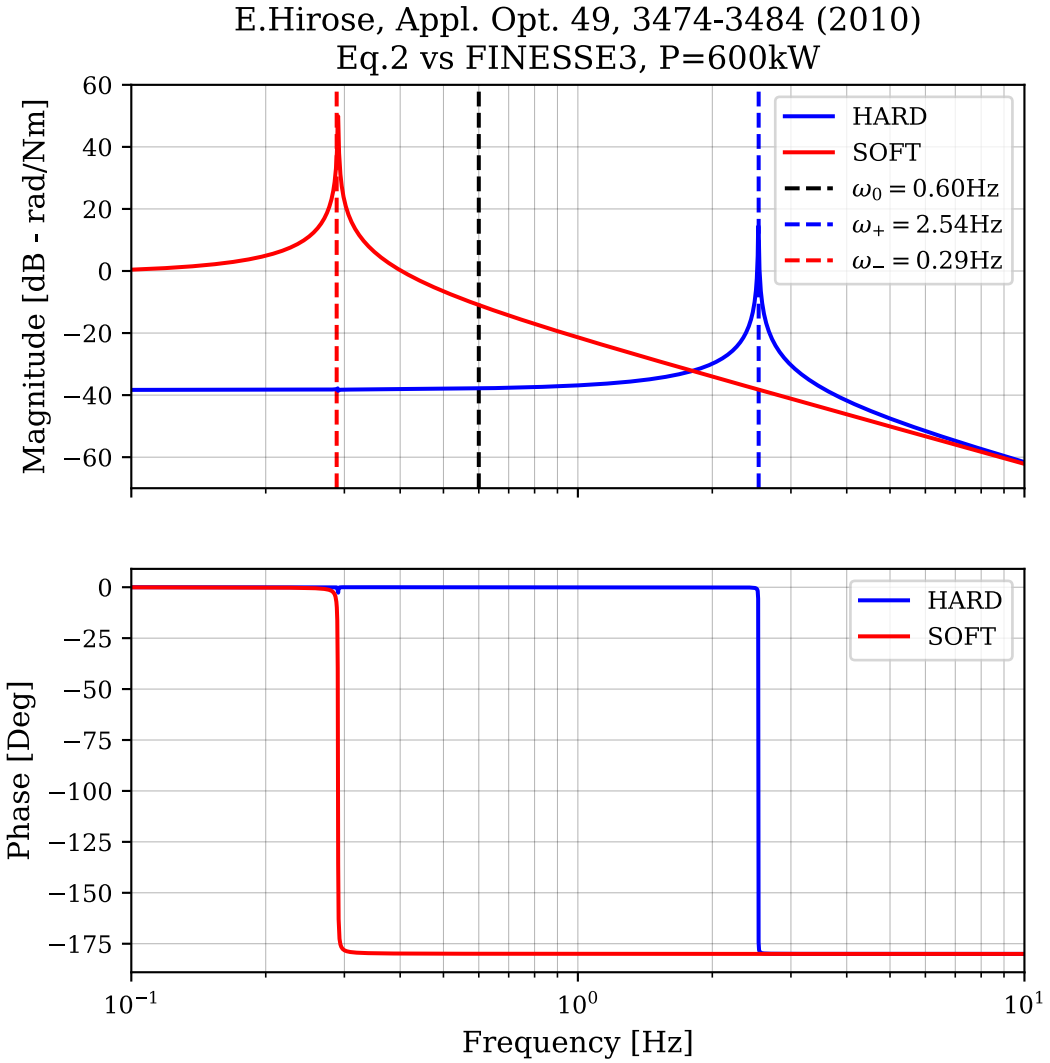
```
omega_0 = 2 * np.pi * f_sus
P = out['P']
# Eq 2 from https://opg.optica.org/ao/fulltext.cfm?uri=ao-49-18-3474
omega_plus = np.sqrt(
    omega_0**2 + P * L / (I * c) * (-(g_1 + g_2) + cmath.sqrt(4+(g_1 - g_2)**2))/ (1 - g_1*g_
↪2)
)
omega_minus = np.sqrt(
    omega_0**2 + P * L / (I * c) * (-(g_1 + g_2) - cmath.sqrt(4+(g_1 - g_2)**2))/ (1 - g_1*g_
↪2)
)

axs = finesse.plotting.bode(sol.f, sol['HARD.AC.o', 'HARD.AC.i'], label='HARD')
axs = finesse.plotting.bode(sol.f, sol['SOFT.AC.o', 'SOFT.AC.i'], label='SOFT', axs=axs)
axs[0].vlines(omega_0 / 2 / np.pi, -70, 60, ls='--', label=f'$\\omega_{{0}} = {omega_0.real /
↪ 2 / np.pi:0.2f}$Hz', color='k', zorder=-100)
```

(continues on next page)

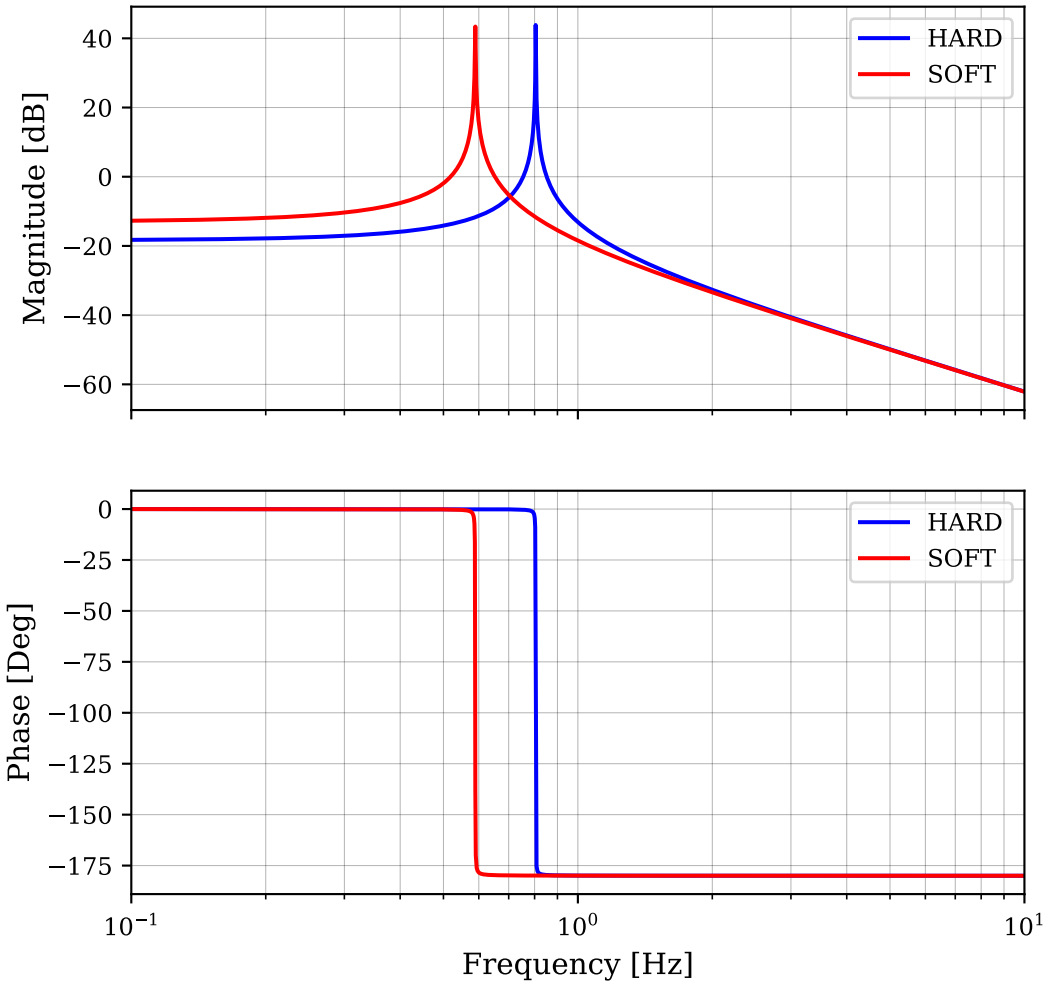
(continued from previous page)

```
axs[0].vlines(omega_plus.real / 2 / np.pi, -70, 60, ls='--', label=f'$\\omega_{{+}} = {omega_
↪plus.real / 2 / np.pi:0.2f}$Hz', zorder=-100)
axs[0].vlines(omega_minus.real / 2 / np.pi, -70, 60, color='red', ls='--', label=f'$\\omega_{
↪{-}} = {omega_minus.real / 2 / np.pi:0.2f}$Hz', zorder=-100)
axs[0].margins(0)
axs[0].set_title(f"E.Hirose, Appl. Opt. 49, 3474-3484 (2010)\nEq.2 vs FINESSE3, P={P / 1e3:0.
↪0f}kW")
axs[0].legend()
axs[0].set_ylabel("Magnitude [dB - rad/Nm]");
```



If we reduce the power we can see the opto-mechanical effects are reduced and the HARD and SOFT resonances begin to return to the nominal suspension resonance frequency.

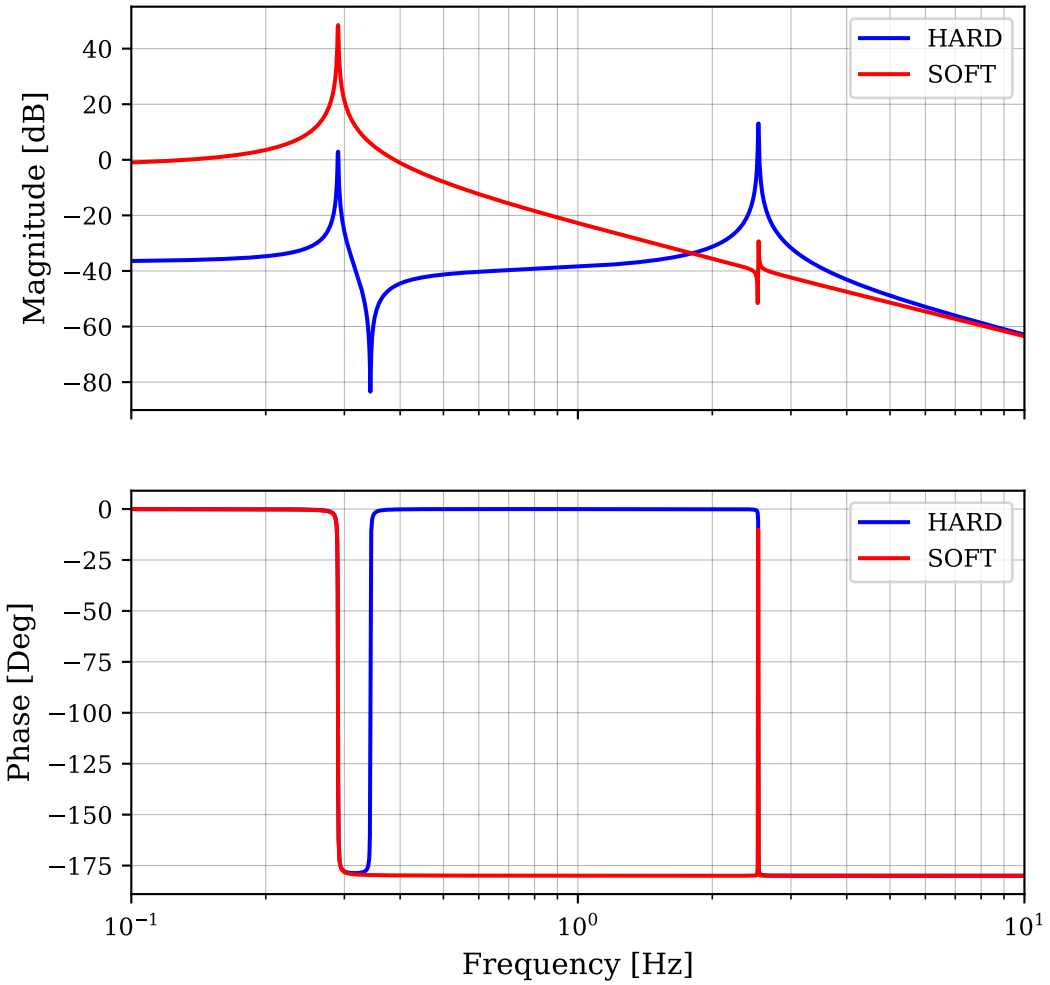
```
LASER.P = 100
sol = model.run(freq_resp)
axs = finesse.plotting.bode(sol.f, sol['HARD.AC.o', 'HARD.AC.i'], label='HARD')
finesse.plotting.bode(sol.f, sol['SOFT.AC.o', 'SOFT.AC.i'], label='SOFT', axs=axs);
```



As a last point to make, due to the asymmetric cavity mirror curvatures the actuation must be properly diagonalised. Here we can see that if we simply set the *M1* and *M2* actuation amplitude to a unit value, that when we drive HARD or SOFT we excite the other mode as well.

```
LASER.P = 1410 * 3/2 # get to roughly 600kW
model.r = 1
# Or alternatively you can replace the amplitudes directly with something like
# HARD.amplitudes[:] = [-1, +1]
# SOFT.amplitudes[:] = [+1, +1]
sol = model.run(freq_resp)
axs = finesse.plotting.bode(sol.f, sol['HARD.AC.o', 'HARD.AC.i'], label='HARD')
finesse.plotting.bode(sol.f, sol['SOFT.AC.o', 'SOFT.AC.i'], label='SOFT', axs=axs)
```

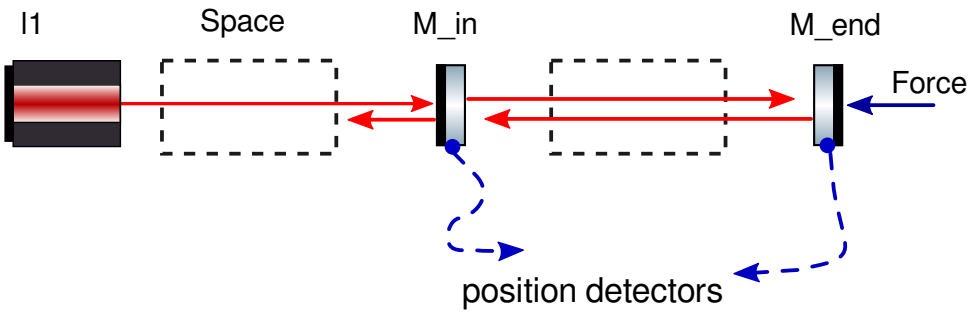
```
array([<Axes: ylabel='Magnitude [dB]'>,
       <Axes: xlabel='Frequency [Hz]', ylabel='Phase [Deg]'>],
      dtype=object)
```



5.4.2 Optical springs

This example shows a simple example for a FINESSE simulation of an optical spring.

The optical configuration



The optical layout shows a laser beam being injected into a Fabry-Perot cavity. What we want to model here is how both of the mirrors move if we apply a force to only one of them. In this example, a force is applied to the end cavity mirror. Note that the cavity must be tuned slightly away from resonance for the optical spring effect to occur.

Note that the mention of position detectors in the figure is for illustrative purposes only. This example uses the `freqresp` (page 221) action to directly calculate the transfer functions between the force signal node `M_end.mech.F_z` and the position signal nodes `M_end.mech.z` and `M_in.mech.F_z`. See Frequency Response (page 219) for more information.

The FINESSE model

```
import numpy as np

import finesse
from finesse.analysis.actions import FrequencyResponse
from finesse.plotting import bode

finesse.configure(plotting=True)

kat = finesse.Model()
kat.parse(
    """
    l l1 P=3
    s s1 l1.p1 M_in.p1 L=1

    # The cavity
    m M_in R=0.9937 L=0
    s scav M_in.p2 M_end.p1 L=1
    m M_end R=1 L=0 phi=-0.048  # Roughly 40kHz detuning

    # Suspend mirrors on pendula
    pendulum sus1 M_in.mech mass=0.25 fz=1 Qz=1000
    pendulum sus2 M_end.mech mass=0.25 fz=1 Qz=1000

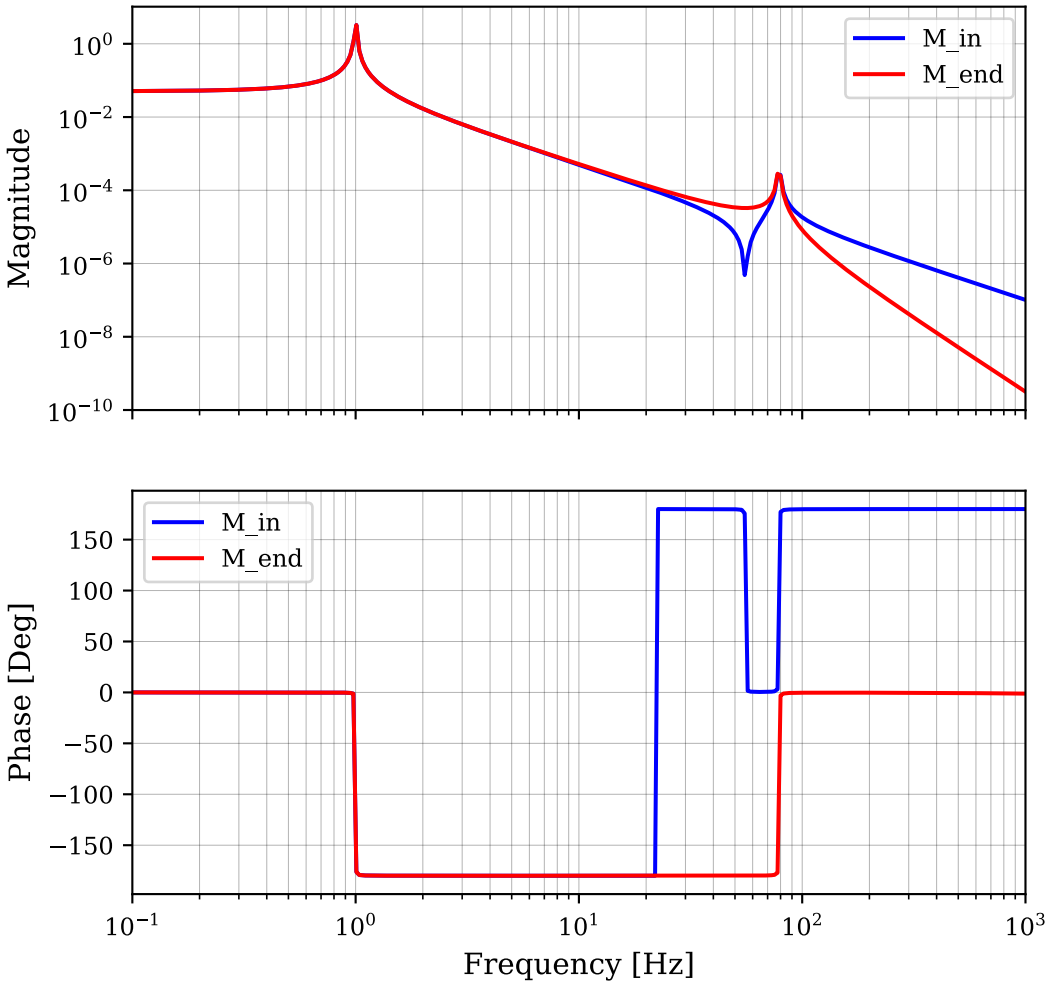
    # Activate signal simulation
    fsig(1)
    """
);
```

Note here that we use the `fsig` (page 196) to activate a signal simulation. See Signals (page 120) for more information.

Output plots

```
out = kat.run(
    FrequencyResponse(
        np.geomspace(0.1, 1e3, 300), ["M_end.mech.F_z"], ["M_end.mech.z", "M_in.mech.z"]
    )
)

axs = bode(out.f, out["M_end.mech.z", "M_end.mech.F_z"], db=False, label="M_in")
bode(out.f, out["M_in.mech.z", "M_end.mech.F_z"], axs=axs, db=False, label="M_end");
```



The plot shows the magnitude and phase of both mirrors' motion. An unexpected result, you may think, is that by applying a force to only one mirror, the other one also moves, despite the absence of any mechanical link between them. However, the optical beam now couples the motions of the two mirrors. This behaviour is called an optical spring: the light field circulating in the cavity essentially creates exactly the same effect as if a physical spring were connecting the mirrors. Concentrating on the magnitude of the motions first, we see two resonances: the first being the mechanical resonance of the mirror suspensions, the second being the resonance of the optical spring. If we consider the phase of the motions we can see that at low frequency the mirrors move exactly in-phase, thus the whole cavity moves together; this is called optical rigidity and can approach the stiffness of diamond! Near the optical spring resonance the behaviour changes and at high frequencies the mirrors begin to move out of phase with one another.

5.4.3 Radiation pressure

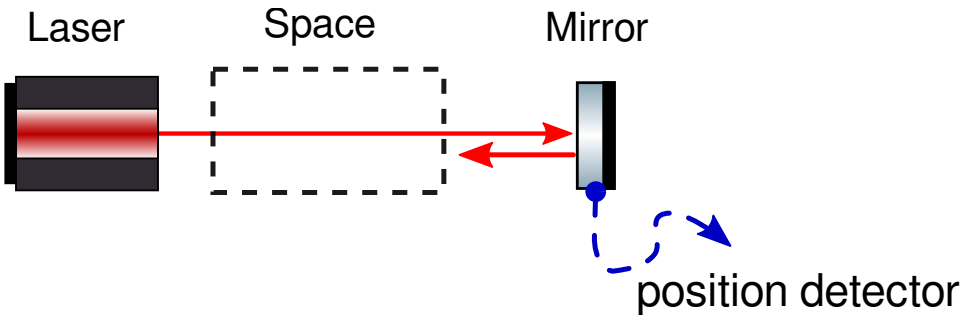
This example shows a simple example for a FINESSE simulation of radiation pressure (page 369) causing a mirror to move.

The optical configuration

The optical layout is very simple, with a laser beam being reflected by a single mirror. The interesting aspects of this setup are hidden in the details: The laser beam includes an amplitude modulation (page 117) 'signal' whose frequency we can tune; secondly, the mirror is suspended on a `Pendulum` (page 193).

The position detector in the figure is for illustrative purposes only. This example uses the `freqresp` (page 221) action to directly calculate the transfer function between the laser amplitude node `l1.amp.i` and the position node `m1.mech.z`.

See for example Modelling transfer functions (page 267) and Calculating general transfer functions (page 272) for discussions on how transfer functions are used in FINESSE.



The FINESSE model

```
import numpy as np

import finesse
from finesse.analysis.actions import FrequencyResponse
from finesse.plotting import bode

finesse.init_plotting()

model = finesse.Model()
model.parse(
    """
    # Optical setup: laser, space and mirror:
    l l1 P=1
    s s1 l1.p1 m1.p1
    m m1 R=1 T=0

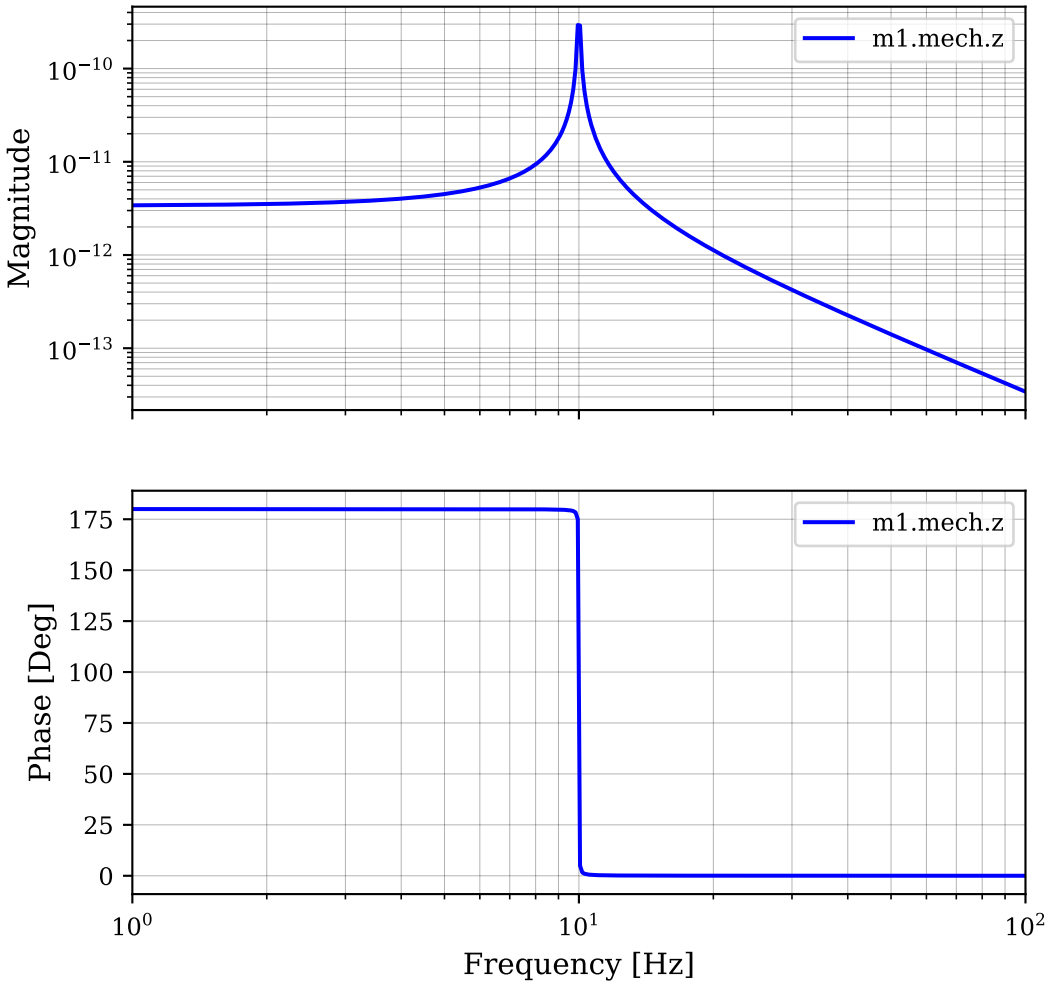
    # Define a pendulum for our mirror, with a z-motion resonance at 10Hz
    # and Q factor of 1000
    pendulum sus1 m1.mech mass=1 fz=10 Qz=1000

    # Set a signal frequency to activate the signal simulation
    # (needed for radiation pressure effects to be observed)
    fsig(1)
    """
);
```

Note here that we use the `fsig` (page 196) to activate a signal simulation. See Signals (page 120) for more information.

Output plots

```
sol = model.run(FrequencyResponse(np.geomspace(1, 100, 400), "l1.amp", "m1.mech.z"))
bode(sol.f, sol["m1.mech.z", "l1.amp"], db=False, label="m1.mech.z");
```



Upon reflection by the mirror the photons reverse their momentum. This momentum transfer gives rise to a force on the mirror, the so-called 'radiation-pressure force'. FINESSE assumes a steady state of the optical system, which in this case means that we assume the static radiation-pressure force to be compensated by another static force, for example via active control or through gravity. The amplitude modulation signal on the laser light, however, creates a modulation of the force which we can model and measure in a steady state. The mechanical transfer function of the mirror determines how the longitudinal force (as a function of frequency) translates into motion. The `freqresp` (page 221) calculates this transfer function, showing a single pole at 10 Hz with a Q factor of 1000.

See also

For further reading, [26] gives a brief but fairly detailed introduction to radiation pressure effects in the context of gravitational wave detectors.

5.5 Quantum mechanics examples

Here you will find a selection of examples for performing various simulation tasks using FINESSE.

5.5.1 Frequency dependant squeezing

Gravitational wave detectors use squeezed light (a special quantum state of light) to improve their sensitivity. Using just squeezed light however can only reduce the radiation pressure (page 369) noise or the shot-noise components of the quantum noise, but not both simultaneously. Using a "filter cavity" [27] rotates the squeezed state in just the right way such that it suppress radiation pressure noise at low frequency and then rotate it to suppress shot-noise at high frequency at the correct points. These additional cavities are being installed into current detectors and will be crucial to realising future generations of detectors—so we need to be able to model them.

In this example we will describe the multiple ways in which you can simulate a filter cavity. This can be done in a rigorous way or in a more approximate manner. Depending on what you want to achieve either option can be used.

To begin with we will need a model of a detector. For this example we will use a model that is similar to Cosmic Explorer, a proposed 40km long detector.

```
import numpy as np
import finesse
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse("""
variable ITM_T 0.014
variable L_ARM 40k
variable FC_detune 7.3 # Hz

l l1 P=165
link(l1, PRM)
# ----------------------------------------------------------------------------
# Power recycling cavity
# ----------------------------------------------------------------------------
m PRM T=0.03 L=40u phi=0
s PRC PRM.p2 BS.p1 L=53.0
bs BS R=0.5 T=0.5
# ----------------------------------------------------------------------------
# X-arm
# ----------------------------------------------------------------------------
s lx BS.p3 ITMX.p1 L=0
m ITMX T=ITM_T L=0u phi=0
s LX ITMX.p2 ETMX.p1 L=L_ARM
m ETMX T=5u L=40u phi=0
# ----------------------------------------------------------------------------
# Y-arm
# ----------------------------------------------------------------------------
s ly BS.p2 ITMY.p1 L=0
m ITMY T=ITM_T L=0u phi=90
s LY ITMY.p2 ETMY.p1 L=L_ARM
m ETMY T=5u L=40u phi=90
# Signal recycling cavity
s SRC BS.p4 SRM.p1 L=20
m SRM T=0.02 L=0 phi=0
# ----------------------------------------------------------------------------
# Output optics
# ----------------------------------------------------------------------------
dbs OFI
link(SRM.p2, OFI.p1)
readout_dc AS OFI.p3.o
# ----------------------------------------------------------------------------
# suspensions
# ----------------------------------------------------------------------------
free_mass ITMX_sus ITMX mass=320
free_mass ETMX_sus ETMX mass=320
free_mass ITMY_sus ITMY mass=320
free_mass ETMY_sus ETMY mass=320
# ----------------------------------------------------------------------------
# Degrees of Freedom
# ----------------------------------------------------------------------------
dof STRAIN LX.dofs.h +1  LY.dofs.h -1

# signal generator
sgen sig STRAIN
```

(continues on next page)

(continued from previous page)

```
qnoised NSR AS.p1.i nsr=True

fsig(1)
""");
```

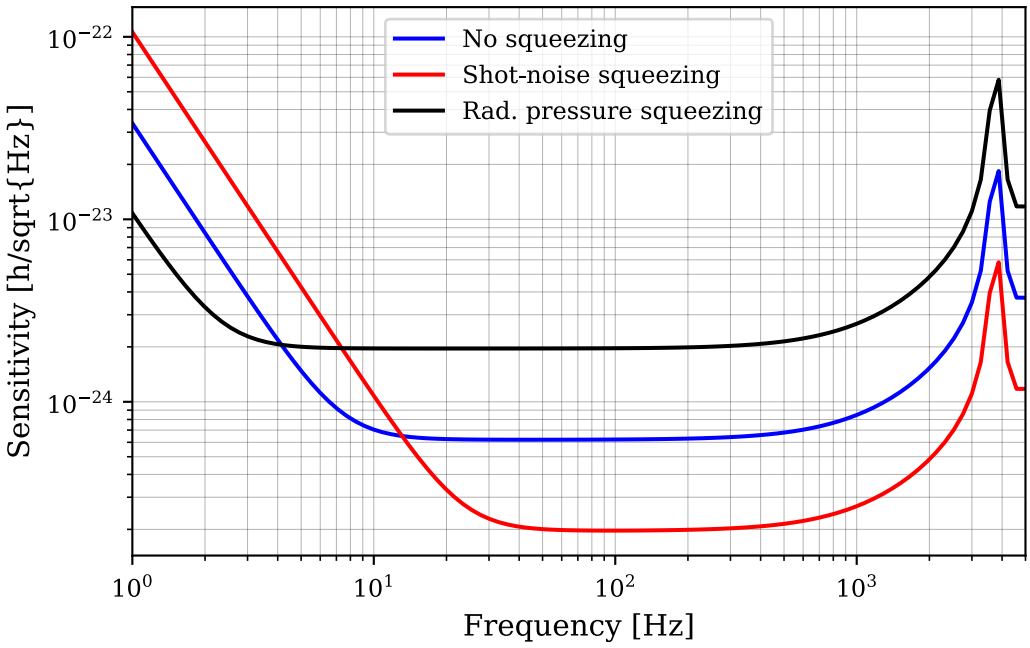
First we make a model which just has a squeezer

```
model = base.deepcopy()
model.parse("""
sq sqz db=10
link(sqz, OFI.p2)
""");
```

We can plot the quantum noise limited sensitivity using the code below. In this code we switch the squeezer on and off and also rotate the squeezing angle to see how the quantum noise shape changes.

```
with model.temporary_parameters():
    # No squeezing
    model.sqz.db = 0
    model.sqz.angle = 0
    sol_no_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")
    # Squeezing 90 degrees
    model.sqz.db = 10
    model.sqz.angle = 90
    sol_shot_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")
    # Squeezing 0 degrees
    model.sqz.db = 10
    model.sqz.angle = 0
    sol_rp_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")

plt.loglog(sol_no_sqz.x1, abs(sol_no_sqz['NSR']), label='No squeezing')
plt.loglog(sol_shot_sqz.x1, abs(sol_shot_sqz['NSR']), label='Shot-noise squeezing')
plt.loglog(sol_rp_sqz.x1, abs(sol_rp_sqz['NSR']), label='Rad. pressure squeezing')
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("Sensitivity [h/sqrt{Hz}]");
```



We can see that by changing the squeezing angle we can improve the noise at high or low frequencies, but not both simultaneously.

The quick method using symbolics

A filter cavity is essentially just rotating the squeezing angle in the correct manner. Therefore we can approximate it by just using a symbolic equation and applying it to the squeezer elements *angle* parameter. Ignoring losses the equation we need is

$$\phi = \phi_0 + \arctan\left(\frac{\Omega^2 - \Delta^2}{\Omega^2 + \Delta^2}\right).$$

Here ϕ_0 is an offset to arctan method to ensure the correct DC rotation is used , $\Omega = 2\pi f_{\text{sig}}$, and Δ is the detuning frequency which specifies at what frequency the squeezed state rotates. This should be around the corner frequency between the radiation pressure and shot noise contributions.

Note

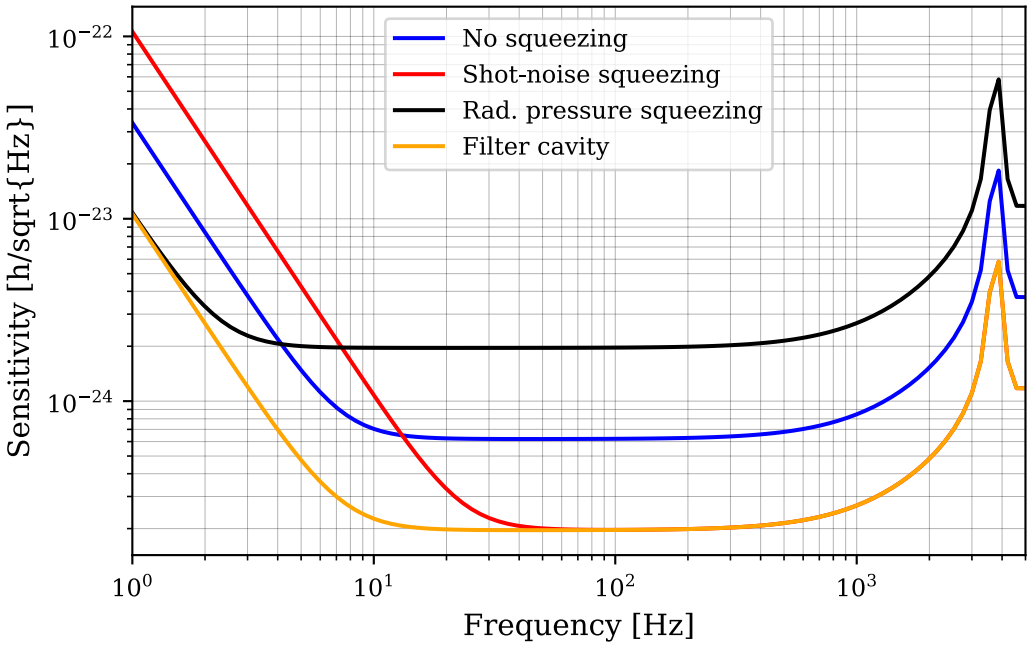
This method is particularly useful if you are not interested in modelling the specific technical details of the filter cavity, such as the filter cavity losses, or mode matching from the cavity into the detector.

We can use normal numpy math function and apply them to the FINESSE symbolic variable references for the signal frequency and detuning frequency. Once that is done we just simply run the model.

```
with model.temporary_parameters():
    model.FC_detune.value = 7.2
    model.sqz.db = 10
    # Give the variable references shorter names
    fsig = model.fsig.f.ref
    delta_f = model.FC_detune.ref
    # set the squeezing angle to be a symbolic function
    model.sqz.angle = (
        45 + 180/np.pi * np.arctan((fsig**2-delta_f**2)/(fsig**2 + delta_f**2))
    )
    print(model.sqz.angle) # print to see how it turned out
    sol_fc_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")

plt.loglog(sol_no_sqz.x1, abs(sol_no_sqz['NSR']), label='No squeezing')
plt.loglog(sol_shot_sqz.x1, abs(sol_shot_sqz['NSR']), label='Shot-noise squeezing')
plt.loglog(sol_rp_sqz.x1, abs(sol_rp_sqz['NSR']), label='Rad. pressure squeezing')
plt.loglog(sol_fc_sqz.x1, abs(sol_fc_sqz['NSR']), label='Filter cavity')
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("Sensitivity [h/sqrt{Hz}]");
```

```
(45+(57.29577951308232*arctan((((fsig.f)**(2)-(FC_detune)**(2))/((fsig.f)**(2)+(FC_
↪detune)**(2))))))) degrees
```

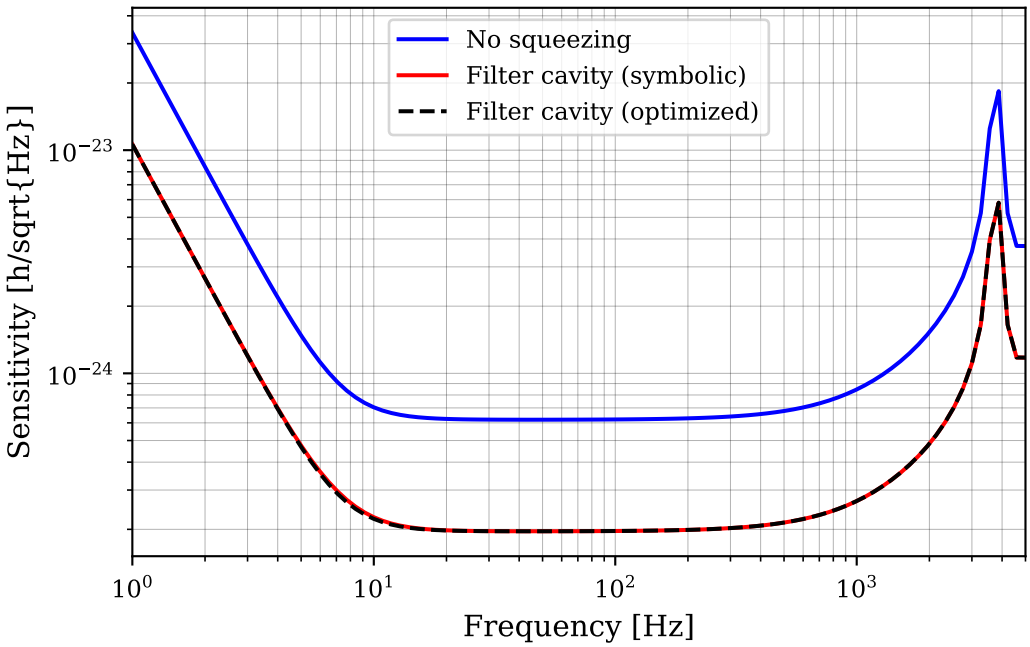


Some experimentation will be needing in choosing ϕ_0 and Δ correctly and once that is done we should see we reduce the noise across the whole frequency range.

An alternative method is to use a minimization routine at each signal frequency and optimise the noise with respect to the squeezing angle. We can easily add this using the *minimize* action on the *xaxis pre_step* event.

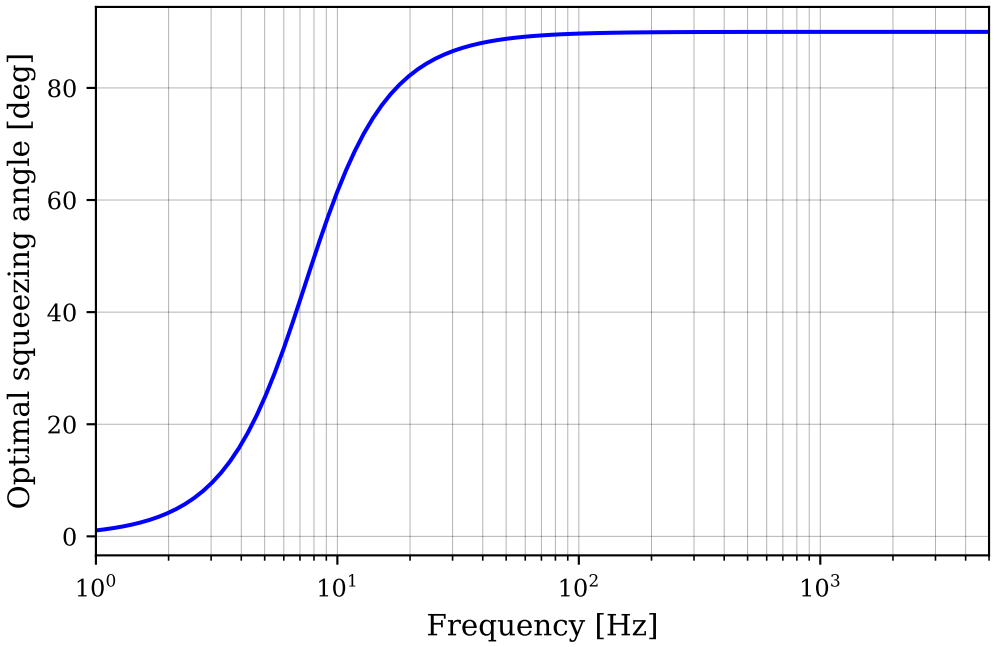
```
# Run an optimiser to find ideal rotation
sol_opt_sqz = model.run("xaxis(fsig, log, 1, 5k, 100, pre_step=minimize(NSR, sqz.angle))")

plt.loglog(sol_no_sqz.x1, abs(sol_no_sqz['NSR']), label='No squeezing')
plt.loglog(sol_fc_sqz.x1, abs(sol_fc_sqz['NSR']), label='Filter cavity (symbolic)')
plt.loglog(sol_opt_sqz.x1, abs(sol_opt_sqz['NSR']), label='Filter cavity (optimized)', ls='--
↪')
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("Sensitivity [h/sqrt{Hz}]");
```



As can be seen the symbolic and optimised version give the same improvements. The optimal squeezing angle can be extracted from the minimizsation solutions using

```
# Extract every optimisisation solution (`x`) from all the pre-step minimize actions
opt_angle = sol_opt_sqz["pre_step", "minimize"].x
plt.semilogx(sol_opt_sqz.x1, opt_angle)
plt.xlabel("Frequency [Hz]")
plt.ylabel("Optimal squeezing angle [deg]");
```



The optimization method can be used to determine the ideal rotation. Consider a case where we have a detuned SRC:

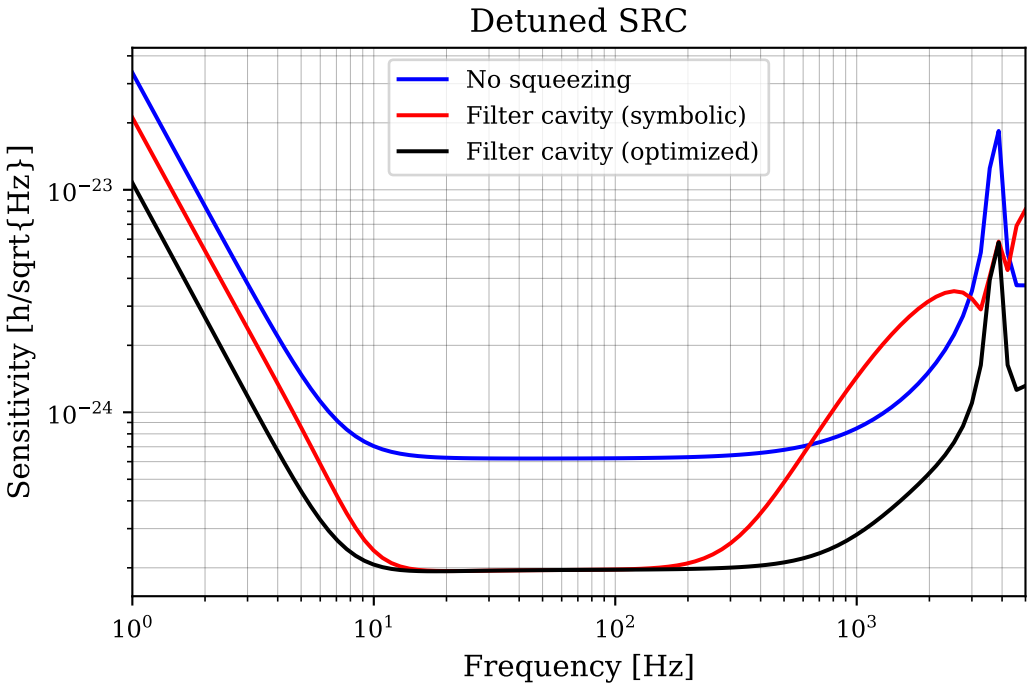
```
with model.temporary_parameters():
    model.FC_detune.value = 7.2
    model.SRM.phi += 0.1 # Detune slightly
    model.sqz.db = 10
    # Give the variable references shorter names
    fsig = model.fsig.f.ref
    delta_f = model.FC_detune.ref
    # set the squeezing angle to be a symbolic function
    model.sqz.angle = (
        45 + 180/np.pi * np.arctan((fsig**2-delta_f**2)/(fsig**2 + delta_f**2))
    )
    sol_fc_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")
    # Run an optimiser to find ideal rotation
    model.sqz.angle = 0
    sol_opt_sqz = model.run("xaxis(fsig, log, 1, 5k, 100, pre_step=minimize(NSR, sqz.angle))
↪")

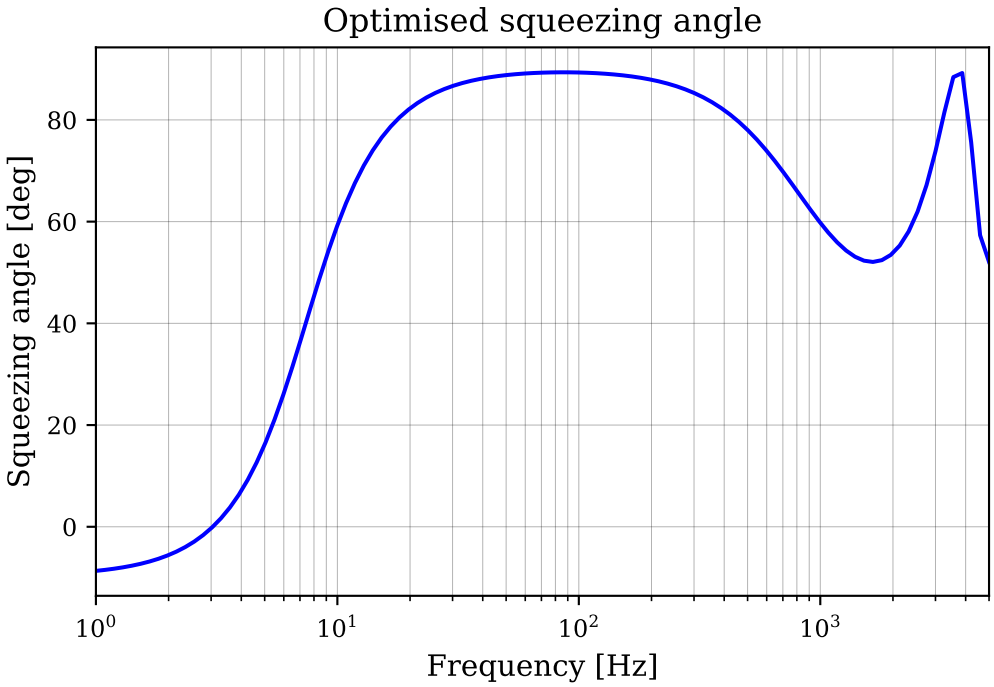
    plt.loglog(sol_no_sqz.x1, abs(sol_no_sqz['NSR']), label='No squeezing')
    plt.loglog(sol_fc_sqz.x1, abs(sol_fc_sqz['NSR']), label='Filter cavity (symbolic)')
    plt.loglog(sol_opt_sqz.x1, abs(sol_opt_sqz['NSR']), label='Filter cavity (optimized)')
    plt.legend()
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Sensitivity [h/sqrt{Hz}]");
    plt.title("Detuned SRC");
    plt.figure()
    # Extract every optimisisation solution (`x`) from all the pre-step minimize actions
    opt_angle = sol_opt_sqz["pre_step", "minimize"].x
    plt.semilogx(sol_opt_sqz.x1, opt_angle)
```

(continues on next page)

(continued from previous page)

```
    plt.xlabel("Frequency [Hz]")
    plt.ylabel("Squeezing angle [deg]")
    plt.title("Optimised squeezing angle")
```





We can see that the optimised version still maintains a perfect reduction in quantum noise. The symbolic case shows how much a filter cavity would be affected by such a detuning. In such a case a single filter cavity cannot correct this squeezing misrotation.

A more realistic model

In some models we are also interested in the details of the filter cavity itself and the propagation between it and the rest of the interferometer. This requires actually specifying the filter cavity in KatScript and optimising its parameters. Here we add a squeezer, an isolator, followed by a linear cavity to a new model building on our *base* model

```
model = base.deepcopy() # make a copy so we can add extra features
model.parse("""
sq sqz db=10
dbs FI
m FC1 L=0 T=0.0015
s sFC FC1.p2 FC2.p1 L=4e3
m FC2 L=0 T=0 phi=-0.022

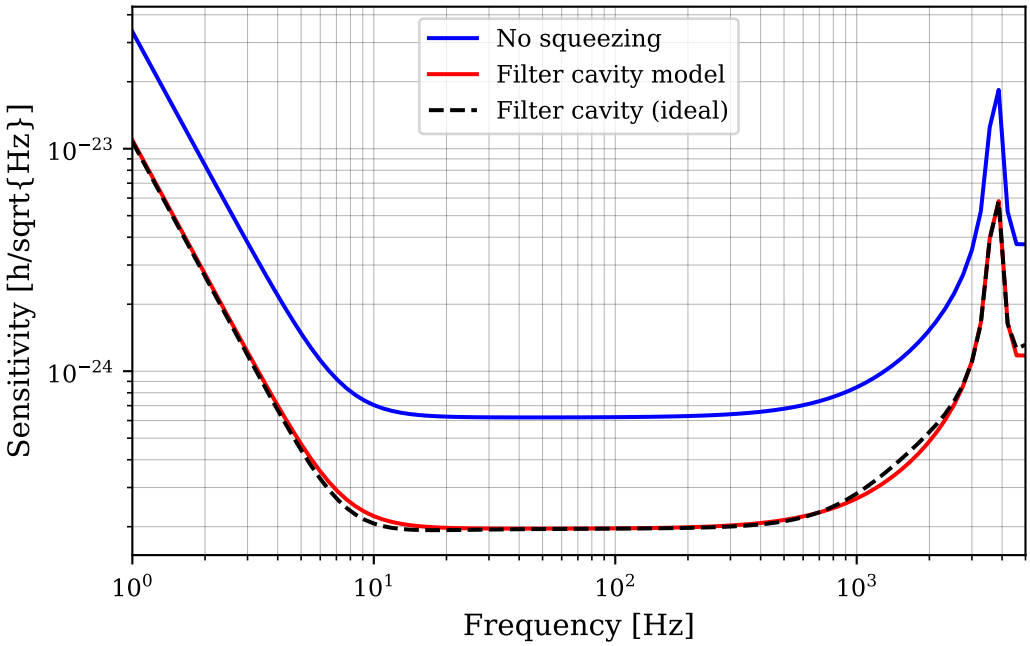
link(sqz, FI.p1)
link(FI.p3, FC1)
link(FI.p4, OFI.p2)
""");
```

Some experimentation and calculations are needed to determine the filter cavity finesse, detuning, and length. All these parameters combine to give the correct frequency dependent rotation to the squeezed light. Once these are found we can run a quick optimisation at the corner frequency a few times to fine tune the aforementioned parameters. More detailed optimisation could also be performed here if necessary over the full frequency range. Below we also compare to the previous results using *minimize* in the *pre_step* event.

```
model.sqz.db = 10
model.sqz.angle = 90
model.FC1.T = 0.0015
model.FC1.R = 1-model.FC1.T.ref # Make sure we update the R correctly when we change T
model.FC2.phi = -0.022
# Optimise the NSR at the corner frequency
model.fsig.f = 7.2
# Run the optimisation a few times to reach a better level
for _ in range(3):
    model.run("minimize(NSR, [FC1.T, FC2.phi, sFC.L])")
# finally compute the NSR
sol_fc2_sqz = model.run("xaxis(fsig, log, 1, 5k, 100)")

plt.loglog(sol_no_sqz.x1, abs(sol_no_sqz['NSR']), label='No squeezing')
plt.loglog(sol_fc2_sqz.x1, abs(sol_fc2_sqz['NSR']), label='Filter cavity model')
plt.loglog(sol_opt_sqz.x1, abs(sol_opt_sqz['NSR']), label='Filter cavity (ideal)', ls='--')
plt.legend()
plt.xlabel("Frequency [Hz]")
plt.ylabel("Sensitivity [h/sqrt{Hz}]");
print("FC1.T =", model.FC1.T)
print("FC2.phi =", model.FC2.phi)
print("FC L =", model.sFC.L)
```

```
FC1.T = 0.00158395913601199
FC2.phi = -0.02328241883258643 degrees
FC L = 3723.892882617261 m
```



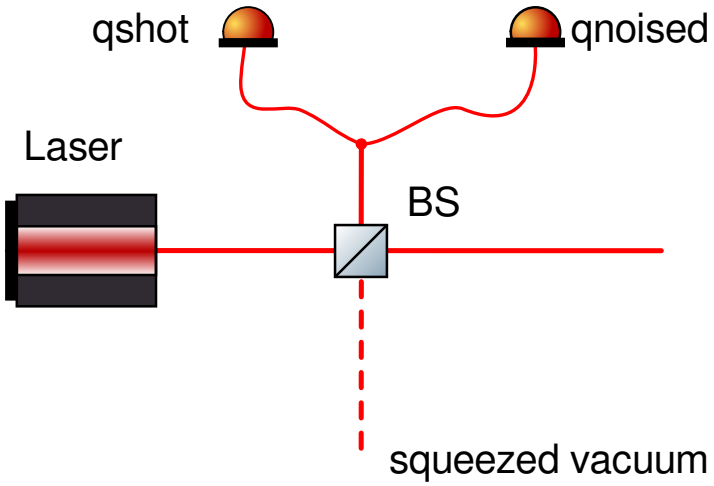
As we can see from the final plot and parameter optimisation results, a more detailed model of a filter cavity can be created and added to a model. This is nearly reaching the ideal filter cavity response, but a more careful selection of parameters is needed.

Including geometric effects would require specifying curvatures of the filter cavity mirrors as well as a mode matching telescope to the rest of the interferometer. If these details are not of interest then we would recommend using the symbolic filter cavity method, as this reduces the computational complexity and setup time.

5.5.2 Squeezing and homodyne detection

This example shows a simple example of modelling squeezed light with FINESSE.

The optical configuration



The optical layout shows a laser and a squeezed light source whose beams are superimposed on a beam splitter. Two quantum noise detectors are used for detection, the `QuantumShotNoiseDetector` (page 182) detector measuring only the shot noise (ignoring the squeezing) and the `QuantumNoiseDetector` (page 181) detector measuring the correct quantum noise. We mix a 10dB squeezed source and a 1W laser at the beamsplitter and measure the result at the photodiode, varying the phase of the laser field.

To mitigate the need for a balanced homodyne, which would require the subtraction of two photocurrents, we can use a highly reflective beamsplitter with a very powerful local oscillator.

The FINESSE model

```
import matplotlib.pyplot as plt
import numpy as np

import finesse
finesse.configure(plotting=True)

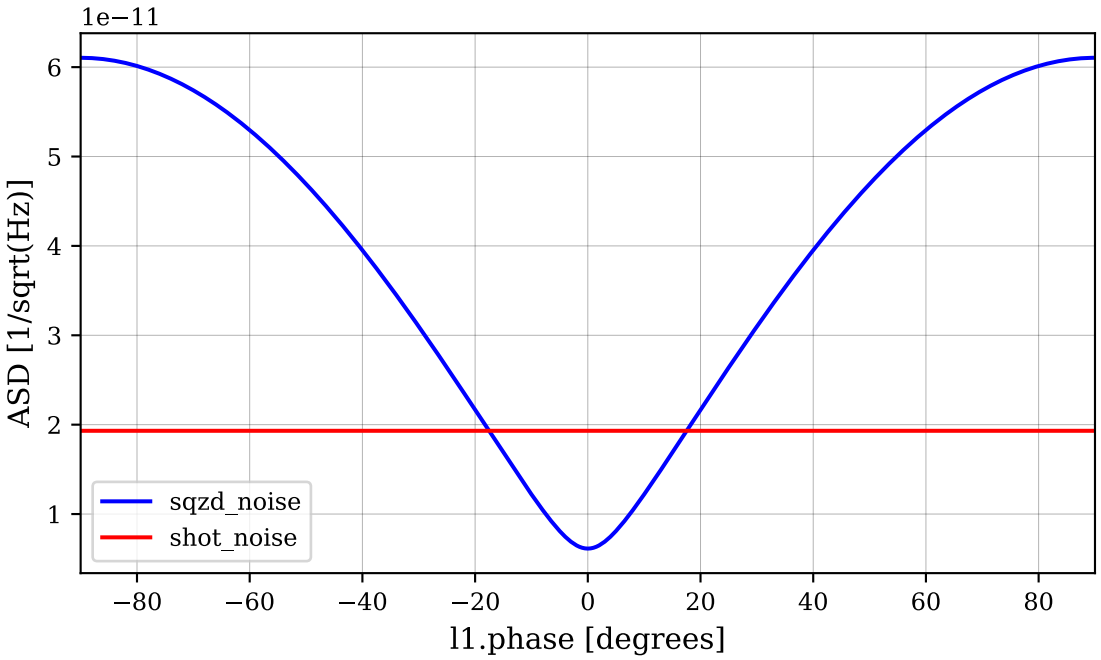
kat = finesse.Model()
kat.parse(
"""
    l l1 P=1-bs1.T
    s s1 l1.p1 bs1.p4 L=1
    sq sq1 db=10
    s s2 sq1.p1 bs1.p1
    bs bs1 R=0.999 L=0 alpha=45

    # Set a signal frequency to activate the signal simulation
    # (needed for squeezing effects to be observed)
    fsig(1)

    # Output the pure shot noise along with the qnoised detector
    # measuring the effects of the squeezing
    qnoised sqzd_noise bs1.p2.o
    qshot shot_noise bs1.p2.o
"""
);
```

Output plots

```
out = kat.run("xaxis(l1.phase, lin, -90, 90, 360)")
out.plot(separate=False);
```



The plot shows that the `QuantumShotNoiseDetector` (page 182) output is flat, whereas the `QuantumNoiseDetector` (page 181) output rises above and falls below the shot noise depending on the phase. You should also see here that you must be careful to ensure that any squeezed field and laser fields have the correct phase and squeezing angle to give the desired effect (try for example to change the length of `s1` and see what happens).

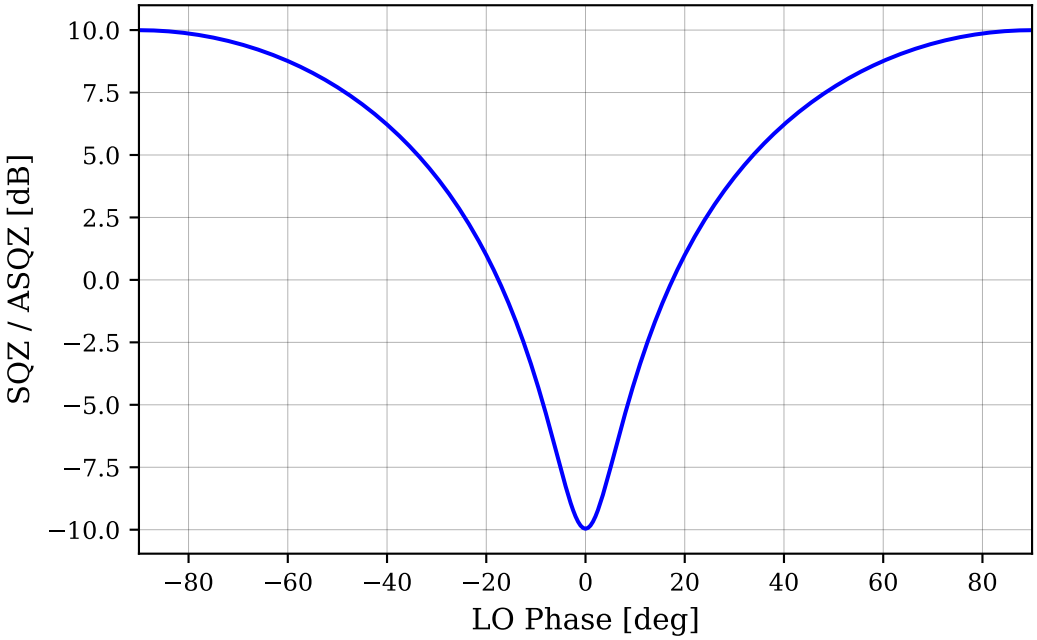
Squeezing is usually measured in decibels. Negative values correspond to squeezing and positive values correspond to antisqueezing. To convert, we can define a quick function and then plot our results. The only loss is on our beamsplitter, and it is 0.1%. With such low losses, we can see almost the full 10 dB of squeezing and antisqueezing.

```
def to_db(quantity, reference, isAmplitude=False):
    factor = 20 if isAmplitude else 10

    return factor*np.log10(quantity/reference)

sqz_db =  to_db(out['sqzd_noise'], out['shot_noise'], isAmplitude=True)

fig, ax = plt.subplots()
ax.plot(out.x1, sqz_db)
ax.set_ylabel('SQZ / ASQZ [dB]')
ax.set_xlabel('LO Phase [deg]');
```



Try reducing the reflectivity of the beamsplitter and see what happens.

5.6 Advanced topics examples

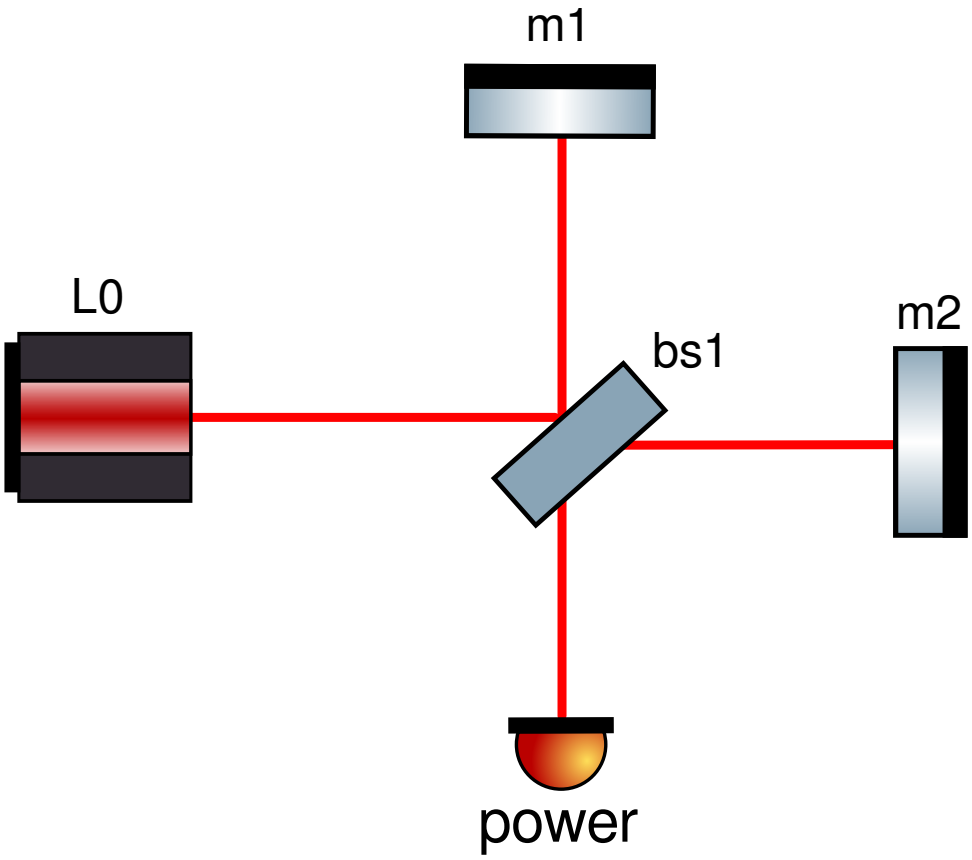
Here you will find a selection of examples for performing various simulation advanced topics using FINESSE.

5.6.1 Quantum-noise limited sensitivity of Advanced LIGO

This example shows how to compute the quantum-noise limited sensitivity of an interferometric gravitational wave detector with FINESSE.

The optical configuration

The optical layout is a very much simplified version of the Advanced LIGO interferometer, a Michelson interferometer with Fabry-Perot cavities in the arms, power recycling and signal recycling. Squeezed light is injected into the so-called dark port, which is also the main detection port, from which we measure the sensitivity.



The FINESSE model

```
import finesse
finesse.configure(plotting=True)

kat = finesse.Model()
kat.parse(
    """
    ###############################################################################
    ###   Variables
    ###############################################################################
    var Larm 3995
    var Mtm  40
    var itmT 0.014
    var lmichx 4.5
    var lmichy 4.45

    ###############################################################################
    ###   Input optics
    ###############################################################################
    l L0 125
    s l_in L0.p1 prm.p1
    # Power recycling mirror
    m prm T=0.03 L=37.5u phi=90
    s prc prm.p2 bs.p1 L=53


    # Central beamsplitter
    bs bs T=0.5 L=0 alpha=45

    ###############################################################################
    ###   X arm
```

(continues on next page)

(continued from previous page)

```
    ################################################################################
    s lx bs.p3 itmx.p1 L=lmichx
    m itmx T=itmT L=37.5u phi=90
    s LX itmx.p2 etmx.p1 L=Larm
    m etmx T=5u L=37.5u phi=89.999875

    pendulum itmx_sus itmx.mech mass=Mtm fz=1 Qz=1M
    pendulum etmx_sus etmx.mech mass=Mtm fz=1 Qz=1M

    ################################################################################
    ###   Y arm
    ################################################################################
    s ly bs.p2 itmy.p1 L=lmichy
    m itmy T=itmT L=37.5u phi=0
    s LY itmy.p2 etmy.p1 L=Larm
    m etmy T=5u L=37.5u phi=0.000125

    pendulum itmy_sus itmy.mech mass=Mtm fz=1 Qz=1M
    pendulum etmy_sus etmy.mech mass=Mtm fz=1 Qz=1M

    ################################################################################
    ###   Output and squeezing
    ################################################################################
    s src bs.p4 srm.p1 L=50.525
    m srm T=0.2 L=37.5u phi=-90

    # A squeezed source could be injected into the dark port
    sq sq1 db=0 angle=90
    s lsqz sq1.p1 srm.p2

    # Differentially modulate the arm lengths
    fsig(1)
    sgen darmx LX.h
    sgen darmy LY.h phase=180

    # Output the full quantum noise limited sensitivity
    qnoised NSR_with_RP srm.p2.o nsr=True
    # Output just the shot noise limited sensitivity
    qshot NSR_without_RP srm.p2.o nsr=True

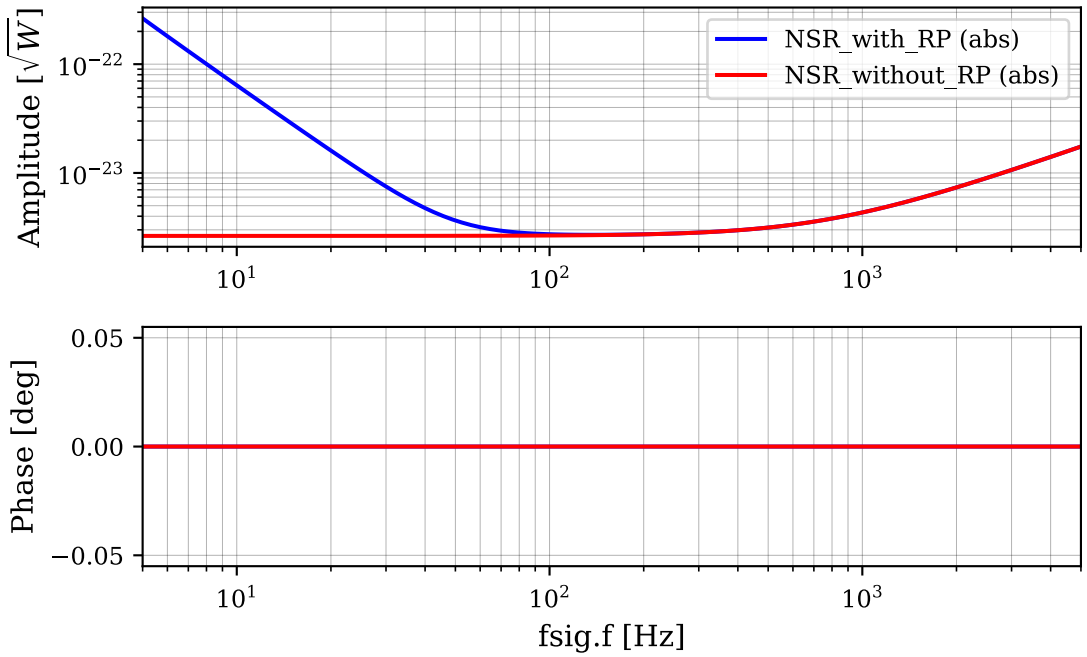
    # We could also display the quantum noise and the signal
    # separately by uncommenting these two lines.
    # qnoised noise srm.p2.o
    # pd1 signal srm.p2.o f=fsig
    """
)
```

```
<finesse.model.Model at 0x788f4a3d2900>
```

The file sets up all the various optical cavities using a plane-wave model. The arm cavity mirrors are suspended from a simple `Pendulum` (page 193) with a resonance at 1 Hz. A gravitational wave signal is injected as a modulation to both arm 'spaces', out of phase by 180 degrees. We then use the `qnoised` (page 181) and `qshot` (page 182) detectors to output the noise-to-signal ratio, or the sensitivity.

Output plots

```
out = kat.run("xaxis(darmx.f, log, 5, 5k, 100)")
out.plot(log=True, separate=False);
```



The model is loosely based on the Advanced LIGO design file, so we expect to see peak sensitivity around 100 Hz at a sensitivity of about $10^{-23}/\sqrt{\text{Hz}}$. We can see that both the `qnoised` (page 181) and `qshot` (page 182) agree at high frequencies, because they both model shot noise correctly. At low frequencies we see that they differ, as only `qnoised` (page 181) takes into account the radiation pressure (page 369) effects.

5.6.2 Inference using Dynesty (NestedSampling)

In this example we look at how to use a random sampler for inference purposes with FINESSE. This is a method that can also be used to infer optical model parameters from measured data. Although here we generate simulated data and compare against some known true values.

Note

Running this complete example takes too long to be included in Finesse's documentation build. Therefore only the first two cells are ran and are guaranteed to run successfully. The other cells containing the `dynesty` code are included statically and are not guaranteed to run successfully in the future.

```
import os
import numpy as np

import finesse
from scipy.stats import truncnorm
import numpy as np
import matplotlib.pyplot as plt
import finesse
from finesse.knm import Map
from finesse.utilities.maps import circular_aperture
```

Define the FINESSE model, parameters. We will generate a mock signal: the power as a function of detuning phase (a cavity scan). We will add some realistic measurement noise on a photodiode in the next cell

```
# create the finesse model with a fixed RoC, and beam offsets xbeta and ybeta

finesse.init_plotting()

model = finesse.Model()
```

(continues on next page)

(continued from previous page)

```
model.parse(
    """
    l l1
    mod mod1 f=9.1M midx=0.1
    m m1 R=0.984 T=0.014 Rc=-1940 xbeta=0.1e-6 ybeta=0.1e-6 ## treat RoC, betas etc... as␣
↪unknowns for MCMC
    m m2 R=1 T=0 Rc=2245
    link(l1, mod1, m1, 3994, m2)
    cav cavity m2.p1.o
    modes(maxtem=10)


    fd E_arm m2.p1.i l1.f  # Field detector
    """
)

sol = model.run(
    """
    series(
        eigenmodes(cavity, -mod1.f, name="l9"),
        eigenmodes(cavity, 0,       name="c0"),
        eigenmodes(cavity, mod1.f,  name="u9"),
        xaxis(m2.phi, lin, -10, 190, 500, name="scan")
    )
    """
)

signal = np.sum(abs(sol["scan"]["E_arm"])**2,axis=1)
```

Now add some simulated noise to make the signal in the data look more realistic. The noise on the power measured by the photodiode is modelled as random draws from a normal distribution with: mean zero and standard deviation $10^{-2.5}$, and truncated to positive values (because measurements of optical power are non-negative). The mock data is then just the sum of the simulated signal and modelled noise.

```
mean = 0
variance = 1e-5
lower_bound = 0  # Ensuring no negative values

# Calculate the standard deviation (since variance = std_dev^2)
std_dev = np.sqrt(variance)

# Define the bounds for the standard normal distribution
a = (lower_bound - mean) / std_dev
b = np.inf



# Generate samples from the truncated normal distribution
num_samples = 501  # You can adjust the number of samples as needed
noise_samples = truncnorm.rvs(a, b, loc=mean, scale=std_dev, size=num_samples)

# Add signal to noise to create mock data.
noise = noise_samples
data = signal + noise
```

Inference on radius of curvature of one of the cavity mirrors

Now assume we do not know the radius of curvature inside the resonant cavity, but a priori, we have an idea of its size, and uncertainty. In this example, we will assume a nominal range of RoC's between

$$-(1940 - 5 \times 10^{-1})\, m \leq R \leq -(1940 + 5 \times 10^{-1})\, m.$$

From our noisy data (the cavity scan) we would like to infer what we can about the radius of curvature. For simplicity, we'll assume that everything else about the cavity (beam offsets, etc…) are known. Concretely, we will use Bayesian inference to perform this measurement. In this framework, we will compute the probability density of different values of the RoC given our data. This is called the "posterior probability density" of the RoC. We will model our a priori knowledge of the radius of curvature as being uniformly, and randomly distributed within this range. In practice, we would want to determine the distribution of measurements empirically. However, this uniform distribution will serve its purpose in this toy model by allowing us define a prior distribution for inference.

```
Rc_max = -(1940+5e-1)
Rc_min = -(1940-5e-1)

def prior_transform(params):
    #sample in curvature (inverse RoC)
    curvature_min = 1./Rc_max
    curvature_max = 1./Rc_min
    curvature_unscaled = params

    curvature = uniform_rescale(curvature_min, curvature_max, curvature_unscaled)

    return curvature

def uniform_rescale(minimum, maximum, val):

    return val*(maximum-minimum) + minimum

def lnLikelihood(params):

    Rc = 1./params
    #dx = params[1]
    #dy = params[2]


    try:

        model.m1.Rc = Rc

        _sol = model.run(
            """
            series(
                eigenmodes(cavity, -mod1.f, name="l9"),
                eigenmodes(cavity, 0,       name="c0"),
                eigenmodes(cavity, mod1.f,  name="u9"),
                xaxis(m2.phi, lin, -10, 190, 500, name="scan")
            )
            """
        )


        _signal = np.sum(abs(_sol["scan"]["E_arm"])**2,axis=1)
        res = data - _signal
        if np.any(res<0):
            return -np.inf
        else:
            mean=0
            a_trunc = 0
            a = (a_trunc - mean) / std_dev
            lnL = truncnorm.logpdf(res, a, b, loc=mean, scale=std_dev).sum()
            return lnL

    except Exception as e:
        return -np.inf
```

```
nlive = 500       # number of live points
bound = 'multi'   # use MutliNest algorithm for bounds
ndims = 1         # two parameters
sample = 'auto'  # random walk sampling
#sample = 'unif'
tol = 0.1         # the stopping criterion
walks = 50
update_interval = 4.1
```

```
import dynesty

sampler = dynesty.NestedSampler(lnLikelihood, prior_transform, ndims,
                  bound=bound, sample=sample, nlive=nlive, update_interval=update_interval)
sampler.run_nested(print_progress=True)

res = sampler.results # get results dictionary from sampler
```

```
1057it [16:56:13, 57.69s/it, +500 | bound: 2 | nc: 1 | ncall: 3767 | eff(%): 47.658 |␣
↪loglstar:   -inf < 2516.248 <    inf | logz: 2512.206 +/-    nan | dlogz:  0.001 >  0.509]
```

```
from dynesty.utils import resample_equal

# draw posterior samples
weights = np.exp(res['logwt'] - res['logz'][-1])
posterior_samples = resample_equal(res.samples, weights)

print('Number of posterior samples is {}'.format(len(posterior_samples)))

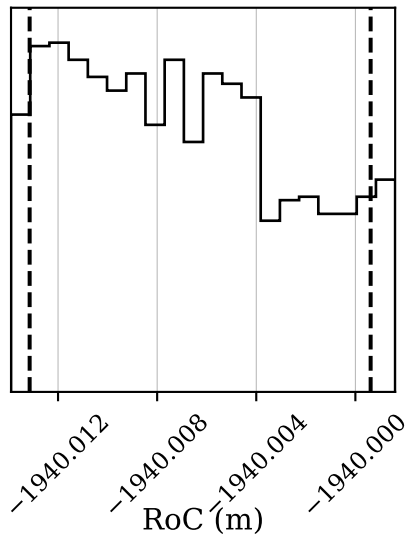
# plot using corner.py
np.save("posterior_samples", posterior_samples)
```

```
Number of posterior samples is 1557
```

```
posterior_samples = np.load("posterior_samples.npy")
posterior_samples.T[0] = 1/posterior_samples.T[0]
```

```
import corner
```

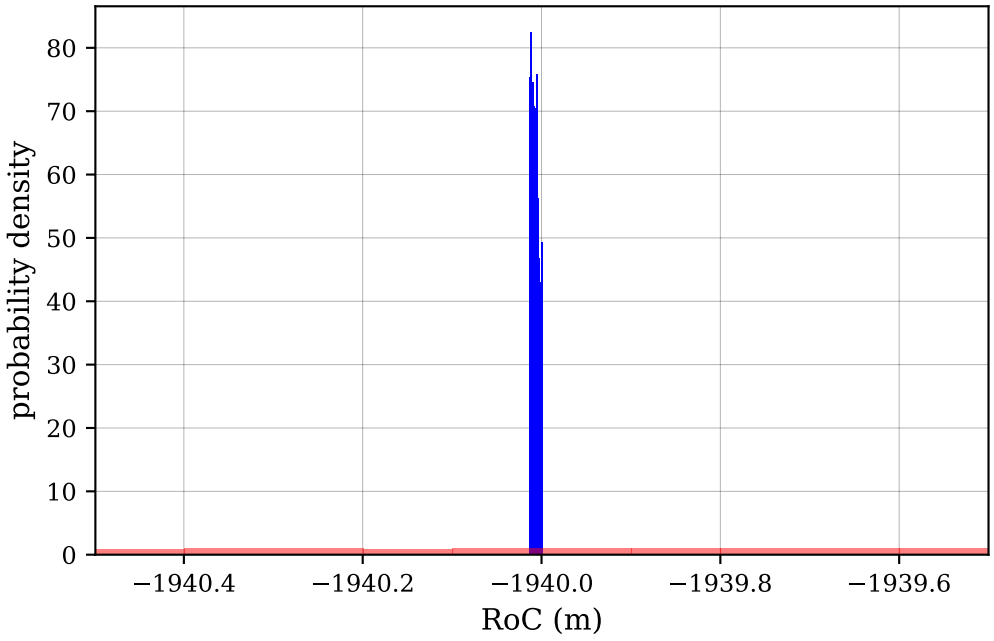
```
corner.corner(posterior_samples,labels=['RoC (m)'],quantiles=(0.95,0.05))
plt.show()
```



The resulting histogram posterior and prior can be plotted using:

```
plt.hist(np.ravel(posterior_samples),density=True)
plt.hist(np.random.uniform(low=Rc_min, high=Rc_max, size=len(np.ravel(posterior_samples))),
         density=True,alpha=0.5)
plt.xlabel("RoC (m)")
plt.ylabel("probability density")
```

```
Text(0, 0.5, 'probability density')
```



Chapter 6

User guide

Here you will find in-depth guides on how to use FINESSE in both a legacy style (for those familiar with previous versions of FINESSE) and more advanced usage via the new script syntax and Python API.

6.1 Plane-wave optics

The following sections provide information about how the various aspects of an interferometer simulation are coded within the FINESSE source code. The analysis of optical systems described here is based on the principle of superposition of light fields: a laser beam can be described as the sum of different light fields. The possible degrees of freedom are:

- frequency,
- geometrical shape and position,
- polarisation.

In the analysis of interferometric gravitational wave detectors, the amplitudes and frequencies of light fields are of principal interest. The polarisation is neglected in the analysis given here, but the formalism can in principle be easily extended to include polarisation also.

This section describes the mathematical formalism based on plane waves only. Later sections expand this to provide the formalism with respect to Hermite-Gauss modes; it is a straightforward extension of the plane wave analysis.

6.1.1 The plane-wave approximation

In many simulations the shape of the light beams or, in general, the geometric properties of a beam transverse to the optical axis are not of interest. In that case one can discard this information and restrict the model to the field on the optical axis. This is equivalent to a model where all light fields are plane waves traveling along one optical axis. This is the default mode of FINESSE and is called the *plane-wave approximation* in the following.

In the plane-wave approximation all light fields are described in one dimension. All beams and optical components are assumed to be centered on the optical axis and of infinite size. Using plane waves, it is very simple to compute interferometer signals depending on the phase and frequency of the light, and of the longitudinal degrees of freedom. Furthermore, it can be easily extended to include other degrees of freedom, such as polarisation or transverse beam shapes.

This section introduces the plane-wave approximation as used in FINESSE by default.

Description of light fields

A laser beam is usually described by the electric component of its electromagnetic field:

$$\vec{E}(t, \vec{x}) = \vec{E}_0 \cos\left(\omega\, t - \vec{k} \cdot \vec{x}\right).$$

In the following calculations, only the scalar expression for a fixed point in space is used. The calculations can be simplified by using the full complex expression instead of the cosine:

$$E(t) = E_0 \exp\left(i(\omega t + \varphi)\right) = a \exp(i\omega t),$$

where $a = E_0 \exp(i\varphi)$. The real field at that point in space can then be calculated as:

$$\vec{E}(t) = \mathrm{Re}\left\{E(t)\right\} \cdot \vec{e}_{\mathrm{pol}}$$

with $\vec{e}_{\mathrm{pol}}$ as the unit vector in the direction of polarisation. Note that FINESSE currently does not model polarization effects.

Each light field is then described by the complex amplitude a and the angular frequency ω. Instead of ω, also the frequency $f = \omega/2\pi$ or the wavelength $\lambda = 2\pi c/\omega$ can be used to specify the light field. It is convenient to define one *default (laser) frequency* f_0, which in FINESSE is done using the `lambda` (page 196) command. The frequency of other light fields is then defined by an offset Δf to that default frequency (see e.g. the `f` parameter of the `laser` (page 165) object). In the following, some functions and coefficients are defined using f_0, ω_0, or λ_0 referring to the previously defined default frequency. The setting of the default frequency is arbitrary, it merely defines a reference for frequency offsets and does not influence the results.

The electric component of electromagnetic radiation is given in Volt per meter. From this, we can calculate the intensity I in $\mathrm{Watt}/\mathrm{m}^2$ as the time-average of the *Poynting vector* S

$$I = \left|\bar{S}\right| = \frac{\epsilon_0 c}{2} EE^*,$$

with ϵ_0 the electric permeability of vacuum and c the speed of light.

In the plane-wave approximation we use the intensity of the field on axis as a stand-in or reference for the total power of the beam (remember, we are not modelling real plane waves, instead the plane-waves approximation just computes the field amplitudes on the optical axis as a model for a real beam). For convenience, unless noted otherwise, we re-define the electric field in units of $\sqrt{\mathrm{Watt}}$. By definition, the total power of a beam P in Watt is then simply calculated as

$$P = EE^*.$$

6.1.2 Mirror and beamsplitter phase relationships

As follows from Fresnel equations, passive optical components, such as mirrors, beamsplitters and lenses, can be described as flat thin layers linearly coupling with the incident light. When light impinges on that surface, both reflection and refraction of the light may occur. A coupling coefficient is the ratio of the reflected, or transmitted, light to the incident light. These ratios are generally complex, describing not only the relative amplitudes but also the phase shifts at the interface.

We define a mirror as the contact plane between two media. We derive the coupling coefficients under normal incidence. To do that, we calculate the phase jump for the transmitted and reflected beam. Finally, we obtain the phase for the transmitted beam as a function pf the position of the mirror. We repeat this for a beamsplitter, which is similar to a mirror except for an angle of incidence.

The phase relationship used is important to conserve energy. A variety exist in literature, the two most common are a symmetric and an anti-symmetric relationship between the two sides of the optic. Many theory papers use the anti-symmetric one which has no phase accumulated on transmission while one of the sides acquires a 180 degree phase shift on reflection. The other - symmetric case - is where neither reflection accumulates any phase, but the transmission from either side has a 90 degree shift, or a factor of complex i. By default, FINESSE uses the symmetric conventions, however the anti-symmetric ones can be used by setting *imaginary_transmission=False* when adding a mirror or beamsplitter.

Both cases should be physically consistent, the differences will often be in the detunings you need to use in a model to place an interferometer at the correct operating point. In more complicated scenarios however, this phase relationship must be expanded. Below also highlights the relationship between an arbitrary reflecting surface with a different refractive index and angle of incidence on each side. Since such a problem is no longer symmetric it must be treated more carefully. By default FINESSE will use the below relationships when it sees such cases. This should be taken into account if you are doing analytic comparisons.

Note

You can revert back to FINESSE v2 behaviour by changing the model settings. E.g. to use FINESSE v2 phase convention on transmission, use *self._settings.phase_config.v2_transmission_phase = True*.

Mirror

The coupling of light field amplitudes with a mirror under normal incidence can be described as follows: there are two coherent input fields, a_1 impinging on the mirror on the front and a_2 on the back surface. Two output fields leave the mirror, b_1 and b_2.

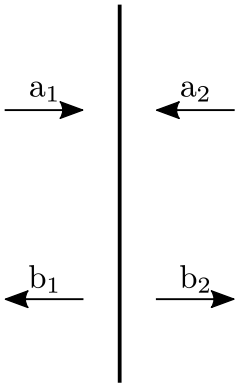


Fig. 6.1: Schematic for two coherent beams falling on the front and back surface of a mirror, a_1 and a_2, along with the outcoming beams reflected off either surface, b_1 and b_2.

The following linear equations can be used to describe the coupling:

$$\begin{pmatrix} b_1 \\ b_2 \end{pmatrix} = \begin{pmatrix} M_{11} & M_{12} \\ M_{21} & M_{22} \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \end{pmatrix}$$

Tuning

We define the *tuning* ϕ_0 of a surface as the shift in the mirror position (expressed in radians) with respect to the reference plane. A tuning of $\phi_0 = 2\pi$ translates the mirror by one vacuum wavelength (default wavelength set in FINESSE): $x = \lambda_0$. A positive tuning is defined to be in the direction of the normal vector on the front surface.

When the mirror shifts its position x meters, the corresponding tuning becomes

$$\phi_0 = k_0 x = \frac{2\pi}{\lambda_0} x = \frac{\omega_0}{c} x$$

A certain displacement results in different changes in the optical path for light fields with different frequencies. To take that into account, ϕ can be generalised as follows:

$$\phi = \phi_0 \frac{\omega}{\omega_0}$$

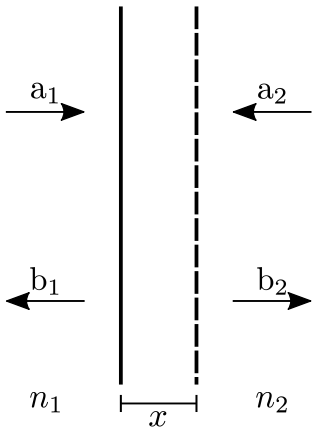


Fig. 6.2: Tuning of a mirror under normal incidence. The solid line is the reference untuned mirror, the dashed one is tuned mirror.

We define the light-mirror coupling coefficients as:

$$\begin{aligned} M_{11} &= re^{i\varphi_{11}(\phi)} \\ M_{22} &= re^{i\varphi_{22}(\phi)} \\ M_{12} = M_{21} &= te^{i\varphi_{12}(\phi)} = te^{i\varphi_{21}(\phi)} \end{aligned}$$

where r is the amplitude reflectance of the mirror, t the mirror transmittance and $\varphi_{ij}(\phi)$ the phase jump at the surface for the impinging light, which depends on the surface tuning. We assume $\varphi_{12}(\phi)$ to be equal to $\varphi_{21}(\phi)$ not to introduce a preferred direction of propagation. For a loss-less surface we can compute conditions for $\varphi_{ij}(\phi)$ from energy conservation:

$$|b_1|^2 + |b_2|^2 = a_1^2 + a_2^2$$

while $|b_1|^2$ and $|b_2|^2$ can be expressed using the definitions of the light-mirror coupling coefficients above as:

$$|b_1|^2 = r^2 a_1^2 + t^2 a_2^2 + 2rt\, a_1 a_2\, \cos(\varphi_{12} - \varphi_{11})$$

$$|b_2|^2 = t^2 a_1^2 + r^2 a_2^2 + 2rt\, a_1 a_2\, \cos(\varphi_{12} - \varphi_{22})$$

Hence energy conservation requires:

$$\cos(\varphi_{12} - \varphi_{11}) = -\cos(\varphi_{12} - \varphi_{22})$$

which in turn requires:

$$\varphi_{12} - \varphi_{11} = (2N+1)\pi - (\varphi_{12} - \varphi_{22})$$

where N is an integer. After some simple algebraic steps, we obtain:

$$\varphi_{12} = (2N+1)\frac{\pi}{2} + \frac{\varphi_{11} + \varphi_{22}}{2}$$

We arbitrarily set $N = 0$ and we will follow this convention throughout this modeling. In general, the conditions for $\varphi_{ij}(\phi)$ are given by the following equation:

$$\varphi_{12}(\phi) = \frac{\pi}{2} + \frac{\varphi_{11}(\phi) + \varphi_{22}(\phi)}{2}$$

Phase jumps in the untuned case

At the reference position of the mirror, we arbitrarily set $\varphi_{11}(0) = \varphi_{22}(0) = 0$ and we will follow this convention for the rest of the modeling. Hence the phase gain for a beam transmitted through the mirror at the reference position is:

$$\varphi_{12}(0) = \varphi_{21}(0) = \frac{\pi}{2}$$

Phase jumps in the tuned case

As the mirror is tuned, the phase of a beam reflected off the surface $\varphi_{ii}(\phi)$ is given as:

$$\varphi_{11}(\phi) = 2n_1\phi$$
$$\varphi_{22}(\phi) = -2n_2\phi$$

where n_1 and n_2 are the indices of refraction of the media on either side of the surface. Substituting $\varphi_{11}(\phi)$ and $\varphi_{22}(\phi)$ in the equation for $\varphi_{ij}(\phi)$, we obtain $\varphi_{12}(\phi)$:

$$\varphi_{12}(\phi) = \varphi_{21}(\phi) = \frac{\pi}{2} + (n_1 - n_2)\phi$$

6.1.3 Beamsplitter

A beamsplitter is similar to a mirror except for the extra parameter α which indicates the tilt angle relative to the incoming beams.

Reflection

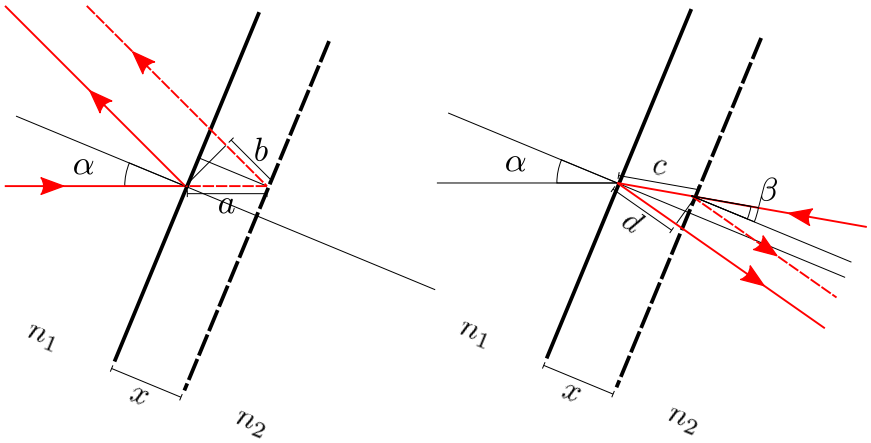


Fig. 6.3: Schematic for the beam reflected off a beamsplitter in the reference (solid lines) and tuned case (dashed lines).

Referring to the figure above, we define the following geometrical paths as:

$$a = \frac{x}{\cos(\alpha)}$$

$$b = a\cos(2\alpha)$$

$$c = \frac{x}{\cos(\beta)}$$

$$d = c\cos(2\beta)$$

where β is the refraction angle given by Snell's law:

$$n_1 \sin(\alpha) = n_2 \sin(\beta)$$

The phase change for a beam reflected on one side of a beamsplitter is:

$$\left(\frac{\omega}{\omega_0}k_0\right) n_1|a+b| = 2n_1\phi\cos\alpha$$

As for a beam reflected on the other side, the phase change is:

$$\left(\frac{\omega}{\omega_0}k_0\right) n_2|c+d| = 2n_2\phi\cos\beta$$

As was done for the mirror, we model the beamsplitter-light coupling via linear coefficients M_{ij}:

$$\begin{aligned} M_{11} &= re^{i\varphi_{11}(\phi)} \\ M_{22} &= re^{i\varphi_{22}(\phi)} \\ M_{12} = M_{21} &= te^{i\varphi_{12}(\phi)} = te^{i\varphi_{21}(\phi)} \end{aligned}$$

The conditions for $\varphi_{ij}(\phi)$ are given by:

$$\varphi_{12}(\phi) = \frac{\pi}{2} + \frac{\varphi_{11}(\phi) + \varphi_{22}(\phi)}{2}$$

The phase change for a beam reflected off either side of the beamsplitter is given as:

$$\begin{aligned} \varphi_{11}(\phi) &= 2n_1\phi\cos\alpha \\ \varphi_{22}(\phi) &= -2n_2\phi\cos\beta \end{aligned}$$

Substituting $\varphi_{11}(\phi)$ and $\varphi_{22}(\phi)$ in the equation for $\varphi_{ij}(\phi)$, we obtain $\varphi_{12}(\phi)$:

$$\varphi_{12}(\phi) = \varphi_{21}(\phi) = \frac{\pi}{2} + (n_1\cos\alpha - n_2\cos\beta)\phi$$

6.1.4 Modulation of light fields

In principle, all parameters of a light field can be modulated. This section describes the modulation of the amplitude, phase and frequency of the light. Any sinusoidal modulation of amplitude or phase generates new field components that are shifted in frequency with respect to the initial field. Basically, light power is shifted from one frequency component, the carrier, to several others, the sidebands. The relative amplitudes and phases of these sidebands differ for different types of modulation and different modulation strengths.

Phase Modulation

Phase modulation can create a large number of sidebands. The number of sidebands with noticeable power depends on the modulation strength (or depth) given by the *modulation index* m. Assuming an input field

$$E_{\text{in}} = E_0\, e^{\mathrm{i}\,\omega_0\, t},$$

a sinusoidal phase modulation of the field can be described as

$$E = E_0\, e^{\mathrm{i}\,(\omega_0\, t + m\cos(\omega_m\, t))}.$$

This equation can be expanded using the *Bessel functions of the first kind* $J_k(m)$. We can write

$$E = E_0\, e^{\mathrm{i}\,\omega_0\, t} \sum_{k=-\infty}^{\infty} \mathrm{i}^{\,k}\, J_k(m)\, e^{\mathrm{i}\,k\,\omega_m\, t}. \tag{6.1}$$

The field for $k = 0$, oscillating with the frequency of the input field ω_0, represents the carrier. The sidebands can be divided into *upper* ($k > 0$) and *lower* ($k < 0$) sidebands. These sidebands are light fields that have been shifted in

frequency by $k\,\omega_m$. The upper and lower sidebands with the same absolute value of k are called a pair of sidebands of order k.

Equation (6.1) shows that the carrier is surrounded by an infinite number of sidebands. However, for small modulation indices ($m < 1$) the Bessel functions rapidly decrease with increasing k, so we can use the approximation:

$$J_k(m) = \frac{1}{k!}\left(\frac{m}{2}\right)^k + O\left(m^{k+2}\right).$$

In this case, only a few sidebands have to be taken into account. For $m \ll 1$ we can write

$$E = E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(J_0(m) - \mathrm{i}\,J_{-1}(m)\, e^{-\mathrm{i}\,\omega_m\,t} + \mathrm{i}\,J_1(m)\, e^{\mathrm{i}\,\omega_m\,t}\right),$$

and with

$$J_{-k}(m) = (-1)^k J_k(m),$$

we obtain

$$E = E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 + \mathrm{i}\,\frac{m}{2}\left(e^{-\mathrm{i}\,\omega_m\,t} + e^{\mathrm{i}\,\omega_m\,t}\right)\right),$$

as the first-order approximation in m. In the above equation the carrier field remains unchanged by the modulation, therefore this approximation is not the most intuitive. It is clearer if the approximation up to the second order in m is given:

$$E = E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 - \frac{m^2}{4} + \mathrm{i}\,\frac{m}{2}\left(e^{-\mathrm{i}\,\omega_m\,t} + e^{\mathrm{i}\,\omega_m\,t}\right)\right),$$

which shows that power is transferred from the carrier to the sideband fields.

Higher-order expansions in m can be performed simply by specifying the highest order of Bessel function, which is to be used in the sum in Equation (6.1), i.e.

$$E = E_0\, e^{\mathrm{i}\,\omega_0\,t} \sum_{k=-\mathrm{order}}^{\mathrm{order}} i^k\, J_k(m)\, e^{\mathrm{i}\,k\omega_m\,t}.$$

Frequency modulation

For small modulation indices, phase modulation and frequency modulation can be understood as different descriptions of the same effect [28]. With the frequency defined as $f = \mathrm{d}\varphi/\mathrm{d}t$ a sinusoidal frequency modulation can be written as:

$$E = E_0\, e^{\mathrm{i}\left(\omega_0\,t + \frac{\Delta\omega}{\omega_m}\cos\left(\omega_m\,t\right)\right)},$$

with $\Delta\omega$ as the frequency swing (how *far* the frequency is shifted by the modulation) and ω_m the modulation frequency (how *fast* the frequency is shifted). The modulation index is defined as:

$$m = \frac{\Delta\omega}{\omega_m}$$

Amplitude modulation

In contrast to phase modulation, (sinusoidal) amplitude modulation always generates exactly two sidebands. Furthermore, a natural maximum modulation index exists: the modulation index is defined to be one ($m = 1$) when the amplitude is modulated between zero and the amplitude of the unmodulated field.

If the amplitude modulation is performed by an active element, for example by modulating the current of a laser diode, the following equation can be used to describe the output field:

$$\begin{aligned} E &= E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 + m\cos\left(\omega_m\,t\right)\right) \\ &= E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 + \tfrac{m}{2}\, e^{\mathrm{i}\,\omega_m\,t} + \tfrac{m}{2}\, e^{-\mathrm{i}\,\omega_m\,t}\right). \end{aligned}$$

However, passive amplitude modulators (like acousto-optic modulators or electro-optic modulators with polarisers) can only reduce the amplitude. In these cases, the following equation is more useful:

$$\begin{aligned} E &= E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 - \tfrac{m}{2}\left(1 - \cos\left(\omega_m\,t\right)\right)\right) \\ &= E_0\, e^{\mathrm{i}\,\omega_0\,t}\left(1 - \tfrac{m}{2} + \tfrac{m}{4}\, e^{\mathrm{i}\,\omega_m\,t} + \tfrac{m}{4}\, e^{-\mathrm{i}\,\omega_m\,t}\right). \end{aligned}$$

Alignment Modulation

As an experimental feature, modelling alignment modulation is possible with the `yaw` and `pitch` options for the `mod_type` argument of `modulator` (page 159), but this has not been tested yet.

6.1.5 Transfer Functions and Error Signals

Two common tasks for interferometer analysis are the computations of error signals and optical transfer functions. In an interferometer, several degrees of freedom for the optical components exist (e.g. positions, alignment angles) and active stabilisation is necessary to enhance the sensitivity.

An error signal is the output of a sensor (or in general a measurable signal) as a function of one degree of freedom (of the interferometer). The transfer function now gives the frequency-dependent coupling of a signal that is present in that particular 'degree of freedom' into the error signal. Transfer functions can be used to compute the coupling of noise in the interferometer and thus to estimate the sensitivity.

Error signals

In general, an error signal is an output of any detector that is suitable for stabilising a certain parameter p with a servo loop. The error signal must be a function of that parameter. In most cases it is preferable to have a bipolar signal with a zero crossing at the operating point p_0. The slope at the operating point is a measure of the sensor gain.

Transfer Functions

A transfer function describes how a system responds to a small periodic disturbance. If we "wiggle" some parameter of the system sinusoidally at a frequency f, the transfer function tells us:

- how large the resulting output oscillation is (amplitude response), and
- how delayed it is relative to the excitation (phase response).

Transfer functions describe the propagation of a periodic signal through a plant and are usually given as frequency plots of amplitude and phase. A transfer function describes the linear coupling of signals inside a system. This means a transfer function is independent of the actual signal size.

For small signals or small deviations, most systems can be linearised and correctly described by transfer functions. Physically, this means we approximate the system as responding proportionally to tiny perturbations around its operating point.

Experimentally, network analysers are commonly used to measure a transfer function: one connects a periodic signal (the source) to an actuator of the plant and compares it to a measured sensor signal. By mixing the source with the sensor signal the analyser determines the amplitude and phase of the response relative to the excitation.

Mathematically, applying a sinusoidal signal $\sin(\omega_s t)$ to the interferometer, for example as a position modulation of a cavity mirror, creates phase-modulation sidebands offset by $\pm\omega_s$ from the carrier light. If detected appropriately, the photodiode output contains a component at ω_s which can be extracted by demodulation.

Modulation-demodulation methods

Several standard techniques exist to generate error signals for controlling an interferometer. Many of them use modulation-demodulation schemes in which a light field is modulated (in phase or amplitude) at a fixed frequency and the detector output is demodulated at that same frequency.

This technique shifts low-frequency information to higher frequencies, where technical noise is often lower, thereby improving the signal-to-noise ratio.

Modelling Transfer Functions

There are two different ways to compute transfer functions in FINESSE which are well described in modelling transfer functions (page 267). That section describes a 'manual' method and a more general method using the frequency response actions (page 272).

The manual method explicitly injects a signal using the `sgen` (page 169) element and measure the response with a demodulated power detector `pd1` (page 180). You can see an example of this in the living review optical spring

example (page 567), which is a direct translation of the FINESSE 2 syntax. While this syntax can be illustrative in simple examples, for signal simulations with multiple inputs and outputs it is cumbersome to add multiple signal generators and detectors. Therefore we recommend using the frequency response actions (page 272) for your simulations.

6.1.6 Input fields or the 'right hand side' vector

After the set of linear equations for a system has been determined, the input fields have to be given by the user. The respective fields are entered into the 'right hand side' (RHS) vector of the set of linear equations. The RHS vector consists of complex numbers that specify the amplitude and phase of every input field. Input fields are initially set to zero, and every non-zero entry describes a source. The possible sources in the carrier simulation are lasers and modulators, and in the signal simulation the only sources are signal generators.

Carriers

Laser

The principal light sources are, of course, the lasers. They are connected to one optical node only. The input power is specified by the user in the input file. For every laser the field amplitude is set as:

$$E_{in} = C\sqrt{(P/\epsilon_c)}\, e^{\mathrm{i}\,\varphi},$$

with

$$P = C^2|E|^2,$$

as the laser power and φ the specified phase. The constant factor C depends on the chosen internal scaling for the field amplitudes that can be set in the init file 'kat.ini'. The default value is $C = 1$. This setting does not yield correct absolute values for light field amplitudes, i.e. when amplitude detectors are used. Instead, one obtains more intuitive numbers from which the respective light power can be easily computed.

Modulators

Modulators produce non-zero entries in the carrier RHS vector for every modulation sideband generated. Depending on the order ($k \geq 0$) and the modulation index (m), the input field amplitude for amplitude modulation is:

$$E_{\text{in}} = \frac{m}{4},$$

and for phase modulation:

$$E_{\text{in}} = (-1)^k\, J_k(m)\ \exp(\mathrm{i}\,\varphi),$$

with φ given as (Equation (6.1)):

$$\varphi = \pm k \cdot (\frac{\pi}{2} + \varphi_{\text{s}}),$$

where φ_{s} is the user-specified phase from the modulator description. The sign of φ is the same as the sign of the frequency offset of the sideband. For 'lower' sidebands ($f_{\text{mod}} < 0$) we get $\varphi = -\ldots$, for 'upper' sidebands ($f_{\text{mod}} > 0$) it is $\varphi = +\ldots$.

Signals

Signals in FINESSE refer to very small, linearised modulations of the optical fields, or mechanical, or electrical states. Optical signal sidebands come in pairs of 'upper' and 'lower' sidebands, and are equally spaced around each carrier optical frequency by a frequency $\Omega = 2\pi f_{\text{sig}}$. In general, at any optical node the optical field near each carrier will be described by:

$$E_c[1 + E_+ \exp(()\mathrm{i}\,\Omega t) + E_- \exp(() - \mathrm{i}\,\Omega t)]\exp(()\omega t)$$

They are used to compute linear time invariant transfer functions in FINESSE. These sidebands describe the modulation of the carrier light field with respect to exciting some other state in the simulation, such as a mechanical or electrical

state. Signal sidebands are always generated at a single frequency, this is the 'signal frequency' or what is seen in the FINESSE code as `fsig` in many locations, which is f_{sig} as mentioned above. `fsig` is in units of Hertz. For mechanical and electrical states only the 'upper' sideband is directly calculated to compute the modulation around the DC value of the state. FINESSE does not handle multiple mechanical or electrical carrier frequencies like with the optical calculations.

As with the carrier field calculations a matrix system is constructed to describe how all the signal sidebands interact with each other, modulate the carrier frequencies, and propagate through the model. This is called the signal matrix. By solving this matrix system you can ask how much signal at one node is generated by exciting another signal node in the system. This output divided by the input computes the transfer function. As the system is linearised, the input signal is often just a unitary value. Therefore it computes how much optical field modulation you might see got one meter of motion of a mirror. Obviously one cannot modulate a mirror position by a meter in an interferometer and the system still work. Experimentally this modulation is very small, small enough so that the linearisation assumptions are true.

Note

Signal sidebands can also be referred to as 'audio' sidebands in literature. Signal is used in FINESSE as it is more general and the frequency range is not technically limited to audio frequencies. The term *audio sidebands* is mostly a carry over from gravitational wave detectors, where the signals are often in the audio frequency range.

It is critical to understand that the ampltiude of signal sidebands are not simply in units of optical field, mechanical motion, or electrical signal. They are the amplitude induced due to some excitation, such as an optic being moved or a voltage being applied. The units of a signal sideband field must be carefully considered as they depend on how you setup your simulation. An optical signal field will have units of $\sqrt{\mathrm{W}}/X$ where X depends on what you excite. If you excite a mirror position, then X is in meters. If you excite a voltage that induces some change in an optical field, then X is in Volts.

Warning

You should be careful not to excite too many signals at once which do not make sense, such as exciting a voltage and a mirror position at the same. The units of that resulting transfer function will be more complicated.

In practice FINESSE actually needs to compute the conjugate of the lower optical signal sidebands. Thus internally the solver computes:

$$E_c[1 + E_+ \exp{(()\mathrm{i}\,\Omega t)} + E_-^* \exp{(()\mathrm{i}\,\Omega t)}] \exp{(()\omega t)}$$

The reason for this is that this allows us to compute non-linear effects such as radiation pressure and photodiode detection inside the matrix solver. In practice the user should not need to ever worry about this and FINESSE will return E_-. However, if you delve into the internals or write your own elements then you should keep this in mind.

Mirror

Mirror motion does not change transmitted light. The reflected light will be modulated in phase by any mirror movement. The relevant parameters are shown in Fig. 6.4. At a reference plane (at the nominal mirror position when the tuning is zero), the field impinging on the mirror is:

$$E_{\text{in}} = E_0 \exp\Big(\mathrm{i}\,(\omega_{\text{c}} t + \varphi_{\text{c}} - k_{\text{c}} x)\Big) = E_0 \exp{(\mathrm{i}\,\omega_{\text{c}} t + \varphi_{\text{c}})}.$$

If the mirror is detuned by x_{t} (here given in meters) then the optical field at the mirror is:

$$E_{\text{mir}} = E_{\text{in}} \exp{(-\mathrm{i}\,k_{\text{c}} x_{\text{t}})}.$$

With the given parameters for the signal frequency, the position modulation can be written as $x_{\text{m}} = a_{\text{s}} \cos(\omega_{\text{s}} t + \varphi_{\text{s}})$ and thus the reflected field at the mirror is:

$$E_{\text{refl}} = r\, E_{\text{mir}} \exp{(\mathrm{i}\, 2 k_{\text{c}} x_{\text{m}})} = r\, E_{\text{mir}} \exp\Big(\mathrm{i}\, 2 k_{\text{c}} a_{\text{s}} \cos(\omega_{\text{s}} t + \varphi_{\text{s}})\Big),$$

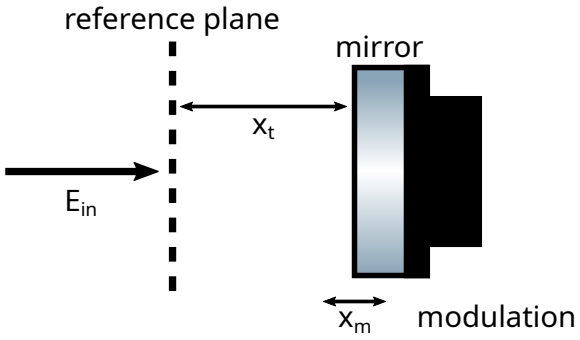


Fig. 6.4: Signal applied to a mirror: modulation of the mirror position.

setting $m = 2k_c a_s$, this can be expressed as:

$$\begin{aligned} E_{\text{refl}} &= r\, E_{\text{mir}} \left(1 + \text{i}\,\tfrac{m}{2} \exp\left(-\text{i}\,(\omega_s t + \varphi_s)\right) + \text{i}\,\tfrac{m}{2} \exp\left(\text{i}\,(\omega_s t + \varphi_s)\right)\right) \\ &= r\, E_{\text{mir}} \left(1 + \tfrac{m}{2} \exp\left(-\text{i}\,(\omega_s t + \varphi_s - \pi/2)\right)\right) \\ &\quad + \left(\tfrac{m}{2} \exp\left(\text{i}\,(\omega_s t + \varphi_s + \pi/2)\right)\right). \end{aligned}$$

This gives an amplitude for both sidebands of:

$$a_{\text{sb}} = r\, m/2\, E_0 = r\, k_c a_s\, E_0.$$

The phase back at the reference plane is:

$$\varphi_{\text{sb}} = \varphi_c + \frac{\pi}{2} \pm \varphi_s - (k_c + k_{\text{sb}})\, x_t, \tag{6.2}$$

where the plus sign refers to the 'upper' sideband and the minus sign to the 'lower' sideband. As in FINESSE the tuning is given in degrees, i.e. the conversion from x_t to ϕ has to be taken into account:

$$\begin{aligned} \varphi_{\text{sb}} &= \varphi_c + \tfrac{\pi}{2} \pm \varphi_s - (\omega_c + \omega_{\text{sb}})/c\, x_t \\ &= \varphi_c + \tfrac{\pi}{2} \pm \varphi_s - (\omega_c + \omega_{\text{sb}})/c\, \lambda_0/360\, \phi \\ &= \varphi_c + \tfrac{\pi}{2} \pm \varphi_s - (\omega_c + \omega_{\text{sb}})/\omega_0\, 2\pi/360\, \phi. \end{aligned}$$

With a nominal signal amplitude of $a_s = 1$, the sideband amplitudes become very large. For an input light field at the default wavelength one typically obtains:

$$a_{\text{sb}} = r\, k_c\, E_0 = r\, \omega_c/c\, E_0 = r\, 2\pi/\lambda_0\, E_0 \approx 6 \cdot 10^6.$$

Numerical algorithms have the best accuracy when the various input numbers are of the same order of magnitude, usually set to a number close to one. Therefore, the signal amplitudes for mirrors (and beam splitters) should be scaled: a natural scale is to define the modulation in radians instead of meters. The scaling factor is then ω_0/c, and setting $a = \omega_0/c\, a'$ the reflected field at the mirror becomes:

$$\begin{aligned} E_{\text{refl}} &= r\, E_{\text{mir}}\, \exp\left(\text{i}\, 2\omega_c/\omega_0\, x_m\right) \\ &= r\, E_{\text{mir}}\, \exp\left(\text{i}\, 2\omega_c/\omega_0\, a'_s \cos(\omega_s t + \varphi_s)\right), \end{aligned}$$

and thus the sideband amplitudes are:

$$a_{\text{sb}} = r\, \omega_c/\omega_0\, a'_s\, E_0,$$

with the factor ω_c/ω_0 typically being close to one. The units of the computed transfer functions are 'output unit per radian'.

When a light field is reflected at the back surface of the mirror, the sideband amplitudes are computed accordingly. The same formulae as above can be applied with $x_m \to -x_m$ and $x_t \to -x_t$, yielding the same amplitude as for the reflection at the front surface, but with a slightly different phase:

$$\begin{aligned} \varphi_{\text{sb,back}} &= \varphi_c + \tfrac{\pi}{2} \pm (\varphi_s + \pi) + (k_c + k_{\text{sb}})\, x_t \\ &= \varphi_c + \tfrac{\pi}{2} \pm (\varphi_s + \pi) + (\omega_c + \omega_{\text{sb}})/\omega_0\, 2\pi/360\, \phi. \end{aligned}$$

Beamsplitter

When the signal frequency is applied to the beam splitter, the reflected light is modulated in phase. In fact, the same computations as for mirrors can be used for beam splitters. However, all distances have to be scaled by $\cos(\alpha)$. Again, only the reflected fields are changed by the modulation and the front side and back side modulation have different phases. The amplitude and phases compute to:

$$\begin{aligned} a_{\text{sb}} &= r\ \omega_c/\omega_0\ a_{\text{s}} \cos(\alpha)\ E_0, \\ \phi_{\text{sb,front}} &= \varphi_c + \frac{\pi}{2} \pm \varphi_{\text{s}} - (k_c + k_{sb}) x_t \cos(\alpha), \\ \phi_{\text{sb,back}} &= \varphi_c + \frac{\pi}{2} \pm (\varphi_{\text{s}} + \pi) + (k_c + k_{sb}) x_t \cos(\alpha). \end{aligned}$$

Space

For interferometric gravitational wave detectors, the 'free space' is an important source for a signal frequency: a passing gravitational wave modulates the length of the space (i.e. the distance between two test masses). A light field passing this length will thus be modulated in phase. The phase change $\phi(t)$ which is accumulated over the full length is (see, for example, [29]):

$$\phi(t) = \frac{\omega_c\ n\ L}{c} + \frac{a_s}{2}\frac{\omega_c}{\omega_s} \sin\left(\omega_s \frac{n\ L}{c}\right) \cos\left(\omega_s \left(t - \frac{n\ L}{c}\right)\right),$$

with L the length of the space, n the index of refraction and a_s the signal amplitude given in strain (h). This results in a signal sideband amplitude of:

$$\begin{aligned} a_{\text{sb}} &= \frac{1}{4}\frac{\omega_c}{\omega_s} \sin\left(\omega_s \frac{n\ L}{c}\right) a_{\text{s}}\ E_0, \\ \phi_{\text{sb}} &= \varphi_c + \frac{\pi}{2} \pm \varphi_s - (\omega_c + \omega_s)\frac{nL}{c}. \end{aligned}$$

Laser

A laser has three types of signal modulation available: phase, frequency and amplitude. These are treated as in Modulation of light fields (page 117).

Frequency

Frequency modulation gives a field

$$E = E_0\ e^{\text{i}\,(\omega_c\,t + a_s/\omega_s \cos(\omega_s\,t + \varphi_s) + \varphi_c)},$$

with sideband amplitude and phase

$$\begin{aligned} a_{sb} &= \tfrac{a_{\text{s}}}{2\omega_{\text{s}}}\ E_0, \\ \phi_{\text{sb}} &= \varphi_c + \tfrac{\pi}{2} \pm \varphi_s. \end{aligned}$$

Phase

With phase modulation, the laser produces a field

$$E = E_0\ \exp\Big(\text{i}\,(\omega_0\,t + a_{\text{s}} \cos(\omega_s\,t))\Big),$$

giving sidebands with

$$\begin{aligned} a_{sb} &= \tfrac{a_{\text{s}}}{2}\ E_0, \\ \phi_{\text{sb}} &= \varphi_c \pm \tfrac{\pi}{2} \pm \varphi_s. \end{aligned}$$

Amplitude

Amplitude modulation gives us

$$E \;=\; E_0\;\exp\left(\mathrm{i}\,\omega_0\,t\right)\;\left(1-\tfrac{m}{2}\left(1-\cos\left(\omega_m\,t\right)\right)\right),$$

with sidebands

$$a_{sb} = \tfrac{a_s}{4}\,E_0,$$
$$\phi_{\text{sb}} = \varphi_c \pm \varphi_s.$$

Modulator

Signal frequencies at a modulator are treated as 'phase noise'. The electric field that leaves the modulator can be written as:

$$E \;=\; E_0 \quad e^{\mathrm{i}\,(\omega_0\,t+\varphi_0)}\;\textstyle\sum_{k=-order}^{order} i^{\,k}\;J_k(m)\;e^{\mathrm{i}\,k(\omega_m\,t+\varphi_m)}$$
$$\times\left(1+\mathrm{i}\,\tfrac{k\;m_2\;a_s}{2}\;e^{-i(\omega_s\,t+\varphi_s)}+\mathrm{i}\,\tfrac{k\;m_2\;a_s}{2}\;e^{\mathrm{i}\,(\omega_s\,t+\varphi_s)}+O((km_2)^2)\right),$$

with

$$E_{\text{mod}} \;=\; E_0\;J_k(m),$$
$$\varphi_{\text{mod}} = \varphi_0 + k\tfrac{\pi}{2} + k\varphi_m.$$

The sideband amplitudes are:

$$a_{sb} = \tfrac{a_s k m_2}{2}\,E_{\text{mod}},$$
$$\phi_{\text{sb}} = \varphi_{\text{mod}} + \tfrac{\pi}{2} \pm \varphi_s.$$

6.1.7 Cavities

- Free spectral range (page 124)

Free spectral range

6.2 Building and interacting with models

Before you can perform simulations of your optical system you must first build a model of it. FINESSE includes a variety of tools and features to make it straight forward to build models ranging from simple single Fabry-Perot cavities to full gravitational wave interferometers.

We offer two ways in which you can build a model. Firstly using KatScript, which is a declarative domain specific language we have developed for building models of interferometers. This removes the complexity of dealing with Python and keeps you focused on the physics of your model. Secondly, we offer a full Python application programming interface (API) that can also be used for building a model for those wishing to use a more programmatic object-orientated interface.

Neither is better or worse, however most of the examples in the documentation will be shown using KatScript. Any feature in KatScript has an equivalent in the Python API, however not all features in FINESSE are guaranteed to have an equivalent KatScript option. This will likely be true for newer or more complex features that users would only really use from a Python interface.

The Python API section also details how you can interact with your model once it has been built. It allows you to easily amend and update the model and parameters.

6.2.1 KatScript

FINESSE includes a built-in definition language called KatScript that provides an alternative way to build models without using the Python *API*. It is concise, simple, and easy to use. The following contains a brief introduction to KatScript syntax and the available instructions.

Supported elements (page 142), commands (page 196) and analyses (page 198) are described in their corresponding sections. There are also command, element and analysis indices for quick reference.

Quick reference

You can quickly get help for particular syntax in an interactive Python environment by calling the `syntax` function:

```
import finesse
print(finesse.syntax("mirror"))
```

```
m / mirror / Mirror: m name R=none T=none L=none phi=0 Rc=inf xbeta=0 ybeta=0␣
↪misaligned=false imaginary_transmission=trueNone
```

From the Command line interface (page 384) you can also use:

```
$ kat3 syntax mirror
$ kat3 syntax pd*  # Wildcards are allowed too.
```

A quick example

Optical models are built by specifying components, detectors, commands, and other types of statement on separate lines:

```
# Example of 10 W of laser power incident upon a Fabry-Perot cavity.
l l1 P=10
s s1 l1.p1 m1.p1
m m1 R=0.99 T=0.01
s s2 m1.p2 m2.p1
m m2 R=0.99 T=0.01
pd pd1 m2.p1.i
modes(even, maxtem=10)
```

To use KatScript in Python you will need to wrap it in a call to `Model.parse()`:

```
from finesse import Model

model = Model()

# Parse the simple model from above
model.parse(
    """
    # Example of 10 W of laser power incident upon a Fabry-Perot cavity.
    l l1 P=10
    s s1 l1.p1 m1.p1
    m m1 R=0.99 T=0.01
    s s2 m1.p2 m2.p1
    m m2 R=0.99 T=0.01
    pd pd1 m2.p1.i
    modes(even, maxtem=10)
    """
)
```

```
<finesse.model.Model at 0x7ed3b42367b0>
```

Note

You can also pass the KatScript directly when creating the model:

```
model = Model(
    """
    # Example of 10 W of laser power incident upon a Fabry-Perot cavity.
    l l1 P=10
    s s1 l1.p1 m1.p1
    m m1 R=0.99 T=0.01
    s s2 m1.p2 m2.p1
    m m2 R=0.99 T=0.01
    pd pd1 m2.p1.i
    modes(even, maxtem=10)
    """
)
```

Most statements are formed from a sequence of space-delimited values; these represent *elements* of the model. Elements always have unique names, which must be defined after the element type, which is specified first. The element type is typically, but not always, represented by a short one or two character name. The arguments following the element name are used to customise the element, such as to set the reflectivity and transmissivity in the case of the mirrors.

The last line is a *command* which sets some properties of the model or simulation. In this case, it's setting which higher order modes to simulate. Commands use "functional" form, with comma-delimited parameters inside parentheses.

Language fundamentals

Whitespace and comments

Whitespace can be simple spaces, tabs and newlines. Comments start with `#`. Everything including and after `#` on the same line is ignored by the parser.

```
# This is a comment!
```

Whitespace is sometimes important in KatScript, for instance between parameters of elements (page 127).

Types, names, and keywords

Component types (like `mirror` (page 142)), names (like `m1`) and keywords (like `lin`) are written directly, without quotation marks:

```
mirror m1  # A mirror named 'm1'.
```

User-defined names have to start with a letter, but can otherwise contain integer numbers and underscores.

Numbers

Numbers can use integer, floating point, scientific and complex notation. The imaginary unit is `j`. Numbers can include _ between digits to assist clarity, e.g. `1_000_000`. SI prefices are supported, but cannot be combined with scientific notation. Integers with leading zeros, e.g. `01`, are not allowed.

Some valid number examples:

```
# Typical number forms.
1
1.
1.2
-3.141
6.5+2.0j
```

(continues on next page)

(continued from previous page)

```
# Scientific notation.
1.1e7   # 11000000.0
7.6e-2  # 0.076

# SI prefices. Any of p, n, u, m, k, M, G, and T are supported.
1m  # 0.001
1k  # 1000.0

# Separation of digits using `_`.
1_234_567       # 1234567
1_234.5_678     # 1234.5678
1_234e6         # 1234000000.0
1_234j          # 1234j
```

Components and detectors

Component and detector definitions, collectively referred to as *elements*, for the most part mirror their Python API signatures. Both positional and keyword arguments are supported. Types of element can be referred to using either their full name or an abbreviated short form. They are defined using syntax of the form `element name arg1 arg2 arg3=val3 ...`:

```
# Definition with positional arguments (Finesse 2 style): a mirror named "m1" with
# reflectivity R=0.99 and transmissivity T=0.01.
mirror m1 0.99 0.01

# Definition with keyword arguments.
# Note that only two of "R", "T" and "L" are required, and the third is calculated
# from the first two. You can still specify all three, but they must all add up to
# 1.
mirror m2 R=0.99 T=0.01

# You can mix positional and keyword arguments as long as they make sense. Like in
# Python, positional arguments must always come before keyword arguments.
mirror m3 1 0 Rc=[1, 1.1]  # Rcx and Rcy defined together.
```

Element definitions cannot span multiple lines.

Many components and detectors have aliases that can be used instead of longer names. For example, `m` can be used instead of `mirror`, or `bs` instead of `beamsplitter`. Available aliases are listed for each instruction in the syntax references for Model elements (page 142), Commands (page 196), and Actions, Analyses, and Solutions (page 198).

Commands and analyses

Commands set some special behaviour in the model. Analyses define one or more procedures to perform on the model defined in the script. Both commands and analyses are defined using syntax in the form `command(arg1, arg2, ...)`:

```
# Command used to set the higher order modes to simulate.
modes(maxtem=3)

# The `modes` command supports various different types of input. You can for example
# specify individual modes...
modes([[1, 0], [0, 1], [2, 0]])

# ...or odd modes up to some maximum order.
modes(odd, maxtem=9)
```

The arguments inside the parentheses in command and analysis definitions can span across multiple lines, and may contain comments:

```
modes(
    even,
    # Set maximum order.
    maxtem=10
)
```

Some analyses can contain other analyses; this is used for example to build a set of nested procedures to perform on the model:

```
# Xaxis function to perform a linear sweep over the value of m1.phi.
xaxis(m1.phi, lin, 0, 10k, 500)

# A nested action tree.
series(
    # First sweep m1.R...
    xaxis(m1.R, lin, 0, 1, 10),
    # ...then L0.P.
    xaxis(L0.P, lin, 10, 100, 20)
)
```

Ports and nodes

Some components define *ports* which act as reference points in the model to which detectors and other components may be connected or certain properties such as beam sizes may be defined. Ports themselves contain *nodes* which provide access to the underlying directions. For a full explanation, see The port and node system (page 139). Some brief examples are shown below:

```
# Spaces connect directly to components' ports.
l l1 P=1
s s1 l1.p1 m1.p1 L=1
m m1 0.99 0.01
s s1 m1.p2 m2.p1 L=1
m m2 0.99 0.01

# Detectors connect to the nodes within ports, allowing the direction to detect to
# be specified.
pd circ m2.p1.i  # Incoming beam at port p1
pd trans m2.p2.o  # Outgoing beam at port p2
```

Paths

Elements and their attributes and parameters can be addressed elsewhere in models using their *path*. Elements are simply referred to by their name; for example, a `mirror` (page 142) named "m1" has path `m1`. Attribute and parameter paths are formed by combining their owner's path with the attribute or parameter name, separated by a `.`; for example, a mirror's *phi* model parameter has path `m1.phi`. Ports and nodes are also addressable this way: the mirror's first port has path `m1.p1` and the incoming node of that port has path `m1.p1.i`.

Commands and analyses do not have names and can thus not be addressed using paths.

Paths are typically useful when specifying spaces, detectors, analyses, and parameter references. Spaces can be used to connect components by referring to the paths of their ports:

```
m m1 R=0.99 T=0.01
m m2 R=0.99 T=0.01
s s1 m1.p2 m2.p1 L=10  # Connect port 2 of m1 to port 1 of m2.
```

Detectors can "look at" a particular port or node in a model:

```
l l1 P=10
m m1 R=0.99 T=0.01
s s1 l1.p1 m1.p1
```

(continues on next page)

(continued from previous page)

```
pd pd1 m1.p1.i  # Look at the power incident on m1 on the port 1 side
pd pd2 m1.p2.o  # Look at the power outgoing from m1 on the port 2 side
ad a m1.p2.o f=0
```

Certain analyses (page 198) can sweep model parameters:

```
l l1 P=1
m m1 R=0.99 T=0.01
m m2 R=0.99 T=0.01
s s1 m1.p2 m2.p1 L=10
xaxis(l1.P, lin, 0, 10, 100)  # Sweep l1's power from 0 to 10 W over 100 points.
```

Expressions and references

Parameters can be given values that contain expressions:

```
m m2 R=0.5 T=0.5 phi=cos(2*pi)
```

When used in element definitions, expressions cannot contain spaces unless they are enclosed in parentheses:

```
# Allowed:
l l2 P=l1.P/10
l l2 P=(l1.P / 10)
l l2 P=cos(l1.P / 10)  # Function arguments can also have whitespace.

# NOT allowed:
# l l2 P=l1.P / 10
```

Supported operators

The following operators are supported:

Operator	Description
`+`	Addition
`-`	Subtraction
`*`	Multiplication
`/`	Division
`//`	Floor division
`**`	Exponentiation

Supported expression functions

The following mathematical functions are supported:

Function	Description
`abs`	Absolute value
`neg`	Negation
`pos`	Positive
`conj`	Conjugate
`real`	Real part
`imag`	Imaginary part
`exp`	Exponential
`sin`	Sine
`arcsin`	Arcsine
`cos`	Cosine
`arccos`	Arccosine
`tan`	Tangent
`arctan2`	(Two-argument) arctangent
`sqrt`	Square root
`radians`	Degrees to radians
`degrees`	Radians to degrees

Functions that require multiple arguments (currently only `arctan2`) should have those arguments separated by a comma:

```
arctan2(1, 0)
```

References

Expressions in KatScript can make references to model parameters using their path (page 128). The value of the target model parameter will then be substituted whenever the expression is evaluated. Since model parameters can themselves be set to expressions, this lets model parameters be linked:

```
m m1 0.99 0.01
m m2 R=m1.R T=m1.T  # m2's R and T parameters take their values from those of m1.
```

References can be included within larger expressions:

```
l l1 P=1
l l2 P=2*l1.P  # Set l2's power to twice that of l1.
```

References can also refer to other parameters within the same element:

```
m m1 R=1-m1.T T=0.01  # Set R to ensure R + T = 1.
```

Model parameter values containing references (either directly or via expressions) get updated on each step of simulations, allowing for a single axis sweep to update many parameters indirectly. This is a powerful feature of KatScript, allowing for compact descriptions of complex models.

Keywords

The following keywords exist in KatScript and serve special purposes. These do not need to be specified in quotes when used as element or function arguments:

`none`
: Used to represent null.

`even`, `odd`, `x`, `y`, `off`
: Used to represent higher order mode sets.

`lin`, `log`
: Used to represent scales.

am, pm
: Used to represent modulation types.

lowpass, highpass, bandpass, bandstop
: Used to represent filter types.

div, gouy, l, length, loss, finesse, fsr, fwhm, pole, modesep, q, g, rc, stability, s, resolution, w, w0, z, r, tau
: Cavity and beam properties.

Arrays

Some parts of the syntax, like the `Rc` parameter of a `mirror` (page 142) or the `modes` (page 197) command, support arrays as inputs. Arrays in KatScript are defined using comma-delimited lists of values contained within square brackets:

```
m m1 R=1 T=0 Rc=[1, 1.1]  # Radius of curvature in tangential (x) and sagittal (y) planes.
```

Arrays can also contain expressions:

```
m m1 R=1 T=0 Rc=[1, 1.1]
m m2 R=1 T=0 Rc=[m1.Rcx, 1.2]
```

Arrays can also contain other arrays. This can be useful for defining the modes to simulate:

```
modes([[1, 0], [0, 1], [2, 0]])  # Simulate only these higher order modes.
```

Generating KatScript from a model

Once you've parsed a script into a model, you can continue to make changes using the Python Interface (page 244). If you wish to then regenerate KatScript for the updated model, you can use `Model.unparse()` or `Model.unparse_file()`. Note that this does not (yet) guarantee to preserve the original parse order.

```
from finesse import Model

model = Model()

# Parse a simple model.
model.parse(
    """
    l l1 P=10
    s s1 l1.p1 m1.p1
    m m1 R=0.995 T=1-m1.R
    s s2 m1.p2 m2.p1 L=10
    m m2 R=0.9999 T=1-m2.R
    pd pd1 m2.p1.i
    pd pd2 m2.p2.o
    """
)

# Change the cavity length.
model.spaces.s2.L = 9

# Unparse the model.
print(model.unparse(warnings=False))
```

```
    l l1 P=10
    s s1 l1.p1 m1.p1
    m m1 R=0.995 T=(1-m1.R) L=(1-(m1.R+m1.T))
    s s2 m1.p2 m2.p1 L=9.0
    m m2 R=0.9999 T=(1-m2.R) L=(1-(m2.R+m2.T))
```

(continues on next page)

(continued from previous page)

```
    pd pd1 m2.p1.i
    pd pd2 m2.p2.o

# Items below could not be matched to original script, or were not present when the model␣
↪was originally parsed.
```

Warning

KatScript generation is an experimental feature and likely contains bugs. It is not (yet!) guaranteed to generate KatScript that parses back to the same model state. Use it with care!

Reviewing the history of parsed files

In larger notebooks or when dealing with multiple files, it may sometimes be useful to review the KatScript files a model is based on. The unparsing (page 131) functionality described above is authoritative, since it is a complete representation of the model. You can use the `finesse.model.Model.print_parsed_files()` to show a quick history:

```
model = Model(loadfile='a.kat')
model.parse_file('b.kat')
model.parse_file('c.kat')
model.print_parsed_files()
```

```
/builds/ifosim/finesse/finesse3/docs/a.kat
/builds/ifosim/finesse/finesse3/docs/b.kat
/builds/ifosim/finesse/finesse3/docs/c.kat
```

The files are also available under the `finesse.model.Model.parsed_files`:

```
model.parsed_files
```

```
(PosixPath('/builds/ifosim/finesse/finesse3/docs/a.kat'),
 PosixPath('/builds/ifosim/finesse/finesse3/docs/b.kat'),
 PosixPath('/builds/ifosim/finesse/finesse3/docs/c.kat'))
```

Warning

Do not use this feature as a replacement for the unparsing (page 131) functionality! The model could also have been changed by parsing strings directly or via the python api. The content of the files listed could also have changed since they were parsed.

Registering custom KatScript directives

New elements, commands and analyses can be registered using the `register_element()`, `register_command()`, and `register_analysis()` methods of `KatSpec`, respectively. These methods require `ItemAdapter` instances as their first argument, which are low-level objects that give a large degree of control over how the new directive should behave. For ordinary elements and analyses the convenience functions `make_element()` and `make_analysis()` are available to simplify the process.

The module variable `finesse.script.spec.KATSPEC` holds an instance of `KatSpec` intended for use in user code to register new elements, commands, and analyses at runtime. Directives registered using this object become

available in the parse/unparse functions and methods throughout FINESSE automatically:

```
from finesse import Model
from finesse.components.nothing import Nothing
from finesse.script.spec import KATSPEC, make_element

# Create a new element type.
class MyElement(Nothing):
    """My custom element."""

# Register a KatScript directive for the new element.
KATSPEC.register_element(make_element(MyElement, "myelement"))

# It's now available in `Model.parse()`:
model = Model()
model.parse("myelement myel1")

print(model.info())
```

```
<finesse.model.Model object at 0x7c001a577e00>

1 optical mode
==============

    - [0 0]

1 component
===========

    - MyElement('myel1')

0 detectors
===========

0 cavities
==========

0 locking loops
===============
```

The registered directive is available for the rest of the current Python kernel's lifetime.

> **Warning**
>
> If you serialize KatScript (page 131) that uses custom directives, before parsing it later you must ensure the directives are again registered with the same arguments.

6.2.2 Replacing KatScript components

For added flexibility when creating or adjusting models, it is possible to replace components in a model. To avoid subtle bugs and difficult to follow code, 'replacing' will always result in a new katscript string that you can use to build a new model. Your existing model will not be touched in any way!

We will first look at the intended usecase of this functionality: extending simplified models with extra complexity. In the example below we prepared a Michelson model (but could also be a full interferometer), with a designated `nothing` component as a 'placeholder'.

```
from finesse import Model
```

(continues on next page)

(continued from previous page)

```
model = Model(
    """
l l1

s LA l1.p1 PLACEHOLDER.p1
nothing PLACEHOLDER
s LB PLACEHOLDER.p2 central_bs.p1

bs central_bs R=0.5 T=0.5
s LW central_bs.p2 WE.p1 L=100
m WE R=0.9 T=0.1
s LN central_bs.p3 NE.p1 L=100
m NE R=0.9 T=0.1
pd pd1 central_bs.p4.o
"""
)
model.plot_graph(network_type="component", layout="dot")
```

```
<IPython.core.display.SVG object>
```

We can insert an entire input mode cleaner by replacing the `nothing` component with some input mode cleaner Katscript.

```
IMC_KATSCRIPT = """
bs bs1 R=0.9 T=0.1
s s1 bs1.p2 bs2.p1
bs bs2 R=0.9 T=0.1
s s2 bs2.p2 bs3.p2
bs bs3 R=0.9 T=0.1
s s3 bs1.p3 bs3.p1
"""

new_kat = model.PLACEHOLDER.replace(IMC_KATSCRIPT, optical_ports=["bs1.p1", "bs3.p3"])
print(new_kat)
```

```
l l1

s LA l1.p1 bs1.p1

bs bs1 R=0.9 T=0.1
s s1 bs1.p2 bs2.p1
bs bs2 R=0.9 T=0.1
s s2 bs2.p2 bs3.p2
bs bs3 R=0.9 T=0.1
s s3 bs1.p3 bs3.p1
s LB bs3.p3 central_bs.p1

bs central_bs R=0.5 T=0.5 L=(1-(central_bs.R+central_bs.T))
s LW central_bs.p2 WE.p1 L=100
m WE R=0.9 T=0.1 L=(1-(WE.R+WE.T))
s LN central_bs.p3 NE.p1 L=100
m NE R=0.9 T=0.1 L=(1-(NE.R+NE.T))
pd pd1 central_bs.p4.o
```

This way you can prepare a simplified 'base' model for your interferometer where you can 'turn on' different parts to model more realistically. The downside to this approach is you will have to split up some of your spaces to insert `nothing` components.

The rest of this page continues with a more detailed explanation of the `replace` functionality. After building your model, the `replace` method is available as an attribute to the component you want to create. Currently you can only replace 1 component (i.e. one katscript line) at a time, but you are allowed to insert as much katscript as you want

```
model = Model(
    """
l laser_1 P=5
"""
)
new_kat = model.laser_1.replace("m m1")
print(new_kat)
```

```
m m1
```

You can see that the method returns a string. You would have to parse this into a new model if you want to use it.

```
new_model = Model(new_kat)
```

You can also call the `replace` method on the `Model` directly, in which case you have to pass the component you want to replace (either as a string or Python object). The advantage is that your editor will recognize the method.

```
model.replace(model.laser_1, "m m1")
```

```
'm m1'
```

```
model.replace("laser_1", "m m1")
```

```
'm m1'
```

Most of the time references to a component are not limited to a single line, since components are always connected to other components. Let's try a more involved example: replacing the first mirror in a cavity with a beamsplitter

```
model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1
s s2 m1.p2 m2.p1
m m2
"""
)
model.plot_dcfields_graph(add_fields=False)
```

```
<IPython.core.display.SVG object>
```

```
PosixPath('/tmp/tmpv99_z2td.svg')
```

```
new_kat = model.m1.replace("bs bs1", optical_ports=["bs1.p1", "bs1.p3"])
print(new_kat)
```

```
l l1
s s1 l1.p1 bs1.p1
bs bs1
s s2 bs1.p3 m2.p1
m m2 R=0.5 T=0.5 L=0.0
```

There are a few things to unpack here. Firstly you can see that for the `m2` component, the default RTL values have been added to the new katscript, even though they were not present in the original katscript. This is because the `replace` method relies on the unparsing functionality, which tries to be as unambiguous as possible.

Secondly, we had to specify some `optical_ports`. This is because the `m1` component appeared in multiple lines, including the lines defining the spaces. If this is the case, you will have to provide a list of strings that will be used as the replacement ports for these optical ports. You only have to supply the ports that actually need replacing.

For instance, in this example we didn't need to provide any replacement for `m1.mech`, since it was not referenced anywhere. Below we will visualize the new model for clarity.

```
new_model = Model(new_kat)
new_model.plot_dcfields_graph(add_fields=False)
```

```
<IPython.core.display.SVG object>
```

```
PosixPath('/tmp/tmppdp7fd7n.svg')
```

The above example will also work when using the `link` command. You will see another artifact from the unparsing: the link command gets unparsed into the explicit space components generated behind the scenes. The autogenerated names will still be based on the old component.

```
model = Model(
    """
l l1
link(l1, m1)
m m1
link(m1, m2)
m m2
"""
)
new_kat = model.m1.replace("bs bs1", optical_ports=["bs1.p1", "bs1.p3"])
print(new_kat)
```

```
l l1

bs bs1

m m2 R=0.5 T=0.5 L=0.0


space l1_p1__m1_p1 portA=l1.p1 portB=bs1.p1
space m1_p2__m2_p1 portA=bs1.p3 portB=m2.p1
```

The ports you are replacing will also automatically replace any nodes associated with that port. This behaviour is currently not configurable. Using the same example as before, we now also have 2 amplitude detectors placed at `m1.p1.i` and `m1.p2.o`. Because we are replacing these ports with `bs1.p1` and `bs1.p3`, the detectors will automatically shift to `bs1.p1.i` and `bs1.p3.o`.

```
model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1
s s2 m1.p2 m2.p1
m m2

ad ad_in m1.p1.i f=0
ad ad_out m1.p2.o f=0
"""
)
new_kat = model.m1.replace("bs bs1", optical_ports=["bs1.p1", "bs1.p3"])
print(new_kat)
```

```
l l1
s s1 l1.p1 bs1.p1
bs bs1
s s2 bs1.p3 m2.p1
```

(continues on next page)

(continued from previous page)

```
m m2 R=0.5 T=0.5 L=0.0

ad ad_in bs1.p1.i f=0
ad ad_out bs1.p3.o f=0
```

Sometimes, the name of the component you are replacing will be mentioned in other lines of katscript. For example, when attaching a `pendulum` to a `mirror`, we use the `connected_to` argument to reference the mirror name. In these cases we need to provide the `replace` function with a `component` argument, which is the name of the new component the `pendulum` needs to be attached to. Let's replace a suspended mirror with a suspended beamsplitter:

```
model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1
pendulum pend1 connected_to=m1 mass=4.0
s s2 m1.p2 m2.p1
m m2

"""
)
new_kat = model.m1.replace(
    "bs bs1", component="bs1", optical_ports=["bs1.p1", "bs1.p3"]
)
print(new_kat)
```

```
l l1
s s1 l1.p1 bs1.p1
bs bs1
pendulum pend1 connected_to=bs1 mass=4.0
s s2 bs1.p3 m2.p1
m m2 R=0.5 T=0.5 L=0.0
```

Finally, we have an example that uses all features at once:

```
model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1
pendulum pend1 connected_to=m1 mass=4.0
link(m1, m2)
m m2
ad ad_circ m1.p2.i f=0
xd m1_motion m1.mech.z
"""
)

new_kat = model.m1.replace(
    "bs bs1",
    component="bs1",
    optical_ports=["bs1.p1", "bs1.p3"],
    mechanical_ports=["bs1.mech"],
    verbose=True,
)
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
<IPython.core.display.HTML object>
```

```
print(new_kat)
```

```
l l1
s s1 l1.p1 bs1.p1
bs bs1
pendulum pend1 connected_to=bs1 mass=4.0

m m2 R=0.5 T=0.5 L=0.0
ad ad_circ bs1.p3.i f=0
xd m1_motion bs1.mech.z


space m1_p2__m2_p1 portA=bs1.p3 portB=m2.p1
```

6.2.3 The port and node system

One of the key structural improvements made to FINESSE is the introduction of a flexible port and node system. This system underpins the code structure of FINESSE as all couplings and interactions are performed via ports and nodes of components.

A `Port` in FINESSE 3 is fundamentally similar to the definition of a node in FINESSE 2; i.e. it represents a point of connections between components in a model. Each port then has single or multiple *nodes*. There are three types of ports/nodes in FINESSE 3, Optical, Electrical, and Mechanical (see `NodeType`) - FINESSE 2 only had optical which limited the type of simulations that were possible. Nodes (`Node`) at each port always have the same type as the port itself. Each node represents a state of the optomechanical system we are modelling. Mathematically this means that each node corresponds to a particular linear equation in the linear set of equations that describe the behaviour of the system we are modelling.

Basic usage of ports and nodes

All components in FINESSE 3 that can be connected together have various `Port` associated with them, which in turn contain `Node` objects representing some optical, or small signal (electrical, or mechanical) state. A port can only contain one type of node.

Each node has a directionality associated with it for the purposes of builing a model and desrcibing the flow of information: input, output, or bidirectional. Input nodes can only be connected to another output node, whereas bidirectional do not care. Every optical port has an input and output node, usually called *i* and *o*. Electrical ports usually just have a single input or output node. Mechanical ports are bidirectional nodes.

Ports and nodes have a standard naming and access convention: *component.port.node*.

Optical Ports are nearly always called something like *p1*, *p2*, …, *pN*. Each optical port always has one input node *i* and one output node *o*. These describe the optical field going into and out of the component. Optical ports between two different elements can only be connected to other optical ports via a *Space* element.

Electrical and Mechanical Ports contain *Signal Nodes*. These nodes represent electronic, mechanical, or other small linearised states in the model. Signal nodes can all be connected together via *Wires* which allow you to form feedback loops. For example, you can use a *Readout* to convert some optical signal into an electronic one, which is then filtered, amplified, and fed back to some mechanical state. These signal states are all small linearised oscillatory states, they do not represent a static effect in the model. These signal nodes can then be used to inject in signals and then measure the corresponding response at another signal node.

Accessing ports

Ports of a component can be accessed via the component through the names that they were assigned during construction. For example, a `Mirror` (page 142) has two ports (*p1* and *p2*) which can be grabbed directly:

```
from finesse.components import Mirror

M1 = Mirror("M1")
print(M1.p1)
print(M1.p2)
```

```
<Port M1.p1 Type=NodeType.OPTICAL @ 0x79cc5ea9c050>
<Port M1.p2 Type=NodeType.OPTICAL @ 0x79cc5ea78910>
```

You can get an simple overview of the ports in a component by using the `port_overview` function.

```
M1.port_overview()
```

```
<finesse.utilities.tables.Table at 0x79cc5ea9c830>
```

And similarly for the nodes by using `node_overview`.

```
M1.node_overview()
```

```
<finesse.utilities.tables.Table at 0x79cc5ea796d0>
```

You can also get a *read-only tuple* of all the ports available at an element using:

```
print(M1.ports)
```

```
(<Port M1.p1 Type=NodeType.OPTICAL @ 0x79cc5ea9c050>, <Port M1.p2 Type=NodeType.OPTICAL @␣
↪0x79cc5ea78910>, <Port M1.mech Type=NodeType.MECHANICAL @ 0x79cc5ea9c2f0>)
```

or all of the optical nodes:

```
print(M1.optical_nodes)
```

```
(<OpticalNode M1.p1.i @ 0x79cc5ea9c1a0>, <OpticalNode M1.p1.o @ 0x79cc5ea78550>,
↪<OpticalNode M1.p2.i @ 0x79cc5ea78a50>, <OpticalNode M1.p2.o @ 0x79cc5ea28510>)
```

or all of the signal nodes:

```
print(M1.signal_nodes)
```

```
(<SignalNode M1.mech.z @ 0x79cc5ea9c440>, <SignalNode M1.mech.yaw @ 0x79cc5ea78b90>,
↪<SignalNode M1.mech.pitch @ 0x79cc5ea78cd0>, <SignalNode M1.mech.F_z @ 0x79cc5ea2a650>,
↪<SignalNode M1.mech.F_yaw @ 0x79cc5ea2a780>, <SignalNode M1.mech.F_pitch @ 0x79cc5ea2db50>)
```

What does a port contain?

In the example code below we show the main properties of a `Port` - namely its type, the nodes that it holds and the component that it is attached to (which is always a `Space` (page 170) between different optical connections):

```
from finesse import Model
from finesse.components import Mirror

M1 = Mirror("M1")
M2 = Mirror("M2")

model = Model()
# connect M1 <-> M2 in a model via a Space of length 1m
model.chain(M1, {"name": "M1_M2", "L":10}, M2)

print(f"Port M1.p2 name = {M1.p2.name}")
print(f"Port M1.p2 type = {M1.p2.type}")
print(f"Port M1.p2 owning component = {M1.p2.component}")
print(f"Port M1.p2 attached component = {M1.p2.attached_to}")
print(f"Port M1.p2 nodes = {M1.p2.nodes}")
```

```
Port M1.p2 name = p2
Port M1.p2 type = NodeType.OPTICAL
Port M1.p2 owning component = Mirror('M1', misaligned=False, ybeta=0.0, xbeta=0.0, Rcy=inf,␣
↪Rcx=inf, phi=0.0, L=0.0, T=0.5, R=0.5)
Port M1.p2 attached component = Space('M1_M2', user_gouy_y=None, user_gouy_x=None, nr=1.0,␣
↪L=10.0)
Port M1.p2 nodes = (<OpticalNode M1.p2.i @ 0x7a143ae90a50>, <OpticalNode M1.p2.o @␣
↪0x7a143ae3c510>)
```

Accessing nodes — via a component

Nodes can be accessed directly through components or via the `Model` instance that their owning component is associated with (see next section for details). To access all the nodes of a component:

```
print(f"All nodes of M1 = {M1.nodes}")
```

```
All nodes of M1 = OrderedDict({'M1.p1.i': <OpticalNode M1.p1.i @ 0x7a143aeb41a0>, 'M1.p1.o':
↪<OpticalNode M1.p1.o @ 0x7a143ae90550>, 'M1.p2.i': <OpticalNode M1.p2.i @ 0x7a143ae90a50>,
↪'M1.p2.o': <OpticalNode M1.p2.o @ 0x7a143ae3c510>, 'M1.mech.z': <SignalNode M1.mech.z @␣
↪0x7a143aeb4440>, 'M1.mech.yaw': <SignalNode M1.mech.yaw @ 0x7a143ae90b90>, 'M1.mech.pitch
↪': <SignalNode M1.mech.pitch @ 0x7a143ae90cd0>, 'M1.mech.F_z': <SignalNode M1.mech.F_z @␣
↪0x7a143ae3e650>, 'M1.mech.F_yaw': <SignalNode M1.mech.F_yaw @ 0x7a143ae3e780>, 'M1.mech.F_
↪pitch': <SignalNode M1.mech.F_pitch @ 0x7a143ae49b50>})
```

Or all the optical nodes:

```
print(f"Optical nodes of M1 = {M1.optical_nodes}")
```

```
Optical nodes of M1 = (<OpticalNode M1.p1.i @ 0x7a143aeb41a0>, <OpticalNode M1.p1.o @␣
↪0x7a143ae90550>, <OpticalNode M1.p2.i @ 0x7a143ae90a50>, <OpticalNode M1.p2.o @␣
↪0x7a143ae3c510>)
```

Get a single optical node of a component by its *direction*:

```
print(f"Input node of port M1.n1 = {M1.p1.i}")
print(f"Output node of port M1.n1 = {M1.p1.o}")
```

```
Input node of port M1.n1 = <OpticalNode M1.p1.i @ 0x7a143aeb41a0>
Output node of port M1.n1 = <OpticalNode M1.p1.o @ 0x7a143ae90550>
```

Accessing nodes — via the model

Nodes play an important role in the `Model` class as the `Node.full_name` property forms the *node_type* of the underlying directed graph object (stored in `Model.network`). Thus, we use Node instances to report on graph data as well as perform operations on this graph - see the networkx DiGraph documentation for details on these methods and attributes.

You can also access all the nodes of a given `NodeType` in a Model instance with (e.g. for optical nodes):

```
print(f"All optical nodes in model: {model.optical_nodes}")
```

```
All optical nodes in model: [<OpticalNode M1.p1.i @ 0x7a143aeb41a0>, <OpticalNode M1.p1.o @␣
↪0x7a143ae90550>, <OpticalNode M1.p2.i @ 0x7a143ae90a50>, <OpticalNode M1.p2.o @␣
↪0x7a143ae3c510>, <OpticalNode M2.p1.i @ 0x7a143ae3eb10>, <OpticalNode M2.p1.o @␣
↪0x7a143ae49d90>, <OpticalNode M2.p2.i @ 0x7a143ae37ac0>, <OpticalNode M2.p2.o @␣
↪0x7a143ae37bd0>]
```

Node names are used as keys in the network of a Model to get data on the node itself and edges connected to the node:

```
print(model.network.nodes[model.M2.p1.i.full_name])
print(model.network[model.M2.p1.i.full_name][model.M2.p1.o.full_name])
```

```
{'weakref': <weakref at 0x7a143aecec50; to 'finesse.components.node.OpticalNode' at␣
↪0x7a143ae3eb10>, 'owner': <weakref at 0x7a143ae75f30; to 'finesse.components.mirror.Mirror
↪' at 0x7a143ae91090>, 'type': <NodeType.OPTICAL: 0>, 'optical': True}
{'name': 'P1i_P1o', 'in_ref': <weakref at 0x7a143aecec50; to 'finesse.components.node.
↪OpticalNode' at 0x7a143ae3eb10>, 'out_ref': <weakref at 0x7a143aececa0; to 'finesse.
↪components.node.OpticalNode' at 0x7a143ae49d90>, 'owner': <weakref at 0x7a143ae75f30; to
↪'finesse.components.mirror.Mirror' at 0x7a143ae91090>, 'length': 1, 'coupling_type':
↪<CouplingType.OPTICAL_TO_OPTICAL: 0>, 'internal': False}
```

6.2.4 Model elements

A model in FINESSE is built up of multiple elements. Each element represents some real-world components such as mirrors, lenses, photodetectors, suspensions, control filters, etc. This section contains descriptions of each element the physics behind FINESSE, with equations provided that match the expressions used within the code itself.

Every model element has a *name* given to it. These can all be referenced in KatScript and through the Model Python object.

Components and detectors

Components and detectors are model elements that can be connected together to build your model.

Optics

Common optical building blocks are provided for building up an optical system. Some have more detailed pages which are listed in the table of contents below.

Mirror

- Component Graph (page 143)
- Phase relationship (page 144)
- RTL relationship (page 144)
  - Three numeric values (page 144)
  - Two numeric values (page 145)
  - Custom symbolic expressions (page 146)
- Parameters (page 146)
- Coordinate systems (page 146)
- Signals (page 147)
  - Longitudinal motions (page 147)
  - Yaw and pitch rotations (page 148)
- Ports and nodes overview (page 148)
  - Ports (page 148)
  - Nodes (page 148)
  - Couplings (page 148)
- Scattering matrix application order (page 148)

`mirror`
`m`
`Mirror`

The mirror component represents a thin dielectric surface with associated properties such as reflectivity, tuning, and radius of curvature. Mirror components are nominally at normal incidence to the beams. It has two optical ports *p1* and *p2* which describes the two beams incident on either side of this surface. The surface normal points out of the mirror on the p1 side. A mirror also has a mechanical port *mech* which has nodes for longitudinal, yaw, and pitch motions. These mechanical nodes are purely for exciting small signal oscillations of the mirror. Static offsets in longitudinal displacements are set by the *phi* parameter (in units of degrees), yaw by the *xbeta* parameter, and pitch the *ybeta* parameter.

Syntax

```
m name R=none T=none L=none phi=0 Rc=inf xbeta=0 ybeta=0 misaligned=false␣
↪imaginary_transmission=true
```

Required

`name`: Name of newly created mirror.

Optional

`R`: Reflectivity of the mirror; defaults to 0.5.

`T`: Transmittance of the mirror; defaults to 0.5.

`L`: Loss of the mirror; defaults to 0.0.

`phi`: Tuning of the mirror (in degrees); defaults to 0.0.

`Rc`: The radius of curvature of the mirror (in metres); defaults to a flat mirror (*Rc=np.inf*). Astigmatic mirrors can also be set with *Rc=(Rcx, Rcy)*. A positive value results in a concave mirror on the p1 side of the mirror.

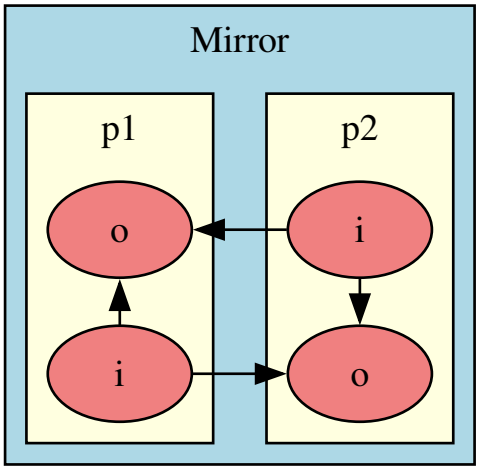
`xbeta`, `ybeta`: Misalignment of the mirror in yaw and pitch in units of radians

`misaligned`: When True the mirror will be significantly misaligned and assumes any reflected beam is dumped. Transmissions will still occur.

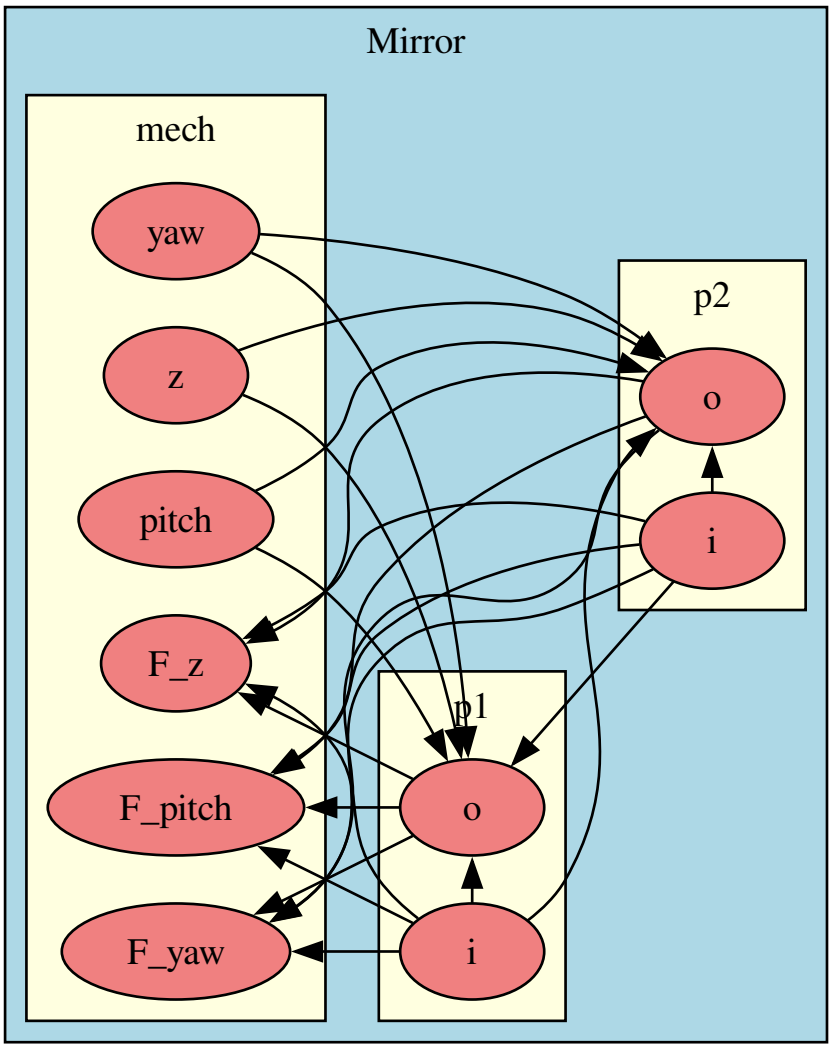
`imaginary_transmission`: Convention for the transmission and reflection reciprocity relations. 'True' uses imaginary transmission and equal real reflection from both sides 'False' uses real transmission, negative reflection from side 1, and positive reflection from side 2. defaults to True

Component Graph

Optical



Full



Phase relationship

See Mirror and beamsplitter phase relationships (page 113) for more details on the phase relationship to ensure energy conservation that is used in FINESSE.

The transmission and reflection phase relationship can be changed per mirror by setting the `imaginary_transmission` to either real or imaginary. By default this is imaginary (True), so a i phase on transmission. If set to be the real convention then the phase is 0 on transmission and 180 degrees on reflection on the port 1 side.

RTL relationship

Energy conservation dictates that the `R`, `T` and `L` parameters should always sum up to 1. There are multiple ways to set these parameters:

Three numeric values

You can assign a numeric value to each parameter. They will be checked on construction of the mirror and any time you update any of the parameters:

```
from finesse.components import Mirror
```

(continues on next page)

(continued from previous page)

```
# invalid construction
mirror = Mirror('m1', R=0.9, T=0.3, L=0.1)
```

```
InvalidRTLError:          (use finesse.tb() to see the full traceback)
Expected R + T + L = 1 in 'm1' but got R + T + L = 1.3
R = 0.9
T = 0.3
L = 0.1
```

```
from finesse.components import Mirror

# correct construction
mirror = Mirror('m1', R=0.6, T=0.3, L=0.1)
# updating value breaks energy conservation
mirror.R = 0.9
```

```
InvalidRTLError:          (use finesse.tb() to see the full traceback)
Expected R + T + L = 1 in 'm1' but got R + T + L = 1.3
R = 0.9
T = 0.3
L = 0.1
```

Since you didn't specify how the other two parameters should respond to a change, you can no longer update the values one at a time. Instead you should use the `finesse.components.surface.Surface.set_RTL()` method to update all three parameters at once:

```
mirror = Mirror('m1', R=0.6, T=0.3, L=0.1)
mirror.set_RTL(R=0.1, T=0.9, L=0.0)
```

Two numeric values

If you only specify two numeric values, for instance `R` and `T`, the third parameter will automatically be set to a symbolic expression (page 238) that satisfies the energy conservation, for instance $L = 1 - R - T$. This allows you to change/sweep any of the parameters you set numerically and the third parameter will update automatically:

```
mirror = Mirror('m1', R=0.2, T=0.3)

print(f"R={mirror.R.value}")
print(f"T={mirror.T.value}")
print(f"L={mirror.L.value}={mirror.L.value.eval()}")

mirror.R = 0.6

print(f"R={mirror.R.value}")
print(f"T={mirror.T.value}")
print(f"L={mirror.L.value}={mirror.L.value.eval()}")
```

```
R=0.2
T=0.3
L=(1-(m1.R+m1.T))=0.5
R=0.6
T=0.3
L=(1-(m1.R+m1.T))=0.10000000000000009
```

Custom symbolic expressions

You can also specify your own symbolic expressions (page 238). An advanced example could be to link the parameters of two mirrors together, such that any change in one mirror is propagated to the second mirror:

```
m1 = Mirror('m1', R=0.2, T=0.3)
m2 = Mirror('m2', R=m1.R.ref, T=m1.T.ref)

print(f"{m1.R.value=}")
print(f"{m2.R.value=}")
print(f"{m2.R.value.eval()=}")

m1.R = 0.6

print(f"{m1.R.value=}")
print(f"{m2.R.value=}")
print(f"{m2.R.value.eval()=}")
```

```
m1.R.value=0.2
m2.R.value=<Symbolic='m1.R' @ 0x7ed018aaae50>
m2.R.value.eval()=0.2
m1.R.value=0.6
m2.R.value=<Symbolic='m1.R' @ 0x7ed018aaae50>
m2.R.value.eval()=0.6
```

Parameters

Listed below are the parameters of the mirror component. Certain parameters can be changed during a simulation and some cannot, which is highlighted in the *can change during simulation* column. These changeable parameters can be used by actions such as `xaxis` or `change`. Those that cannot must be changed before a simulation is run.

```
<IPython.core.display.HTML object>
```

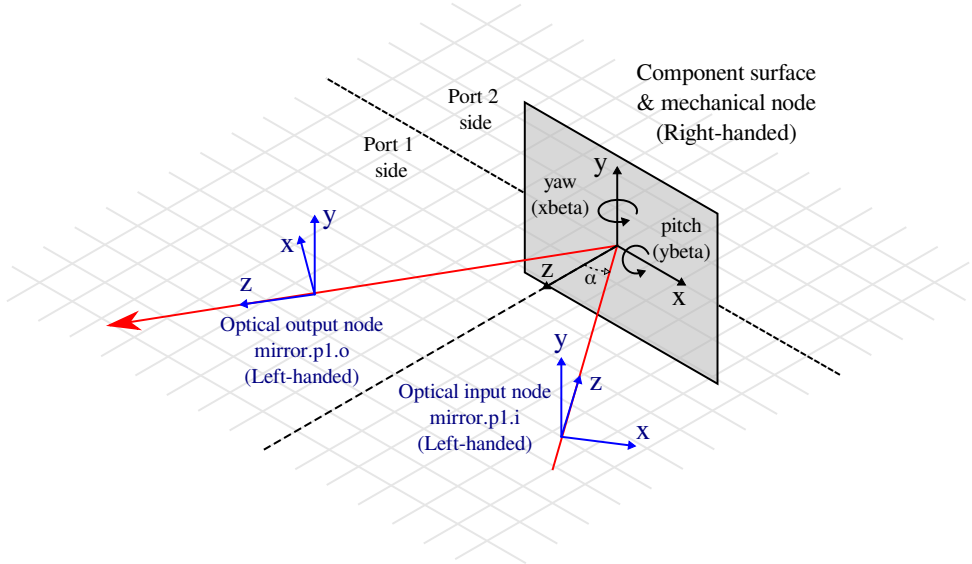
Coordinate systems

There are various coordinate systems involved in modelling the mirror component. Each mechanical and optical node has an associated coordinate system. Shown in the figure below are the coordinate systems for the optical `p1` (Port 1) and the mechanical mode `mech`.

The optical nodes, as for all optical nodes for any component, are in a left-handed coordinate system, shown in blue. The incoming and output going nodes represent the beam in the incident plane. For a mirror the angle of incidence is fixed to zero, $\alpha = 0$. A beamsplitter component is exactly the same as a mirror component, except that the angle of incidence can be non-zero. The outgoing node is a reflection of the incoming in the z and x coordinates.



Fig. 6.5: Andreas and Daniel applying a left-handed coordiate system.



The mechanical node coordiante system is shown in black and is a right-handed coordinate system. A positive z motion is the surface normal of mirror surface going in the port 1 direction. Yaw and pitch are right-handed rotations around the y and x axes respectively.

Signals

Signals can be excited by injecting a signal into the relevant electrical or mechanical node at a component. The mirror component only has mechanical signals.

Longitudinal motions

Small longitudinal oscillations can be excited directly by using the `mirror.mech.z` (whose units are meters) or by applying a force `mirror.mech.F_z` (units of Newtons). If no suspension is attached to the mirror then applying a force will not move the mirror.

When the longitudinal motion is driven any carrier light reflected from the mirror surface will be phase modulated. This creates upper and lower signal sidebands around each carrier. Mathematically, the amount of sideband generated,

$a_\pm$, is linearly proportional to induced motion, z. The higher order mode vector for the upper or lower sideband generated at the outgoing node of port 1 will be

$$\hat{a}_\pm = -\mathrm{i}\, k_\pm r z^\pm \hat{E}_c^i \exp\left(\mathrm{i}\,\frac{f_\mathrm{sig}}{f_0}\phi\right)$$

Where $k_\pm$ is the sideband wavenumber, f_0 is the reference optical frequency which can be set using the `lambda` (page 196) command and defaults to $f_0 = 1064$nm, $\hat{E}_c^i$ is a vector of carrier higher order modes, r the mirror reflectivity, and ϕ the mirror tuning in radians. $z^\pm$ is defined as $z^+ \equiv z$ and $z^- \equiv z^*$. Sidebands generated from reflection on the port 2 side will have a 180 degree phase shift relative to those generated on the port 1 side.

Yaw and pitch rotations

Small rotational oscillations can be excited by using the `mirror.mech.yaw` and `mirror.mech.pitch` both in units of radians. Torques can also be applied by using the `mirror.mech.F_yaw` or `mirror.mech.F_pitch`. If no suspension is attached to the mirror then applying a force will not rotate the mirror.

When either yaw or pitch is driven any carrier light reflected from the surfaces has a linear phase shifted applied to its wavefront, thus tilting it.

$$\hat{a}_\pm = -\mathrm{i}\, k_\pm r \beta^\pm \mathbf{K}_{y/p} \hat{E}_c^i \exp\left(\mathrm{i}\,\frac{f_\mathrm{sig}}{f_0}\phi\right)$$

The terms are the same as for the longitudinal equation except for $\beta^\pm$ which is the misalignment equivalent of $z^\pm$ for either pitch or yaw, and $\mathbf{K}_{y/p}$ which is the mode scattering matrix for pitch or yaw for some linearised unit radian alignment change.

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7ed018df3610>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7ed018df2e90>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7ed018bfa9e0>
```

Scattering matrix application order

If the `mirror` (page 142) has a surface map applied and there is a mode mismatch across one of its connections, the `knm_apply_bayer_helms_first` attribute defines the order of the multiplication in which these contributions are combined. See How Bayer-Helms and map contributions are combined (page 361).

Beamsplitter

- Component Graph (page 149)
- RTL relationship (page 152)
  - Phase relationship (page 152)
  - Ports (page 152)

  - Parameters (page 152)
- Ports and nodes overview (page 152)
  - Ports (page 152)
  - Nodes (page 153)
  - Couplings (page 153)
- Scattering matrix application order (page 153)

beamsplitter
bs
Beamsplitter

The beamsplitter component represents a thin dielectric surface with associated properties such as reflectivity, tuning, and radius of curvature. It has four optical ports p1, p2, p3, and p4 which describe the four beams incident on either side of this surface. p1 and p2 are on side 1 and p3 and p4 are on side 2. A 100% transmissive beamsplitter will transmit all of the light incident at p1 to p3.

It also has a mechanical port *mech* which has nodes for longitudinal, yaw, and pitch motions. These mechanical nodes are purely for exciting small signal oscillations of the mirror. Static offsets in longitudinal displacements are set by the *phi* parameter (in units of degrees), misalignments in yaw by the *xbeta* parameter, and pitch the *ybeta* parameter. Macroscopic angle of incidence of the beamsplitter is set by the `alpha` parameter. Beamsplitters physically operate the same as mirror components, except for the non-normal angle of incidence option. See RTL relationship (page 144) for more information on how the RTL relationship is handled.

Syntax

```
bs name R=none T=none L=none phi=0 alpha=0 Rc=inf xbeta=0 ybeta=0 plane=0␣
↪misaligned=false imaginary_transmission=true
```

Required

`name`: Name of newly created beamsplitter.

Optional

`R`: Reflectivity of the beamsplitter.

`T`: Transmissivity of the beamsplitter.

`L`: Loss of the beamsplitter.

`phi`: Microscopic tuning of the beamsplitter (in degrees).

`alpha`: Angle of incidence (in degrees)

`Rc`: Radius of curvature (in metres); defaults to `numpy.inf` to indicate a planar surface.

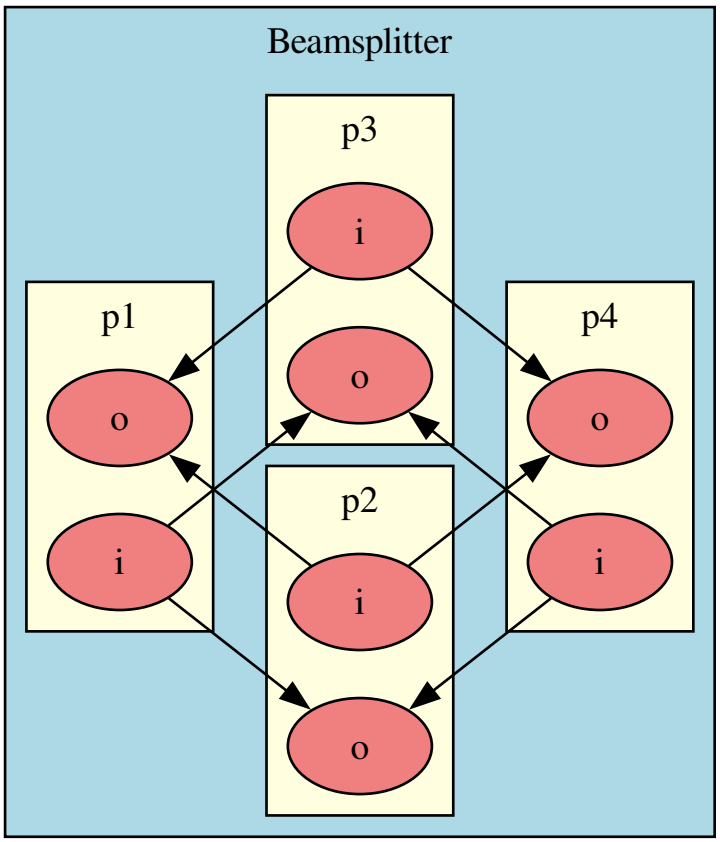
`xbeta`, `ybeta`: Angle of misalignment in the yaw plane (xbeta) and pitch (ybeta), respectively (in radians); defaults to *0*.

`plane`: Plane of incidence, either 'xz' or 'yz'. Defaults to 'xz'.

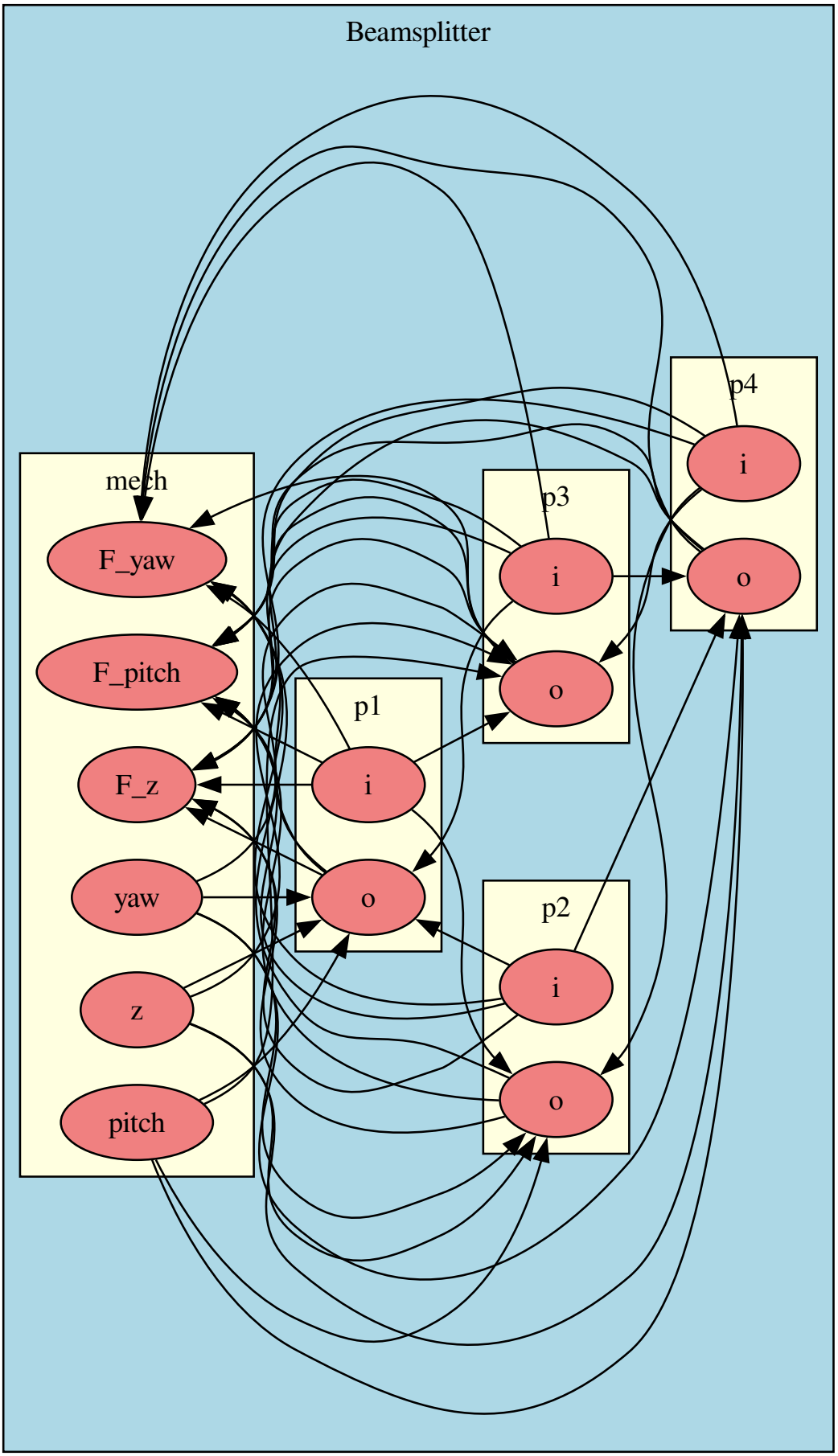
`misaligned`: When True the beamsplitter will be significantly misaligned and assumes any reflected beam is dumped. Transmissions will still occur.

Component Graph

Optical



Full


Beamsplitter
mech
F_yaw
F_pitch
F_z
yaw
z
pitch
p1
i
o
p2
i
o
p3
i
o
p4
i
o

RTL relationship

See RTL relationship (page 144)

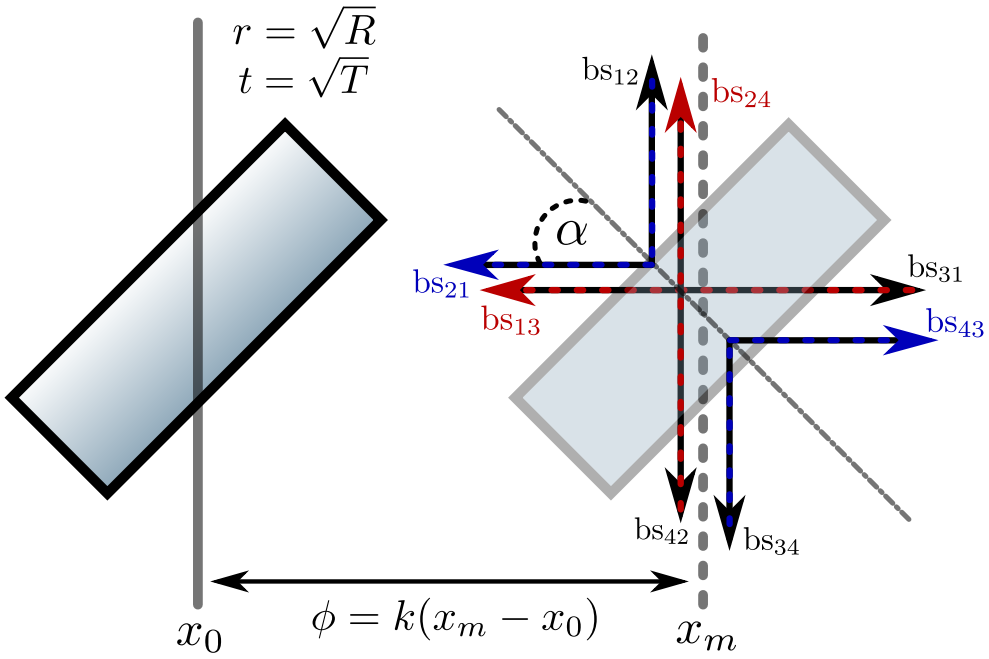
Phase relationship

See Mirror and beamsplitter phase relationships (page 113) for more details on the phase relationship to ensure energy conservation that is used in FINESSE.

The transmission and reflection phase relationship can be changed per mirror by setting the `imaginary_transmission` to either real or imaginary. By default this is imaginary (True), so a i phase on transmission. If set to be the real convention then the phase is 0 on transmission and 180 degrees on reflection on the port 1 side.

Ports

The order of the ports is shown below. In a typical Michelson interferometer the order of the ports is: input-reflection-transmission-output.



Parameters

Listed below are the parameters of the beamsplitter component. Certain parameters can be changed during a simulation and some cannot, which is highlighted in the *can change during simulation* column. These changeable parameters can be used by actions such as `xaxis` or `change`. Those that cannot must be changed before a simulation is run.

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7be4ec7e1450>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7be4ec7e1950>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7be4ec78efd0>
```

Scattering matrix application order

If the `beamsplitter` (page 149) has a surface map applied and there is a mode mismatch across one of its connections, the `knm_apply_bayer_helms_first` attribute defines the order of the multiplication in which these contributions are combined. See How Bayer-Helms and map contributions are combined (page 361).

- Directional Beamsplitter (page 153)
- Isolator (page 155)
- Lens (page 156)
- Scattering matrix application order (page 158)
- AstigmaticLens (page 158)
- Modulator (page 159)
- Optical Bandpass (page 161)
- Frequency Loss (page 162)

Directional Beamsplitter

directional_beamsplitter
dbs
DirectionalBeamsplitter

Represents a directional beamsplitter optical component. Connections made between ports:

- port 1 to port 3 * port 3 to port 4 * port 4 to port 2 * port 2 to port 1

Syntax

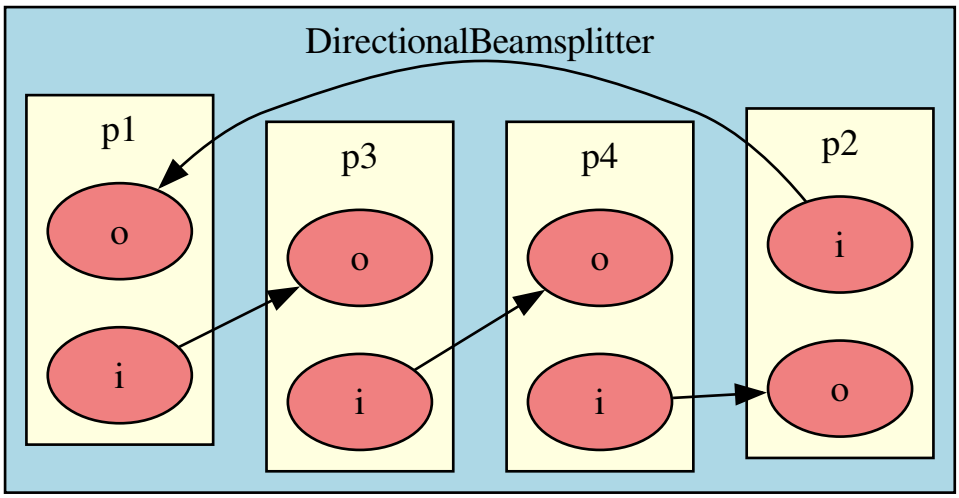
```
dbs name
```

Required

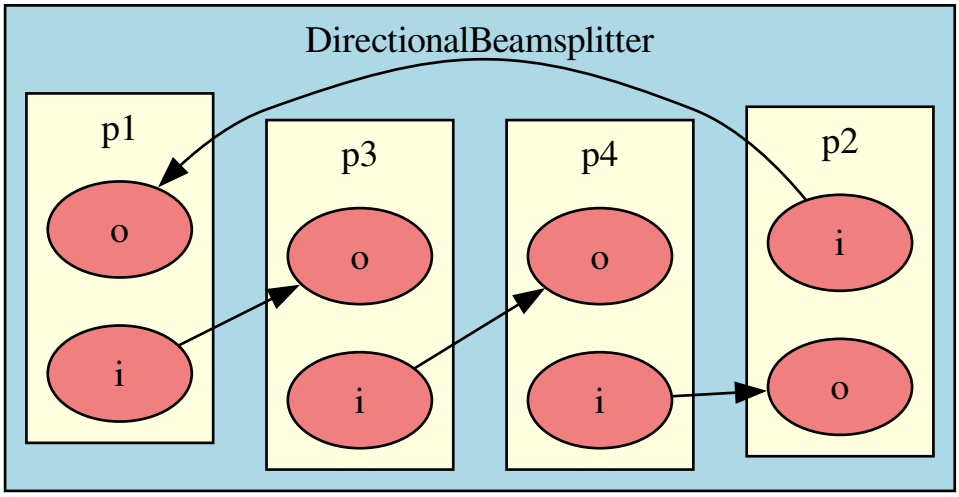
`name`: Name of newly created directional beamsplitter.

Component Graph

Optical



Full



Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70e0c2f0>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70df8f50>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70df9450>
```

Isolator

isolator
isol
Isolator

Represents an isolator optical component with a suppression factor. Suppresses the light field transmitted from p2 to p1. The field from p1 to p2 is.

Syntax

```
isol name S=0.0
```

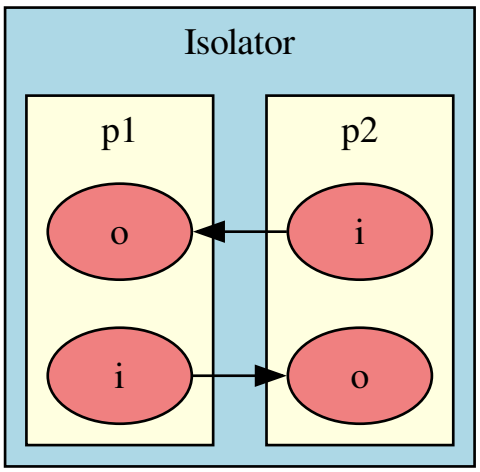
Required

`name`: Name of newly created isolator.

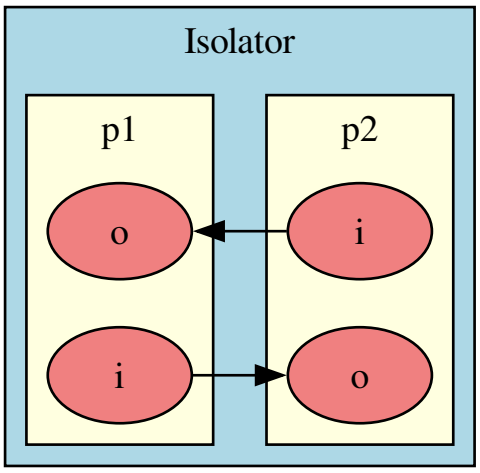
`S`: Power suppression in dB. Defaults to 0

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70da6780>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70daac30>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70d7bbd0>
```

Lens

lens
Lens

Represents a thin lens optical component with an associated focal length.

Syntax

```
lens name f=inf
```

Required

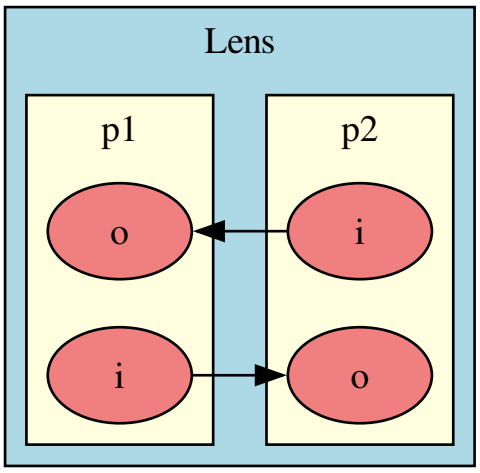
`name`: Name of newly created lens.

Optional

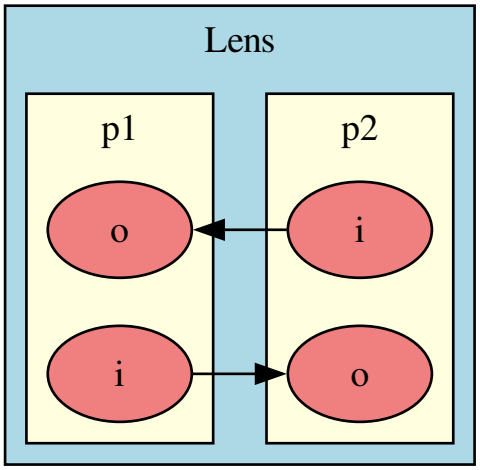
`f`: Focal length of the lens in metres; defaults to infinity.

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70d6c750>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70d6d150>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780ca2d90e60>
```

Scattering matrix application order

If the `lens` (page 156) has a surface map applied and there is a mode mismatch across one of its connections, the `knm_apply_bayer_helms_first` attribute defines the order of the multiplication in which these contributions are combined. See How Bayer-Helms and map contributions are combined (page 361).

AstigmaticLens

alens
AstigmaticLens

Represents a thin astigmatic lens optical component with an associated focal lengths.

Syntax

```
alens name fx=inf fy=inf
```

Required

`name`: Name of newly created lens.

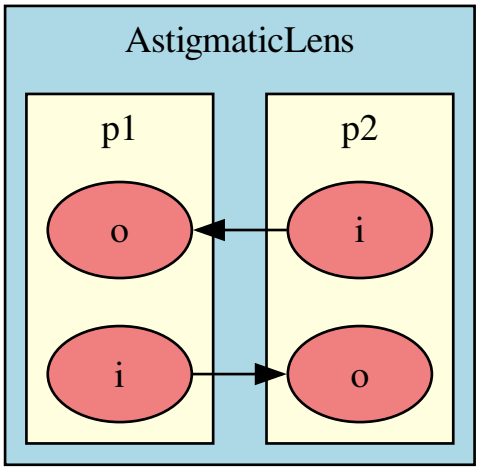
Optional

`fx`: Focal length in x-z plane of the lens in metres; defaults to infinity.

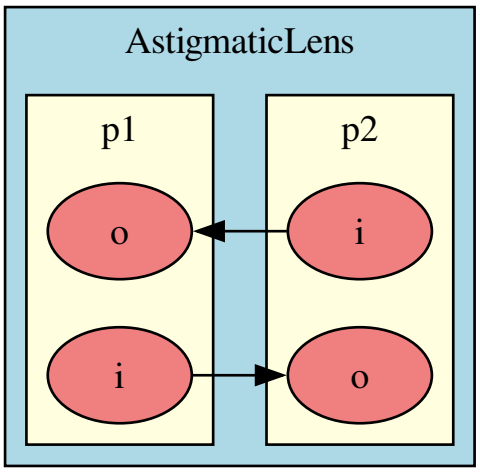
`fy`: Focal length of y-z plane the lens in metres; defaults to infinity.

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70e18210>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70e18f30>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780ca2838940>
```

Modulator

modulator
mod
Modulator

Represents a modulator optical component with associated properties such as modulation frequency, index and order.

Syntax

```
mod name f midx order=1 mod_type=pm phase=0.0 positive_only=false
```

Required

`name`: Name of newly created modulator.

`midx`: Modulation index, >= 0.

Optional

`f`: Frequency of the modulation (in Hz) or `Frequency` object.

`order`: Maximum order of modulations to produce. Must be 1 for amplitude modulation. Defaults to 1.

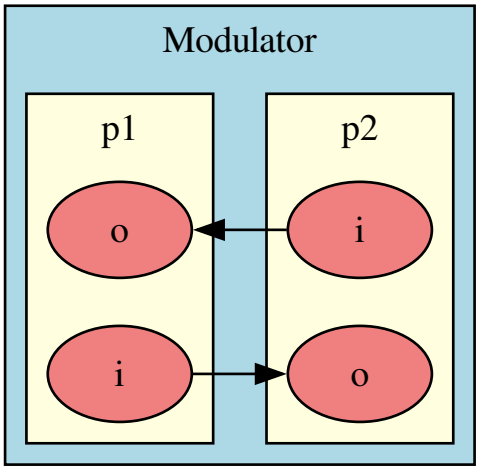
`mod_type`: Modulation type: - 'am' (amplitude modulation (page 118)) - 'pm' (phase modulation (page 117)) - 'yaw' (alignment modulation (page 119)) (EXPERIMENTAL) - 'pitch' (alignment modulation (page 119)) (EXPERIMENTAL) Defaults to 'pm'.

`phase`: Relative phase of modulation (in degrees). Defaults to 0.0.

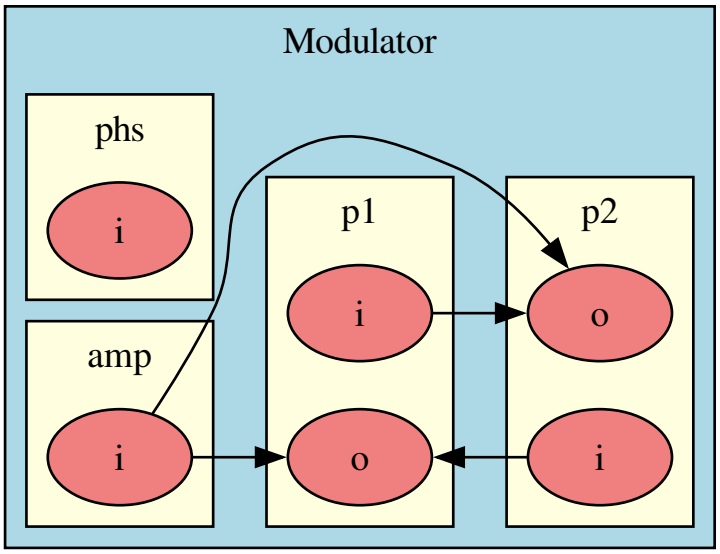
`positive_only`: If True, only produce positive-frequency sidebands. Defaults to False.

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70d89010>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70e205d0>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70e207e0>
```

Optical Bandpass

optical_bandpass
obp
OpticalBandpassFilter

An idealised optical bandpass filter that will transmit an optical frequency around some central frequency with a 3dB bandwidth.

Syntax

```
obp name fc=0 bandwidth=1000 filter_hom=none
```

Required

`name`: Name of element

`fc`: Central frequency

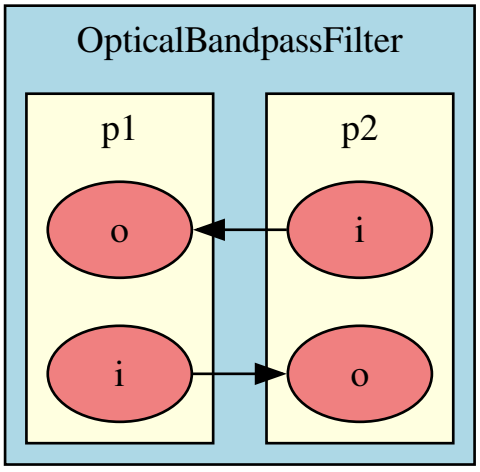
`bandwidth`: Filter 3dB bandwidth

Optional

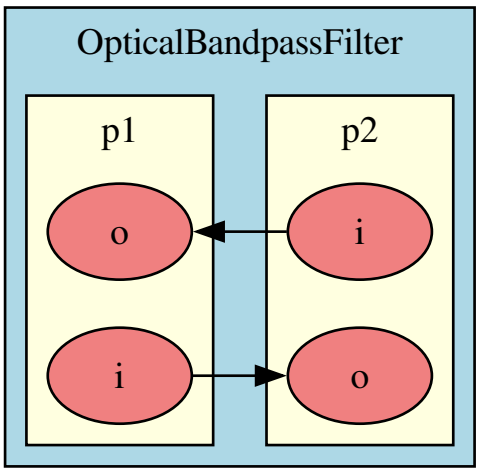
`filter_hom`: Individual higher order modes can be filtered and transmitted by setting this to a tuple of (n, m). If `None` then no mode filtering is done. This cannot be changed during a simulation.

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70e01bd0>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70dad490>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70dfdc50>
```

Frequency Loss

floss
FrequencyLoss

Represents an unphysical element which introduces a loss and/or phase for a particular frequency.

Syntax

```
floss name f loss=0 phase=0
```

Required

`name`: Name of newly created lens.

Optional

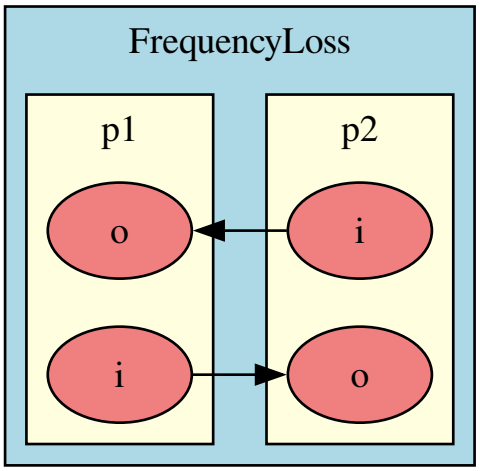
`f`: Frequency to apply loss and phase to.

`loss`: Fractional loss at the frequency

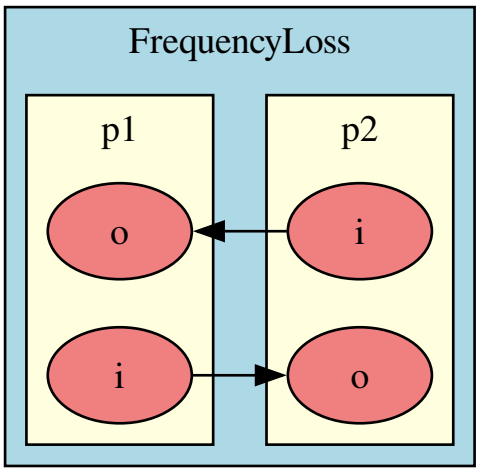
`phase`: Phase change at the frequency

Component Graph

Optical



Full



Parameters

```
<IPython.core.display.HTML object>
```

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70dfeb50>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70dfe550>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70dfcdd0>
```

Telescope

telescope
tel
Telescope

This component represents a perfectly matching and adapting telescope. It will not change the beam parameter on transmission of the component and any field coming in will be transmitted to the other side into the other beam parameter. This should be used in cases when you want to connect to optical systems but do not want to design a complicated optical telescope system to mode-match them both, or in cases where one system has a changing parameter but you don't care about the details of the telescope.

No beam is traced through the telescope so you must specify a beam parameter or a cavity on both sides of the telescope. If not, a tracing error will occur. Currently no losses, mismatch, or misalignments can be induced at this telescope. There is no accumulated Gouy phase or plane-wave propagation phase.

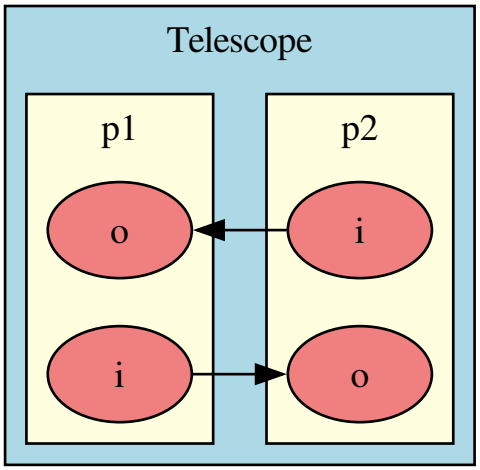
Syntax

```
tel name
```

Required

`name`: Name of newly created telescope

Component Graph



Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x780c70dfeed0>
```

Nodes

```
<finesse.utilities.tables.Table at 0x780c70dfca50>
```

Couplings

```
<finesse.utilities.tables.Table at 0x780c70dfead0>
```

Sources

Sources are components which generate some kind of input, such as a laser which generates some optical field, or a signal generator which produces some electronic signal.

- Laser (page 165)
  - Component Graph (page 166)
  - Ports and nodes overview (page 168)
- Squeezer (page 168)
  - Component Graph (page 168)
  - Ports and nodes overview (page 169)
- Signal generator (page 169)
  - Ports and nodes overview (page 170)
- Noise source (page 170)

Laser

laser
l
Laser

Represents a laser producing a beam with associated properties such as power and frequency.

Syntax

```
l name P=1 f=0 phase=0 signals_only=false
```

Required

`name`: Name of the newly created laser.

Optional

`P`: Power of the laser (in Watts), defaults to 1 W.

`f`: Frequency-offset of the laser from the default (in Hz) or `Frequency` object. Defaults to 0 Hz offset.

`phase`: Phase-offset of the laser from the default, defaults to zero.

`signals_only`: When True, this laser component will only inject signal sidebands. They will use the current carrier value as a scaling terms but the carrier will not be injected into the simulation. This allows a user to just inject signal sidebands into a model.

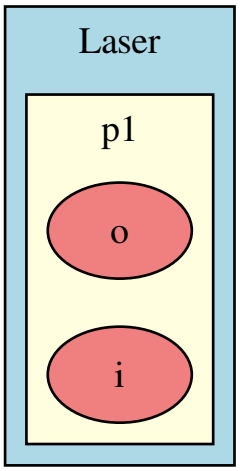
Example

Create a 10 W laser at the carrier frequency with zero phase:

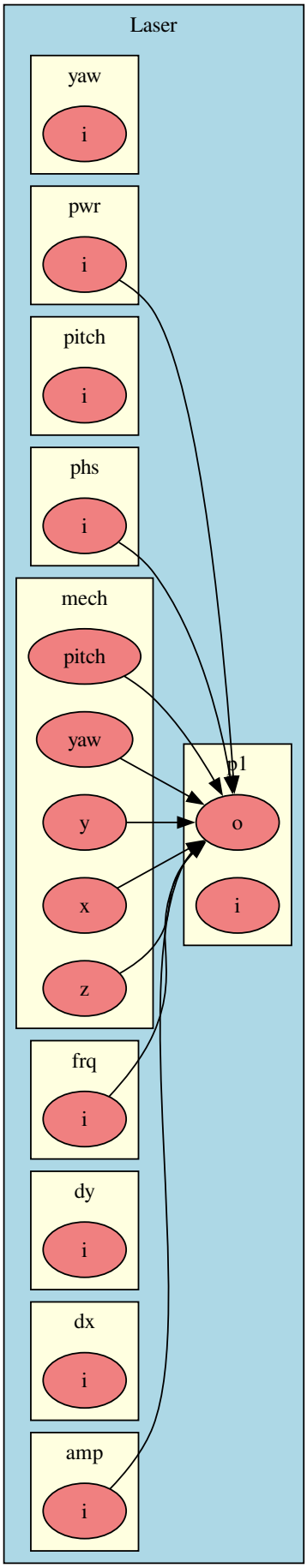
```
laser l1 P=10 f=0 phase=0
```

Component Graph

Optical



Full


Laser
yaw
i
pwr
i
pitch
i
phs
i
mech
pitch
yaw
y
x
z
p1
o
i
frq
i
dy
i
dx
i
amp
i

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7aab59fdc830>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7aab59fc9310>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7aab59fc9810>
```

Squeezer

squeezer
sq
Squeezer

Represents a squeezer producing a squeezed-light beam with a given squeezing in decibels and angle. The upper and conjugate of the lower sidebands can be excited in a signal analysis by injecting into the *.upper* and *.lower_conj* signal nodes.

Syntax

```
sq name db f=0 angle=0
```

Required

`name`: Name of the newly created squeezer.

`db`: Squeezing factor (in amplitude decibels). 6dB gives a factor of 2 reduction in noise.

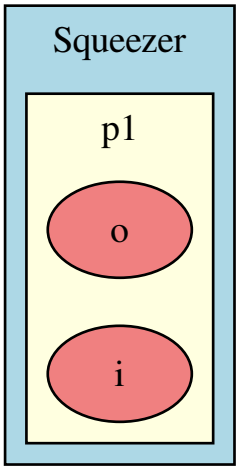
Optional

`f`: Frequency-offset of the squeezer from the default (in Hz) or `Frequency` object. Defaults to 0 Hz offset.

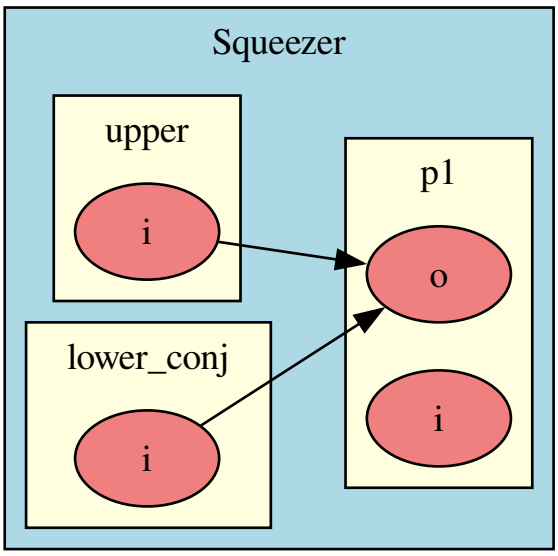
`angle`: Squeezing angle (in degrees). Defaults to zero.

Component Graph

Optical



Full



Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7aab59f76b10>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7aab59f768b0>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7aab59f7da30>
```

Signal generator

signal_generator
sgen
SignalGenerator

Represents a signal generator which produces a signal with a given amplitude and phase.

Syntax

```
sgen name node amplitude=1 phase=0
```

Required

`name`: Name of newly created signal generator.

`node`: A node to inject a signal into.

Optional

`amplitude`: Amplitude of the signal in volts.

`phase`: Phase-offset of the signal from the default in degrees, defaults to zero.

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7aab5a006030>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7aab5a006360>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7aab59dce250>
```

Noise source

noise
ClassicalNoise

Syntax

```
noise name node ASD
```

Connectors

- Space (page 170)
  - Ports and nodes overview (page 171)
- Nothing (page 171)
  - Ports and nodes overview (page 172)
- Wire (page 172)

Space

space
s
Space

Represents a space between two components in the interferometer configuration, with a given length and index of refraction.

There can be many spaces in a model which are not of much interest and do not need to be referred to. For example, the `link` command will create spaces between components without giving an explicit name, just specifying a distance between them. All Space elements are added to the Model's `.spaces` namespace so they can all easily be iterated over. You can also find spaces which had no name specified and thus have an autogenerated name. User specified Space's with names will be put in the main Model namespace so that they can be accessed like any other element in the model.

Syntax

```
s name portA portB L=0.0 nr=1.0 user_gouy_x=none user_gouy_y=none
```

Required

`portA`, `portB`: Ports to connect.

Optional

`name`: Name of newly created space. If not specified, a name is automatically generated.

`L`: Geometric length of newly created `Space` (page 170) instance; defaults to 0.

`nr`: Index of refraction of newly created `Space` (page 170) instance; defaults to 1.0.

`user_gouy_x`, `user_gouy_y`: User-defined gouy phase to override the calculated value.

Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7f79cea952b0>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7f79cea81bd0>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7f79cea820d0>
```

Nothing

nothing
Nothing

Represents an empty point in the interferometer configuration.

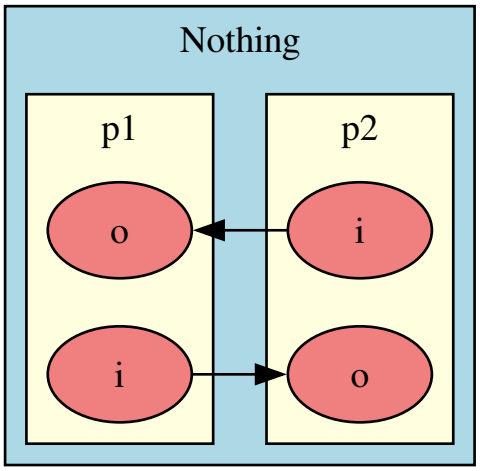
Nothing is just some point in space that can be connected to. For example, you can use this to propagate a beam from some component out to some arbitrary point to make a measurement at. You can also use this to split spaces up, if you wanted to measure something inbetween two components. It can also be used to replace a component, for example if you want to remove a lens or a mirror in some beam path.

Syntax

```
nothing name
```

Required

`name`: Name of newly created nothing.



Ports and nodes overview

Ports

```
<finesse.utilities.tables.Table at 0x7f79cea2ad70>
```

Nodes

```
<finesse.utilities.tables.Table at 0x7f79cea2b490>
```

Couplings

```
<finesse.utilities.tables.Table at 0x7f79cea332f0>
```

Wire

wire
w
Wire

A wire represents a flow of information between signal nodes. It connects two specific signal nodes. Signal nodes have a direction associated with them: input, output, or bidirectional. Depending on what direction each node has depends on what the wire will setup when it is connected. When connecting ports the wire will look at the name of each node in the port and connect nodes with the same name.

Syntax

```
w name nodeA nodeB delay=0 gain=1
```

Required

`nodeA`, `nodeB`: Signal nodes to connect together.

Optional

`name`: Name of newly created wire.

`delay`: A delay time for the signal to flow from A to B in seconds

`gain`: A scaling factor that can be used to scale the output relative to the input.

Detectors

amplitude_detector
ad
AmplitudeDetector

Represents an amplitude detector which calculates the amplitude and phase of light fields at one frequency.

Syntax

```
ad name node f n=none m=none
```

Required

`name`: Name of newly created detector.

`node`: Node to read output from.

`f`: Frequency of light to detect (in Hz).

Optional

`n`: Tangential mode index to probe. Defaults to None such that all fields of the given frequency are summed.

`m`: Sagittal mode index to probe. Defaults to None such that all fields of the given frequency are summed.

astigd
AstigmatismDetector

Detector for astigmatism figure-of-merit at a given node.

The computed quantity is given via one minus `finesse.gaussian.BeamParam.overlap()`.

Syntax

```
astigd name node
```

Required

`name`: Name of the detector.

`node`: Node to compute astigmatism at.

balanced_homodyne_detector
bhd
BalancedHomodyneDetector

Represents a balanced homodyne detector including the beampslitter and photocurrent subtraction of the two photodiodes.

Syntax

```
bhd name node P_lo=1 phase=0
```

Required

`name`: Name of newly created detector.

`node`: Node to read the output from

`P_lo: float, optional; default: 1 W`: Power of the local oscillator [W]

Optional

`phase`: Phase of the local oscillator [deg]

beam_property_detector
bp
BeamPropertyDetector

Probe for detecting the properties of a beam at a given node.

The valid values for *prop* are: * `"w"`: beam size at *node* [metres], * `"w0"`: waist size as measured at *node* [metres], * `"z"`: distance to the waist from *node* [metres], * `"zr"`: the Rayleigh range [metres], * `"gouy"`: the Gouy phase of the beam at *node* [radians], * `"div"`: divergence angle of the beam at *node* [radians], * `"rc"`: radius of curvature of wavefront at *node* [metres], * `"s"`: curvature of wavefront at *node* [1 / metres], * `"q"`: beam parameter at *node*. .. note:: The `"gouy"` target property here detects the Gouy phase as derived from the beam parameter q at the specified node, i.e: .. math:: psi = arctan{left(frac{myRe{q}}{myIm{q}}right)}. It does **not** compute any Gouy phase accumulation. Use Gouy (page 177) to detect the accumulated Gouy phase over a path.

Syntax

```
bp name node prop direction=x q_as_bp=false
```

Required

`name`: Name of newly created detector.

`node`: Node to read output from.

`prop`: The property of the beam to detect. See above for options.

Optional

`direction`: Plane to detect in - 'x' for tangential, 'y' for sagittal.

`q_as_bp`: If detecting q, should the detector output return `BeamParam` object instead of just a complex number.

ccd
CCD

Camera for measuring the intensity of a beam, $I = |E(x, y)|^2$, where the unscaled x and y coordinate arrays used are `finesse.detectors.camera.Image.xdata` and `finesse.detectors.camera.Image.ydata`, respectively. Note that this is just the intensity at the points (x,y), not an integrated power over some finite pixel size.

Syntax

```
ccd name node xlim ylim npts w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

`xlim`: Limits of the x-dimension of the image. If a single number is given then this will be computed as $x_{\text{lim}} = [-|x|, +|x|]$.

`ylim`: Limits of the y-dimension of the image. If a single number is given then this will be computed as $y_{\text{lim}} = [-|y|, +|y|]$.

`npts`: Number of points in both axes.

Optional

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

ccdline
CCDScanLine

Camera for measuring the intensity of a beam, $I = |E(x, y)|^2$, along a 1D slice. Where the unscaled x and y coordinate arrays used are `finesse.detectors.camera.ScanLine.xdata` and `finesse.detectors.camera.ScanLine.ydata`, respectively. Note that this is just the intensity at the points (x,y), not an integrated power over some finite pixel size.

The `ScanLine.direction` (i.e. axis of slice) is determined from which of *xlim* or *ylim* is specified.

Syntax

```
ccdline name node npts x=none y=none xlim=none ylim=none w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

`npts`: Number of points in slice axis.

`x`: The x coordinate of the slice. If ylim is given and this is not specified then defaults to zero. If xlim is given and this is also specified then it is ignored.

`y`: The y coordinate of the slice. If xlim is given and this is not specified then defaults to zero. If ylim is given and this is also specified then it is ignored.

`xlim`: The limits of the x-axis scan lines. A single number gives $x_{\text{axis}} \in [-|x|, +|x|]$, or a tuple of size two gives $x_{\text{axis}} \in [x[0], x[1]]$.

`ylim`: The limits of the y-axis scan lines. A single number gives $y_{\text{axis}} \in [-|y|, +|y|]$, or a tuple of size two gives $y_{\text{axis}} \in [y[0], y[1]]$.

Optional

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

ccdpx
CCDPixel

Camera for measuring the intensity of a beam, $I = |E(x, y)|^2$, at a single point. Where the unscaled x and y coordinate used is `finesse.detectors.camera.Pixel.xdata` and `finesse.detectors.camera.Pixel.ydata`, respectively. Note that this is just the intensity at (x,y), not an integrated power over some finite pixel dimension.

Syntax

```
ccdpx name node x=0 y=0 w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

Optional

`x`: The x co-ordinate of the pixel.

`y`: The y co-ordinate of the pixel.

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

cavity_property_detector
cp
CavityPropertyDetector

Probe for detecting the properties of a cavity.

The valid values for *prop* are: * `"length"` or `"l"`: round-trip cavity length [metres], * `"loss"`: round-trip loss as a fraction, * `"finesse"`: the cavity finesse, * `"fsr"`: free spectral range [Hz], * `"fwhm"`: full-width at half-maximum (i.e. linewidth) [Hz], * `"pole"`: cavity pole frequency [Hz], * `"tau"`: photon storage time [s], * `"abcd"`: round-trip ABCD matrix, * `"g"` or `"stability"`: stability as g-factor, * `"gouy"`: round-trip Gouy phase [deg], * `"modesep"`: mode-separation frequency [Hz], * `"resolution"`: cavity resolution [Hz], * `"q"`: eigenmode, * `"w"`: beam size at the cavity source node [metres], * `"w0"`: waist size [metres], * `"z"`: distance to the waist from the cavity source node [metres], * `"zr"`: the Rayleigh range of the eigenmode [metres], * `"div"`: divergence angle of cavity mode [radians], * `"rc"`: radius of curvature of wavefront at cavity source node [metres], * `"s"`: curvature of wavefront at cavity source node [1 / metres].

Syntax

```
cp name cavity prop direction=x q_as_bp=false
```

Required

`name`: Name of newly created cavity property detector.

`cavity`: The cavity to probe. If the name is provided then the `CavityPropertyDetector.cavity` attribute will point to the corresponding `Cavity` (page 195) object when adding this detector to a `Model` instance.

`prop`: Property of the cavity to probe. See above for options.

Optional

`direction`: Plane to detect in.

`q_as_bp`: If detecting q, should the detector output return `BeamParam` object instead of just a complex number.

fcam
FieldCamera

Camera for detecting the full image of the beam in terms of amplitude and phase.

Get the unscaled x and y coordinate data via `finesse.detectors.camera.Image.xdata` and `finesse.detectors.camera.Image.ydata`, respectively.

Syntax

```
fcam name node xlim ylim npts f=0 w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

`xlim`: Limits of the x-dimension of the image. If a single number is given then this will be computed as $x_{\text{lim}} = [-|x|, +|x|]$.

`ylim`: Limits of the y-dimension of the image. If a single number is given then this will be computed as $y_{\text{lim}} = [-|y|, +|y|]$.

`npts`: Number of points in both axes.

Optional

`f`: Field frequency offset from the carrier to detect.

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

field_detector
fd
FieldDetector

Outputs an array of the higher order modes amplitudes at a particular node and frequency. The mode ordering is given by *Model.homs*. Plane wave models will output a single element array.

This detector can only be used on optical nodes.

Syntax

```
fd name node f
```

Required

`name`: Name of newly created detector.

`node`: Node to read output from.

`f`: Frequency of light to detect (in Hz).

fline
FieldScanLine

Camera for detecting a slice of the beam in terms of amplitude and phase.

The `ScanLine.direction` (i.e. axis of slice) is determined from which of *xlim* or *ylim* is specified. Get the unscaled x and y coordinate data via `finesse.detectors.camera.ScanLine.xdata` and `finesse.detectors.camera.ScanLine.ydata`, respectively.

Syntax

```
fline name node npts x=none y=none xlim=none ylim=none f=0 w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

`npts`: Number of points in slice axis.

`x`: The x coordinate of the slice. If ylim is given and this is not specified then defaults to zero. If xlim is given and this is also specified then it is ignored.

`y`: The y coordinate of the slice. If xlim is given and this is not specified then defaults to zero. If ylim is given and this is also specified then it is ignored.

`xlim`: The limits of the x-axis scan lines. A single number gives $x_{\text{axis}} \in [-|x|, +|x|]$, or a tuple of size two gives $x_{\text{axis}} \in [x[0], x[1]]$.

`ylim`: The limits of the y-axis scan lines. A single number gives $y_{\text{axis}} \in [-|y|, +|y|]$, or a tuple of size two gives $y_{\text{axis}} \in [y[0], y[1]]$.

Optional

`f`: Field frequency offset from the carrier to detect.

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

fpx
FieldPixel

Camera for detecting a single pixel of the beam in terms of the amplitude and phase.

Get the unscaled x and y coordinate data via `finesse.detectors.camera.Pixel.xdata` and `finesse.detectors.camera.Pixel.ydata`, respectively.

Syntax

```
fpx name node x=0 y=0 f=0 w0_scaled=true
```

Required

`name`: Unique name of the camera.

`node`: Node at which to detect.

Optional

`x`: The x co-ordinate of the pixel.

`y`: The y co-ordinate of the pixel.

`f`: Field frequency offset from the carrier to detect.

`w0_scaled`: Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

gouy
Gouy

Detector to measure the accumulated gouy phase across a sequence of spaces.

This detector can operate in one of two modes, depending upon args given: * automatically determine the spaces through an arbitrary path by specifying the *from_node* and *to_node* arguments, * OR provide a pre-determined sequence of spaces (or their names) as positional arguments. Whichever option is chosen, this detector will compute the same fundamental quantity; that is the sum of the Gouy phases accumulated over each space (in degrees).

Syntax

```
gouy name *args from_node=none to_node=none via_node=none direction=x
```

Required

`name`: Name of newly created gouy detector.

`\*args`: A sequence of spaces or space names.

`from_node`: An `OpticalNode` instance, or a data type which can be converted to an optical node.

`to_node`: An `OpticalNode` instance, or a data type which can be converted to an optical node.

`via_node`: An `OpticalNode` instance, or a data type which can be converted to an optical node or a sequence of the these types.

`direction`: Plane to detect in - 'x' for tangential, 'y' for sagittal. Defaults to 'x'.

knmd

KnmDetector

Direct probe of coupling coefficients at a component.

This detector has several "modes" which depend upon the values given for the mode indices *n1*, *m1* and *n2*, *m2*. If: * all of *n1*, *m1*, *n2*, *m2* are specified then the detector will output a single complex coefficient corresponding to the coupling from `(n1, m1) -> (n2, m2)`, * just *n1* and *m1* are specified then it will output a vector of complex coefficients corresponding to each coupling from `(n1, m1) -> (n, m)` for each mode `(n, m)` in the model, * only *n2* and *m2* are specified then it will output a vector of complex coefficients corresponding to each coupling from `(n, m) -> (n2, m2)` for each mode `(n, m)` in the model, * none of *n1*, *m1*, *n2*, *m2* are specified then the detector outputs the full matrix of complex coupling coefficients. .. hint:: When using this detector in "full-matrix" mode (i.e. by not giving the values for any of the mode indices), it can be useful to combine the output with the `KnmMatrix` object to obtain a more convenient representation of the scattering matrix. An example of this is shown below, where the output of a detector of this type is wrapped using `KnmMatrix.from_buffer()`. .. jupyter-execute:: import finesse finesse.configure(plotting=True) from finesse.knm.matrix import KnmMatrix IFO = finesse.Model() IFO.parse(''' l L0 P=1 link(L0, ITM) m ITM R=0.99 T=0.01 Rc=-2090 xbeta=0.3u s LARM ITM.p2 ETM.p1 L=4k m ETM R=0.99 T=0.01 Rc=2090 cav ARM ITM.p2 modes(x, maxtem=6) knmd K_itm_r ITM 22 ''') out = IFO.run('noxaxis()') # Make a KnmMatrix wrapper around the output from the detector k_mat = KnmMatrix.from_buffer(out["K_itm_r"], IFO.homs) # Now we can perform operations such as plotting the scattering matrix k_mat.plot(cmap="bone"); See Computing arbitrary scattering matrices (page 353) for some examples on the utility that the `KnmMatrix` object provides.

Syntax

```
knmd name comp coupling n1=none m1=none n2=none m2=none
```

Required

`name`: Name of newly created KnmDetector.

`comp`: A component which can scatter modes.

`coupling`: Coupling direction string, e.g. "11" for coupling coefficients on reflection from the front surface of a mirror.

`n1, m1, n2, m2: int or None`: From (n1, m1) and To (n2, m2) mode indices of the coupling coefficient(s) to retrieve. See above for the options.

math_detector
mathd
MathDetector

A detector that performs some math operation and outputs the result.

Syntax

```
mathd name expression dtype='O' dtype_shape=[] unit='arb.' label=none
```

Required

`name`: Name of detector

`expression`: Symbolic expression to evaluate as the detectors output

`dtype_shape`: The expected array shape when evaluating *expression*. Defaults to an empty tuple. This must be an empty tuple if *dtype* is a scalar type or "O".

`unit`: The unit of the output for plotting. Defaults to "arb."

`label`: How to label the axis when plotting this detector. Defaults to *None*.

Optional

`dtype`: The expected data type when evaluating expression. Defaults to "O".

mmd

ModeMismatchDetector

Detector for mode mismatch figure-of-merit for a specified node coupling.

The computed quantity is given in `finesse.gaussian.BeamParam.mismatch()` where q_1 is the input beam parameter at *node1* propagated via the associated ABCD matrix to *node2*, and q_2 is the beam parameter at *node2*. As mode mismatches cannot occur over spaces in Finesse, *node1* must be an input node and *node2* must be an output node.

Syntax

```
mmd name node1 node2 direction=x percent=false
```

Required

`name`: Name of the detector.

`node1`: Input node or port. If a `Port` instance is given then this node will be the input node of that port. Note that this node cannot be an output node.

`node2`: Output node or port. If a `Port` instance is given then this node will be the output node of that port. Note that this node cannot be an input node.

Optional

`direction`: Plane of computation, defaults to "x" for the tangential plane. Changing to "y" will compute the mode mismatch in the sagittal plane.

`percent`: Whether to calculate the mode mismatch as a fraction (default behaviour) or a percentage.

motion_detector
xd
MotionDetector

Represents a motion detector which calculates the amplitude and phase of surface motion.

Syntax

```
xd name node
```

Required

`name`: Name of newly created motion detector.

`node`: Node to read output from.

optimal_q_detector
optbp
OptimalQ

This detector tries to compute an optimal beam parameter (q) for a specified optical frequency at a node.

Output of this detector into an array solution will be a tuple of `BeamParam` in each transverse direction, (qx, qy). If the optimisation process fails beam parameter objects will NaN values will be returned.

Syntax

```
optbp name node f fix_spot_size=false astigmatic=false accuracy=1e-09␣
↪direction=both
```

Required

`name`: Name of the detector

`node`: Node name or object to put this detector at

`f`: Frequency component tro compute the optimal beam parameter for.

Optional

`fix_spot_size`: When True the optimised will keep the current spot size at the node fixed and just optimise the curvature.

`astigmatic`: When True qx and qy will be optimised separately

`accuracy`: Approximate mismatch accuracy to try and compute the optimised beam parameter to. mismatch(q_actual, q_optimal) < accuracy

`direction`: Return either *both* or just the *x* or *y* modes

power_detector_dc
pd
PowerDetector

Represents a power detector with no RF demodulations. It calculates the DC laser power at a node in Watts of optical power.

Syntax

```
pd name node pdtype=none
```

Required

`name`: Name of newly created power detector.

`node`: Node to read output from.

power_detector_demod_1
pd1
PowerDetectorDemod1

Represents a power detector with one RF demodulation. It calculates the RF beat power at a node in Watts of optical power.

If no demodulation phase is specified then this detector outputs a complex value *I+1j*Q*.

Syntax

```
pd1 name node f phase=none pdtype=none
```

Required

`name`: Name of newly created power detector.

`node`: Node to read output from.

`f`: Demodulation frequency in Hz

Optional

`phase`: Demodulation phase in degrees

power_detector_demod_2
pd2
PowerDetectorDemod2

Represents a power detector with two RF demodulation. It calculates the RF beat power at a node in Watts of optical power.

If no demodulation phase is specified for the final demodulation this detector outputs a complex value *I+1j*Q* where I and Q are the in-phase and quadrature parts of the signal.

Syntax

```
pd2 name node f1 phase1 f2 phase2=none pdtype=none
```

Required

`name`: Name of newly created power detector.

`node`: Node to read output from.

`f1`: First demodulation frequency in Hz

`phase1`: First demodulation phase in degrees

`f2`: Second demodulation frequency in Hz

Optional

`phase2`: Second demodulation phase in degrees

quantum_balanced_homodyne
qnoisebhd
QuantumBHD

Represents the quantum noise of a balanced homodyne detector including the beampslitter and photocurrent subtraction of the two photodiodes.

Syntax

```
qnoisebhd name node P_lo=1 phase=0 nsr=false sources=none exclude_
↪sources=none
```

Required

`name`: Name of newly created quantum BHD noise detector.

`node`: Node to read the output from.

Optional

`P_lo`: Power of the local oscillator [W]

`phase`: Phase of the local oscillator [deg]

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

`sources`: If given, only detect quantum noise contributions from these components.

`exclude_sources`: If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

quantum_noise_detector
qnoised
QuantumNoiseDetector

Represents a quantum noise detector with no RF demodulations.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency.

Syntax

```
qnoised name node nsr=false sources=none exclude_sources=none
```

Required

`name`: Name of newly created quantum noise detector.

`node`: Node to read output from.

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

`sources`: If given, only detect quantum noise contributions from these components.

`exclude_sources`: If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

quantum_noise_detector_demod_1
qnoised1
QuantumNoiseDetectorDemod1

Represents a quantum noise detector with 1 RF demodulation.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency.

Syntax

```
qnoised1 name node f phase nsr=false sources=none exclude_sources=none
```

Required

`name`: Name of newly created quantum noise detector.

`node`: Node to read output from.

`f`: Demodulation frequency in Hz

`phase`: Demodulation phase in degrees

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

`sources`: If given, only detect quantum noise contributions from these components.

`exclude_sources`: If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

quantum_noise_detector_demod_2
qnoised2
QuantumNoiseDetectorDemod2

Represents a quantum noise detector with 2 RF demodulations.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency.

Syntax

```
qnoised2 name node f1 phase1 f2 phase2 nsr=false sources=none exclude_
↪sources=none
```

Required

`name`: Name of newly created quantum noise detector.

`node`: Node to read output from.

`f1`: First demodulation frequency in Hz

`phase1`: First demodulation phase in degrees

`f2`: Second demodulation frequency in Hz

`phase2`: Second demodulation phase in degrees

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

`sources`: If given, only detect quantum noise contributions from these components.

`exclude_sources`: If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

quantum_shot_noise_detector
qshot
QuantumShotNoiseDetector

Represents a quantum shot noise detector with no RF demodulations.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency, considering only vacuum noise contributions (ignoring radiation pressure and squeezing effects).

Syntax

```
qshot name node nsr=false
```

Required

`name`: Name of newly created quantum shot noise detector.

`node`: Node to read output from.

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

quantum_shot_noise_detector_demod_1
qshot1
QuantumShotNoiseDetectorDemod1

Represents a quantum shot noise detector with 1 RF demodulation.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency, considering only vacuum noise contributions (ignoring radiation pressure and squeezing effects).

Syntax

```
qshot1 name node f phase nsr=false
```

Required

`name`: Name of newly created quantum shot noise detector.

`node`: Node to read output from.

`f`: Demodulation frequency in Hz

`phase`: Demodulation phase in degrees

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

quantum_shot_noise_detector_demod_2
qshot2
QuantumShotNoiseDetectorDemod2

Represents a quantum shot noise detector with 2 RF demodulations.

It calculates the amplitude spectral density of the photocurrent noise of a photodiode output demodulated at the signal frequency, considering only vacuum noise contributions (ignoring radiation pressure and squeezing effects).

Syntax

```
qshot2 name node f1 phase1 f2 phase2 nsr=false
```

Required

`name`: Name of newly created quantum shot noise detector.

`node`: Node to read output from.

`f1`: First demodulation frequency in Hz

`phase1`: First demodulation phase in degrees

`f2`: Second demodulation frequency in Hz

`phase2`: Second demodulation phase in degrees

Optional

`nsr`: If true, calculate the noise-to-signal ratio rather than the absolute noise value.

Readouts

- DC readout (page 186)
- RF photodetector readout (page 187)
- Balanced Homodyne Readout (page 188)

Readouts are a hybrid component in FINESSE 3 which combine together both a detector and an optical component. In essence, they represent a photodetector or some other measurement device that produces an electronic signal.

Readouts have one or more optical ports that are coupled to electrical ports. This coupling represents the calculation done in a `PowerDetector` (page 180) for the *ReadoutDC* and a `PowerDetectorDemod1` (page 180) for the *ReadoutRF*.

These electrical ports have two main usecases:

- Calculating transfer functions with the `FrequencyResponse` (page 221) action, see Readouts for frequency domain analysis (page 270)
- Modelling closed loop systems (page 281) by taking the output at the electrical node and feeding, (optionally) filtering it and feeding it back into another component's electrical port. The loop is usually closed with Locking actions (page 72). Filters can be found in `finesse.components.electronics`

Readouts can be connected to the model in two different ways:

- *Detector-style* by using the `optical_node` argument. This allows you to place the Readout anywhere in your model and to place multiple readouts on the same optical node.
- *Component-style* by using a `Space` (page 170) or the `finesse.model.Model.link()` method to connect the optical node of the readout to another optical_node in your model. Since FINESSE only allows a port to be connected to one other port, you can only connect one readout per port.

```
from finesse import Model
from finesse.components import Laser, ReadoutDC, ReadoutRF

# Detector style
model = Model()
l1 = model.add(Laser("l1"))
readout1 = model.add(ReadoutDC("readout1", optical_node=l1.p1.i))
readout2 = model.add(ReadoutDC("readout2", optical_node=l1.p1.i))

# readout 'borrows' nodes
print(f"{readout1.p1.i=}")
print(f"{readout1.p1.o=}")
print(f"{readout2.p1.i=}")
print(f"{readout2.p1.o=}")
```

```
readout1.p1.i=<OpticalNode l1.p1.i @ 0x7f454f3a8ad0>
readout1.p1.o=<OpticalNode l1.p1.o @ 0x7f454f388a50>
readout2.p1.i=<OpticalNode l1.p1.i @ 0x7f454f3a8ad0>
readout2.p1.o=<OpticalNode l1.p1.o @ 0x7f454f388a50>
```

Above you see that the readout only 'borrows' the nodes of the port it is connected to and this 'borrowing' can be done as many times as you want.

Alternatively you can connect explicitly:

```
# Component style
model = Model()
l1 = model.add(Laser("l1"))
readout1 = model.add(ReadoutDC("readout1"))
model.link(l1.p1, readout1.p1, verbose=True)
```

```
Connecting l1 to readout1
```

But this does not allow you to connect a second readout to the same port:

```
readout2 = model.add(ReadoutDC("readout2"))
model.link(l1.p1, readout2.p1, verbose=True)
```

```
Connecting l1 to readout2
```

```
FinesseException:          (use finesse.tb() to see the full traceback)
Whilst trying to connect <Port l1.p1 Type=NodeType.OPTICAL @ 0x7f454f33aea0> to <Port␣
↪readout2.p1 Type=NodeType.OPTICAL @ 0x7f454f3bc490>, the following exception occurred:
Port <Port l1.p1 Type=NodeType.OPTICAL @ 0x7f454f33aea0> has already been connected to
```

Warning

Mixing the two styles will lead to an exception and the following should be avoided:

```
l l1
ReadoutDC readout l1.p1.o
link(l1.p1, readout.p1)
```

The `output_detectors` argument can be used to add relevant detectors to the solution object:

- `ReadoutDC`: A `Powerdetector` and a `QuantumShotNoiseDetector` will be added pointing to `ReadoutDC.p1.i`

```
model = Model()
l1 = model.add(Laser("l1"))
readout = model.add(
    ReadoutDC("readout", optical_node=l1.p1.i, output_detectors=True)
)
sol = model.run()
for det in sol.detectors:
    print(det, sol[det])
```

```
readout_DC 0.0
```

- `ReadoutRF`: Two `PowerDetectorDemod1` (`I` and `Q` quadratures), a `PowerDetector` and a `QuantumShotNoiseDetectorDemod1`

```
model = Model()
l1 = model.add(Laser("l1"))
readout = model.add(
    ReadoutRF("readout", optical_node=l1.p1.o, output_detectors=True)
)
sol = model.run()
for det in sol.detectors:
    print(det, sol[det])
```

```
readout_I 1.0000000000000002
readout_Q 6.123233995736767e-17
readout_DC 1.0000000000000002
```

Note that the quantum detectors will only be added if you are running a signal model (sgen or frequency response)

Note

Currently `output_detectors` only makes detectors appear in `Solution` objects, and there will be no detectors listed in `finesse.model.Model.detectors()`.

DC readout

`readout_dc`
`ReadoutDC`

A Readout component which represents a photodiode measuring the intensity of some incident field. Audio band intensity signals present in the incident optical field are converted into an electrical signal and output at the `self.DC` port, which has a single `self.DC.o` node.

See Readouts (page 183) for more information.

Syntax

```
readout_dc name optical_node=none pdtype=none output_detectors=false
```

Required

`name`: Name of the readout component

Optional

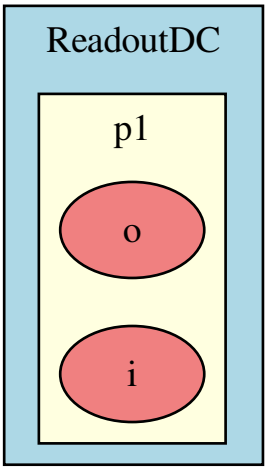
`optical_node`: Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `finesse.model.Model.link()` command (and then only a single readout can be connected per port).

`pdtype`: A name of a pdtype definition or a dict representing a pdtype definition, by default None. See Segmented photodiodes (page 310)

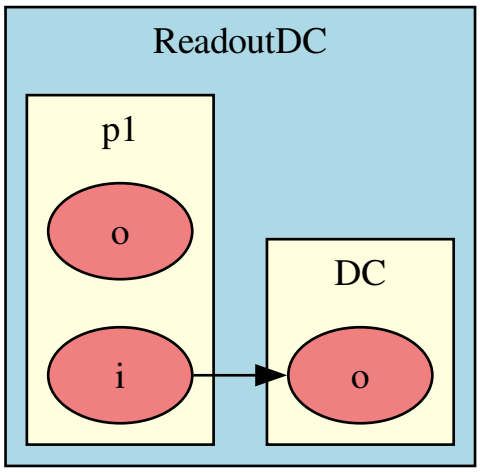
`output_detectors`: Whether to add a `PowerDetector` and a `QuantumShotNoiseDetector` to the solution object, by default False

Component Graph

Optical



Full



RF photodetector readout

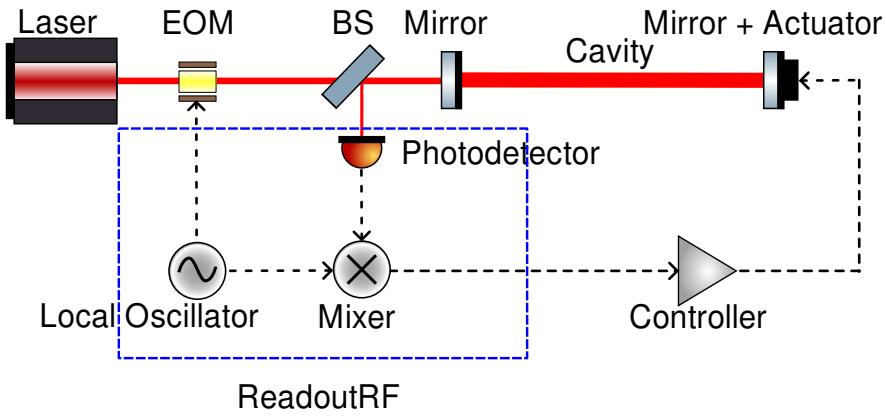


Fig. 6.6: In this control loop, the *ReadoutRF* represents a combination of the oscillator, mixer and photodetector.

readout_rf
ReadoutRF

A readout component which represents a demodulated photodiode. The `self.I` and `self.Q` electrical ports contain the signal of the two quadratures.

See Readouts (page 183) for more information.

Syntax

```
readout_rf name optical_node=none f=0 phase=0 output_detectors=false␣
↪pdtype=none
```

Required

`name`: Name of the readout component

Optional

`optical_node`: Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `link` (page 196) command (and then only a single readout can be connected per port).

`f`: Demodulation frequency in Hz, by default 0

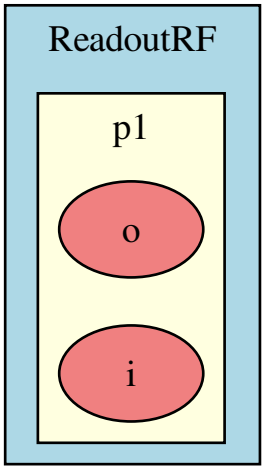
`phase`: Demodulation phase in degrees, by default 0

`output_detectors`: Whether to add a `PowerDetector` (DC), two `PowerDetectorDemod1` 's (I and Q) and a `QuantumShotNoiseDetectorDemod1` to the solution object, by default False

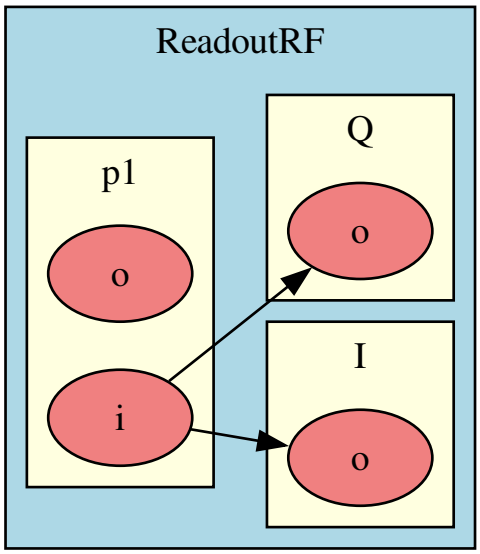
`pdtype`: A name of a pdtype definition or a dict representing a pdtype definition, by default None. See Segmented photodiodes (page 310)

Component Graph

Optical



Full



Balanced Homodyne Readout

balanced_homodyne_readout
BalancedHomodyneReadout

A readout component representing a balanced homodyne detector (BHD).

Syntax

```
balanced_homodyne_readout name optical_node=none P_lo=1 phase=0 output_
↪detectors=false
```

Required

`name`: Name of the readout component

Optional

`optical_node`: Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `finesse.model.Model.link()` command (and then only a single readout can be connected per port).

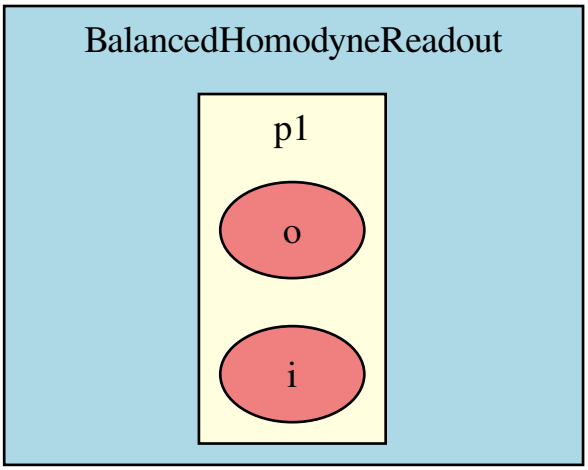
`P_lo`: Power of the local oscillator in W, by default 1 W

`phase`: Phase of the local oscillator in deg, by default 0 deg

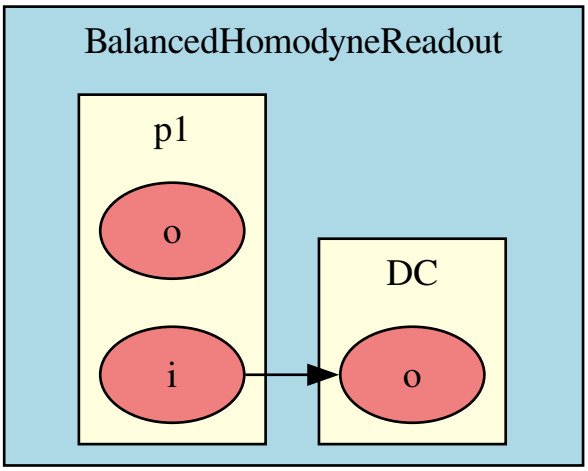
`output_detectors`: Whether to add a `BalancedHomodyneDetector` to the solution object, by default False

Component Graph

Optical



Full



Control and filtering

lock
Lock

A simple lock which computes and applies the feedback to a given parameter using an error signal.

Syntax

```
lock name error_signal feedback gain accuracy enabled=true offset=0
```

Required

`name`: Name of newly created lock.

`error_signal`: An error signal parameter or an object capable of producing a real-type error signal. This is typically a demodulated `PowerDetector` instance (or the name of the instance).

`feedback`: A parameter of the model to apply the locks' feedback signal to.

`gain`: Control loop gain.

`accuracy`: Threshold to decide whether the loop is locked.

`enabled`: If true this lock will run when the *RunLocks()* action is used. Explicitly specifying the name of the lock will override this setting, e.g. *RunLocks(name)*.

`offset`: An offset that is applied to the error signal before it is used.

degree_of_freedom
dof
DegreeOfFreedom

The degree of freedom element represents degrees of freedom in your optical setup. It is used to control multiple parameters in your model in a coherent way.

Syntax

```
dof name *node_amplitude_pairs DC=0 lock_parameters=true
```

Required

`name`: Name of the degree of freedom.

`\*node_amplitude_pairs: tuple[str, float | Symbol]`: String referring to a parameter or a local degree of freedom, coupled with the amplitude with which this DOF will drive the parameter. The parameter value will be set to the product of this (symbolic) amplitude and the DOF.DC parameter. In case multiple DOFS drive the same parameter, their contributions will be summed.

Optional

`DC`: DC value for the degree of freedom, by default 0

`lock_parameters`: Whether to 'lock' the parameters driven by this degree of freedom. This prevents accidental overriding of parameter values that are meant to be controlled by a dof, by default True.

See Also

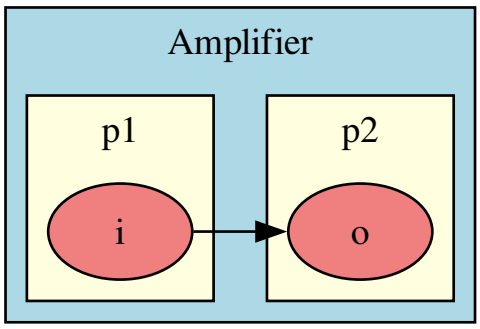
Using degrees of freedom (page 286)

amplifier
Amplifier

Syntax

```
amplifier name gain=1
```

Component Graph



filter_zpk
zpk
ZPKFilter

A zero-pole-gain filter element that is used for shaping signals in simulations. It is a two port element. *p1* is the input port and *p2* is the output port. Each one has a single node: *p1.i* and *p2.o*.

Syntax

```
zpk name z p k=none fQ=false gain=1
```

Required

`name`: Name of element in the model

`z`: A 1D-array of zeros. Use *[]* if none are required. By default these are provided in units of radians/s, not Hz.

`p`: A 1D-array of poles. Use *[]* if none are required. By default these are provided in units of radians/s, not Hz.

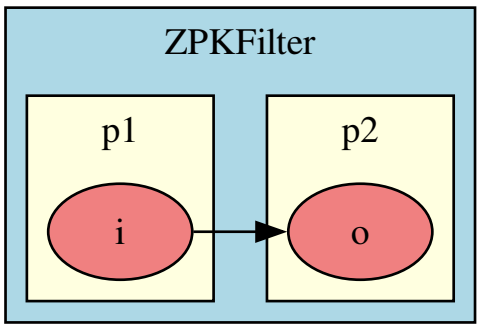
`gain`: Overall gain for the filter. Differs from *k* as this is a *Parameter* so can be easily switched on/off or varied during a simulation.

Optional

`k`: Gain factor for the zeros and poles. If *None* then its value is automatically set to generate a unity gain at DC.

`fQ`: When True the zeros and poles can be specified in a tuple of (frequency, quality factor) for each pole and zero. This automatically adds both roots of each pair.

Component Graph



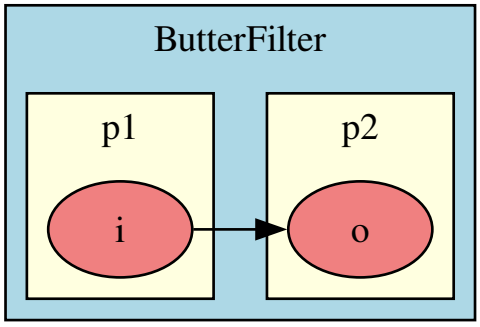
filter_butter
butter

ButterFilter

Syntax

```
butter name order btype frequency gain=1 analog=true
```

Component Graph

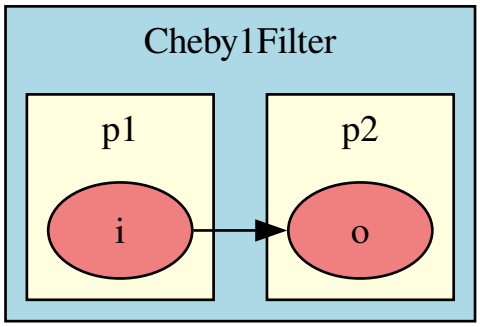


filter_cheby1
cheby1
Cheby1Filter

Syntax

```
cheby1 name order rp btype frequency gain=1 analog=true
```

Component Graph



test_point
test
TestPoint

A simple component which has an arbitrary number of test nodes that can be connected to and from.

Syntax

```
test name *ports
```

Mechanical Elements

free_mass
FreeMass

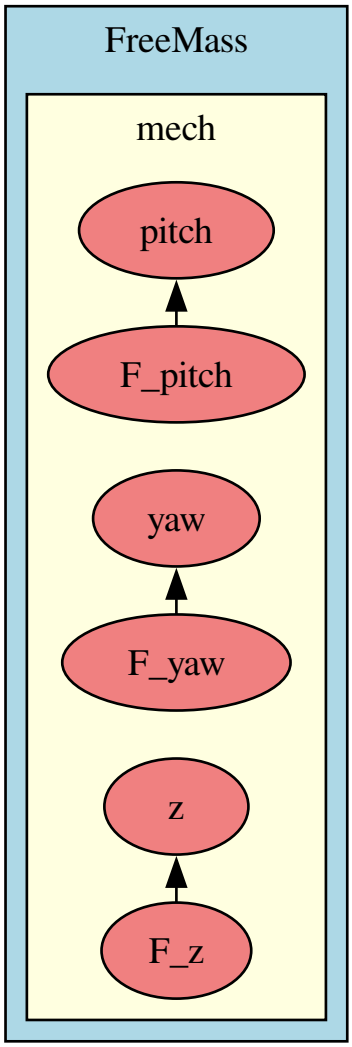
Simple free mass suspension of an object.

The object being suspended must have a mechanical port with nodes z, pitch, and yaw and forces F_z, F_pitch, and F_yaw.

Syntax

```
free_mass name connected_to mass=inf I_yaw=inf I_pitch=inf
```

Component Graph



pendulum
Pendulum

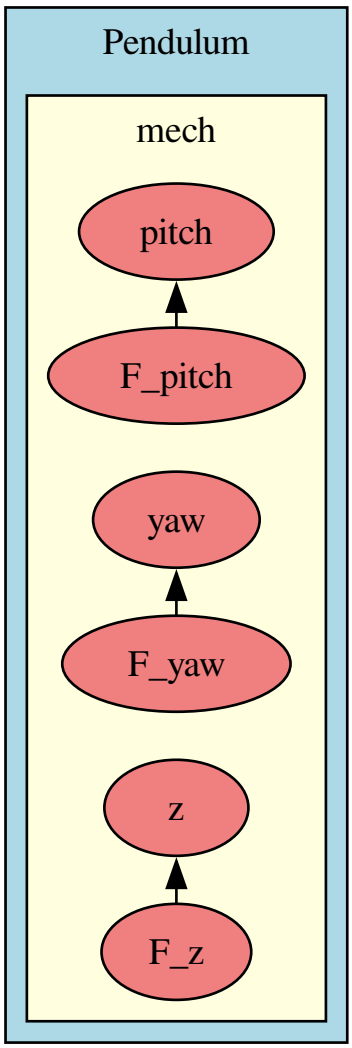
Simple pendulum suspension of an object.

The object being suspended must have a mechanical port with nodes z, pitch, and yaw and forces F_z, F_pitch, and F_yaw.

Syntax

```
pendulum name connected_to mass=inf Qz=1000 fz=1 I_yaw=inf Qyaw=1000␣
↪fyaw=1 I_pitch=inf Qpitch=1000 fpitch=1
```

Component Graph



suspension_zpk
sus_zpk
SuspensionZPK

A suspension that models multiple poles and zeros for the z, yaw, or pitch motion of an optic. The user must ensure that minus signs are correct for this transfer function as well as defining complex conjugae pairs for physically correct behaviour.

ZPK terms are in units of radians/s.

Syntax

```
sus_zpk name connected_to zpk_plant
```

Required

`name`: Element name

`connected_to`: Mechanical port or element to attach this suspension to

`zpk_plant`: Dictionary of {(output, input):(z,p,k)}

Other elements

There are also other elements which do not represent physical components that can be used in a model.

Variables

variable
var

The variable element acts slightly different to other elements. When added to a model it creates a new `finesse.parameter.Parameter` in the model it has been added to. This does the same as calling `finesse.model.Model.add_parameter()`. This new parameter can be used like a variable for making symbolic links to or for storing some useful number about the model.

See `finesse.model.Model.add_parameter()` for more details.

Syntax

```
var name value description=none units='' is_geometric=false changeable_
↪during_simulation=true
```

Gaussian beams

cavity
cav
Cavity

Represents a cavity in an interferometer configuration.

This class stores the shortest round-trip path from the start node of the cavity (via a given node) back to the same node, and holds symbolic expressions for each physical attribute of the cavity. Numeric values corresponding to these attributes are obtained through the relevant properties. Adding a Cavity to a `Model` results in the beam parameters of all nodes in the cavity path being set according to the cavity eigenmode (`finesse.components.cavity.Cavity.q`) when a beam trace is performed (e.g. at the start of a modal based simulation). The mode of the cavity is then also used as a trace starting point when setting beam parameters at nodes outside of the cavity - see Tracing the beam (page 306) for details on the beam tracing algorithm.

Syntax

```
cav name source via=none priority=0
```

Required

`name`: Name of newly created cavity.

`source`: Node / Port that the cavity path starts from. If no *via* node is specified, then the cavity path will be given by the shortest path from source back to the component that owns source. If a port is given then the *output* optical node of that port will be used as the source.

Optional

`via`: Node that the cavity path must traverse via; defaults to *None*. Note that, unlike *source*, this cannot be a `Port` object as this would be ambiguous for beamsplitter type components - i.e. determination of which node to use cannot be assumed automatically.

`priority`: Priority value for beam tracing. Beam tracing dependencies are sorted in descending order of priority - i.e. higher priority value dependencies will be traced first. Any dependency with a priority value of zero will be traced, after non-zero priority dependencies, in alphabetic order of the dependency names.

gauss

Beam parameter at a specific node of a model.

The following are legal initialisations of a *Gauss* object:: # Non-astigmatic q = Gauss('name', node, q=q) q = Gauss('name', node, w0=w0, z=z) # waist size and position q = Gauss('name', node, zr=zr, z=z) # Rayleigh range and waist position q = Gauss('name', node, w=w, Rc=Rc) # beam size and RoC q = Gauss('name',

node, w=w, S=S) # beam size and curvature # Astigmatic q = Gauss('name', node, qx=qx, qy=qy) q = Gauss('name', node, w0x=wx, zx=zx, w0y=wy, zy=zy) q = Gauss('name', node, zrx=zrx, zx=zx, zry=zry, zy=zy) q = Gauss('name', node, wx=wx, Rcx=Rcx, wy=wy, Rcy=Rcy) q = Gauss('name', node, wx=wx, Sx=Sx, wy=wy, Sy=Sy) See `gaussian.BeamParam` for more detailed descriptions of these variables.

Syntax

```
gauss name node priority=0 **kwargs
```

```
<IPython.core.display.HTML object>
```

6.2.5 Commands

Commands set some properties of the model or its elements. Some commands may only be specified once per model, others multiple times.

`fsig`

This element represents the signal frequency (`fsig`) used in a model. It is a unique element, which means only one can be added to any given model. This is done automatically with the name `fsig`. It has a single parameter `f` for the frequency of the signal.

The signal frequency must be set by the user to enable transfer functions and noise projections to be simulated.

Syntax

```
fsig(f)
```

Required

`f`: Signal frequency to use in a model [Hz]. If set to `None` then no signal frequencies will be modelled in the simulation.

`lambda`

Set the reference wavelength used in a model.

Syntax

```
lambda(value)
```

Required

`value`: Reference wavelength to use in a model [Hz]. If not specified, a reference wavelength of 1064 nm is used.

`link`

Connect multiple components together in one quick command. In many models a collection of components just need to be connected together without having to specify each port exactly. This command accepts multiple components as arguments, each is connected to the next. Interally the *link* command is creating spaces and wires between components but giving them automatically generated names. Therefore, the *link* command is useful when you are not interested in what the spaces or wires are called, which is often the case in readout paths or signal feedback loops.

This command will try to connect components in the "obvious" way. For example, a collection of two-port optical components will be connected one after another, the second port of the first item connected to the first port of the second component, etc. Explicit ports can also be provided if exact connections are required. For example, at a beamsplitter, if you want to link through on transmission then use *..., BS.p1, BS.p3, ...*. Using just *BS* here would result in using the first and second ports, a reflection. Links can contain a mix of optical and signal nodes. When going from optical to signal nodes they must be specified verbosely. For example, a DC readout component *PD*, you would need to specify *..., PD, PD.DC, ...*.

Syntax

```
link(*args, verbose=false)
```

Required

`\*args`: Separate arguments of either components or ports. A float value will create a space or wire with the provided length or time delay.

Optional

`verbose`: Print out what the link command is doing

See Also

`space` (page 170)

`modes`

Select the HOM indices to include in the model.

See Selecting the modes to model (page 327) for examples on using this method.

Syntax

```
modes(modes=none, maxtem=none, include=none, remove=none)
```

Optional

`modes`: Identifier for the mode indices to generate. This can be: - An iterable of mode indices, where each element in the iterable must unpack to two integer convertible values. - A string identifying the type of modes to include, must be one of "off", "even", "odd", "x" or "y".

`maxtem`: Optional maximum mode order.

`include`: A single mode index pair, or an iterable of mode indices, to include. Each element must unpack to two integer convertible values.

`remove`: A single mode index pair, or an iterable of mode indices, to remove. Each element must unpack to two integer convertible values.

Note: at least one of `modes`, `maxtem`, `include` and `remove` must be specified.

`phase_config`

Coupling coefficient and Gouy phase scaling:

- phase_level 3 == zero_k00=True, zero_tem00_gouy=True - phase_level 2 == zero_k00=False, zero_tem00_gouy=True - phase_level 1 == zero_k00=True, zero_tem00_gouy=False - phase_level 0 == zero_k00=False, zero_tem00_gouy=False See also Phase configuration settings (page 299) This can be used to change the computation of light field phases in the Hermite-Gauss mode. In general, in the presence of higher order modes, the "macroscopic length" of a space is no longer an integer number of wavelengths for the TEM00 mode because of the Gouy phase. Furthermore, the coupling coefficients k_{nmnm} contribute to the phase when there is a mode mismatch. For correct analysis these effects have to be taken into account. On the other hand, these extra phase offsets make it very difficult to set a resonance condition or operating point intuitively. In most cases another phase offset can be added to all modes so that the phase of the TEM00 becomes zero. This method allows setting these phase offsets for the propagation through free space, for the coupling coefficients, or both. Regardless of the setting, the phases for all higher modes are changed accordingly such that the relative phases remain correct.

Syntax

```
phase_config(zero_k00=false, zero_tem00_gouy=true)
```

Optional

`zero_k00`: Scale phase for k0000 (TEM00 to TEM00) coupling coefficients to 0. Defaults to True.

`zero_tem00_gouy`: Ensure that all Gouy phases for TEM00 are 0. Defaults to True.

`tem`

Distributes power into the mode HGnm.

Syntax

```
tem(laser, n, m, factor, phase=0.0)
```

Required

`laser`: The `Laser` (page 165) to set mode power for.

`n`, `m`: Mode indices.

`factor`: Relative power factor, modes with equal *factor* will have equivalent power distributed to them.

Optional

`phase`: Phase offset for the field, in degrees.

6.2.6 Actions, Analyses, and Solutions

The way in which simulations are defined and run is significantly different in FINESSE 3. Users familiar with `xaxis` in FINESSE 2 can happily continue using it in version 3. However, for users wishing to perform more complicated modelling tasks, actions, analyses, and solutions provide a new interface to take your simulation modelling to the next level. These enable more advanced physics simulations with greater computational efficiency, all with a simple to use interface.

The introduction is a must-read for getting the most out of FINESSE 3. More technical details can be found in other sections for those users wanting to know more.

Introduction

FINESSE 3 introduces a new way in which to run simulations. The primary idea is to allow multiple numerical studies/simulations to be performed on a model in a more adaptable and computationally efficient way.

It is worthwhile revisiting briefly what this involved in FINESSE 2 and Pykat to see why we made this change. In FINESSE 2 the only analyses that could be performed were `noxaxis`, `xaxis`, and `x2axis`. All these did was simulate the model over some linear or logarithmic parameter range. If you wanted to explore some parameter space in a non uniform way or many more dimensions, you would have to use `noxaxis` and call the FINESSE 2 C binary multiple times in Python, via a for-loop, for example. Computing multiple xaxis at a time was also not possible due to the structure of the C code.

Calling the v2 binary builds the model each time it is invoked; it allocated memory, scanning some parameters, computing detectors outputs, writing to an ASCII output file, freeing memory, then loading the output file into Python for processing. Obviously for a large model like a LIGO file each binary call was quite costly when using many higher-order modes and locks. Take a common example such as computing multiple input and output transfer functions. In v2 only a single input could be computed at once. So for each excitation the model would have to be built, the operating point found with locks (which was costly), and the transfer functions calculated. That state of the simulation would be thrown away each time between binary calls.

Using actions to compose an analyses to perform on a model is a solution to the problems mentioned above. They allow a user to build custom analyses from simpler blocks that would have taken many lines of code in Pykat and FINESSE 2. This page aims to introduce the basics of actions and how to start composing more complicated analyses.

The 'xaxis' and 'noxaxis' action

In FINESSE 3 we have introduced the concept of an *analysis* that is to be performed on a model. An analysis can be simple or very complex. They are composed of single or multiple atomic simulation tasks we call `actions`. For example, `xaxis` (page 209) is an action - its task is to sweep some parameter and computes all the detector outputs and returns a solution object with its result, and finally reset all the values it has changed during the sweep.

Take the following simple example consisting of a laser beam incident on a photodetector reading out the power measured as we vary the power:

> **Note**
>
> Action KatScript uses function style syntax (page 127). This allows actions to be nested, to span multiple lines, and to contain comments.

```
import finesse
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")
```

```
<finesse.model.Model at 0x7ef47f79fe00>
```

To run an action you can simply pass it as an argument to run:

```
sol = model.run("xaxis(l1.P, lin, 0, 1, 10)")
print(sol)
print(sol['P'])
```

```
- Solution Tree
● xaxis - ArraySolution
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

Running the model returns a solution generated by the analysis - if one is generated. An `Xaxis` (page 209) action produces an `ArraySolution`. The array solution allows you to access the detector outputs using the detector names. Remember that different actions can produce different types of solutions.

Stored actions

The last action that was run is stored in the `analysis` member. When you run a model with no arguments it will perform whatever analysis is set in `analysis`.

```
model.analysis
```

We can see from the `xaxis` (page 209) used above that it was set to an `Xaxis` (page 209) object. You can parse a new action KatScript if you wish and that will store it in the model.

```
sol = model.run("noxaxis()")
print(model.analysis)
print(sol)
print(sol['P'])
```

```
None
- Solution Tree
● noxaxis - ArraySolution
1.0000000000000002
```

This pattern of running an action is not recommended and we suggest passing the action directly to the run method. This is so that it is clear what is being run to generate a particular solution.

Python API

As with the rest of FINESSE 3 any feature that is accessible via KatScript can also be accessed and interacted with via the Python API. All the built-in actions are available in `finesse.analysis.actions`. Each KatScript command is associated with a similarly named class in this module. The class names usually use a longer *CamelCase* style.

Reusing our simple example, the `Xaxis` (page 209) action can be used in a similar manner. Rather than adding the analysis to the model we just run it directly and provide the model to run it on.

```
import finesse
from finesse.analysis.actions import Xaxis

model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")
sweep_power = Xaxis('l1.P', 'lin', 0, 1, 10)
sol = model.run(sweep_power)
print(sol['P'])
```

```
[0.  0.1 0.2 0.3 0.4 0.5 0.6 0.7 0.8 0.9 1. ]
```

Which produces exactly the same result as the KatScript interface. The Python API allows you to programmatically build up actions in scripts if needed. Action events are similarly set using the keyword-argument of the action classes.

Solutions

Solutions replace the the `.out` file generated in FINESSE 2, there we were very limited by the type of data we could output. In FINESSE 3 solutions are introduce to fix this issue and enable a wide variety of outputs to be produced.

As mentioned, each action may or may not produce a solution. Some actions just perform some task but do not generate any particular output. Others like `xaxis` (page 209) generate an array of detector outputs. Others like `freqresp` (page 221) can compute multiple-input-multiple-output transfer function matrices.

Each solution generated is domain specific and can contain very different data so it is worthwhile referring to the documentation of each action and its solution to determine how to use it. Broadly those solutions that generate data also contain helper functions such as *plot()* to quickly make standard plots.

Analysis with multiple actions - 'series'

The real power now comes from being able to chain together multiple actions to perform in a single analysis. This enables us to run more complex simulations without having to resort to writing a lot of boilerplate code that was required in Pykat. Say we want to run two `xaxis` (page 209) actions but only build the model once. For this we use the `series` (page 207) action. This runs each action it is given sequentially in the same simulation.

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")

sol = model.run("""
    series(
        xaxis(l1.P, lin, 0, 1, 10),
        xaxis(l1.P, log, 1, 10, 10)
    )
""")
print(sol)
display(sol)
```

```
- Solution Tree
o series - SeriesSolution
├──● xaxis - ArraySolution
└──● xaxis - ArraySolution
```

```
<SeriesSolution of series @ 0x7ef47f6d1400 children=2>
```

Running it we now see our solution is a bit more complicated and it has two children, as we have two array solutions.

The solutions are essentially a tree structure, where each solution can have multiple children, depending on what analysis you create.

```
sol.children
```

```
[<ArraySolution of series/xaxis @ 0x7ef47f6a6400 children=0>,
 <ArraySolution of series/xaxis @ 0x7ef47f6a4ba0 children=0>]
```

The child solutions can also be accessed in multiple ways, by numerical index:

```
sol[0], sol[1]
```

```
(<ArraySolution of series/xaxis @ 0x7ef47f6a6400 children=0>,
 <ArraySolution of series/xaxis @ 0x7ef47f6a4ba0 children=0>)
```

Or by name:

```
sol['xaxis']
```

```
<finesse.solutions.array.ArraySolutionSet at 0x7ef47f8381a0>
```

You'll notice that this last option returns an `ArraySolutionSet` which is combined set of array solutions generated by an `xaxis` (page 209). This requires the number of steps to be the same in each `xaxis` (page 209) used as a single large array is generated. You can then access outputs from this combined set:

```
sol['xaxis']['P']
```

```
array([[ 0.        ,  0.1       ,  0.2       ,  0.3       ,  0.4       ,
         0.5       ,  0.6       ,  0.7       ,  0.8       ,  0.9       ,
         1.        ],
       [ 1.        ,  1.25892541,  1.58489319,  1.99526231,  2.51188643,
         3.16227766,  3.98107171,  5.01187234,  6.30957344,  7.94328235,
        10.        ]])
```

We can give actions names for easier access too:

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")

sol = model.run("""
    series(
        xaxis(l1.P, lin, 0, 1, 10, name='xaxis1'),
        xaxis(l1.P, log, 1, 10, 10, name='xaxis2')
    )
""")
print(sol['xaxis1'])
print(sol['xaxis2'])
```

```
- Solution Tree
● xaxis1 - ArraySolution
- Solution Tree
● xaxis2 - ArraySolution
```

Actions and model state changes

The series (page 207) action internally passes the state of the model and simulation from one action to the next. The model state refers to the value of all the model elements, parameters, settings, and model layout. This allows one action to change the state of the model, then the next one acts on that. We can try this with the same example from above:

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")

sol = model.run("""
    series(
        xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis1'),
        xaxis(l1.P, lin, 0, 10, 1, relative=true, name='xaxis2')
    )
""")
print(sol['xaxis1']['P'])
print(sol['xaxis2']['P'])
```

```
[1. 2.]
[ 1. 11.]
```

The xaxis (page 209) actions change the laser power relative to its initial value (`xaxis*` in KatScript 2). You'll see from the output that the initial value is the same. This is because the xaxis (page 209) actions revert the state of the model after it has swept the parameters. If we wanted to change the state of the model so that the laser power is different we can use the change (page 235) action:

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")
sol = model.run("""
    series(
        xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis1'),
        change(l1.P=2),
        xaxis(l1.P, lin, 0, 10, 1, relative=true, name='xaxis2')
    )
""")
print(sol['xaxis1']['P'])
print(sol['xaxis2']['P'])
```

```
[1. 2.]
[ 2. 12.]
```

Now we see the second solution starts from 2. The change (page 235) action can alter several variables at once `change(l1.p=2, l1.phase=45)`.

It is important to remember that series (page 207) does not deep-copy the model it is asked to run on, therefore any state changes made will still be present at the end of the analysis.

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")
sol = model.run("""
    series(
```

(continues on next page)

(continued from previous page)

```
        change(l1.P=2),
        noxaxis(),
        change(l1.P=3),
        noxaxis()
    )
""")
print(model.l1.P)
```

```
3.0 W
```

We see that the laser power in our original model is now 3W instead of 1W. The reason that actions can leave behind state changes is so that more advanced actions can be used. For example, running locks, optimising parameters, or finding the correcting operating point of an interferometer. Overall reducing the amount of times a model is copied is ideal as it is computationally expensive for larger models.

Analysis with multiple actions - 'parallel'

Note

The `parallel` (page 207) action is not multithreaded in the current version. This feature will be added in an upcoming version. See **#834**

Along with `series` (page 207) there is also `parallel` (page 207). This creates copies of the model and restarts the simulation for each action in the parallel arguments. Essentially this is equivalent to writing python code to deep-copy the model object and run each action one after another. Eventually, this will automatically handle running multiple simulation concurrently, via `multiprocessing`, threads, or distributing them to some computing cluster. As the `parallel` (page 207) action copies the model, any state changes made will not be present in the initial model that was run, unlike `series` (page 207).

Take our previous example, using `parallel` (page 207) we see that our relative change in the laser power is not doing anything different.

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")
sol = model.run("""
    parallel(
        change(l1.P=2),
        xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis1'),
        xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis2')
    )
""")
print(sol['xaxis1']['P'])
print(sol['xaxis2']['P'])
```

```
[1. 2.]
[1. 2.]
```

To run multiple actions in parallel we need to build up our action using `series` (page 207). You should note that the solution has a more complicated structure now.

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
```

(continues on next page)

(continued from previous page)

```
""")
sol = model.run("""
    parallel(
        series(
            change(l1.P=2),
            xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis1')
        ),
        series(
            change(l1.P=10),
            xaxis(l1.P, lin, 0, 1, 1, relative=true, name='xaxis1')
        )
    )
""")
print(sol)
```

```
- Solution Tree
O parallel - ParallelSolution
├──O series - SeriesSolution
|   └──● xaxis1 - ArraySolution
└──O series - SeriesSolution
    └──● xaxis1 - ArraySolution
```

Care must be taken when extracting out solutions from these states. There are various ways to extract each one, or select multiple

```
display(sol[0, 'xaxis1']) # the first parallel action xaxis solution
display(sol['series']) # a tuple of the solutions generated by a series
display(tuple(s['xaxis1'] for s in sol['series'])) # Generator syntax to extract each xaxis␣
↪solution
```

```
<ArraySolution of parallel/series/xaxis1 @ 0x7ef47ec15be0 children=0>
```

```
<finesse.solutions.base.SolutionSet at 0x7ef47ea8c440>
```

```
(<ArraySolution of parallel/series/xaxis1 @ 0x7ef47ec15be0 children=0>,
 <ArraySolution of parallel/series/xaxis1 @ 0x7ef47ec16c20 children=0>)
```

Improved indexing will likely be added in later versions. Currently slices can only be used in the final index which means `sol[:, 'xaxis1']` in the above scenario would not work.

Extracting model states from an analysis

In some instances we might want to perform a complex analysis on a model which involves copying it but still access the final state of the model generated. For example, an analysis that might perform multiple optimisations in a parallel action where we want to get the final model states.

To return these internally generated states we can set the `return_state` flag in `run`:

```
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")

sol, state = model.run("""
    parallel(
        change(l1.P=2),
```

(continues on next page)

(continued from previous page)

```
        change(l1.P=3)
    )
""", return_state=True)
print("Initial model =", model)
print(state)
```

```
Initial model = <finesse.model.Model object at 0x7ef47ece9a90>
O AnalysisState <finesse.model.Model object at 0x7ef47ece9a90>
├──O AnalysisState <finesse.model.Model object at 0x7ef47ece9950>
└──O AnalysisState <finesse.model.Model object at 0x7ef47ece9e50>
```

The returned `state` object is another tree structure. It is the root state of the analysis tree that is internally generated. What we can do with this tree is dig into the children and extract the model. Currently there is no accessing the states via the names of the actions, which may be implemented at a later date.

```
state.children[0].model.l1.P
```

```
<l1.P=2.0 @ 0x7ef47ea814e0>
```

Action events

Certain actions have *events* which you can run actions on when they occur. Please refer to the documentation on each action for detailed descriptions of any events they may expose.

Let's look at two events which will are commonly used in the `xaxis` (page 209) action, *pre_step* and *post_step*. These events are called before and after each step along the axis. Therefore we are able to run an action after each step, such as running locks, or computing a whole other `xaxis` (page 209) action. Here we simply demonstrate this using the `print_model_attr` (page 237) action, which just prints the current value of some model value. We see the first pre-step is the initial value of the laser power.

```
import finesse
model = finesse.Model()
model.parse("""
l l1 P=1
pd P l1.p1.o
""")

sol = model.run("""
    xaxis(
        l1.P, lin, 10, 100, 3,
        pre_step=print_model_attr(l1.P),
        post_step=print_model_attr(l1.P)
    )
""")
```

```
 l1.P=10.0
 l1.P=10.0
 l1.P=40.0
 l1.P=40.0
 l1.P=70.0
 l1.P=70.0
 l1.P=100.0
 l1.P=100.0
```

In this example we run a whole other xaxis for each post-step along the initial xaxis.

```
import finesse
model = finesse.Model()
model.parse("""
```

(continues on next page)

(continued from previous page)

```
l l1 P=1
pd P l1.p1.o
""")
sol = model.run("""
    xaxis(
        l1.P, lin, 10, 100, 10,
        post_step=xaxis(
            l1.phase, lin, 0, 90, 100
        )
    )
""")
print(sol)
```

```
- Solution Tree
● xaxis - ArraySolution
└──○ post_step
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   ├──● xaxis - ArraySolution
   └──● xaxis - ArraySolution
```

If we look at the solution generated we see there is an additional entry called `post_step` where all our xaxis solutions are now stored. We are now able to access them all as a set of array solutions extracting detector outputs from all of the solutions

```
sol['post_step', 'xaxis']['P']
```

```
array([[ 10.,  10.,  10., ...,  10.,  10.,  10.],
       [ 19.,  19.,  19., ...,  19.,  19.,  19.],
       [ 28.,  28.,  28., ...,  28.,  28.,  28.],
       ...,
       [ 82.,  82.,  82., ...,  82.,  82.,  82.],
       [ 91.,  91.,  91., ...,  91.,  91.,  91.],
       [100., 100., 100., ..., 100., 100., 100.]], shape=(11, 101))
```

Or the solution objects individually as an iterable:

```
sol['post_step', :]
```

```
[<ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb602b0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60380 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60450 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60520 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb605f0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb606c0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60790 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60860 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60930 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60a00 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60ad0 children=0>]
```

Or via the usual solution tree node object:

```
sol['post_step'].children
```

```
[<ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb602b0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60380 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60450 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60520 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb605f0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb606c0 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60790 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60860 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60930 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60a00 children=0>,
 <ArraySolution of xaxis/post_step/xaxis @ 0x7ef47eb60ad0 children=0>]
```

Groups

series
Series

A sequential series of actions to apply during a simulation.

Syntax

```
series(*actions, flatten=true, name=none)
```

Required

`actions`: A collection of Actions to run in series

`flatten`: When True, each action will be stored in the top level of the solution tree. When False, each action will be a child of the previous solution generated.

`name`: Optional name for the solution generated by these actions

See Also

`parallel` (page 207)

parallel
Parallel

Syntax

```
parallel(*actions)
```

See Also

`series` (page 207)

for
For

An action changes a parameter value and runs a set of actions for each value in the array. Essentially the same as *Series* combined with *Change*. The parameter value is not reset to its original value once this action has finished.

Generates a `ForSolution` output when run.

Syntax

```
for(param, values, *actions)
```

Required

`param`: Parameter to change in the model

`values`: Array of values to use for the parameter

`\*actions`: Actions to run for each parameter value.

See Also

`series` (page 207)

`change` (page 235)

Axes

The following actions are all variations of a parameter 'sweep' or 'axis'; you want to modify the values of one more parameters of the model and solve for the detector outputs for each value.

- Noxaxis (page 208)
- Xaxis (page 209)
- X2axis (page 213)
- X3axis (page 215)
- Sweep (page 218)

Noxaxis

noxaxis
Noxaxis

The default Action for a model when no action is specified. Solves the model without changing any parameters. Will return a solution with a single value for every detector in the model.

Syntax

```
noxaxis(pre_step=none, post_step=none, name='noxaxis')
```

Optional

`pre_step`: An action to perform before the Noxaxis is run, by default None

`post_step`: An action to perform after the Noxaxis is run, by default None

`name`: Name of the action, used to find the solution in the final output, by default "noxaxis"

This action will be run when you call `Model.run` without specifying an Action.

```
from finesse import Model
from finesse.analysis.actions import Noxaxis

model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1 R=0.5 T=0.5
pd p_in l1.p1.o
pd p_refl m1.p1.o
"""
)
# the following are equivalent
sol = model.run()
sol = model.run(Noxaxis())

sol.print_detector_info()
print(sol["p_in"])
print(sol["p_refl"])
```

```
{
  "p_in": {
    "name": "p_in",
    "detector_type": "finesse.detectors.powerdetector.PowerDetector",
    "dtype": "numpy.float64",
    "unit": "W",
    "label": "Power"
  },
  "p_refl": {
    "name": "p_refl",
    "detector_type": "finesse.detectors.powerdetector.PowerDetector",
    "dtype": "numpy.float64",
    "unit": "W",
    "label": "Power"
  }
}
1.0000000000000002
0.5000000000000002
```

This produces a single value for every detector in the model. Note that `sol.plot` will not work in this case.

```
sol.plot()
```

```
/usr/local/lib/python3.13/site-packages/finesse/plotting/plot.py:897: UserWarning:␣
↪No x-axes have been defined, unable to plot the outputs of any detectors of type␣
↪<class 'finesse.detectors.powerdetector.PowerDetector'>
  warn(
```

```
{}
```

Xaxis

xaxis
Xaxis

Sweeps a parameter and solves the model for every value in the sweep. Will a solution with an array of values for every detector in the model.

Syntax

```
xaxis(
    parameter,
    mode,
    start,
    stop,
    steps,
    relative=false,
    pre_step=none,
    post_step=none,
    name='xaxis'
)
```

Required

`parameter`: Parameter to sweep over.

`mode`: Whether to sweep linearly or logarithmically

`start`: Starting value of the parameter sweep

`stop`: Final value of the parameter sweep.

`steps`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional

`relative`: Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False

`pre_step`: An action to perform before each step is computed, by default None

`post_step`: An action to perform after each step is computed, by default None

`name`: Name of the action, used to find the solution in the final output, by default "xaxis"

See Also

`x2axis` (page 213), `x3axis` (page 215), `sweep` (page 218)

Sweeps a parameter and solves the model for each step. The returning `ArraySolution` contains a NumPy array for every detector, and some useful metadata of the analysis that was performed.

```
from finesse import Model, init_plotting
from finesse.analysis.actions import Xaxis

from matplotlib import pyplot as plt

init_plotting()

model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1 R=0.5 T=0.5
pd p_in l1.p1.o
pd p_refl m1.p1.o
"""
)
sol = model.run(
    Xaxis(
        parameter=model.l1.P,
        mode="lin",
        start=1,
        stop=10,
        steps=3
    )
)
sol.print_detector_info()
print(f"{sol.axis_info=}")
print(f"{sol.params=}")
print(f"{sol["p_in"]=}")
print(f"{sol["p_refl"]=}")
```

```
{
  "p_in": {
    "name": "p_in",
    "detector_type": "finesse.detectors.powerdetector.PowerDetector",
    "dtype": "numpy.float64",
    "unit": "W",
    "label": "Power"
  },
  "p_refl": {
    "name": "p_refl",
    "detector_type": "finesse.detectors.powerdetector.PowerDetector",
    "dtype": "numpy.float64",
    "unit": "W",
    "label": "Power"
  }
}
```

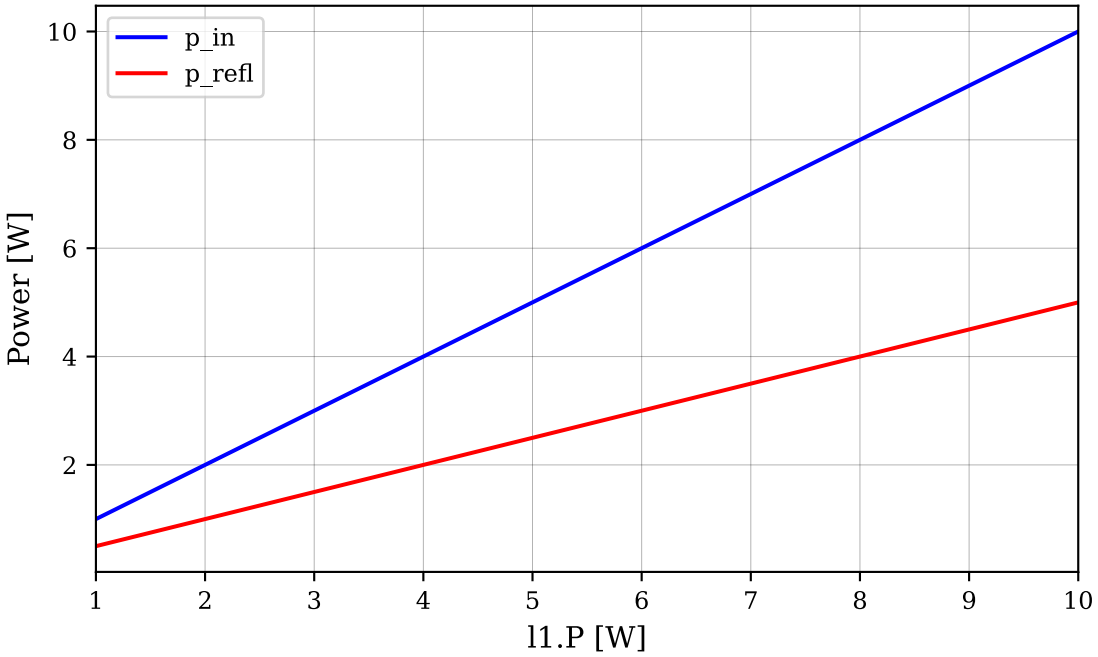
(continues on next page)

(continued from previous page)

```
sol.axis_info={'l1.P': {'name': 'l1.P', 'component': 'l1', 'unit': 'W'}}
sol.params=('l1.P',)
sol["p_in"]=array([ 1.,  4.,  7., 10.])
sol["p_refl"]=array([0.5, 2. , 3.5, 5. ])
```

Default plotting will work on an `ArraySolution`. Detectors of the same type will be plotted on the same plot:

```
fig_dict = sol.plot()
```



The dictionary returned by the `plot()` method contains the figures produced. The dictionary is indexed by two types of keys:

- The string names of the detectors with the corresponding `matplotlib.figure.Figure` object as value
- The *types* of detectors with a list of all `matplotlib.figure.Figure` objects plotting that type of detector

```
for key, val in fig_dict.items():
    print(f"{key}: {val}")
```

```
<class 'finesse.detectors.powerdetector.PowerDetector'>: Figure(576x355.968)
p_in: Figure(576x355.968)
p_refl: Figure(576x355.968)
```

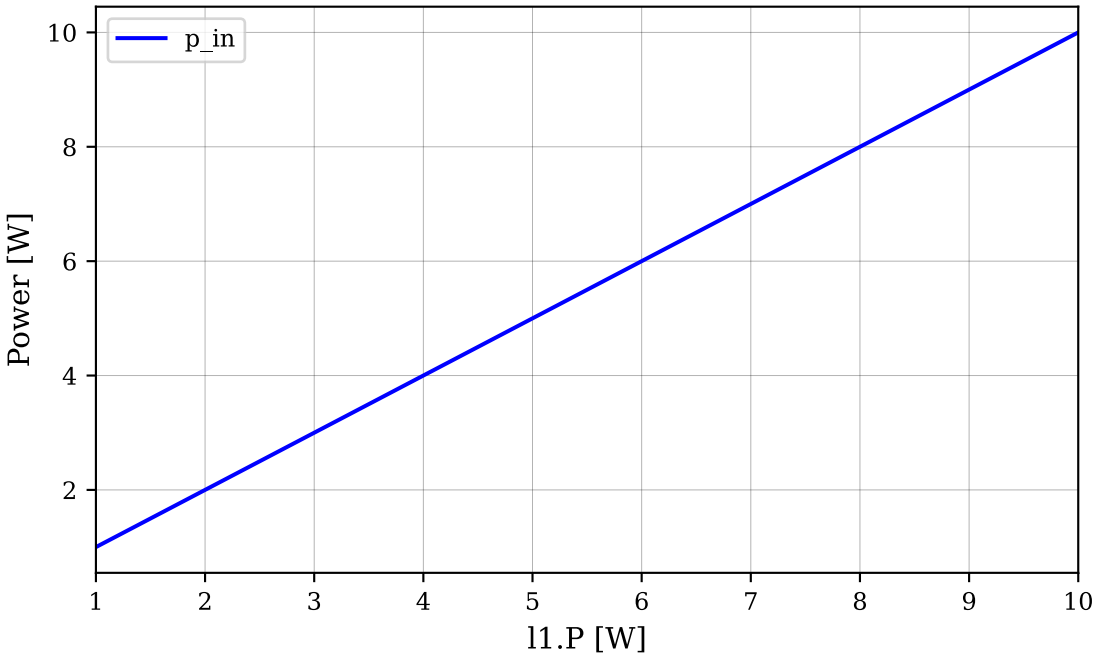
You can use this to customise the figures created by FINESSE:

```
fig = fig_dict["p_in"]
fig.axes[0].set_title("New title")
fig.axes[0].grid(False)

plt.show()
```

You can also only plot specific detectors by specifying their names in the `plot()` call.

```
sol.plot("p_in");
```



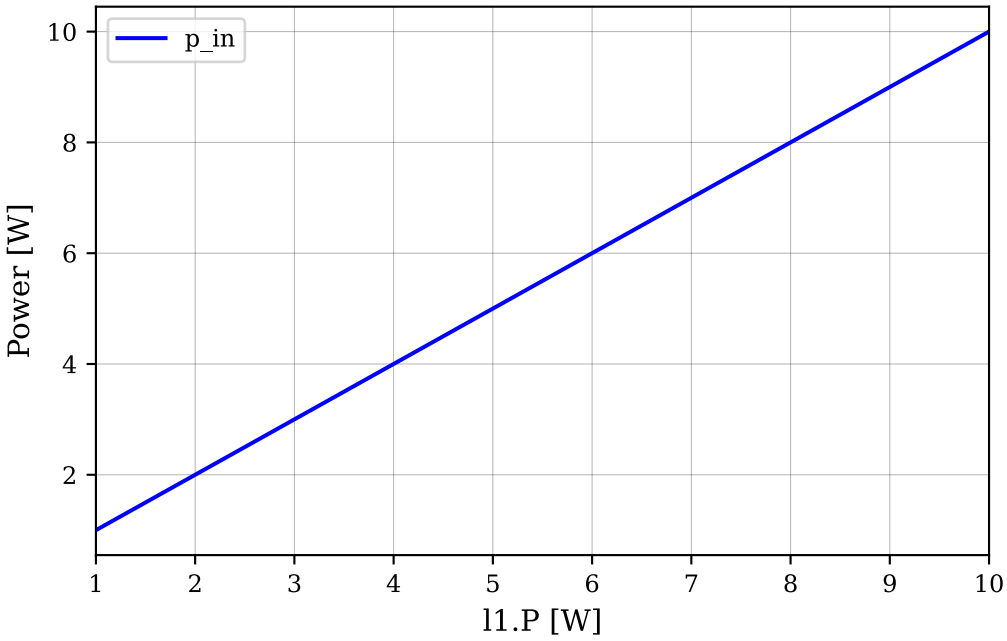
The original sweep axis is available under the `x` attribute. This attribute contains a tuple of arrays, to be compatible with multi-dimensional sweeps. You can access the first axis with the short hand `x1` or by indexing the tuple `x[0]`.

```
print(sol.x)  # tuple of arrays
print(sol.x1)  # shorthand
print(sol.x[0])  # first sweep array
```

```
(array([ 1.,  4.,  7., 10.]),)
[ 1.  4.  7. 10.]
[ 1.  4.  7. 10.]
```

Together with the information in `trace_info` and `axis_info`, we can use these attributes to create our own plot:

```
detector = "p_in"
trace_info = sol.trace_info["p_in"]
axis_info = sol.axis_info[sol.params[0]]
plt.xlabel(f"{axis_info['name']} [{axis_info['unit']}]")
plt.ylabel(f"{trace_info['label']} [{trace_info['unit']}]")
plt.plot(sol.x[0], sol[detector], label=detector)
plt.legend()
plt.show()
```



X2axis

x2axis
X2axis

Two-dimensional version of the Xaxis action. Will solve the model for every point on a grid of values defined by the two sets of sweep arguments. Returns a solution with a two-dimensional array for every detector in the model.

Significantly faster than calling Xaxis in a Python for loop.

Syntax

```
x2axis(
    parameter1,
    mode1,
    start1,
    stop1,
    steps1,
    parameter2,
    mode2,
    start2,
    stop2,
    steps2,
    relative=false,
    pre_step=none,
    post_step=none,
    name='x2axis'
)
```

Required

`parameter1`: Parameter to sweep over.

`mode1`: Whether to sweep linearly or logarithmically

`start1`: Starting value of the parameter sweep

`stop1`: Final value of the parameter sweep.

`steps1`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

`parameter2`: Parameter to sweep over.

`mode2`: Whether to sweep linearly or logarithmically

`start2`: Starting value of the parameter sweep

`stop2`: Final value of the parameter sweep.

`steps2`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional

`relative`: Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False, by default False

`pre_step`: An action to perform before each step is computed, by default None

`post_step`: An action to perform after each step is computed, by default None

`name`: Name of the action, used to find the solution in the final output, by default "x2axis"

See Also

`xaxis` (page 209), `x3axis` (page 215), `sweep` (page 218)

A two-dimensional version of `xaxis` (page 209). Each detector in the solution will produce a two-dimensional array.

```
from finesse import Model, init_plotting
from finesse.analysis.actions import X2axis

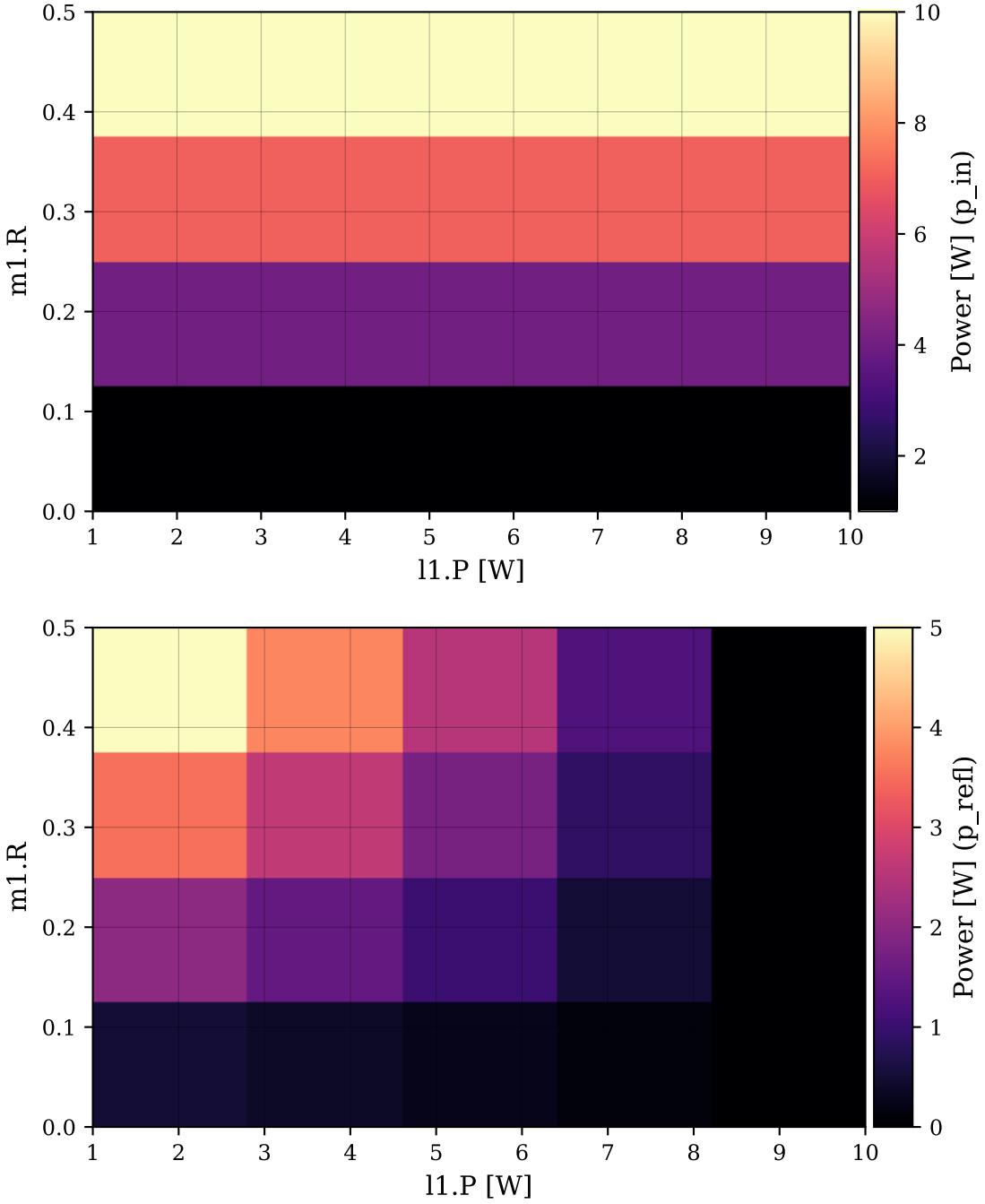
init_plotting()

model = Model(
    """
l l1
s s1 l1.p1 m1.p1
m m1 R=0.5 T=0.5
pd p_in l1.p1.o
pd p_refl m1.p1.o
"""
)
sol = model.run(
    X2axis(
        parameter1=model.l1.P,
        mode1="lin",
        start1=1,
        stop1=10,
        steps1=3,
        parameter2=model.m1.R,
        mode2="lin",
        start2=0.5,
        stop2=0.0,
        steps2=4,
    )
)
print(f"{sol.shape=}")  # 3x4 steps will produce an array of size 4x5
print(sol["p_in"])  # two-dimensional array
print(sol["p_in"][:, 0])  # 1st column is equal to the first sweep over laser power
print(sol["p_in"][0, :])  # 1st row is equal to the first sweep over mirror reflectivity
```

```
sol.shape=(4, 5)
[[ 1.  1.  1.  1.  1.]
 [ 4.  4.  4.  4.  4.]
 [ 7.  7.  7.  7.  7.]
 [10. 10. 10. 10. 10.]]
[ 1.  4.  7. 10.]
[1. 1. 1. 1. 1.]
```

The default `plot` method will produce a 2D histogram for every detector.

```
fig_dict = sol.plot()
```



X3axis

x3axis
X3axis

Three-dimensional version of the Xaxis action. Will solve the model for every point on a cube of values defined by the three sets of sweep arguments. Returns a solution with a three-dimensional array for every detector in the model.

Significantly faster than calling Xaxis in a nested Python for loop.

Syntax

```
x3axis(
    parameter1,
    mode1,
    start1,
    stop1,
    steps1,
    parameter2,
    mode2,
    start2,
    stop2,
    steps2,
    parameter3,
    mode3,
    start3,
    stop3,
    steps3,
    relative=false,
    pre_step=none,
    post_step=none,
    name='x3axis'
)
```

Required

`parameter1`: Parameter to sweep over.

`mode1`: Whether to sweep linearly or logarithmically

`start1`: Starting value of the parameter sweep

`stop1`: Final value of the parameter sweep.

`steps1`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

`parameter2`: Parameter to sweep over.

`mode2`: Whether to sweep linearly or logarithmically

`start2`: Starting value of the parameter sweep

`stop2`: Final value of the parameter sweep.

`steps2`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

`parameter3`: Parameter to sweep over.

`mode3`: Whether to sweep linearly or logarithmically

`start3`: Starting value of the parameter sweep

`stop3`: Final value of the parameter sweep.

`steps3`: Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional

`relative`: Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False, by default False

`pre_step`: An action to perform before each step is computed, by default None

`post_step`: An action to perform after each step is computed, by default None

`name`: Name of the action, used to find the solution in the final output, by default "x3axis"

See Also

`xaxis` (page 209), `x2axis` (page 213), `sweep` (page 218)

Three-dimensional version of x2axis (page 213). Solves the model for a cube of different parameter values. The cubic scaling can mean very long runtimes or memory requirements if you are not careful with specifying the number of steps per axis.

```
from finesse import Model, init_plotting
from finesse.analysis.actions import X3axis

init_plotting()

model = Model(
    """
l l1
s s1 l1.p1 m1.p1

m m1 R=0.5 L=0.0
s s2 m1.p2 m2.p1
m m2 R=0.5 L=0.0

pd p_circ m1.p2.i
"""
)
sol = model.run(
    X3axis(
        parameter1=model.l1.P,
        mode1="lin",
        start1=1.0,
        stop1=0.5,
        steps1=4,
        parameter2=model.m2.R,
        mode2="lin",
        start2=0.5,
        stop2=0.99,
        steps2=5,
        parameter3=model.m1.phi,
        mode3="lin",
        start3=0.0,
        stop3=90.0,
        steps3=10,
    )
)
print(f"{sol.shape=}")  # 3x4x5 steps will produce an array of size 4x5x6
```

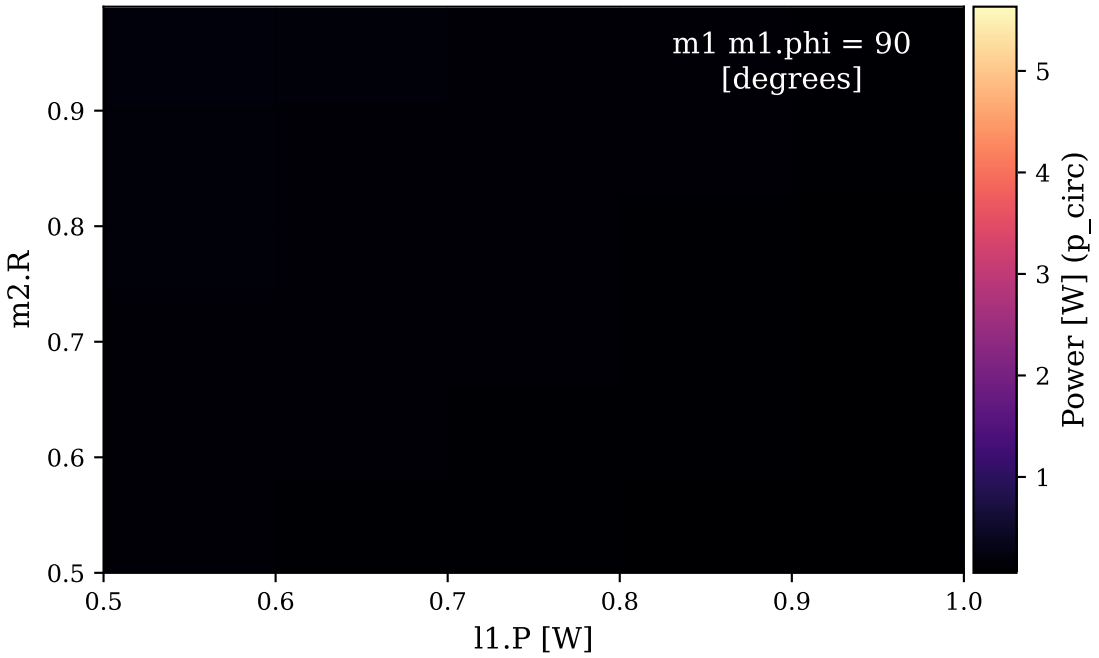
```
sol.shape=(5, 6, 11)
```

The default plot method will produce an animated 2D-plot, using the third axis specified as the time axis.

```
from IPython.display import HTML

fig_dict = sol.plot(show=False)
anim = fig_dict[1]['p_circ']
HTML(anim.to_jshtml())
```

```
<IPython.core.display.HTML object>
```



Sweep

sweep
Sweep

An action that sweeps N number of parameters through the values in N arrays.

Syntax

```
sweep(
    *args,
    pre_step=none,
    post_step=none,
    reset_parameter=true,
    name='sweep'
)
```

Required

`args`: Expects 3 arguments per axis. The first is a full name of a Parameter or a Parameter object. The second is an array of values to step this parameter over, and lastly a boolean value to say whether this is a relative step from the parameters initial value.

`name`: Name of the action, used to find the solution in the final output.

Optional

`pre_step`: An action to perform before the step is computed

`post_step`: An action to perform after the step is computed

`reset_parameter`: When true this action will reset the all the parameters it changed to the values before it ran.

See Also

`xaxis` (page 209), `x2axis` (page 213), `x3axis` (page 215)

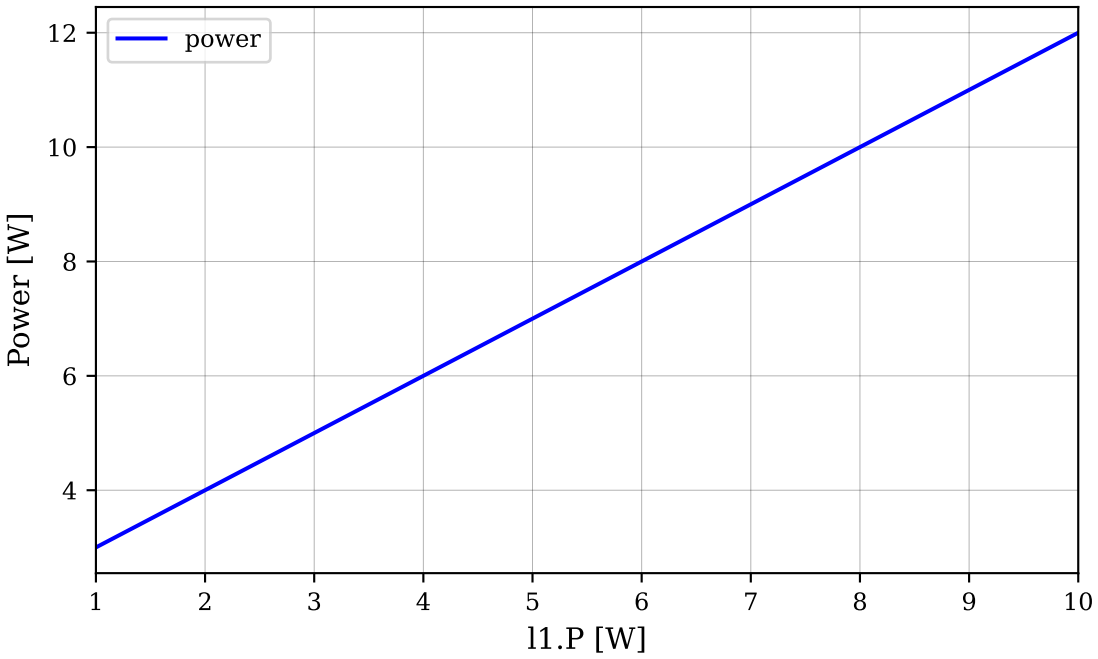
All of the 'xaxis' Actions are based on 'Sweep'. It simply takes groups of three arguments representing a parameter, an array of values to sweep over and a boolean to specify if the values should be applied on top of the current parameter value. Can be useful if you want to sweep over parameter values from a data file or specify a distribution that is not linear or logarithmic.

```
from finesse import Model, init_plotting
from finesse.analysis.actions import Sweep

init_plotting()

model = Model(
    """
l l1 P=2
pd power l1.p1.o
"""
)
sol = model.run(Sweep(model.l1.P, [1, 5, 10], True))
print(f"{sol.x1=}")
print(f"{sol["power"]=}")
sol.plot()
```

```
sol.x1=array([ 1.,  5., 10.])
sol["power"]=array([ 3.,  7., 12.])
```




```
{finesse.detectors.powerdetector.PowerDetector: <Figure size 576x355.968 with 1 Axes>,
 'power': <Figure size 576x355.968 with 1 Axes>}
```

Frequency Response

- Input and output nodes (page 220)
- Solution indexing (page 220)
- Model setup (page 220)
- Frequency Response (page 221)
- Frequency Response 2 (page 223)
- Frequency Response 3 (page 225)
- Frequency Response 4 (page 227)

This page gives an overview of the four different "FrequencyResponse" actions which can be used to calculate transfer functions (page 119) in FINESSE. See also the section on signal modelling (page 120) for additional background information

For a detailed overview with example code, see Calculating general transfer functions (page 272). The following examples also showcase `FrequencyResponse`:

- Angular radiation pressure (page 85)
- Optical springs (page 90)
- Radiation pressure (page 92)

Finally you can find a more general overview on transfer functions on Transfer Functions and Error Signals (page 119).

The different frequency response actions differ mainly in the type of nodes used as input and output for the transfer function calculation. See Basic usage of ports and nodes (page 139) for an overview of the difference between node types.

Input and output nodes

Action Name	Input Nodes	Output nodes
`FrequencyResponse` (page 221)	Electrical/Mechanical	Electrical/Mechanical
`FrequencyResponse2` (page 223)	Optical	Electrical/Mechanical
`FrequencyResponse3` (page 225)	Optical	Optical
`FrequencyResponse4` (page 227)	Electrical/Mechanical	Optical

Solution indexing

Below is a quick overview of the indexing of the `out` attribute of the `FrequencyResponseSolution` returned by the analyses on this page.

Action Name	Index 1	Index 2	Index 3	Index 4	Index 5
`FrequencyResponse` (page 221)	frequencies	outputs	inputs	N/A	N/A
`FrequencyResponse2` (page 223)	frequencies	outputs	inputs	HOMs (input)	N/A
`FrequencyResponse3` (page 225)	frequencies	outputs	inputs	HOMs (output)	HOMs (input)
`FrequencyResponse4` (page 227)	frequencies	outputs	inputs	HOMs (output)	N/A

Model setup

Note that it is also possible to calculate transfer functions 'manually' using the `sgen` (page 169) to inject signals, this is described in Computing transfer functions - manual method (page 267).

For this page we'll use a simple cavity setup where one of the mirrors is suspended to demonstrate the differences between the different frequency response actions.

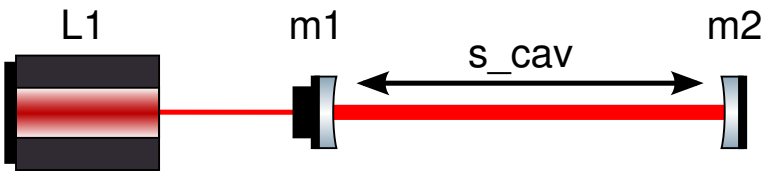


Fig. 6.7: Cavity setup

```
import numpy as np

import finesse
from finesse.analysis.actions import (
    FrequencyResponse,
    FrequencyResponse2,
    FrequencyResponse3,
    FrequencyResponse4,
)
from finesse.plotting import bode

finesse.init_plotting()

model = finesse.Model("""
l l1 P=1
s s1 l1.p1 m1.p1 L=1
m m1 R=0.9 T=0.1
s s2 m1.p2 m2.p1 L=1
m m2 R=0.9 T=0.1

pendulum pend m1 mass=1

fsig(1)
""")
```

Frequency Response

frequency_response
freqresp
FrequencyResponse

Computes the frequency response of a signal injected at various nodes to compute transfer functions to multiple output nodes. Inputs and outputs should be electrical or mechanical nodes. It does this in an efficient way by using the same model and solving for multiple RHS input vectors.

This action does not alter the model state. This action will ignore any currently definied signal generator elements in the model. Produces an output transfer matrix from each input node to some readout output. The shape of the output matrix is: [frequencies, inputs, outputs] To inject into optical nodes please see `FrequencyResponse2` (page 223) and `FrequencyResponse3` (page 225). To read out optical nodes please see `FrequencyResponse3` (page 225) and `FrequencyResponse4` (page 227).

Syntax

```
freqresp(f, inputs, outputs, carrier_solution=none, name='frequency_
↪response')
```

Required

`f`: Frequencies to compute the transfer functions over

`inputs`: Mechanical or electrical nodes to inject signal at

`outputs`: Mechanical or electrical nodes to measure output at

Optional

`carrier_solution`: When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.

`name`: Solution name

See Also

`freqresp2` (page 223), `freqresp3` (page 225), `freqresp4` (page 227)

As mentioned above (page 220), `FrequencyResponse` calculates transfer functions between signal nodes (electrical or mechanical). We define a range of frequencies to sweep over and the input nodes (where the signal will be injected) and the output nodes (where the signal will be "measured"). In this case we will modulate the laser amplitude to see the mechanical resonance of the suspended mirror.

```
f = np.geomspace(1e-1, 1e2, 100)
sol = model.run(
    FrequencyResponse(f, inputs=[model.l1.amp.i], outputs=[model.m1.mech.z])
)
```

Since both the input and the output nodes are signal nodes, their value is single complex number and the resulting shape of the `sol.out` array is `N_f` x `N_inputs` x `N_outputs` (see also Solution indexing (page 220)).

```
print(sol.out.shape)
print(sol.f.shape)
print(sol.inputs)
print(sol.outputs)
```

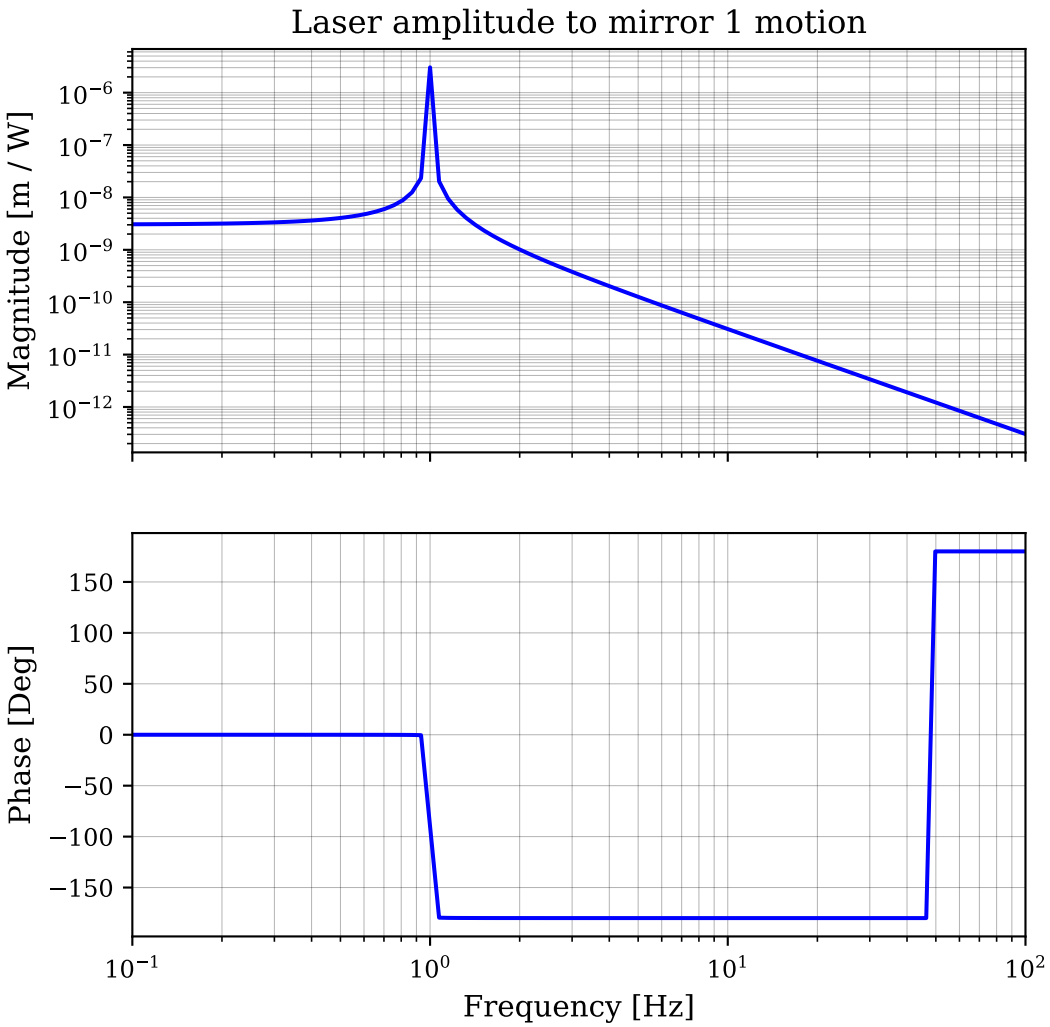
```
(100, 1, 1)
(100,)
['l1.amp.i']
['m1.mech.z']
```

To plot the result, we use the `bode` utility function. We index the `sol.out` array with `[:, 0, 0]` to select the transfer function from `l1.amp.i` to `m1.mech.z`. Note that we can also select the transfer function using the node names, which is useful if we are calculating multiple transfer functions at once.

```
np.allclose(sol["m1.mech.z", "l1.amp.i"], sol.out[:, 0, 0])
```

```
True
```

```
axs = bode(sol.f, sol.out[:, 0, 0], db=False)
axs[0].set_ylabel("Magnitude [m / W]")
axs[0].set_title("Laser amplitude to mirror 1 motion");
```



Frequency Response 2

frequency_response2
freqresp2
FrequencyResponse2

Computes the frequency response of a signal injected at an optical port at a particular optical frequency. This differs from `FrequencyResponse` (page 221) in the way the inputs and outputs are prescribed. For `FrequencyResponse2` (page 223) you specify optical input nodes and a signal output node.

This action does not alter the model state. This action will ignore any currently definied signal generator elements in the model. Produces an output transfer matrix from each HOM at a particular frequency and optical node to some readout output. The shape of the output matrix is: [frequencies, outputs, inputs, HOMs] It should be noted that when exciting a lower signal sideband frequency it will actually return the operator for propagating the conjugate of the lower sideband. This is because FINESSE is internally solving for the conjugate of the lower sideband to linearise non-linear optical effects. To inject into mechanical and electrical nodes please see `FrequencyResponse` (page 221) and `FrequencyResponse4` (page 227). To readout optical nodes please see `FrequencyResponse3` (page 225) and `FrequencyResponse4` (page 227).

Syntax

```
freqresp2(
    f,
    inputs,
    outputs,
    carrier_solution=none,
```

(continues on next page)

(continued from previous page)

```
    name='frequency_response2'
)
```

Required

`f`: Frequencies to compute the transfer functions over

`inputs`: Optical node and frequency tuple to inject at. A symbolic refence to the model's fsig.f parameter should always be used when defining a frequency to look at.

`outputs`: Mechanical or electrical (signal)nodes to measure output to

Optional

`carrier_solution`: When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.

`name`: Solution name

See Also

`freqresp` (page 221), `freqresp3` (page 225), `freqresp4` (page 227)

`FrequencyResponse2` calculates transfers functions between optical nodes and signal nodes. We can calculate the same transfer function as above with `freqresp` (page 221), but now we use the outgoing optical field from the laser as an input node. Since we are now injecting into an optical field, we have to add the frequency at which we are injecting the signal. In most cases this will be the signal frequency (page 120).

```
f = np.geomspace(1e-1, 1e2, 100)
sol = model.run(
    FrequencyResponse2(
        f, inputs=[(model.m1.p1.i, model.fsig.f.ref)], outputs=[model.m1.mech.z]
    )
)
```

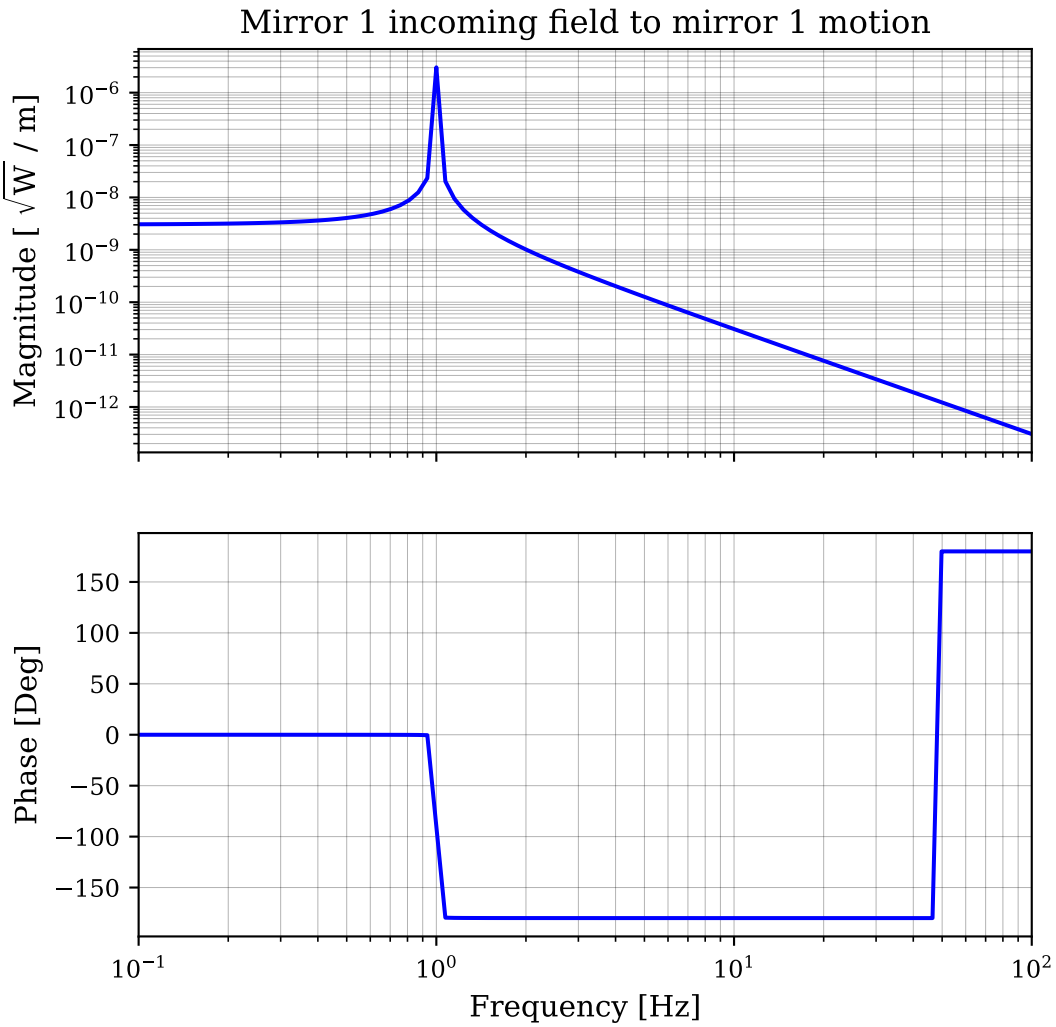
The output shape has gained another dimension as opposed to `freqresp` (page 221), since the optical field that we are injecting the signal into can in principle contain multiple higher order modes. `FrequencyResponse2` calculates the transfer function from each mode of the input to every output node separately. In our case we are modelling plane wave, therefore the size of this extra dimension is 1.

```
print(sol.out.shape)
print(sol.f.shape)
print(sol.inputs)
print(sol.outputs)
```

```
(100, 1, 1, 1)
(100,)
[(<OpticalNode m1.p1.i @ 0x7c49da472210>, <Symbolic='fsig.f' @ 0x7c49da482ab0>)]
[<SignalNode m1.mech.z @ 0x7c49da4b1a90>]
```

To plot the result, we provide an extra index. Note that the plot looks identical to the example for `freqresp` (page 221).

```
axs = bode(f, sol.out[:, 0, 0, 0], db=False)
axs[0].set_ylabel("Magnitude [ $\\sqrt{\\mathrm{W}}$ / m]")
axs[0].set_title("Mirror 1 incoming field to mirror 1 motion");
```



Frequency Response 3

frequency_response3
freqresp3
FrequencyResponse3

Computes the frequency response of a signal injected at an optical port at a particular optical frequency. This differs from `FrequencyResponse` (page 221) in the way the inputs and outputs are prescribed. For `FrequencyResponse3` (page 225) you specify optical input nodes and optical output nodes.

This action does not alter the model state. This action will ignore any currently definied signal generator elements in the model. Produces an output transfer matrix from each HOM at a particular frequency and optical node to some other optical node and frequency. The shape of the output matrix is: [frequencies, outputs, inputs, HOMs, HOMs] It should be noted that when exciting a lower signal sideband frequency it will actually return the operator for propagating the conjugate of the lower sideband. This is because FINESSE is internally solving for the conjugate of the lower sideband to linearise non-linear optical effects. To inject into mechanical and electrical nodes please see `FrequencyResponse` (page 221) and `FrequencyResponse4` (page 227). To readout mechanical and electrical nodes please see `FrequencyResponse` (page 221) and `FrequencyResponse2` (page 223).

Syntax

```
freqresp3(
    f,
    inputs,
    outputs,
```

(continues on next page)

(continued from previous page)

```
    carrier_solution=none,
    name='frequency_response3'
)
```

Required

`f`: Frequencies to compute the transfer functions over

`inputs`: Optical node and frequency tuple to inject at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.

`outputs`: Optical node and frequency tuple to measure output at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.

Optional

`carrier_solution`: When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.

`name`: Solution name

See Also

`freqresp` (page 221), `freqresp2` (page 223), `freqresp4` (page 227)

`FrequencyResponse3` calculates transfer functions between optical nodes. Since optical nodes have different complex amplitudes for every frequency and higher order mode, we specify the inputs and outputs differently. We have to provide the frequency at which we want to inject and measure the signal for every node. In most cases this will be the signal frequency (page 120). We measure the transfer function between the laser output field and the circulating field in the cavity. Since the cavity is on resonance, the response will be strongest around the resonance frequency of the suspended mirror.

```
f = np.geomspace(1e-1, 1e1, 100)
sol = model.run(
    FrequencyResponse3(
        f,
        inputs=[(model.l1.p1.o, model.fsig.f.ref)],
        outputs=[(model.m1.p2.o, model.fsig.f.ref)],
    )
)
```

The output shape has grown, since it contains separate transfer functions for every combination of higher order modes. In this example, we are modelling in plane wave, such that that the two extra dimension are both of size one, but in principle `FrequencyResponse3` allows us to calculate transfer functions between every mode combination all at once. To plot the result, we index the `.out` array with `[:, 0, 0, 0, 0]`, meaning: "all frequencies", "first output node", "first input node", "first output mode", "first input mode".

```
print(sol.out.shape)
print(sol.f.shape)
print(sol.inputs)
print(sol.outputs)
```

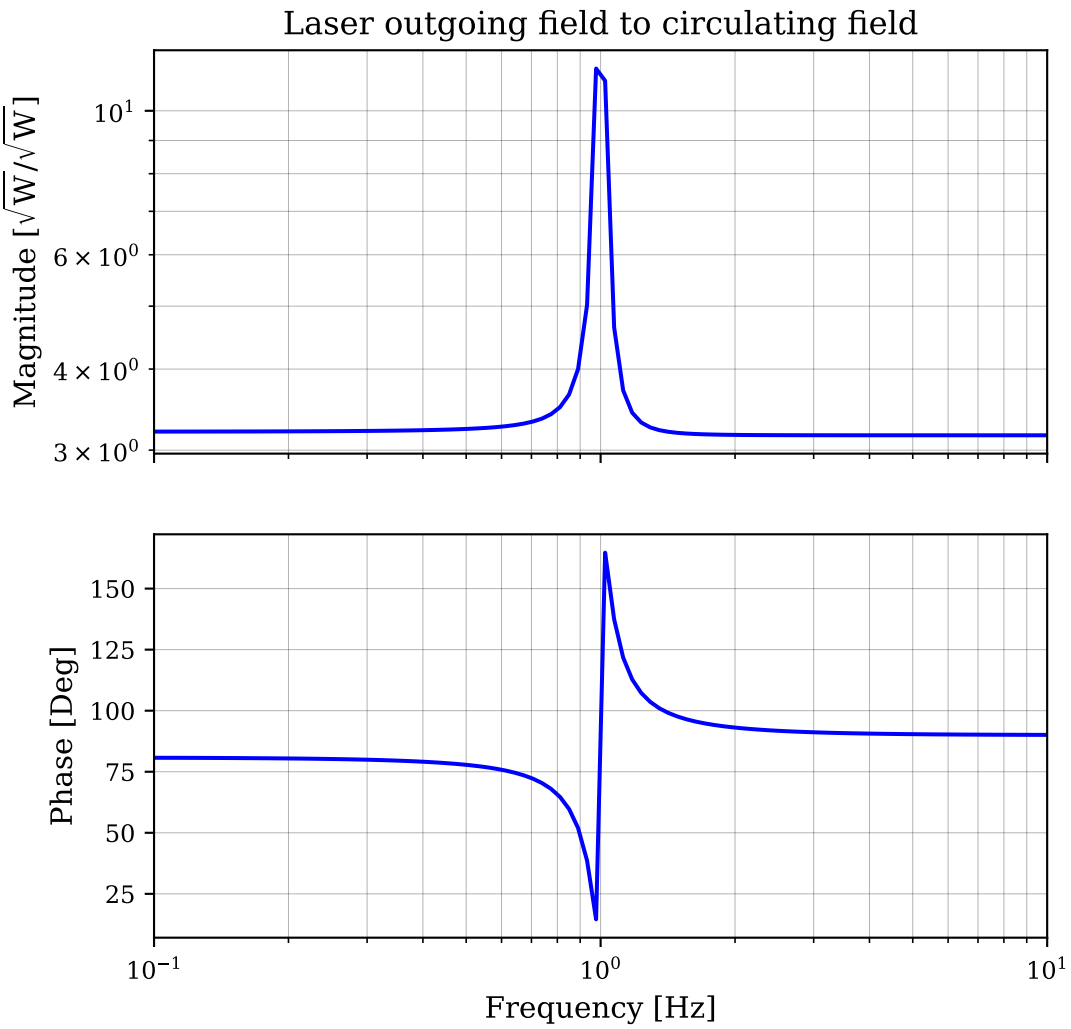
```
(100, 1, 1, 1, 1)
(100,)
[(<OpticalNode l1.p1.o @ 0x7c49da471310>, <Symbolic='fsig.f' @ 0x7c49d9588650>)]
[(<OpticalNode m1.p2.o @ 0x7c49da4a0b90>, <Symbolic='fsig.f' @ 0x7c49da482bd0>)]
```

```
f = np.geomspace(1e-1, 1e1, 100)
sol = model.run(
    FrequencyResponse3(
        f,
        inputs=[(model.l1.p1.o, model.fsig.f.ref)],
        outputs=[(model.m1.p2.o, model.fsig.f.ref)],
```

(continues on next page)

(continued from previous page)

```
    )
)
axs = bode(f, sol.out[:, 0, 0, 0], db=False)
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}} / \\sqrt{\\mathrm{W}}$]");
axs[0].set_title("Laser outgoing field to circulating field");
```



Frequency Response 4

frequency_response4
freqresp4
FrequencyResponse4

Computes the frequency response of a signal injected at an electrical or mechanical port. This differs from `FrequencyResponse` (page 221) in the way the inputs and outputs are prescribed. For `FrequencyResponse4` (page 227) you specify signal input nodes and optical output nodes.

This action does not alter the model state. This action will ignore any currently defined signal generator elements in the model. Produces an output transfer matrix from each signal node to each HOM at a particular frequency and optical node. The shape of the output matrix is: [frequencies, outputs, inputs, HOMs] It should be noted that when exciting a lower signal sideband frequency it will actually return the operator for propagating the conjugate of the lower sideband. This is because FINESSE is internally solving for the conjugate of the lower sideband to linearise non-linear optical effects. To inject into optical nodes please see `FrequencyResponse2` (page 223) and `FrequencyResponse3` (page 225). To readout mechanical and electrical nodes please see `FrequencyResponse` (page 221) and `FrequencyResponse2` (page 223).

Syntax

```
freqresp4(
    f,
    inputs,
    outputs,
    carrier_solution=none,
    name='frequency_response4'
)
```

Required

`f`: Frequencies to compute the transfer functions over

`inputs`: Mechanical or electrical node to inject signal at

`outputs`: Optical node and frequency tuple to measure output at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.

Optional

`carrier_solution`: When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.

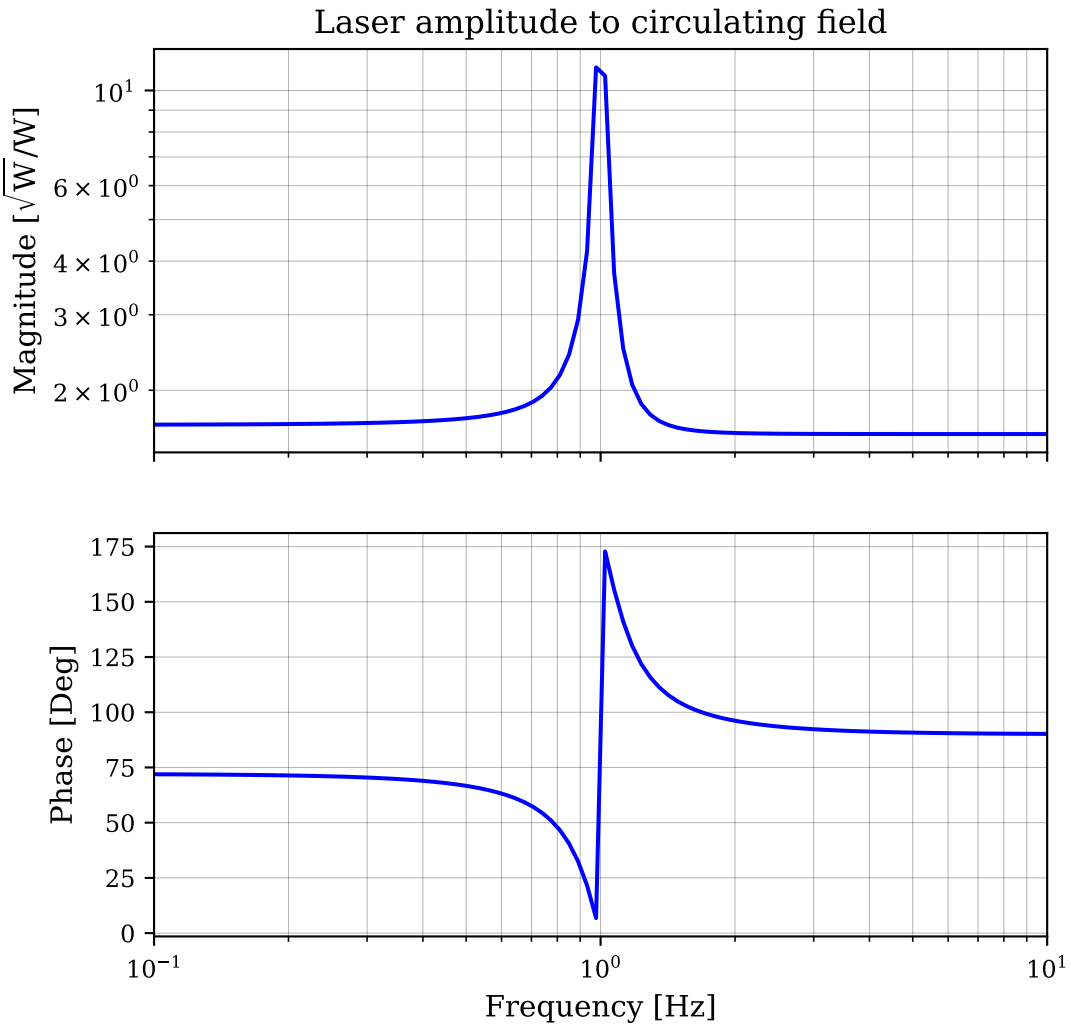
`name`: Solution name

See Also

`freqresp` (page 221), `freqresp2` (page 223), `freqresp3` (page 225)

`FrequencyResponse4` calculates transfer functions between signal nodes and optical nodes, making it basically the reverse of `freqresp2` (page 223). The same comments about specifying the signal frequency (page 120) (this time for output node) and the extra indexing for the higher order modes also apply. We recreate the same transfer function as for `freqresp3` (page 225), but this time we are injecting into the laser amplitude electrical input node instead of the laser outgoing field.

```
f = np.geomspace(1e-1, 1e1, 100)
sol = model.run(
    FrequencyResponse4(
        f,
        inputs=[model.l1.amp.i],
        outputs=[(model.m1.p2.o, model.fsig.f.ref)],
    )
)
axs = bode(f, sol.out[:, 0, 0, 0], db=False)
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}} / \\mathrm{W}$]");
axs[0].set_title("Laser amplitude to circulating field");
```



Tuning

Tuning actions are used to tune a model and prepare it for locking with `RunLocks` (page 231). See also Locking actions (page 72) for a demonstration of how these actions are used.

`opt_rf_readout_phase`
`OptimiseRFReadoutPhaseDC`

This optimises the demodulation phase of `ReadoutRF` (page 187) elements relative to some `DegreeOfFreedom` (page 190) or driven `Parameter` in the model. The phases are optimised by calculating the DC response of the readouts. This `Action` changes the state of the model by varying the readout demodulation phases. If no arguments are given it will try to automatically optimise any lock element in the model that is using an RF readout with respect to the lock feedback parameter.

Syntax

```
opt_rf_readout_phase(
    *args,
    d_dof=1e-10,
    name='optimise_demod_phases_dc',
    verbose=false
)
```

Required

`args`: Pairs of `DegreeOfFreedom` (page 190) or `Parameter` and `ReadoutRF` elements, or pairs of their names. If none are provided *OptimiseRFReadoutPhaseDC* will automatically search for `Lock` (page 190) elements which have `ReadoutRF` (page 187) error signal and op-

timise them. Every readout can only be optimised to a single `DegreeOfFreedom` (page 190) or `Parameter`. If a readout is paired with multiple drives, its phase is optimised to the first and the others are ignored.

`verbose`: If True this will print the initial and optimised readout phases. Defaults to False

Optional

`d_dof`: A small offset applied to the DOFs to compute the gradients of the error signals

sensing_matrix_dc
SensingMatrixDC

Computes the sensing matrix elements for various degrees of freedom and readouts that should be present in the model. The solution object for this action then contains all the information on the sensing matrix. This can be plotted in polar coordinates, displayed in a table, or directly accessed.

The sensing gain is computed by calculating the gradient of each readout signal, which means it is a DC measurement. This will not include any suspension or radiation pressure effects. This action does not modify the states model.

Syntax

```
sensing_matrix_dc(dofs, readouts, d_dof=1e-09, name='sensing_matrix_dc')
```

Required

`dofs`: String names of degrees of freedom

`readouts`: String names of readouts

Optional

`d_dof`: Small step used to compute derivative

set_lock_gains
SetLockGains

An action that computes the optimal lock gains using the sensing matrix found with `SensingMatrixDC` (page 230). This action computes the error signal gradient for each lock with respect to its drive and sets the gain as *-gain_scale/sensing*.

Syntax

```
set_lock_gains(
    *locks,
    d_dof_gain=1e-10,
    gain_scale=1,
    name='set gains',
    verbose=false
)
```

Required

`name`: Name of the action.

`verbose`: If True this will print the name of the enabled locks and their gains.

Optional

`\*locks`: A list of locks for which to set the gain. If none provided, all enabled locks in model are used. Disabled locks that are explicitly listed will have their gains set.

`d_dof_gain`: Step size to use when calculating the gain for each error signal/DOF pair.

`gain_scale`: Extra gain scaling factor applied to the gain calculation: *-gain_scale/sensing* In multiple lock models where the locks are cross coupled using a *gain_scale* < 1 can improve the stability of the locking algorithm to stop excessively large steps.

See Also

`opt_rf_readout_phase` (page 229), `sensing_matrix_dc` (page 230)

get_error_signals
GetErrorSignals

An action that quickly calculates the current error signals for all or a subset of locks in a model.

Syntax

```
get_error_signals(*locks, name='get error signals')
```

Required

`name`: Name of the action.

Optional

`\*locks`: A list of lock names to compute the error signals for. If not provided, all locks in model are used.

run_locks
RunLocks

An action that iteratively moves the system to lock. Currently, lock error signals must be readouts, not detectors, for use in this action.

Syntax

```
run_locks(
    *locks,
    method='proportional',
    scale_factor=1,
    sensing_matrix=none,
    max_iterations=10000,
    display_progress=false,
    d_dof_phase=1e-09,
    d_dof_gain=1e-09,
    exception_on_fail=true,
    no_warning=false,
    pre_step=none,
    show_progress_bar=none,
    name='run locks'
)
```

Required

`method`: Which method to use in the locking iterations.

`scale_factor`: Factor by which to multiply all DOF changes. Should be set below 1 if it is desired to minimize overshooting.

`sensing_matrix`: Sensing matrix of gains used in locking, of the type that would be returned by state.apply(SensingMatrixDC(lock_dof_names, readout_names) If None, the sensing matrix is recalculated. Recommended to be None except when locking multiple times in a row, e.g. with DragLocks.

`max_iterations`: The maximum number of locking steps in each execution of RunLocks.

`display_progress`: When true, displays the status of the error signals during locking iterations.

`d_dof_phase`: Step size to use when optimizing the demodulation phase for each error signal/DOF pair.

`d_dof_gain`: Step size to use when calculating the gain for every pair of error signals and DOFs.

`exception_on_fail`: When true, raise exception if maximum iterations are surpassed.

`no_warning`: When true, don't even raise a warning if maximum iterations are reached. Recommended to be false unless repeatedly testing locking.

`pre_step`: Action to apply on each step of the lock

`show_progress_bar`: Will enable the progress bar when true.

`name`: Name of the action.

Optional

`\*locks`: A list of locks to use in each RunLocks step. If not provided, all locks in model are used.

pseudo_lock_cavity
PseudoLockCavity

An action that locks a cavity defined by a *Cavity* element to a specific mode without using any radio-frequency sensing scheme. This will only work on simple cavities that are not coupled in any way. You can specify whether to try and lock to a particular HG mode with the *mode=[n,m]* keyword argument, or just pick the lowest loss mode, *lowest_loss=True*.

Syntax

```
pseudo_lock_cavity(
    cavity,
    mode=none,
    lowest_loss=false,
    feedback=none,
    name='pseudo_lock_cavity'
)
```

Required

`cavity`: Cavity element describing some Fabry-Perot like optical cavity

Optional

`mode`: HG mode to try and lock to, default is [0,0]

`lowest_loss`: Select the eigenmode which has the lowest loss, most likely the fundamental mode of the cavity. Using lowest loss will override the *mode* selection.

`feedback`: If *None* the required cavity tuning to lock to the calculated mode will be determined from the cavity objects source node element, and the relevant *phi* parameter will be used. Alternatively you can specify which tuning parameter is used instead. Which should be a *phi* of some mirror in the cavity or a *DegreeOfFreedom* which controls the cavity length.

`name`: Name of the solution generated by this action

pseudo_lock_drfpmi
PseudoLockDRFPMI

Pseudo-locking is attempting to find an operating point for a LIGO like model without needing to use RF sidebands and readouts. Although it is not physically accurate it does provide a useful tool for analysing detectors from a more theoretical basis. This generates a `PseudoLockDRFPMISolution` solution containing various operators and results.

This action is hardcoded to work with a LIGO like model. Mirrors should be named with ITMX, ETMX, PRM, etc. This currently action only really works for finding the lock points for the PRC, SRC, XARM, and YARM – which all have defined cavity roundtrips which allow eigendecomposition of roundtrip operators. The eigenvectors describe the HOM mix for each resonant mode in a cavity and the eigenvalues the roundtrip phase and loss of the mode. This code looks for eigenvectors with the largest HG00 content and then uses the eigenvalues to compute what cavity tunings need to make this mode resonant. The corner is the most complicated here as the PRC and SRC are coupled via the beamsplitter. The eigendecomposition is performed on the 2x2 operator matrix for the PRC, SRC, and the coupling matrices between them. When the coupling is small the results are the same as performing the decomposition on each SRC and PRC separately. Note that the only degree of freedom that this does not handle currently is MICH. MICH is awkward because it is not a cavity. It is essentially the beamsplitter position that makes the anti-symmetric port dark.

Syntax

```
pseudo_lock_drfpmi(frequency=0, apply_tunings=true, name='operator_lock')
```

Optional

`frequency`: Frequency to use for calculating the operators

`apply_tunings`: When True the action will modify the model tunings

`name`: Name of the solution generated by this action

Physics

eigenmodes
Eigenmodes

For a given Cavity defined in a model, this action will compute the roundtrip operator and calculate the eigenvalues and -vectors of the cavity. This will not give correct solutions for coupled cavities as these need to include additional effects.

This can be used to determine what modes combination of modes are resonating in a cavity and the required tuning to make that mode resonate.

Syntax

```
eigenmodes(cavity, frequency, name='eigenmodes', method='schur_qr')
```

Required

`cavity`: cavity name or `Cavity` (page 195) instance

`frequency`: Optical carrier or signal frequency to use for calculating the operators

Optional

`method`: Method used to compute the eigenvalues and eigenvectors, options are - 'eig': using numpy's eig function - 'schur_qr': using scipy's 'schur' and 'qr' decomposition for the eigenvectors - 'schur_svd': using scipy's 'schur' and 'svd' decomposition for the eigenvectors

`name`: Name of the solution generated by this action

operator
Operator

This action can be used to extract operators out from a simulation for external use. The operators are defined by a path in the network between two nodes (via some other if more direction is required).

The *model.path* method can be used to test which nodes are traversed before using this to extract operators if needed.

Syntax

```
operator(start_node, end_node, via=none, frequency=0, name='operator')
```

Required

`start_node`: Start node name

`end_node`: End node name

Optional

`via`: Via node (or sequence of via nodes) to use to specify a path with multiple options

`frequency`: Optical carrier or signal frequency to use for calculating the operators

`name`: Name of the solution generated by this action

abcd
ABCD

Computation of an ABCD matrix over a path.

See `compute_abcd()` for details.

Syntax

```
abcd(name=abcd, **kwargs)
```

beam_trace
BeamTrace

Full beam tracing on a complete model.

See `Model.beam_trace()` for details.

Syntax

```
beam_trace(name='beam_trace', **kwargs)
```

propagate_beam
PropagateBeam

Propagation of a beam, in a single plane, through a given path.

See `Model.propagate_beam()` for details.

Syntax

```
propagate_beam(name='propagation', **kwargs)
```

propagate_beam_astig
PropagateAstigmaticBeam

Propagation of a beam, in both planes, through a given path.

See `Model.propagate_beam_astig()` for details.

Syntax

```
propagate_beam_astig(name='astig_propagation', **kwargs)
```

antisqueezing
AntiSqueezing

Computes the amount of anti-squeezing at an output from a squeezing element.

Syntax

```
antisqueezing(f, squeezer, readout, signal=none, name='antisqueezing')
```

Required

`f`: Signal frequencies to compute the anti-squeezing over

`squeezer`: Name of squeezing component

`readout`: Name of readout port to compute squeezing at

`signal`: Name of signal drive to calculate a the signal transfer function of. This is returned in *sol.signal* and can be used to scale the noise into equivalent units of some signal.

Maps

update_maps
UpdateMaps

Update any maps that might be changing in the simulation.

Syntax

```
update_maps(*args, name='update_maps', **kwargs)
```

Utilities

change
Change

Changes a model Parameter to some value during an analysis.

Syntax

```
change(change_dict=none, relative=false, name='change', **kwargs)
```

Required

`\**kwargs`: Alternative method to specify parameter:value pairs to change

Optional

`change_dict`: Dictionary of parameter:value pairs to change.

`relative`: Whether to increment from the parameters current value or not

`name`: Name of action

dc_fields
DCFields

An action that saves the DC (carrier) fields at all nodes, frequencies, and higher order modes for the current state of the simulation.

Syntax

```
dc_fields(name='dcfields')
```

Optional

`name`: Name of the solution generated by this action

debug
Debug

An action that will start an IPython debug shell.

To access the current model use *state.model*.

Syntax

```
debug(name='Debug')
```

maximize
Maximize

An action that maximizes some detector output by applying some feedback to multiple targets in a model. Extra keyword arguments are passed on to the Scipy method:

minimize This action offers a simplified interface that allows an optimization to be done during a simulation. By default the the Nelder-Mead optimization method is used but can be overridden. The user should read the Scipy documentation to determine which options should be used which are method dependant.

Syntax

```
maximize(detector, parameter, *args, name='maximize', **kwargs)
```

Required

`detector`: The name of the detector output to maximize / minimize.

`parameter`: The parameter or name of the parameter to optimize, or a tuple of parameters when using multiple targets to optimize over.

`kwargs`: Optional parameters passed to the Scipy optimisation routine as the *options* input. See Scipy method documentation to determine what is available.

Optional

`bounds`: A pair of (lower, upper) bounds on the parameter value. Requires a method that uses bounds.

`offset`: An offset applied to the detector output when optimizing, defaults to 0.

`kind`: Either 'max' for maximization or 'min' for minimization, defaults to 'max'.

`max_iterations`: Maximum number of solver iterations, defaults to 10000.

`method`: Optimisation method to use, see Scipy documentation for options.

`name`: The name of this action, defaults to 'maximize'.

`update_maps`: If you are changing some parameter or variable that a *Map* depends on then setting this flag to *True* will recompute the *Map* data for each iteration of the optimiser.

`pre_step`: Action to run on each step of the optimisation.

See Also

`minimize` (page 236)

minimize
Minimize

An action that minimizes some detector output by applying some feedback to multiple targets in a model. Extra keyword arguments are passed on to the Scipy method:

minimize This action offers a simplified interface that allows an optimization to be done during a simulation. By default the the Nelder-Mead optimization method is used but can be overridden. The user should read the Scipy documentation to determine which options should be used which are method dependant.

Syntax

```
minimize(detector, parameter, *args, name='minimize', **kwargs)
```

Required

`detector`: The name of the detector output to maximize / minimize.

`parameter`: The parameter or name of the parameter to optimize, or a tuple of parameters when using multiple targets to optimize over.

`kwargs`: Optional parameters passed to the Scipy optimisation routine as the *options* input. See Scipy method documentation to determine what is available.

Optional

`bounds`: A pair of (lower, upper) bounds on the parameter value. Requires a method that uses bounds.

`offset`: An offset applied to the detector output when optimizing, defaults to 0.

`kind`: Either 'max' for maximization or 'min' for minimization, defaults to 'max'.

`max_iterations`: Maximum number of solver iterations, defaults to 10000.

`method`: Optimisation method to use, see Scipy documentation for options.

`name`: The name of this action, defaults to 'maximize'.

`update_maps`: If you are changing some parameter or variable that a *Map* depends on then setting this flag to *True* will recompute the *Map* data for each iteration of the optimiser.

`pre_step`: Action to run on each step of the optimisation.

See Also

`maximize` (page 235)

noise_projection

NoiseProjection

Syntax

```
noise_projection(f, *output_nodes, scaling=none, name='loop')
```

plot
Plot

Syntax

```
plot(name=abcd)
```

print
Printer

Syntax

```
print(*args, name='printer', eval=true)
```

print_model
PrintModel

An action that prints the model object being currently used to run actions.

Syntax

```
print_model(name='print_model')
```

print_model_attr
PrintModelAttr

Prints an attribute of the model being currently used.

Syntax

```
print_model_attr(*args, eval=true, prefix='')
```

Required

`args`: Strings input for the attribute to print. If a function is given, that funcion will be applied to the attribute before printing.

Optional

`eval`: When *True* symbolic expressions will be evaluated before printing. Defaults to *True*.

`prefix`: Optional string to print before the attributes

store_model_attr
StoreModelAttr

Action that stores value of model attributes during a simulation. The resulting `finesse.analysis.actions.random.SaveModelAttrSolution` will have a `.values` dictionary with the stored values. Most useful to execute between other actions in a `series` (page 207).

Syntax

```
store_model_attr(*args, name='store')
```

save_matrix
SaveMatrix

An action that will save the current state of the matrix being used by the simulation.

Not something that should be used lightly in loops or multiple times in a large simulation. Using this in something like a full LIGO model with many HOMs and sidebands will quickly fill up memory.

Syntax

```
save_matrix(name='savematrix')
```

6.2.7 Symbolics

Finesse 3 introduces a new symbolic math feature that can be used to easily define mathematical expressions, transformations, and build mode complex models. For those coming from Finesse 2, this engine has replaced commands such as *func*, *var*, *put*, and *set* with a more powerful framework.

The symbolic engine provided in Finesse is custom written and, whilst similar to packages such as Mathematica and Sympy, is not as feature complete or powerful. The symbolics have both a KatScript and Python interface to provide maximum flexibility. Here we will detail some of the main use cases for symbolic expressions in Finesse.

Referencing parameters

The most common use case for symbols is to quickly allow you to link different elements' parameters together. Take this simple example of two lasers, using KatScript, we can quickly make *l2* always have the same power as *l1* by simply writing:

```
import finesse
model = finesse.script.parse("""
l l1 P=1
l l2 P=l1.P
""")

model.l1.P, model.l2.P
```

```
(<l1.P=1.0 @ 0x7d1fe7e4b100>, <l2.P=l1.P @ 0x7d1fe7e4b6a0>)
```

We can see that *l1.P* has a value of 1 whereas *l2.P* has a value of *l1.P*. To explore this more, we can print the value of the parameter:

```
model.l2.P.value, type(model.l2.P.value)
```

```
(<Symbolic='l1.P' @ 0x7d1fe7c6cad0>, finesse.parameter.ParameterRef)
```

We can see this is a *Symbolic* object which is a *ParameterRef* type, which means it is a symbol that references the current value of *l1.P*. If we want to get the current value of a symbol we can simply evaluate it:

```
model.l2.P.value.eval()
```

```
1.0
```

and we see we get a value of *1*. If *l1.P* changes and we evaluate it again, we see we get the new value.

```
model.l1.P = 10
model.l2.P.value.eval()
```

```
10.0
```

We can also achieve this using the Python API:

```
model = finesse.script.parse("""
l l1 P=1
l l2 P=2
""")

print(model.l1.P.ref)
model.l2.P = model.l1.P.ref
print(model.l2.P)
```

```
l1.P
l1.P W
```

Note that here we can take any *ModelParameter* and get a symbolic reference to it by using the *.ref* attribute. These references can be used to make more complicated expressions if you need to:

```
import numpy as np
expression = 10 * model.l2.P.ref + np.cos(model.l1.P.ref)

print(expression)
print(expression.eval())
```

```
((10*l2.P)+cos(l1.P))
10.54030230586814
```

Note above we can easily do standard mathematical operations on symbols in Python and also in KatScript:

```
model = finesse.script.parse("""
l l1 P=1
l l2 P=10*cos(l1.P)
""")
```

KatScript will always assume you are making a symbolic operation, there is no need to specify *.ref* in KatScript. The benefit of this behaviour is that when you run simulations the second laser power will automatically follow the expressions when the power of *l1* is changed.

Another common use case is if you wanted to vary the reflectivity or transmission of a mirror. For example, you can sweep the transmission of a mirror and always keep the reflectivity the correct value:

```
model = finesse.script.parse("""
l l1 P=1
m m1 R=1-m1.T T=0 L=0
link(l1, m1)

xaxis(m1.T, lin, 0, 1, 100)
""")

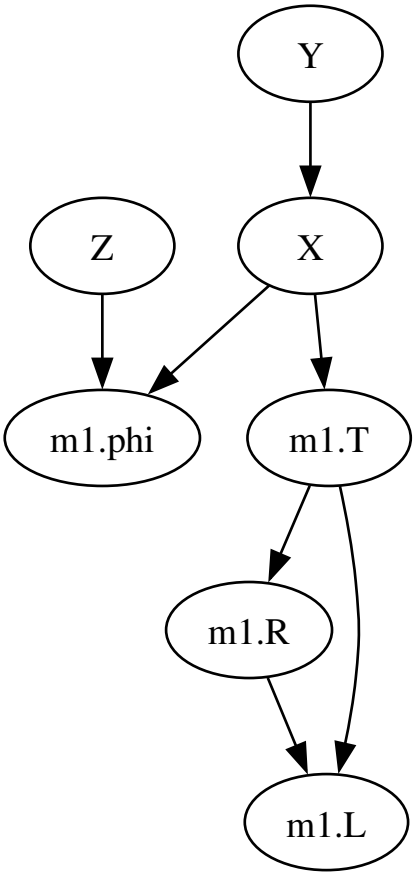
# or with Python
model.m1.R = 1 - model.m1.T.ref
```

Visualizing the symbolic graph

Symbolic references in your model can grow quite complex. Sometimes you may be wondering which parameters will be impacted when another parameter or variable is updated. The symbolic relationships can be retrieved with the `finesse.model.Model.symbolic_network` attribute.

It can be visualized with `finesse.model.Model.plot_symbolic_network()`:

```
model = finesse.Model()
model.parse(
    """\
var Y 0.5
var X Y
var Z 10
m m1 R=1-m1.T T=X phi=X+Z
    """
)
model.plot_symbolic_network()
```



This object returned by `finesse.model.Model.symbolic_network` is a `networkx.DiGraph` object, which you can explore with standard networkx functions. Note that the nodes in the graph are the full names of the parameters (e.g. `m1.R`), not the `finesse.parameter.Parameter` objects themselves. For example, you can find out whether two parameters are linked symbolically:

```
import networkx as nx

nx.has_path(model.symbolic_network, "Y", "m1.L")
```

```
True
```

Or which parameters depend on a certain parameter:

```
nx.descendants(model.symbolic_network, "X")
```

```
{'m1.L', 'm1.R', 'm1.T', 'm1.phi'}
```

See the networkx documentation for more options.

Variables vs ParameterRef

There are two types of variables in the FINESSE symbolics, *ParameterRef* and *finesse.symbol.Variable*. The former is described above and is a symbol whose value is attached to some model elements parameter. It is also possible to use a simple variable which is not.

Broadly they both work in the same way. The difference is mostly when evaluating some expression. A *ParameterRef* will evaluate to the current value of the parameter, whereas a *Variable* will not. The only way to give a variable a value to numerically evaluate an expression is to substitute one in.

Substitution

Substitution involves mapping a symbol with another expression or numeric value. There are two ways to do this in FINESSE. The first is a pure substitution:

```
a = finesse.symbols.Variable('a')
b = finesse.symbols.Variable('b')
y = a+b
y.substitute({a: b+1})
```

```
<Symbolic='((b+1)+b)' @ 0x7d1fe5b97460>
```

Alternatively you can also evaluate and substitute at the same time

```
a = finesse.symbols.Variable('a')
b = finesse.symbols.Variable('b')
y = a+b
y.eval(subs={a: b+1})
```

```
<Symbolic='((b+1)+b)' @ 0x7d1fe5b974d0>
```

Note that this is the same result when using *Variables*, as they are not evaluated to a specific value. Note that when using parameter references this behaviour will not be the same:

```
model = finesse.script.parse("""
var a 1
var b 2
""")
y = model.a.ref + model.b.ref
y.eval(subs={a: b+1})
```

```
3.0
```

a is replaced with *b+1* then evaluated as $b = 2$. Using *substitute* you can avoid the evaluation of the parameter references:

```
y.substitute({a: b+1})
```

```
<Symbolic='(a+b)' @ 0x7d1fe5b975b0>
```

You can also override *b* in the substitution if needed

```
y.eval(subs={a: b+1, b: 3})
```

```
3.0
```

even with arrays

```
y.eval(subs={a: b+1, b: np.array([1,2,3,4])})
```

```
3.0
```

Some substitutions/mappings are not unique and if applied recursively in different orders will result in different results. FINESSE again does not apply the mappings recursively to avoid this. Therefore a substitution like below has a definite answer but still has a *b* term in it, despite the mapping *b=>a* being applied.

```
(a+b).substitute({a: a+b, b: a})
```

```
<Symbolic='((a+b)+a)' @ 0x7d1fe5b97f50>
```

Making a function from an expression

It is possible to convert a symbolic expression into some evaluable function. This process is called "lambdifcation", or generating some anonymous *lambda* function that can be called. This is particular useful if you have some expression, say from the ABCD tracing, and you want to run it in some optimiser.

```
model = finesse.script.parse("""
var a 1
laser L1 P=10
""")
y = 2 * model.a.ref * model.L1.P.ref

print("y =", y, "=", y.eval())
```

```
y = ((2*a)*L1.P) = 20.0
```

We can convert this expression into a function we can just call.

```
f = y.lambdify(model.a.ref, model.L1.P.ref)
print(f(1, 10))
print(f(2, 10))
print(f(2, 20))
```

```
20
40
80
```

If you need something that evaluates faster, you can also look into using *numba* JIT to compile the generated function into something that evaluates faster.

Or if you want a function that is just using one parameter and the rest are the

```
f = y.lambdify(model.a.ref, model.L1.P.ref)
print(f(1, 10))
print(f(2, 10))
print(f(2, 20))
```

```
20
40
80
```

Preserving intent

One feature of the Finesse symbolics which differs from many other Symbolic packages is that there is no automated simplification. The key idea here is that we want to record exactly what the user originally intended. This for example, is useful for keeping track of factors of two or pi, etc, which when cancelled or simplified maybe lost.

```
model = finesse.script.parse("""
l l1 P=2*2/2-pi+pi
""")

model.l1.P
```

```
<l1.P=((2.0-π)+π) @ 0x7d1fe5b5a2f0>
```

two-arg vs N-arg operators

In our symbolic computation everything is broken down into operations and their arguments. These in turn then form a tree structure. Two varieties exist, binary-trees or operators accepting only two arguments, and a general tree structure, where operators can have N-arguments. The Finesse symbolic feature actually support both. The binary-tree structure is used by default, and generates an expression tree that fully preserves the original intent of the user.

The binary type of tree is not ideal for performing symbolic simplification procedures though. Therefore there is also a generic tree structure option as well. General FINESSE users should not really be able to tell which is being used at any particular time, nor should it make large differences in the behaviour of the code. It may be important for users performing more complex analysis though. More details on symbolic computation can be read up on in [30], which the FINESSE implementation is based upon.

In general the binary tree structure is apparent when you print longer expressions (You can also check the symbols *_is_narg_expression_tree* attribute):

```
a = finesse.symbols.Variable('a')
y = 2+a+2+a+2*a*a/2
y
```

```
<Symbolic='((((2+a)+2)+a)+(((2*a)*a)/2))' @ 0x7d1fe5bbda20>
```

You should note that the brackets collect together always two terms in each operation. you can convert between the two tree structures using *.to_nary_add_mul()* and *.to_binary_add_mul()*. These are named as such as the general tree structure only really applies to the add and multiplication operator.

```
a = finesse.symbols.Variable('a')
y = 2+a+2+a + 2*a*a/2
y.to_nary_add_mul()
```

```
<Symbolic='(2+2+2*a+(a)**(2))' @ 0x7d1fe5bbde80>
```

During the above conversion processes some obvious simplification happens. Negation and division operators also do not exist in the N-argument trees. Those can always be represented by multiply and power operators which makes simplification and comparing two expressions significantly easier.

```
a = finesse.symbols.Variable('a')
y = 1/a
print(y)
print(y.to_nary_add_mul())
```

```
(1/a)
(a)**(-1)
```

This however highlights why the N-argument option is not the default, as we have lost the intent of a division. This has subtle numeric implications as a floating point division is not numerically identical to a power operation in all cases.

Basic simplification

In cases where you do want some basic simplification of FINESSE symbolics there are only a few options, but they should cover most basic algebraic requirements. As needs evolve more may be added:

- *expand* - expand operators
- *collect* - collect together like terms

```
model = finesse.script.parse("""
var b 1
l l1 P=b+2-b+(b+b)
""")

print(model.l1.P.value)
print(model.l1.P.value.expand())
print(model.l1.P.value.collect())
```

```
(((b+2)-b)+(b+b))
(2+2*b)
(2+2*b)
```

The simplification should also work on functions:

```
(np.cos(a) - np.cos(a)).collect()
```

```
0
```

6.2.8 Python Interface

Models and Components

Using the python interface to construct your FINESSE model can be more verbose than using KatScript (page 125), but gives you access to syntax highlighting and autocompletion, especially when using a modern IDE like VSCode or PyCharm. It also offers the most feature rich API to interact with FINESSE models. All features are supported in Python API, whereas only a subset are support in KatScript.

Note

If you want to know how to map a KatScript command to a Python method or class you can inspect the `_register_constructs` method in **script/spec.py** which sets up the various links between KatScript and Python API.

Adding components to a model

All the KatScript components (page 142), have a Python class associated in either the `finesse.components` or the `finesse.detectors` module. Documentation can also be accessed using the python `help()` method

```
from finesse import Model
from finesse.components import Laser, Mirror, Space, Beamsplitter
from finesse.detectors import PowerDetector

# help(Laser) # would print the help for the Laser
```

Adding components is done using `finesse.model.Model.add()`, which returns the component added.

```
model = Model()
l1 = model.add(Laser("l1", P=10))
l1
```

```
<'l1' @ 0x7c81dc6d0830 (Laser)>
```

Adding a component automatically creates an attribute on the `finesse.model.Model` object with the name of the component.

```
model.l1
```

```
<'l1' @ 0x7c81dc6d0830 (Laser)>
```

Note

When writing a python script, your IDE might warn you that these attributes don"t exist and not provide auto-completion. When working in an interactive/Jupyter context, these attributes will be available for autocompletion after the `add` method has been executed.

Component names need to be unique:

```
model = Model()
model.add(Laser("l1", P=10))
model.add(Laser("l1", P=10))
```

```
FinesseException:         (use finesse.tb() to see the full traceback)
An element with the name l1 is already present in the model (Laser('l1', signals_only=False,␣
↪f=0.0, phase=0.0, P=10.0))
```

The `finesse.model.Model.add()` function also accepts an iterable of components:

```
model = Model()
l1, l2, l3 = model.add([Laser(f"l{i}", P=i) for i in range(3)])
model.components
```

```
(<'l0' @ 0x7c81dbd3f390 (Laser)>,
 <'l1' @ 0x7c81dbd3f4d0 (Laser)>,
 <'l2' @ 0x7c81dbd3f610 (Laser)>)
```

Connecting components

For any useful simulation, components have to be connected together. Many spaces must be made in more complicated models and often you do not need to give them a name. Therefore the python interface provides multiple ways of doing so depending on your use case.

Specifying ports on Space components

Directly adding a Space element is the most low-level option. We first create and add the components for a simple cavity:

```
m = Model()

l1, m1, m2 = m.add(
    [Laser("l1"),
     Mirror("m1", R=0.1, T=0.9),
     Mirror("m2", R=0.1, T=0.9)
    ]
)
```

And then we create `Space` (page 170) components, connecting their ports (page 139) to the ports (page 139) of existing components.

```
s1 = m.add(Space("s1", portA=l1.p1, portB=m1.p1))
s2 = m.add(Space("s2", portA=m1.p2, portB=m2.p1))
```

Using the connect method

The `finesse.model.Model.connect()` connects the provided components or ports together and returns the created space/wire component. It tries to connect in a "smart" way, so it can be useful to use the verbose argument to see what is happening.

```
m = Model()

l1 = Laser("l1")
m1 = Mirror("m1", R=0.1, T=0.9)
m2 = Mirror("m2", R=0.1, T=0.9)

m.add([l1, m1, m2])
```

(continues on next page)

(continued from previous page)

```
s1 = m.connect(l1, m1, L=1, name="s1", verbose=True)
s2 = m.connect(m1, m2, L=1, name="s2", verbose=True)
```

```
Selecting port <Port l1.p1 Type=NodeType.OPTICAL @ 0x7c81dbbe9850> for <'l1' @␣
↪0x7c81dbbfc550 (Laser)>
Selecting port <Port m1.p1 Type=NodeType.OPTICAL @ 0x7c81dbbea150> for <'m1' @␣
↪0x7c81dbbfc690 (Mirror)>
Selecting port <Port m1.p2 Type=NodeType.OPTICAL @ 0x7c81dbbd4e10> for <'m1' @␣
↪0x7c81dbbfc690 (Mirror)>
Selecting port <Port m2.p1 Type=NodeType.OPTICAL @ 0x7c81dbbd5180> for <'m2' @␣
↪0x7c81dbbfc910 (Mirror)>
```

Using the link method

The `finesse.model.Model.link()` method calls the `finesse.model.Model.connect()` method between each pair of components provided. Connector names will be autogenerated, but it is possible to specify the length/time delay of the connector by adding integers to the argument list. This is useful when you have a long list of components to connect quickly.

```
m = Model()

l1 = Laser("l1")
m1 = Mirror("m1", R=0.1, T=0.9)
m2 = Mirror("m2", R=0.1, T=0.9)
# Connect with distances l1 - 10m -> m1 - 20m -> m2
m.link(l1, 10, m1, 20, m2, verbose=True)
```

```
Connecting l1 to m1
Selecting port <Port l1.p1 Type=NodeType.OPTICAL @ 0x7c81dbd72060> for <'l1' @␣
↪0x7c81dbbfcf50 (Laser)>
Selecting port <Port m1.p1 Type=NodeType.OPTICAL @ 0x7c81dbd72330> for <'m1' @␣
↪0x7c81dbbfd090 (Mirror)>
Connecting m1 to m2
Selecting port <Port m1.p2 Type=NodeType.OPTICAL @ 0x7c81dbd723c0> for <'m1' @␣
↪0x7c81dbbfd090 (Mirror)>
Selecting port <Port m2.p1 Type=NodeType.OPTICAL @ 0x7c81dbd72450> for <'m2' @␣
↪0x7c81dbbfd1d0 (Mirror)>
```

You can also be more verbose with the ports linked `m.link(l1.p1, 10, m1.p2, m2.p1, 20, m2.p2, verbose=True)`. Note that when connecting through a component you need to specify which port to in and which to go out, as seen with `m1` above.

Using the chain method

This method is documented for completeness but maybe removed later as link and connect above are used far more frequently.

The `finesse.model.Model.chain()` method is similar to the `finesse.model.Model.link()` method. One difference is it accepts dictionaries that specify space components, allowing for named connectors.

```
m = Model()
l1, m1, m2 = m.chain(
    Laser("l1"), Mirror("m1"), {"name": "s1", "L": 1}, Mirror("m2")
)
print(m.s1)
```

```
Space('s1', user_gouy_y=None, user_gouy_x=None, nr=1.0, L=1.0)
```

Visualizing the model

After creating and adding your components, it can be helpful to visualize your model to ensure your components are connected like you would expect. For this example we will create a simple interferometer.

```
m = Model()

l1 = Laser("l1")
bs = Beamsplitter("bs", R=0.5, T=0.5)
m.link(l1, bs)

ETMx = Mirror("ETMx", R=1, T=0)
m.link(bs.p3, ETMx)

ETMy = Mirror("ETMy", R=1, T=0)
m.link(bs.p2, ETMy)

signal = PowerDetector("signal", m.bs.p4.o)
m.add(signal)
```

```
<'signal' @ 0x7c81dc5a12b0 (PowerDetector)>
```

Using component_tree

We can use `finesse.model.Model.component_tree()` to draw a tree of the model using ASCI characters.

```
print(m.component_tree())
```

```
o l1
└──o bs
   ├──o ETMy
   └──o ETMx
```

For larger models, it can be clearer to only display a subsection of the model, by using the `root` and `radius` arguments to select a subset of the graph.

```
# Only display components that are connect directly to the end mirror
print(m.component_tree("ETMx", radius=1))
```

```
o ETMx
└──o bs
```

We can exchange some clarity with verbosity by showing the ports components are connected to by using the `show_ports` argument.

```
print(m.component_tree(show_ports=True))
```

```
o l1
└──o bs (l1.p1 ↔ bs.p1)
   ├──o ETMy (bs.p2 ↔ ETMy.p1)
   └──o ETMx (bs.p3 ↔ ETMx.p1)
```

We can also include the detectors in the component tree, although they are strictly speaking not components since they are not part of the optical network.

```
print(m.component_tree(show_detectors=True))
```

```
o l1
└──o bs
```

(continues on next page)

(continued from previous page)

```
   ├──O ETMy
   ├──O ETMx
   └──O signal
```

Finally we can also use `component_tree` to visualize other networks besides the component network, using the `network_type` argument:

```
from finesse.utilities.network_filter import NetworkType

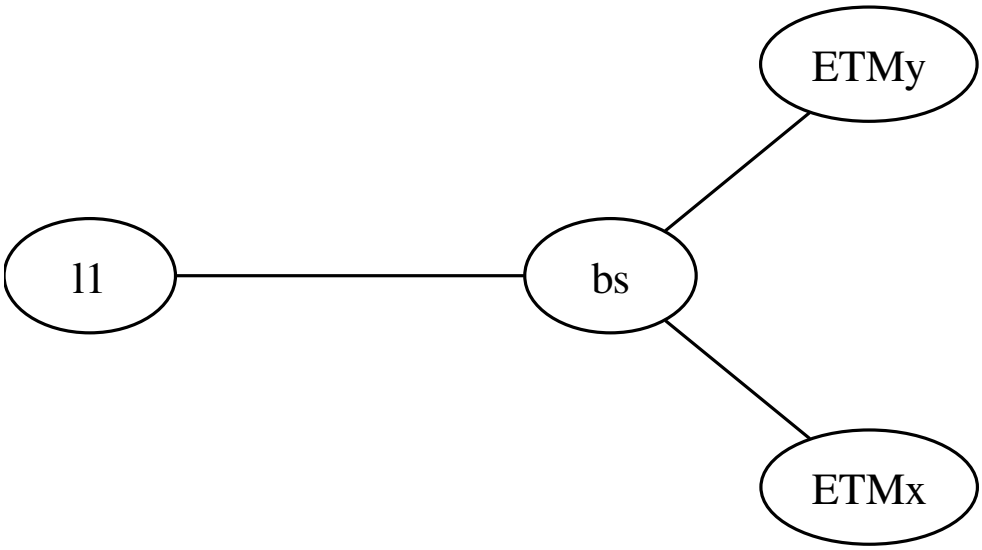
# note you can also use 'network_type="optical"'
print(m.component_tree(m.l1.p1.o, network_type=NetworkType.OPTICAL))
```

```
O l1.p1.o
└──O bs.p1.i
   ├──O bs.p2.o
   │  └──O ETMy.p1.i
   │     ├──O ETMy.p1.o
   │     │  └──O bs.p2.i
   │     │     ├──O bs.p1.o
   │     │     │  └──O l1.p1.i
   │     │     └──O bs.p4.o
   │     └──O ETMy.p2.o
   └──O bs.p3.o
      └──O ETMx.p1.i
         ├──O ETMx.p1.o
         │  └──O bs.p3.i
         └──O ETMx.p2.o
```

Using graphviz

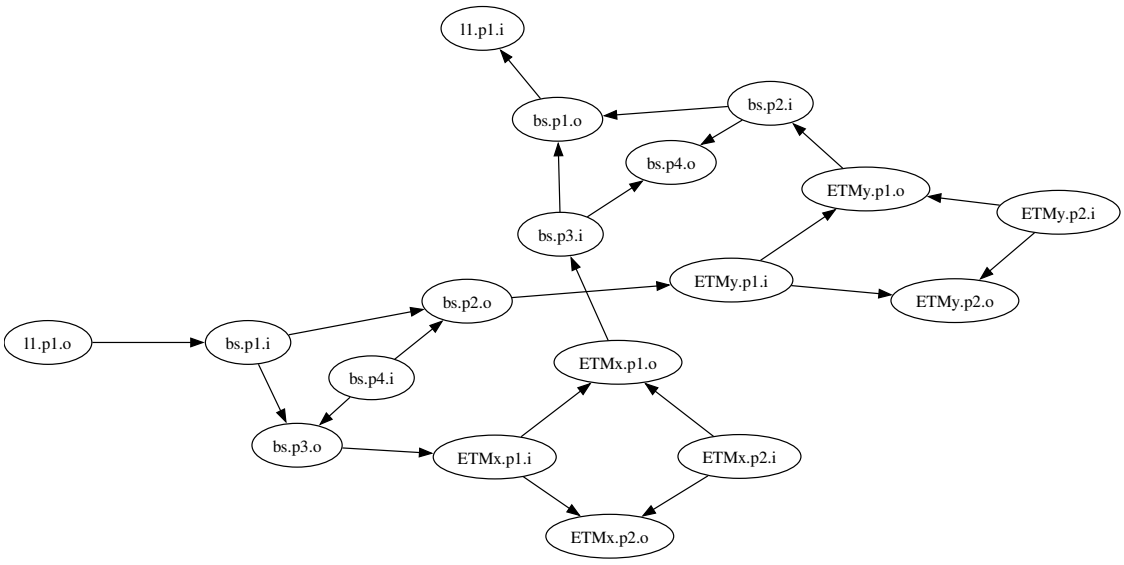
You can also directly visualize the The model graph (page 35) by using `finesse.model.Model.plot_graph()`.

```
m.plot_graph()
```



To see more information about how the optical ports are connected, you can specify the network type `optical`.

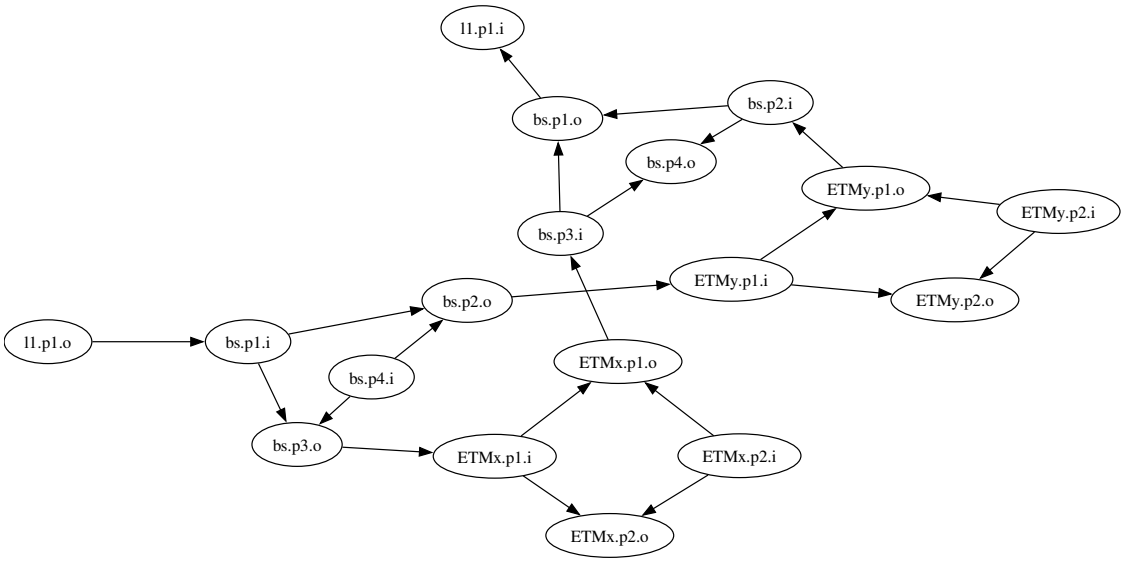
```
m.plot_graph(network_type="optical")
```



Note

This command may fail if you have not installed FINESSE via `conda`. See here to install the graphviz dependency separately.

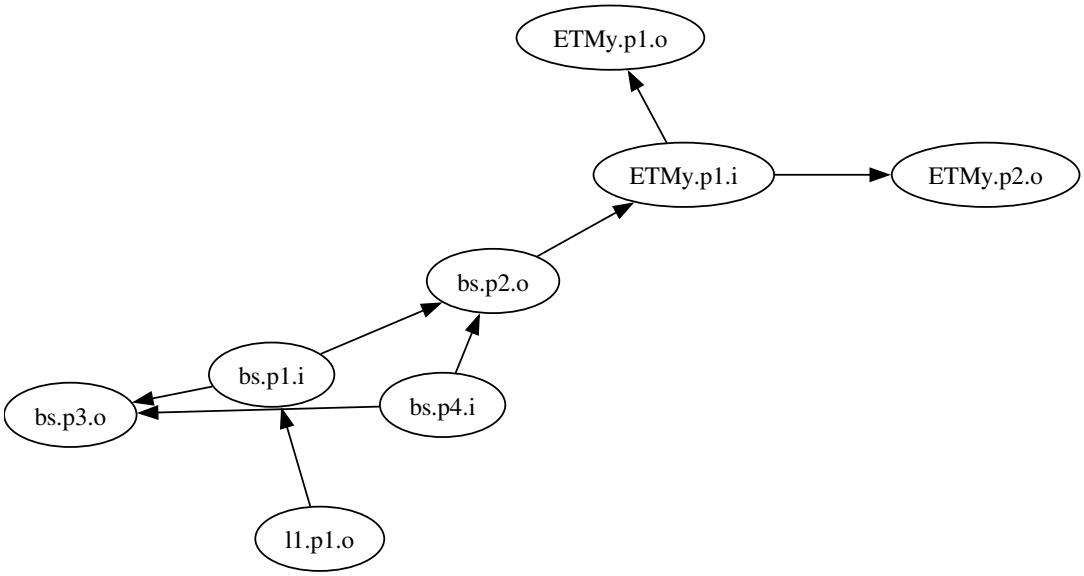
```
m.plot_graph(network_type="optical", graphviz=True)
```



Note that the `graphviz` argument controls whether graphviz will be used. If not available, it will automatically revert to `networkx` and emit a warning.

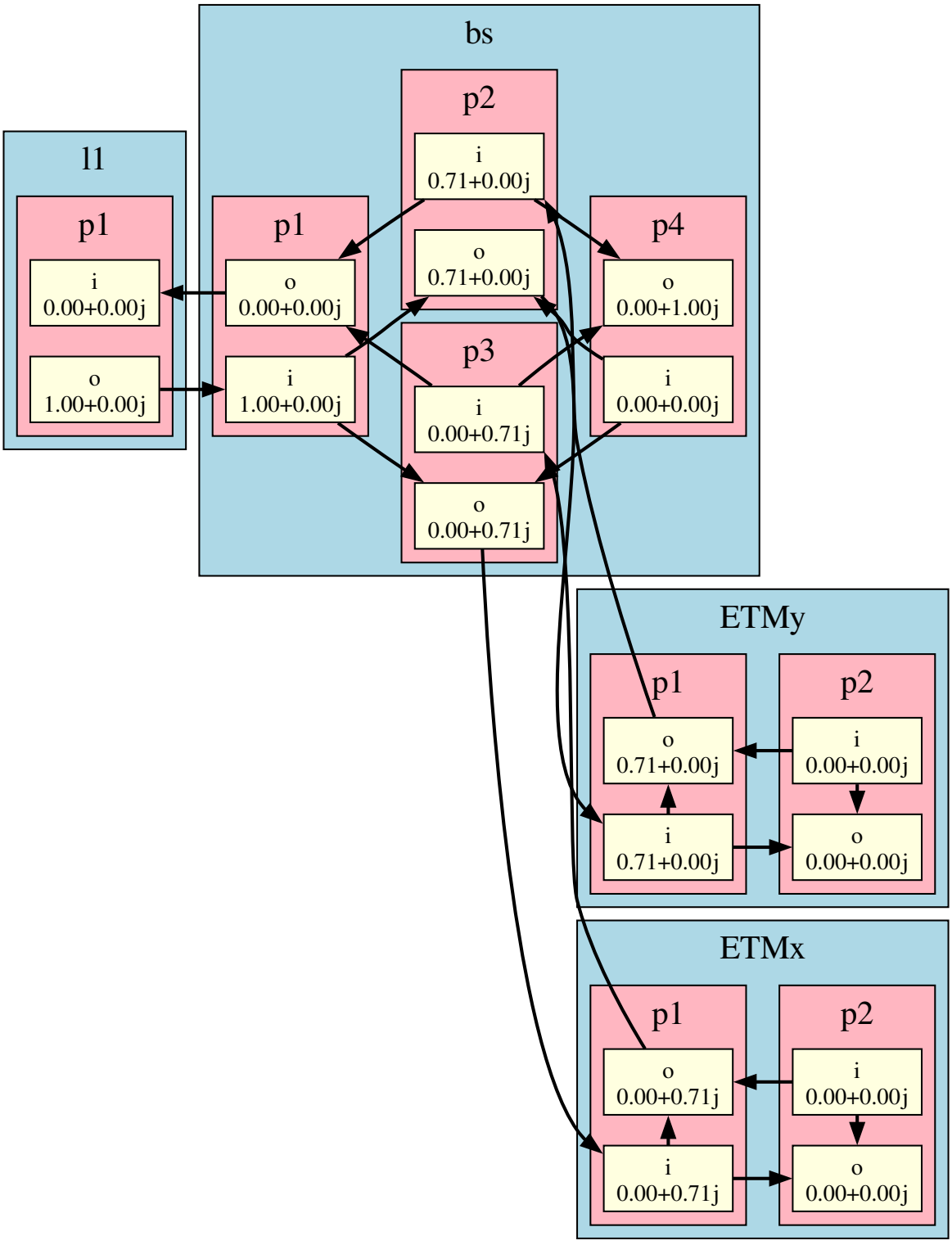
`plot_graph` also supports the `radius` and `root` arguments, which should be used in tandem.

```
m.plot_graph(network_type="optical", graphviz=True, root="bs.p2.o", radius=2)
```



An alternative visualization method is `finesse.model.Model.plot_dcfields_graph()`, which includes the output of the `DCFields` (page 235) action in the visualization:

```
m.plot_dcfields_graph()
```




```
PosixPath('/tmp/tmp4g45bjp_.svg')
```

Component freezing and unfreezing

Freezing and unfreezing components is a way to prevent the user from adding new attributes of a model element accidentally. This stops the user from making commong typos, such as setting `mirror.r` instead of `mirror.R`. The `ModelElement` base class, which all components and detectors inherit from, has a `_freeze` and `_unfreeze` method.

If the user wants to store additional attributes for a component, they can unfreeze it, set the attributes, and then freeze it again. This can be useful for storing auxiliary information about the component, such as experimental data, metadata, or other information that is not part of the model itself.

```
model = Model()
l1 = model.add(Laser("l1", P=10))
l1._unfreeze()
l1.my_attr = "test"
l1._freeze()
print(l1.my_attr)
```

```
test
```

> **Note**
>
> Elements are only frozen after they have been added to a model. Before that the elements are unfrozen and can be modified freely, which the user should be careful with.

Examining components

After parsing a component, you can examine the values of the component parameters by using the `finesse.element.ModelElement.info()` method:

```
import finesse
from finesse.components import Laser

m = finesse.Model()
m.add(Laser("l1", P=1))
print(m.l1.info())
```

```
Laser l1

Parameters:
┌──────────────┬──────────────┐
│ Description  │ Value        │
╞══════════════╪══════════════╡
│ Power        │ 1.0 W        │
├──────────────┼──────────────┤
│ Phase        │ 0.0 degrees  │
├──────────────┼──────────────┤
│ Frequency    │ 0.0 Hz       │
├──────────────┼──────────────┤
│ Signals only │ False        │
└──────────────┴──────────────┘
```

If you have defined a parameter value using a reference, this will be reflected in the info table:

```
import finesse
from finesse.components import Laser

m = finesse.Model()
m.add(Laser("l1", P=1))
```

(continues on next page)

(continued from previous page)

```
m.add(Laser("l2", P=m.l1.P.ref))
print(m.l2.info())
```

```
Laser l2

Parameters:
┌─────────────┬──────────────┐
│ Description │ Value        │
╞═════════════╪══════════════╡
│ Power       │ l1.P W       │
├─────────────┼──────────────┤
│ Phase       │ 0.0 degrees  │
├─────────────┼──────────────┤
│ Frequency   │ 0.0 Hz       │
├─────────────┼──────────────┤
│ Signals only │ False       │
└─────────────┴──────────────┘
```

If you want to see the current numerical value for all the parameters, pass `eval_refs=True`:

```
import finesse
from finesse.components import Laser

m = finesse.Model()
m.add(Laser("l1", P=1))
m.add(Laser("l2", P=m.l1.P.ref))
print(m.l2.info(eval_refs=True))
```

```
Laser l2

Parameters:
┌─────────────┬──────────────┐
│ Description │ Value        │
╞═════════════╪══════════════╡
│ Power       │ 1.0 W        │
├─────────────┼──────────────┤
│ Phase       │ 0.0 degrees  │
├─────────────┼──────────────┤
│ Frequency   │ 0.0 Hz       │
├─────────────┼──────────────┤
│ Signals only │ False       │
└─────────────┴──────────────┘
```

Saving and loading models

> **Note**
>
> The Python API is still in development and the serialisation format may change in the future. It is recommended to only use Pickle for temporary storage and not for long term storage.

KatScript offers one way to easily store a particular optical layout which describes the components, detectors, and analyses to be performed. Whilst this is convenient and can be stored as a simple ASCII file for long term storage, it is limited in how complex a model can be described. The Python API offers full access to make models which use advanced features in Finesse as well as custom user code. In such cases the `finesse.model.Model` objects much be serialised and deserialised and saved somewhere.

In Python, `Pickle` is the standard library for serialising objects. Pickling tries to save the object as a binary representation of the object, which can then be loaded back into memory. The benefit of this is that arbitrarily complex objects can be saved and loaded. However, there are several downsides to Pickle:

- Pickle is not human readable, so it is not suitable for long term storage
- Pickle is not secure, loading a Pickle file can execute arbitrary code so care should be taken on the origin of the Pickle file
- Pickle is not guaranteed to be compatible between different versions of Python
- Pickle is not guaranteed to be compatible between different versions of FINESSE
- Pickle is not guaranteed to be compatible between different versions of the Python API
- Pickle is not guaranteed to be compatible between different operating systems, such as Windows and Linux, or different architectures, such as x86 and ARM.

With this being said, it is still very useful to be able to save and load models for temporary short term storage, such as when running an optimisation and saving the state of the model at each iteration. Or serialising models to distribute to other processors in a cluster or multi core system (page 254).

The interface to load and save models is very simple. The `finesse.model.Model.save()` and `finesse.model.load()` methods can be used like so:

```
import finesse
from finesse import Model
from finesse.components import Laser, Mirror, Space, Beamsplitter
from finesse.detectors import PowerDetector

model = Model()
l1 = model.add(Laser("l1", P=10))
pd1 = model.add(PowerDetector("pd1", l1.p1.o))

model.save("model.pkl")
loaded_model = finesse.model.load('model.pkl')

print(model, loaded_model)
```

```
<finesse.model.Model object at 0x78d0132afe00> <finesse.model.Model object at 0x78d01332d310>
```

Warning

Pickling is not always a fool proof way to save and load models. Sometimes the saving works and the loading does not. This is usually due to internal issues in FINESSE that can be fixed. Please report any issues you have to the Gitlab project issue tracker.

Parallel processing

Parallel processing is a broad topic and can be implemented in many different ways depending on the problem you are trying to solve. This section will cover a few different ways to run simulations in parallel using the Python API. It will not be complete and will not cover all possible ways to run simulations in parallel.

Local multiprocessing

Note

FINESSE is not compatible with the standard multiprocessing library in Python due to the way that FINESSE models are not picklable. The dill library must be used which is implemented in pathos.

The Python API is compatible with the pathos library, which provides a simple way to run simulations in parallel. The following examples demonstrates how to run simulations in parallel using the *ProcessPool* class.

Random parameter values with fixed model

Here is a simple case where we have a single model that we want to randomly vary some parameter and compute the result in parallel. In this case we will vary the power of a laser and compute the power at the output of the laser. The *run* method is called in a separate process and the model is passed to it. A fixed model in this case means one that you are not adding any new components to.

```
import os
import finesse
import finesse.analysis.actions as fa
import finesse.components as fc
import finesse.detectors as fd
import matplotlib.pyplot as plt
import numpy as np

from pathos.multiprocessing import ProcessPool

def run(job_number):
    # This is important to ensure that the random number generator is seeded
    # with a different seed for each process. Otherwise, all processes will use
    # the same random numbers.
    np.random.seed()
    model.LASER.P = np.random.normal(1, 0.1)
    return model.run(fa.Noxaxis())


model = finesse.Model()
LASER = model.add(fc.Laser("LASER", P=1))
model.add(fd.PowerDetector("P", LASER.p1.o))

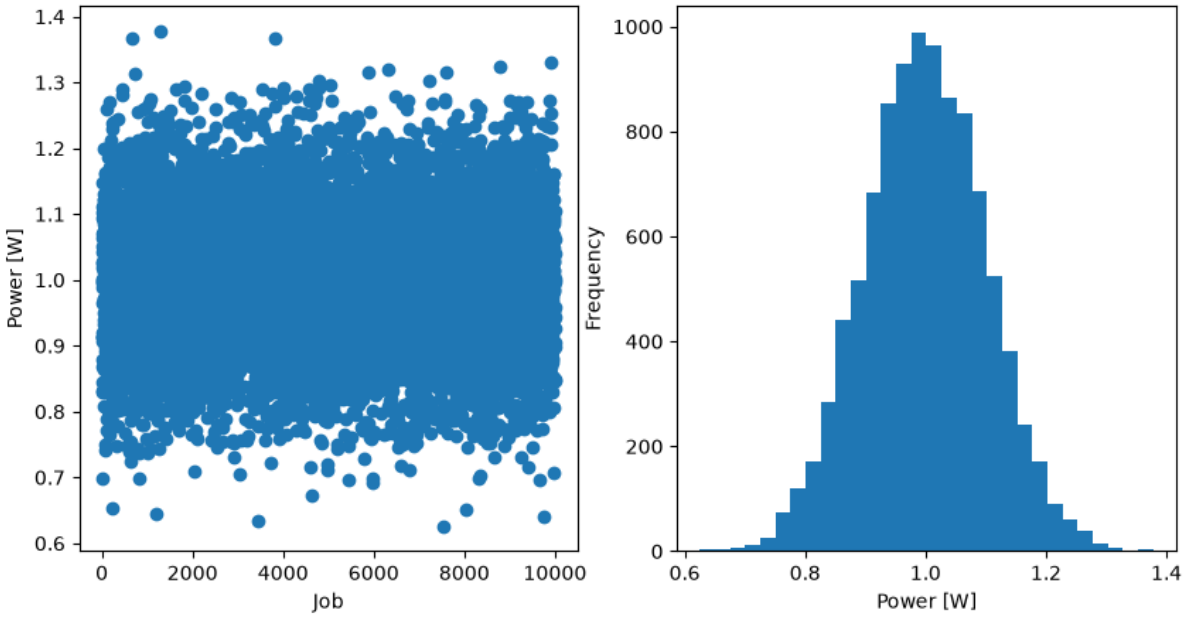
action = fa.Noxaxis()
N = 10000

pool = ProcessPool(nodes=12)
solutions = pool.map(run, range(N))

fig, axs = plt.subplots(1, 2, figsize=(10, 5))

axs[0].scatter(np.arange(N), [sol["P"] for sol in solutions])
axs[0].set_xlabel("Job")
axs[0].set_ylabel("Power [W]")

axs[1].hist([sol["P"] for sol in solutions], bins=30)
axs[1].set_xlabel("Power [W]")
axs[1].set_ylabel("Frequency");
```



There are a few important things to note in this example:

- You should not make any changes to the model in the *run* function. This is because the model is not being copied, but being shared between processes.
- The *np.random.seed()* is called in the *run* function to ensure that each process has a different seed for the random number generator. This is important to ensure that each process generates different random numbers.

Random parameter values with new model

If you want to make new models for each process it is good practice to pass the model as an argument to the *run* function. This way you can be sure that each model instance in the *run* function is independent of the others.

```
def run(model):
    import os

    np.random.seed()
    model.LASER.P = np.random.normal(1, 0.1)
    return model.run(fa.Noxaxis()), hex(id(model)), os.getpid()


model = finesse.Model()
LASER = model.add(fc.Laser("LASER", P=1))
model.add(fd.PowerDetector("P", LASER.p1.o))

action = fa.Noxaxis()
N = 3
print("Original", hex(id(model)), os.getpid())
pool = ProcessPool(nodes=12)
solutions = pool.map(run, [model] * N)
solutions
```

```
Original 0x7a4e50016ad0 1950
```

```
[(<ArraySolution of noxaxis @ 0x7a4e46cee8e0 children=0>,
  '0x7a4e46eaa210',
  2034),
 (<ArraySolution of noxaxis @ 0x7a4e46cef1d0 children=0>,
```

(continues on next page)

(continued from previous page)

```
 '0x7a4e46eaa210',
 2036),
(<ArraySolution of noxaxis @ 0x7a4e46cef2a0 children=0>,
 '0x7a4e46eaaad0',
 2036)]
```

Here we can see in *solutions* that the model instance is different for each process and that the process id is different for each process. A solution is returned for each.

In this case we can also do mode advanced changes to the model, such as adding different components and running unique analyses.

MPI

MPI is a standard for message passing between processes and is widely used in high performance computing. The mpi4py library provides a Python interface to MPI. The following example demonstrates how to run simulations in parallel using MPI. This is a common way to run FINESSE models on a supercomputer or cluster that supports MPI parallelism.

Note

The following example will not easily work in a Jupyter notebook. It is recommended to run this example in a Python script.

```
import finesse
import finesse.analysis.actions as fa
import finesse.components as fc
import finesse.detectors as fd
import numpy as np
from mpi4py import MPI

comm = MPI.COMM_WORLD
rank = comm.Get_rank()
size = comm.Get_size()

def run(job_number):
    np.random.seed()
    model = finesse.Model()
    LASER = model.add(fc.Laser("LASER", P=1))
    model.add(fd.PowerDetector("P", LASER.p1.o))
    model.LASER.P = np.random.normal(1, 0.1)
    return model.run(fa.Noxaxis())

N = 100
solutions = []

# Each rank will run a different set of simulations
# The rank will start at 0 and increment by the size of the communicator
# until it reaches N
for i in range(rank, N, size):
    solutions.append(run(i))

all_solutions = comm.gather(solutions, root=0)

if rank == 0:
    # The rank 0 process will gather all the solutions and analyse them,
    # typically you would save the results to disk and then write another
    # script to analyse the results.
    pass
```

To run this example you will need to use the *mpirun* command. For example, to run this example with 4 processes you would use the following command:

```
mpirun -n 4 python my_script.py
```

This will run the script *my_script.py* with 4 processes. The *rank* variable will be different for each process and the *size* variable will be the total number of processes.

The *run* function is the same as in the previous example. The only difference is that the model is created inside the *run* function. There is no pickling or serialisation of the model needed for MPI. You should create the model inside the function call and have it return the result of the simulation.

Displaying tabulated data

This page contains several examples how to use the classes in `finesse.utilities.tables`

The first class is *Table*. It is intended for already readable data, that means strings or objects with a useful *__str__* method.

```
import numpy as np
import finesse
from finesse.utilities.tables import Table, NumberTable

data = [['a', 'b', 'c'],
        [1,2,3],
        [4,5,6]
       ]
tab = Table(data)
```

The table can be displayed with pretty printing or as html. The *print* method will prefer html display if thwat is supported and fall back to a sring representation if not. Calling the built-in *print* function always uses the string version.

```
tab.print()
```

```
<finesse.utilities.tables.Table at 0x7b955e59fe00>
```

```
print(tab)
```

```
┌───┬───┬───┐
│ a │ b │ c │
╞═══╪═══╪═══╡
│ 1 │ 2 │ 3 │
├───┼───┼───┤
│ 4 │ 5 │ 6 │
└───┴───┴───┘
```

By default the first row is considered a header. This can also be enabled for the first column or disabled.

```
print(Table(data, headerrow=True, headercolumn=True))
```

```
┌───╥───┬───┐
│ a ‖ b │ c │
╞═══╬═══╪═══╡
│ 1 ‖ 2 │ 3 │
├───╫───┼───┤
│ 4 ‖ 5 │ 6 │
└───╨───┴───┘
```

```
print(Table(data, headerrow=False, headercolumn=True))
```

```
┌───╥───┬───┐
│ a ╫ b │ c │
├───╫───┼───┤
│ 1 ╫ 2 │ 3 │
├───╫───┼───┤
│ 4 ╫ 5 │ 6 │
└───╨───┴───┘
```

```
print(Table(data, headerrow=False, headercolumn=True))
```

```
┌───╥───┬───┐
│ a ╫ b │ c │
├───╫───┼───┤
│ 1 ╫ 2 │ 3 │
├───╫───┼───┤
│ 4 ╫ 5 │ 6 │
└───╨───┴───┘
```

The textcolor, backgroundcolor and the text alignment can be changed. They can be specified in four different ways:

1. A single value that will be applied to all cells
2. One value per row in a nested list
3. One value per column in a nested list
4. A value for every cell in an array with the same dimensions as the data

The colors must be defined as RGB values, possible alignments are “left”, “right” and “center”.

```
red = (255, 0, 0)
green = (0, 255, 0)
blue = (0, 0, 255)

Table(data, color=red)
```

```
<finesse.utilities.tables.Table at 0x7b955e5cc510>
```

```
Table(data,
color=[
    [red],
    [green],
    [blue]
])
```

```
<finesse.utilities.tables.Table at 0x7b955e5d6b10>
```

```
Table(data,
    color=[
        [red, green, blue]
    ])
```

```
<finesse.utilities.tables.Table at 0x7b955e5c7ac0>
```

```
Table(data,
color=[
```

(continues on next page)

(continued from previous page)

```
    [red, green, blue],
    [blue, green, red],
    [green, red, blue]
])
```

```
<finesse.utilities.tables.Table at 0x7b955e5c7ce0>
```

```
Table(data, backgroundcolor=green)
```

```
<finesse.utilities.tables.Table at 0x7b955e5a0e50>
```

If any number of the RGB triple is smaller than zero, it is treated as if no color was given.

```
Table(data,
color=[
    [blue, (-1,0,0), red]
])
```

```
<finesse.utilities.tables.Table at 0x7b955e5a3150>
```

```
tab=Table([['abc','def','ghi'],['x','y','z']], alignment=[['left','right','center']])
print(tab)
tab.print()
```

```
┌─────┬─────┬─────┐
│ abc │ def │ ghi │
╞═════╪═════╪═════╡
│ x   │   y │  z  │
└─────┴─────┴─────┘
```

```
<finesse.utilities.tables.Table at 0x7b95a05f94f0>
```

It is also possible to generate latex code for the table.

```
print(tab.latex())
```

```
\begin{table}
\begin{tabular}{|l|r|c|}
\hline
abc & def & ghi\\
\hline
x & y & z\\
\hline
\end{tabular}
\end{table}
```

If the table cells have colored text, the *xcolor* package must be used in the latex file, the backgroundcolor is not used.

For handling different alignments in a column the *multicolumn* command is used.

```
tab = Table(
        [
            ['abc','def','ghi'],
            ['x','y','z']
        ],
        color=[
            [red, green, blue],
```

(continues on next page)

(continued from previous page)

```
            [blue, green, red]
            ],
        alignment=[
            ['left','right','center'],
            ['right', 'left', 'center']
        ]
)
print(tab.latex())
```

```
\begin{table}
\definecolor{color000000255}{RGB}{000, 000, 255}
\definecolor{color000255000}{RGB}{000, 255, 000}
\definecolor{color255000000}{RGB}{255, 000, 000}
\begin{tabular}{|l|r|c|}
\hline
\textcolor{color255000000}{abc} & \textcolor{color000255000}{def} & \textcolor
↪{color000000255}{ghi}\\
\hline
\multicolumn{1}{|r|}{\textcolor{color000000255}{x}} & \multicolumn{1}{|l|}{\textcolor
↪{color000255000}{y}} & \textcolor{color255000000}{z}\\
\hline
\end{tabular}
\end{table}
```

You can also save the data of the table as a csv file using the syntax of the *csv* package. The file will not contain any formatting or color information. The function returns the used csv writer, so you can write further data to the file.

```
import sys
tab.write_csv(sys.stdout)
```

```
abc,def,ghi
x,y,z
```

```
<_csv.writer at 0x7b955e620400>
```

If you want to process some other data as a table or change how something is shown, you can create a new class inheriting from *Table* and overload the necessary methods. One such class already exists: *NumberTable*. It displays data only containing numbers with string headers.

```
data = np.random.random((2,3))
NumberTable(data)
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e59fe00>
```

```
NumberTable(data, colnames=['a','b','c'])
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e62ca50>
```

```
NumberTable(data, rownames=['x','y'])
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e62c910>
```

```
NumberTable(data, rownames=['x','y'], colnames=['a','b','c'])
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e5ce650>
```

You can format the numbers in several wayswith a python formatting string or a function that transforms numbers to strings. These can be given for every cell at once, column- or row-wise or for every cell individually.

```
print(
    NumberTable(
        data,
        numfmt='{:.5g}'
    )
)
```

```
┌─────────┬─────────┬─────────┐
│  0.9968 │ 0.10551 │ 0.75884 │
├─────────┼─────────┼─────────┤
│ 0.49458 │ 0.33251 │ 0.73633 │
└─────────┴─────────┴─────────┘
```

```
print(
    NumberTable(
        data,
        numfmt=lambda x: round(x,2)
    )
)
```

```
┌──────┬──────┬──────┐
│  1.0 │ 0.11 │ 0.76 │
├──────┼──────┼──────┤
│ 0.49 │ 0.33 │ 0.74 │
└──────┴──────┴──────┘
```

```
print(
    NumberTable(
        data,
        numfmt=[
            ['{:.5g}','{:.2g}','{:.1g}']
        ]
    )
)
```

```
┌─────────┬──────┬─────┐
│  0.9968 │ 0.11 │ 0.8 │
├─────────┼──────┼─────┤
│ 0.49458 │ 0.33 │ 0.7 │
└─────────┴──────┴─────┘
```

```
print(
    NumberTable(
        data,
        numfmt=[
            ['{:.5g}'],
            ['{:.2g}']
        ]
    )
)
```

```
┌─────────┬─────────┬─────────┐
│ 0.9968 │ 0.10551 │ 0.75884 │
├─────────┼─────────┼─────────┤
│   0.49 │    0.33 │    0.74 │
└─────────┴─────────┴─────────┘
```

```
print(
    NumberTable(
        data,
        numfmt=[
            ['{:.5g}','{:.2g}','{:.1g}'],
            ['{:.1e}',lambda x: round(x,2), lambda x: str(x**2)]
        ]
    )
)
```

```
┌──────────┬──────┬──────────────────────┐
│  0.9968  │ 0.11 │                  0.8 │
├──────────┼──────┼──────────────────────┤
│ 4.9e-01  │ 0.33 │ 0.5421816841699789 │
└──────────┴──────┴──────────────────────┘
```

You can also color the numbers and their background by applying a function to the data array. It has to have the structure of a *matplotlib* colormap.

```
import matplotlib as mpl
cmap1 = mpl.colormaps.get_cmap('viridis')
cmap2 = mpl.colormaps.get_cmap('Greys')
tab = NumberTable(
    data,
    rownames=['x','y'],
    colnames=['a','b','c'],
    numfmt='{:.5g}',
    colfunc = cmap1,
    bgcolfunc = cmap2
)
tab.print()
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e5a3750>
```

As the colormaps expect values between 0 and 1, you can provide a normalization function.

```
NumberTable(
    [[200,500,-300]],
    colfunc=cmap1,
    norm=mpl.colors.Normalize(vmin=-300, vmax=500))
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e717d40>
```

As *NumberTable* is a child class of *Table*, it has the same display functions.

```
print(tab.latex())
```

```
\begin{table}
\definecolor{color000000000}{RGB}{000, 000, 000}
\definecolor{color033143141}{RGB}{033, 143, 141}
\definecolor{color049104142}{RGB}{049, 104, 142}
\definecolor{color072038119}{RGB}{072, 038, 119}
\definecolor{color078078078}{RGB}{078, 078, 078}
\definecolor{color085085085}{RGB}{085, 085, 085}
\definecolor{color086198103}{RGB}{086, 198, 103}
\definecolor{color099203095}{RGB}{099, 203, 095}
\definecolor{color152152152}{RGB}{152, 152, 152}
\definecolor{color198198198}{RGB}{198, 198, 198}
\definecolor{color242242242}{RGB}{242, 242, 242}
```

(continues on next page)

(continued from previous page)

```
\definecolor{color253231037}{RGB}{253, 231, 037}
\begin{tabular}{|c|c|c|c|}
\hline
 & a & b & c\\
\hline
x & \multicolumn{1}{|r|}{\textcolor{color253231037}{0.9968}} & \multicolumn{1}{|r|}{\
↪textcolor{color072038119}{0.10551}} & \multicolumn{1}{|r|}{\textcolor{color099203095}{0.
↪75884}}\\
\hline
y & \multicolumn{1}{|r|}{\textcolor{color033143141}{0.49458}} & \multicolumn{1}{|r|}{\
↪textcolor{color049104142}{0.33251}} & \multicolumn{1}{|r|}{\textcolor{color086198103}{0.
↪73633}}\\
\hline
\end{tabular}
\end{table}
```

```
tab.write_csv(sys.stdout);
```

```
,a,b,c
x,0.9968,0.10551,0.75884
y,0.49458,0.33251,0.73633
```

These classes are used in finesse by several functions to display information. For example the distances matrix of a model can be shown in table format. The function supports keyword arguments for *NumberTable* to customize the table.

```
model = finesse.Model()
model.parse("""
### L0 -> BS -> YARM of ET-LF
# input
l L0 P=1
s l0 L0.p1 BS.p1 L=10

# Main beam splitter
bs BS T=0.5 L=37.5u alpha=60
s BSsub1 BS.p3 BSAR1.p1 L=0.07478 nr=nsilica
s BSsub2 BS.p4 BSAR2.p1 L=0.07478 nr=nsilica
bs BSAR1 R=50u L=0 alpha=-36.6847
bs BSAR2 R=50u L=0 alpha=36.6847

# Y arm telescope
s lBS_ZM1 BS.p2 ZM1.p1 L=70
bs ZM1 T=250u L=37.5u Rc=-50
s lZM1_ZM2 ZM1.p2 ZM2.p1 L=50
bs ZM2 T=0 L=37.5u Rc=-82.5
s lZM2_ITMlens ZM2.p2 ITM_lens.p1 L=52.5

lens ITM_lens 75
s lITM_th2 ITM_lens.p2 ITMAR.p1 L=0

# Y arm input mirror
m ITMAR R=0 L=20u
s ITMsub ITMAR.p2 ITM.p1 L=0.2 nr=nsilicon
m ITM T=7000u L=37.5u Rc=-5580

# Y arm length
s l_arm ITM.p2 ETM.p1 L=10k

# Y arm end mirror
m ETM T=6u L=37.5u Rc=5580
```

(continues on next page)

(continued from previous page)

```
s ETMsub ETM.p2 ETMAR.p1 L=0.2 nr=nsilicon
m ETMAR R=0 L=500u

# SRM
s lBS_SRM BSAR2.p3 SRM.p1 L=10

m SRM T=0.2 L=0 Rc=-9410
s SRMsub SRM.p2 SRMAR.p1 L=0.0749 nr=nsilicon
m SRMAR R=0 L=50n

# cavities
cav cavARM ITM.p2
cav cavSRC SRM.p1 ITM.p1.i

var nsilica 1.44963098985906
var nsilicon 3.42009

lambda(1550n)
""")

ps = model.propagate_beam(model.ITM.p1.o, model.SRM.p1.i)

import matplotlib as mpl
cmap = mpl.colormaps.get_cmap('viridis')
norm = mpl.colors.Normalize(vmin=-200, vmax=200)

tab = ps.distances_matrix_table(
    colfunc = cmap,
    norm = norm
    )
tab.print()
```

```
<finesse.utilities.tables.NumberTable at 0x7b955e266990>
```

Tracebacks in Jupyter notebooks

Andreas Freise, 16/04/2020

The following demonstrates how Finesse handles parse errors. When run from IPython or Jupyter the full traceback is suppressed and only the parser error itself is shown. When run from Python it always returns the full traceback.

You can print the latest full traceback with *finesse.tb()*.

If you want to always see the full traceback, for example, during debugging, you can set this with *finesse.show_tracebacks()*

We use a very simple example model for the demonstration:

```
from finesse.env import is_interactive

print(is_interactive())
```

```
True
```

```
import finesse

code = """
l laser P=1
pd tmp laser.p1.o
"""
```

(continues on next page)

(continued from previous page)

```
kat = finesse.Model()
kat.parse(code)
```

```
<finesse.model.Model at 0x782901f4c050>
```

A syntax error in the katscript gives a parser error:

```
code = """
l1 laser P=1
pd tmp laser.p1.o
"""

kat = finesse.Model()
kat.parse(code)
```

```
KatSyntaxError:          (use finesse.tb() to see the full traceback)
line 2: unknown element 'l1'. Did you mean 'l'?
   1:
-->2: l1 laser P=1
      ^^
   3: pd tmp laser.p1.o
```

```
code = """
l1 laser P=1
pd tmp laser.p1.o
"""

kat = finesse.Model()
kat.parse(code)
```

```
KatSyntaxError:          (use finesse.tb() to see the full traceback)
line 2: unknown element 'l1'. Did you mean 'l'?
   1:
-->2: l1 laser P=1
      ^^
   3: pd tmp laser.p1.o
```

finesse.show_tracebacks() can also switch tracebacks off again:

```
finesse.show_tracebacks(False)
```

```
False
```

6.3 Frequency-domain modelling

One of the primary tasks undertaken with FINESSE is simulating the frequency domain behaviour of complex optical systems. This involves computing various transfer functions of small signals injected into the system and computing how they affect various outputs. With knowledge of how these small signals behave in an optical system we can then model how classical and quantum noise propagates throughout an optical system and the behaviour of control systems designed to keep the an optical system operating.

Modelling linear time invariant (LTI) systems is a broad subject and covered in significant and better detail elsewhere. As background reading primarily targeted at modelling LTI systems for sensing and control of interferometers we recommend Chapter 8, *Interferometric length sensing and control*, of the [18] — much of this section of the manual will assume physics knowledge from this chapter.

Note

Those with previous experience in modelling LTI responses in FINESSE 2 will be familiar with the `fsig` (page 196) command that was used to inject a signals. FINESSE 3 still keeps the same name `fsig` (page 196) but this is now purely the frequency of the small signal excitation `Model.fsig.f` which globally sets what signal frequency is used. Injecting signals is now done using signal generator components or special analyses like `frequency_response` (page 221).

FINESSE 3 introduces a significantly changed interface from FINESSE 2 for modelling transfer functions. In this section we will cover these modelling tasks from the basic to more advanced features.

6.3.1 Modelling transfer functions

Transfer functions are frequency domain functions that relate how much signal is measured at an output relative to some injected signal. When modelling a *linear time invariant* (LTI) system we must assume any signal is small enough that there are no non-linear behaviours introduced by injecting the signal in the first place. In FINESSE, when we say a *signal* we refer to some small sinusoidal oscillation in either a model parameter or detector output. And when we say we inject a signal, we mean that we are exciting some property of the model by shaking it some small amount at a frequency we refer to as `fsig` (page 196) — the signal frequency (page 120). This frequency is set by the `model.fsig.f` parameter and is in units of Hertz. In various documentation we may also refer to the signal frequency in units of radians-per-second using the symbol $\Omega \equiv 2\pi f_{\text{sig}}$. The signal frequency must always be greater than zero as it must be oscillating.

For example, the output signal in an interferometric system is usually some optical power measurement, a small oscillation in units of Watts of optical power measured. An input signal would be a small mirror displacement in units of meters. Therefore a transfer function from mirror displacement to measured optical power would be in W/m, and will typically vary in amplitude and phase at different frequencies.

Operating points: Carrier and signals — DC and AC

It is important at this stage to understand some of FINESSE's nomenclature when it comes to LTI systems. Optomechanical systems are typically non-linear in nature with respect to any of its defining parameters. They are only approximately linear for small changes in these parameters. Mathematically, this is equivalent to taking the non-linear equations describing the system and Taylor expanding around a specific point. This point is known as the *operating point*, the *DC state* of the model, or the linearisation point. In FINESSE this operating point is set by the current value of all the various elements and their parameters that are included in the model. For example, the laser power, the mirror tunings, the modulator frequency, etc. For those familiar to Simulink, this is what the `linmod` functions are performing. FINESSE does this linearisation for you.

The operating point determines the steady state of the optical fields in a model (we do not model DC changes in electrical or mechanical systems currently). For example, the circulating power in a Fabry-Perot cavity depends non-linearly on the microscopic tunings of the cavity mirrors. We refer to this as *solving the carrier fields*, by carrier fields we mean any static optical fields generated by lasers or modulators, so this includes radio-frequency sidebands at all optical nodes in the model.

The optical signal fields are the small sidebands around these carrier fields generated by any injected signals in the model. For example, you can inject a signal into a mirror mechanical `z` node, to oscillate its position along the optical axis. This in turn will create optical signal sidebands around any carrier fields being reflected from the mirror to model an oscillating phase modulation. Or you could inject a signal into a laser's electronic amplitude node to modulate the power of the outputted optical field by some small amount. Or you could inject a signal into an optical node to directly modulate the signal sidebands around a carrier field. Signal fields are sometimes referred to as the *AC fields*.

Computing transfer functions - manual method

Note

The `fsig` command from FINESSE 2 previously set the signal frequency, amplitude, phase, and a target for injecting. This is now split between `fsig` (page 196) and `signal_generator` (page 169). `fsig` (page 196)

only sets the signal frequency in version 3.

Let us consider a simple example of a laser field reflected from a mirror 100m away then back towards the laser. We add some power detectors to look at the DC and AC state of the optical fields.

```
import finesse
finesse.init_plotting()

model = finesse.Model()
model.parse("""
l l1 P=1
s s1 l1.p1 m1.p1 L=100
m m1 R=1 T=0

fsig(1) # model 1Hz oscillations

pd P_DC l1.p1.i          # a power detector looking at the DC power
pd1 P_AC l1.p1.i f=fsig # a power detector looking at the fsig frequency fluctuations|
""")
sol = model.run("noxaxis()")
sol['P_DC'], sol['P_AC']
```

```
(np.float64(1.0000000000000002), np.complex128(0j))
```

We see that we measure 1W of DC power, and 0W of AC fluctuations. To inject a signal we need to use the signal_generator (page 169) (sgen (page 169) for short) component and connect it to some electrical or mechanical node. As we are measuring optical power we can modulate the laser amplitude:

```
model = finesse.Model()
model.parse("""
l l1 P=1
s s1 l1.p1 m1.p1 L=100
m m1 R=1 T=0

fsig(1) # model 1Hz oscillations
sgen sig l1.pwr # inject a signal to modulate the laser power

pd P_DC l1.p1.i          # a power detector looking at the DC power
pd1 P_AC l1.p1.i f=fsig # a power detector looking at the fsig frequency fluctuations|
""")
sol = model.run("noxaxis()")
sol['P_DC'], sol['P_AC']
```

```
(np.float64(1.0000000000000002),
 np.complex128(0.9999999999912151-4.19169004389109e-06j))
```

Now we see that the DC state is not affected, signals do not change carrier fields or the DC state of the simulation. But we do see that the AC detector is measuring a complex value which absolute value of 1 and a small phase component. So what is this complex value? It is a single point of a transfer function, it tells us the relationship between a 1Hz oscillation applied at the laser amplitude input relative to the optical power fluctuation at the power detector. 1V applied at the laser amplitude node gives 1W of power fluctuation. Therefore our transfer function has units of W/V.

The absolute value tells us how much signal is generated, 1W/V, nothing particularly exciting, but all we are doing is reflecting the laser from the mirror. If the mirror was not a perfect reflector we see we get a small transfer function

```
model.m1.set_RTL(R=0.5, T=0.5)
sol = model.run("noxaxis()")
sol['P_AC']
```

```
np.complex128(0.4999999999956076-2.095845021945545e-06j)
```

This phase component is from the time delay taken by injecting the signal at the laser and it propagating to the mirror and back to the detectors. If we make it longer we can see we accumulate more delay

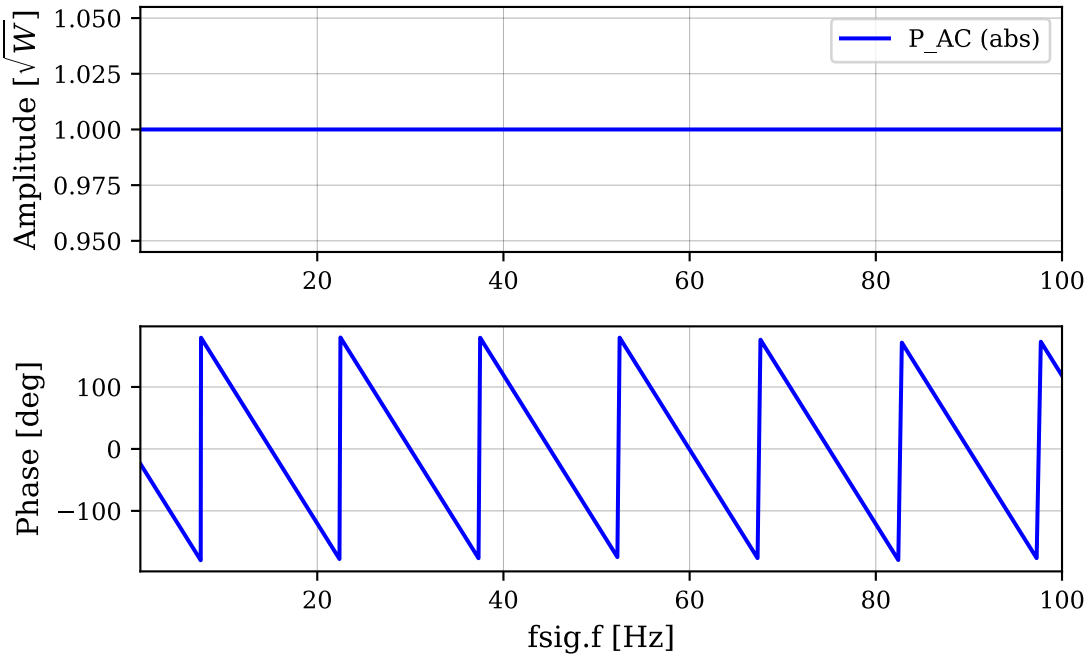
```
model.m1.set_RTL(R=1, T=0)
model.spaces.s1.L = 10e6
sol = model.run("noxaxis()")
sol['P_AC']
```

```
np.complex128(0.9134274721511403-0.407001539456029 85j)
```

The process described above is the same for any model, no matter how complicated. If you introduce more signal generators they will all be coherently injected! If you want to model multiple signal injections with this manual method you'll need to switch off certain signal generators and switch on others then recompute the transfer functions.

If you want to compute a plot we can simply sweep the signal frequency. From this we can see that the amplitude remains constant, but there is an evolving phase delay with frequency.

```
sol = model.run("xaxis(fsig, log, 1, 100, 1000)")
sol.plot('P_AC');
```



We can also apply some electronic filters to shape our inputs if required. For example, take a Mephisto NPRO laser, the curent drive modulation has a 5kHz bandwidth on it. Below we can quickly add this in with a 5kHz lowpass Butterworth filter (see `butter` (page 191)) and then excite the input to that instead.

```
model = finesse.Model()
model.parse("""
l l1 P=1
s s1 l1.p1 m1.p1 L=10k
m m1 R=1 T=0

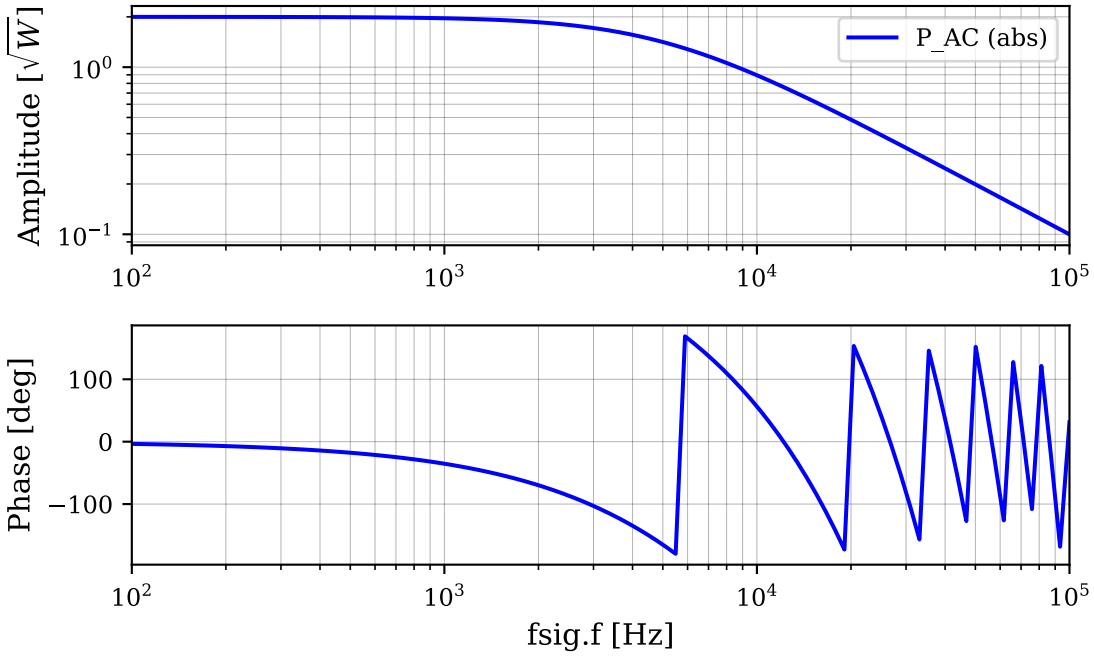
fsig(1) # model 1Hz oscillations

# 5kHz bandwidth with a butter filter
butter current_drive 1 lowpass 5k
link(current_drive, l1.amp)
sgen sig current_drive.p1.i
```

(continues on next page)

(continued from previous page)

```
pd P_DC l1.p1.i          # a power detector looking at the DC power
pd1 P_AC l1.p1.i f=fsig  # a power detector looking at the fsig frequency fluctuations|
""")
sol = model.run("xaxis(fsig, log, 100, 100k, 100)")
sol.plot('P_AC', logx=True, logy=True);
```



6.3.2 Readouts for frequency domain analysis

As seen in the previous section we can manually compute single input to multiple output (SIMO) transfer functions with a few lines of KatScript. If we wanted to inject multiple signals separately from one another to compute a full multiple input multiple output (MIMO) it can get awkward with the manual method. This would require adding and removing signal generators in some for-loop, then concatenating all the results together.

In FINESSE 3 we have introduced the Readout (page 183) component. This is a mix of a detector and a component. It has one optical port for light to be incident on, then it can have one or several output electrical ports. This means instead of taking transfer functions from some signal generator to a detector, we can specify the electrical or mechanical nodes to inject a signal at, then simply read the result out at another node.

Several readout components exist:

- `ReadoutDC` (page 186): A DC photodiode readout with a single `readout.DC.o` signal node
- `ReadoutRF` (page 187): A single RF demodulated readout which has `I` and `Q` demodulated signals

Consider a similar example to what we discussed in the previous section: a laser beam reflected from a mirror far away then back on to a photodiode. Instead of specifying a power detector `pd1` (page 180) we now use a `readout_dc` (page 186) component which we'll call `REFL`. This models a photodiode which measures signal fluctuations and converts it into an electrical output at the port `REFL.DC`. You'll notice in this we do not need to specify the signal frequency like we did previously with `pd1` (page 180)

```
import finesse
finesse.init_plotting()

model = finesse.Model()
model.parse("""
l l1 P=1
s s1 l1.p1 m1.p1 L=10k
```

(continues on next page)

(continued from previous page)

```
m m1 R=1 T=0
readout_dc REFL l1.p1.i

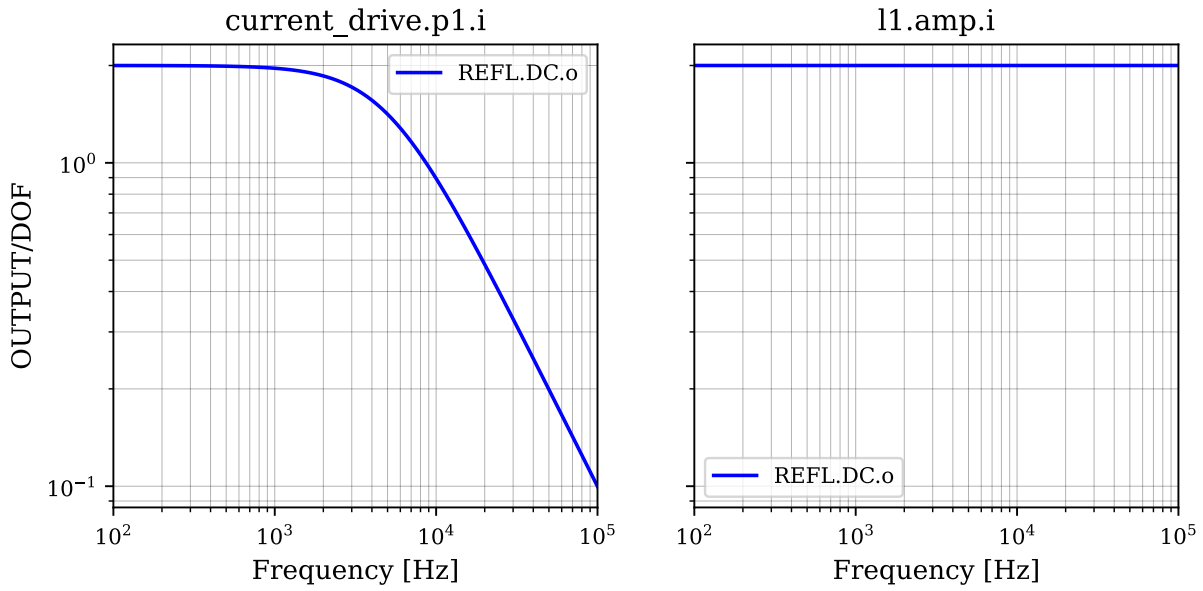
butter current_drive 1 lowpass 5k
link(current_drive, l1.amp)

fsig(1)
""")
```

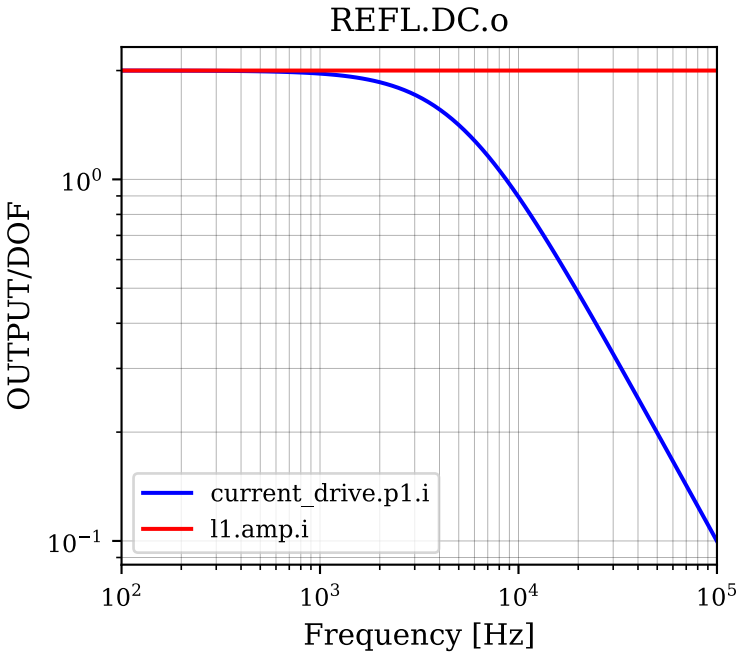
```
<finesse.model.Model at 0x7d12746ce900>
```

Now we introduce a new analysis that can be performed on a model, the `frequency_response` (page 221) action. This single action allows you to inject multiple signals and read them out at multiple outputs over a generic frequency vector. Above we use the `geomspace(start, stop, steps)` (see also `numpy.geomspace()`) to create a logarithmic vector (`numpy.linspace()` is also available as well as specify arbitrary ones, e.g. `[1, 10, 11.5, 18]`). Next we provide an array of inputs nodes, followed by an array (or singular in this case) output node.

```
sol = model.run("""
    frequency_response(
        geomspace(100, 100k, 100),       # frequency vector
        [current_drive.p1.i, l1.amp.i],  # injection nodes
        REFL.DC.o                        # output nodes
    )
""")
# sol.plot() defaults to .plot_inputs
sol.plot_inputs();
```




```
sol.plot_outputs(); # alternate plot method
```



Note

In cases where a port only has a single node you can just write the port for shorthand `REFL.DC.o` is equivalent to `REFL.DC` when some element or command expects a node.

Computing frequency domain responses is a key feature of FINESSE, much more details can be found in the following section on the variety of analyses that can be performed, Calculating general transfer functions (page 272).

6.3.3 Calculating general transfer functions

There are three types of nodes in FINESSE that inject and probe different types of signals (see also Basic usage of ports and nodes (page 139)):

- Optical: Optical modes.
- Mechanical: Mechanical degrees of freedom such as lengths and angles (pitch and yaw), and forces and torques.
- Electrical: electrical signals such as outputs from photodetectors and `DegreeOfFreedom` (page 190) components.

Note

Whilst differentiated in descriptions, internally electrical and mechanical nodes are both `SignalNode` attached to an element. Such as the laser amplitude signal `laser.amp`, or *mirror.mech.z*. Signal nodes represent a small AC oscillations at the `model.fsig` frequency.

The previous section described how transfer functions are computed from electrical and mechanical nodes to electrical and mechanical nodes using the `FrequencyResponse` (page 221) action. There are a total of four frequency response actions which calculate transfer functions between optical nodes as well so that optical modes can be probed directly. This is especially useful when analyzing quantum noise. The four actions are

Action Name	Input Nodes	Output nodes
`FrequencyResponse` (page 221)	Electrical/Mechanical	Electrical/Mechanical
`FrequencyResponse2` (page 223)	Optical	Electrical/Mechanical
`FrequencyResponse3` (page 225)	Optical	Optical
`FrequencyResponse4` (page 227)	Electrical/Mechanical	Optical

In this section we use these four actions to analyze a simple Fabry-Perot cavity with Pound-Drever-Hall [31] readout.

Setting up the model

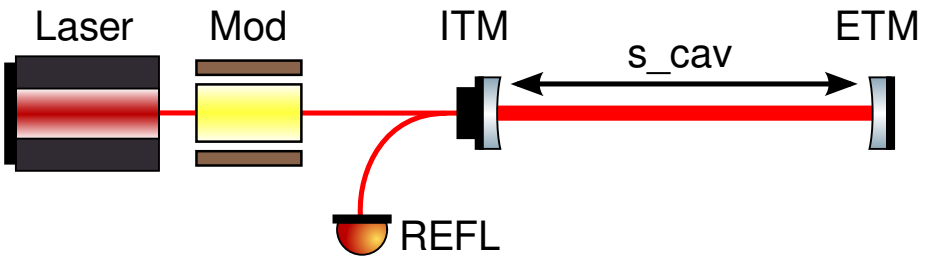


Fig. 6.8: The Fabry-Perot setup

```
import numpy as np
import finesse
import finesse.components as fc
import finesse.analysis.actions as fa
from finesse.plotting import bode

finesse.init_plotting()

Fmod_Hz = 9e6
model = finesse.Model()
ETM = fc.Mirror("ETM", T=0, L=0, Rc=2245)
ITM = fc.Mirror("ITM", T=0.014, L=0, Rc=1940)
Laser = fc.Laser("Laser", P=5e3)
Mod = fc.Modulator("Mod", f=Fmod_Hz, midx=0.2, order=1)
REFL = fc.ReadoutRF("REFL", ITM.p2.o, f=Fmod_Hz, output_detectors=True)

model.add((Laser, Mod, ETM, ITM, REFL))
model.link(Laser.p1, Mod, ITM.p2, ITM.p1, 4e3, ETM.p1)
```

Note

This example uses the Python API to construct the model. See Python Interface (page 244) for more details.

If you are using higher order modes, you would also add a `Cavity` to use the mode of the cavity to define the modal basis. See Defining the modal basis (page 329) for details. Here we manually set a mismatched beam parameter for illustrative purposes so that there will be coupling between the HOMs and the fundamental.

```
# Usually we would do this
# model.add(fc.Cavity("cavARM", model.ETM.p1.o))
# and not this
model.ITM.p1.i.q = finesse.BeamParam(w=55e-3, Rc=1943)
model.modes("even", maxtem=2)
print(model.mismatches_table())
```

```
╒═══════════════════════╤═══════════════╤═══════════════╕
│       Coupling        │ Mismatch (x)  │ Mismatch (y)  │
╞═══════════════════════╪═══════════════╪═══════════════╡
│ ETM.p1.i -> ETM.p1.o  │        0.0033 │        0.0033 │
├───────────────────────┼───────────────┼───────────────┤
│ ETM.p2.i -> ETM.p2.o  │        0.0033 │        0.0033 │
├───────────────────────┼───────────────┼───────────────┤
│ ITM.p1.i -> ITM.p1.o  │        0.0001 │        0.0001 │
├───────────────────────┼───────────────┼───────────────┤
│ ITM.p2.i -> ITM.p2.o  │        0.0001 │        0.0001 │
╘═══════════════════════╧═══════════════╧═══════════════╛
```

This is fair bit of mismatch, so we need to add a `Lock` (page 190) to make sure the cavity is on the correct operating point before calculating transfer functions. See Locking actions (page 72) for details.

```
model.add(
finesse.locks.Lock("cav_lock", model.REFL.outputs.I, model.ETM.phi, 1e-4,
1e-6)
);
```

Calculating the transfer functions

The `FrequencyResponseN` actions are called like

```
FrequencyResponseN(F_Hz, [input_nodes], [output_nodes], name="action name")
```

For `FrequencyResponse{2,3,4}` which include optical nodes, input and output optical nodes need to be listed as a tuple `(node, freq)` where `freq` is the frequency the sideband is calculated at. This frequency must be a symbolic reference to the model's `fsig.f` parameter. See the section on signal frequencies (page 120) for more information.

Note

Positive frequencies correspond to upper sidebands while negative frequencies correspond to the *conjugate* of the lower sidebands, not the lower sidebands. Therefore, propagation of both result in a phase delay.

For example, with nodes defined as

```
fsig = model.fsig.f.ref
carrier_nodes = [
    (node, +fsig),
    (node, -fsig),
]
rf_sideband_nodes = [
    (node, Fmod_Hz + fsig),
    (node, Fmod_Hz - fsig),
    (node, -Fmod_Hz + fsig),
    (node, -Fmod_Hz - fsig),
]
```

the first node in `carrier_nodes` is the upper sideband of the carrier and the second node is the conjugate of the lower sideband of the carrier. The first two nodes listed in `rf_sideband_nodes` are the upper sideband and conjugate of the lower sideband around the RF sideband at `+Fmod_Hz`. The second two nodes are the upper and conjugate of the lower sideband around the RF sideband at `-Fmod_Hz`.

In our example we calculate the response at the reflection of the ITM to signals excited at or by the ETM for the different types of signals.

```
# Run the four FrequencyResponse actions
F_Hz = np.geomspace(1, 2e3, 200)
model.fsig.f = 1  # make sure to set this or you'll get a warning
fsig = model.fsig.f.ref  # Make a shorthand reference to the fsig symbol

# FrequencyResponse actions need a symbolic reference to frequency
sol = model.run(
    fa.Series(
        # First lock the cavity
        fa.RunLocks(),
        # ETM motion to PDH readout
        fa.FrequencyResponse(
            F_Hz,
            [model.ETM.mech.z],
            [model.REFL.I.o, model.REFL.Q.o],
            name="fresp",
        ),
        # ETM optical fields to PDH readout
        fa.FrequencyResponse2(
            F_Hz,
            [
                (model.ETM.p1.o, +fsig),  # upper sideband
                (model.ETM.p1.o, -fsig),  # conjugate lower sideband
            ],
            [model.REFL.I.o, model.REFL.Q.o],
            name="fresp2",
        ),
        # ETM optical fields to ITM optical fields
        fa.FrequencyResponse3(
            F_Hz,
            [
                (model.ETM.p1.o, +fsig),
                (model.ETM.p1.o, -fsig),
            ],
            [
                (model.ITM.p2.o, +fsig),
                (model.ITM.p2.o, -fsig),
            ],
            name="fresp3",
        ),
        # ETM motion to ITM optical fields
        fa.FrequencyResponse4(
            F_Hz,
            [model.ETM.mech.z],
            [
                (model.ITM.p2.o, +fsig),
                (model.ITM.p2.o, -fsig),
            ],
            name="fresp4",
        ),
    )
)
```

The transfer functions are returned in the `out` attribute of the solutions. The dimensions returned for each action are

Action Name	Index 1	Index 2	Index 3	Index 4	Index 5
`FrequencyResponse` (page 221)	frequencies	outputs	inputs	N/A	N/A
`FrequencyResponse2` (page 223)	frequencies	outputs	inputs	HOMs (input)	N/A
`FrequencyResponse3` (page 225)	frequencies	outputs	inputs	HOMs (output)	HOMs (input)
`FrequencyResponse4` (page 227)	frequencies	outputs	inputs	HOMs (output)	N/A

The transfer functions computed with `FrequencyResponse` (page 221) can also be accessed through the output and input keys like `sol["fresp"]["out", "in"]`. In this example each of the solutions has three HOMs where

```
for idx, hom in enumerate(model.homs):
    print(f"index {idx} corresponds to HG{''.join(hom.astype(str))}")
```

```
index 0 corresponds to HG00
index 1 corresponds to HG20
index 2 corresponds to HG02
```

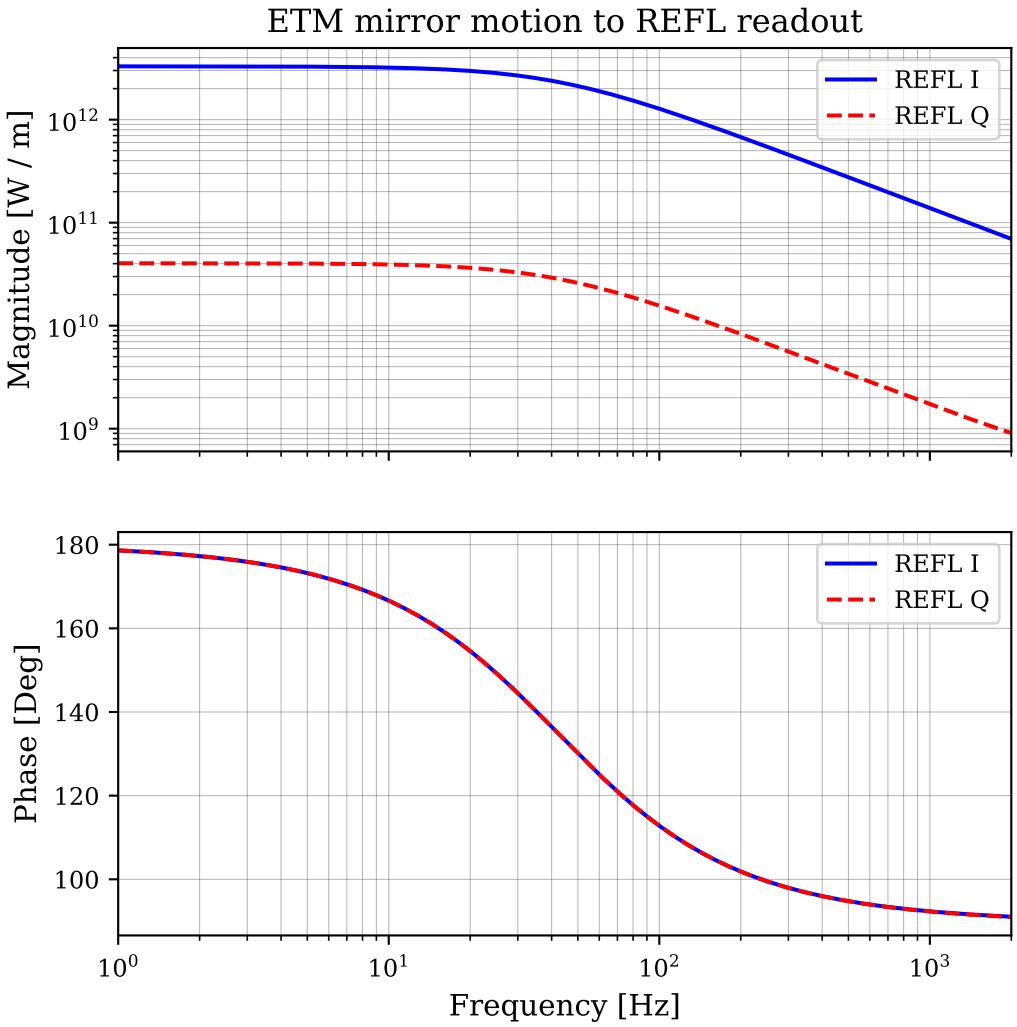
Frequency response of the fundamental mode

In this example, `FrequencyResponse` (page 221) calculates ETM mirror motion to the PDH REFL readout. There are two outputs (the PDH quadratures) and one input (ETM motion). Here we plot mirror motion to the I and Q PDH readout.

```
print("FrequencyResponse shape", sol["fresp"].out.shape)

axs = bode(F_Hz, sol["fresp"].out[..., 0, 0], db=False, label="REFL I")
bode(F_Hz, sol["fresp"].out[..., 1, 0], axs=axs, db=False, label="REFL Q", ls="--")
axs[0].set_ylabel("Magnitude [W / m]")
axs[0].set_title("ETM mirror motion to REFL readout");
```

```
FrequencyResponse shape (200, 2, 1)
```



As noted above, transfer functions calculated with `FrequencyResponse` (page 221) can also be accessed by their

input and output keys:

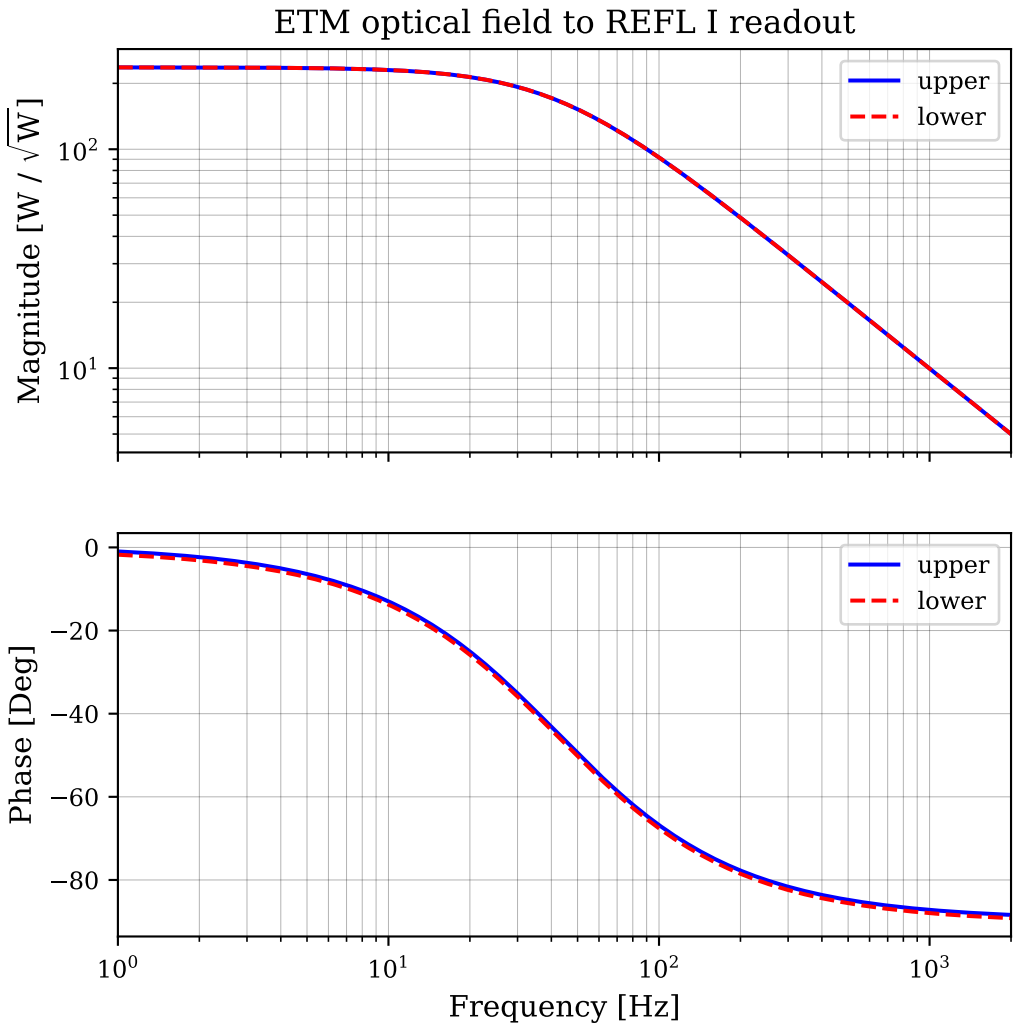
```
print(np.all(sol["fresp"]["REFL.I.o", "ETM.mech.z"] == sol["fresp"].out[..., 0, 0]))
print(np.all(sol["fresp"]["REFL.Q.o", "ETM.mech.z"] == sol["fresp"].out[..., 1, 0]))
```

```
True
True
```

In this example, `FrequencyResponse2` (page 223) calculates optical fields to PDH REFL readout. It has the same two outputs as `FrequencyResponse` (page 221) and two inputs (the upper and lower sidebands of the optical fields at the ETM). Here we plot the upper and conjugate lower sidebands of the optical fields at the front of the ETM to the REFL I PDH readout.

```
print("FrequencyResponce2 shape", sol["fresp2"].out.shape)
axs = bode(F_Hz, sol["fresp2"].out[..., 0, 0, 0], db=False, label="upper")
bode(F_Hz, sol["fresp2"].out[..., 0, 1, 0], axs=axs, db=False, label="lower", ls="--")
axs[0].set_ylabel("Magnitude [W / $\\sqrt{\\mathrm{W}}$]")
axs[0].set_title("ETM optical field to REFL I readout");
```

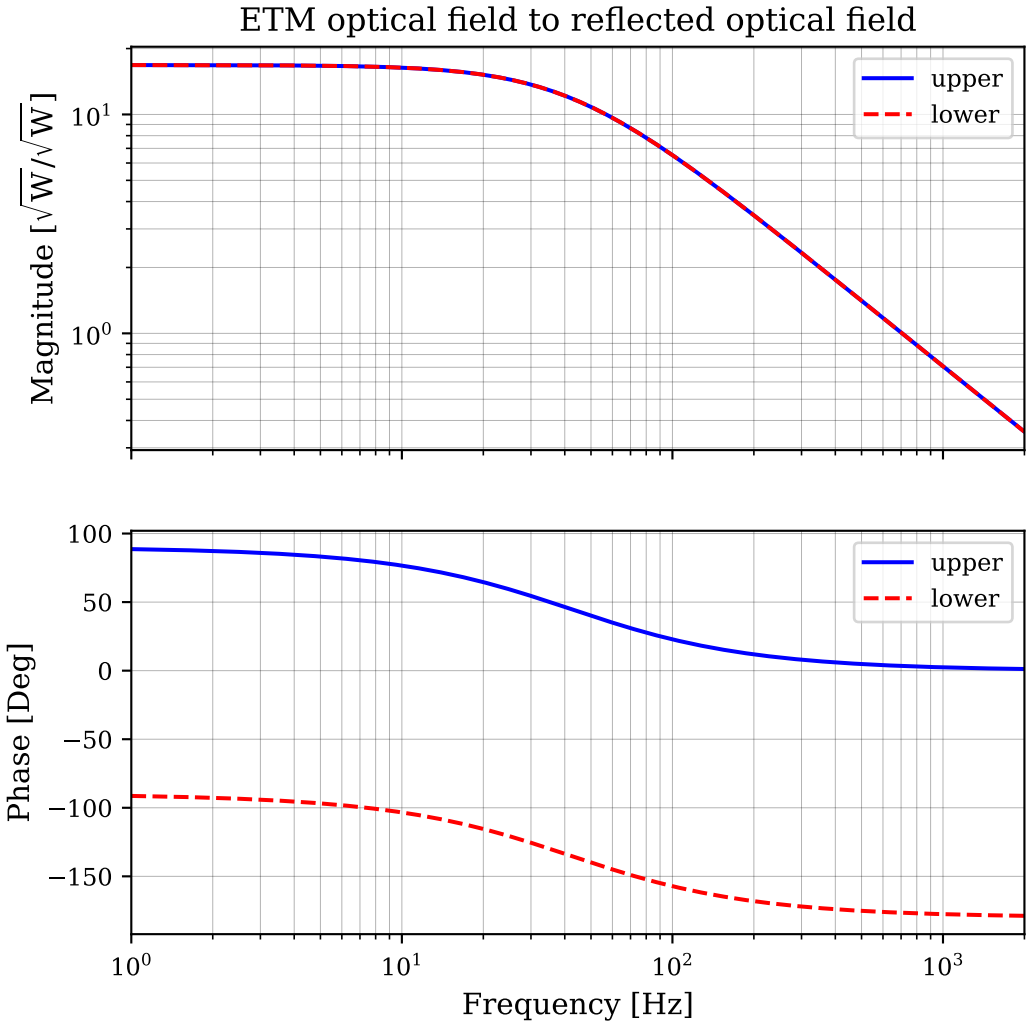
```
FrequencyResponce2 shape (200, 2, 2, 3)
```



In this example, `FrequencyResponse3` (page 225) calculates optical fields at the ETM to optical fields at the ITM. There are two outputs (upper and lower sidebands of the fields at the ITM) and two inputs (upper and lower sidebands of the fields at the ETM). Here we plot the upper and conjugate lower sidebands at the front of the ETM to the sidebands at the back of the ITM.

```
print("FrequencyResponse3 shape", sol["fresp3"].out.shape)
axs = bode(F_Hz, sol["fresp3"].out[..., 0, 0, 0, 0], db=False, label="upper")
bode(F_Hz, sol["fresp3"].out[..., 1, 1, 0, 0], axs=axs, db=False, label="lower", ls="--")
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}} / \\sqrt{\\mathrm{W}}$]")
axs[0].set_title("ETM optical field to reflected optical field");
```

```
FrequencyResponse3 shape (200, 2, 2, 3, 3)
```



There is no mixing between the uppper and lower sidebands in this case:

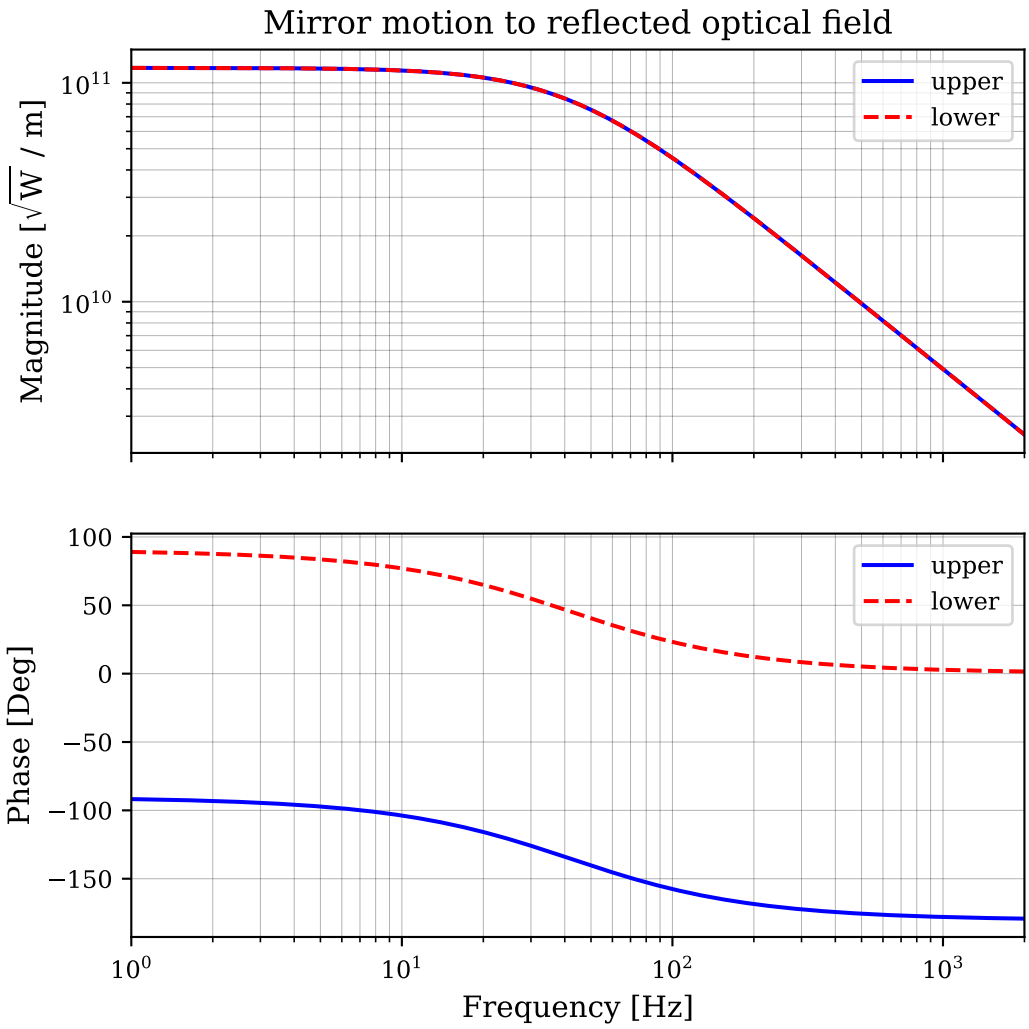
```
print(np.allclose(sol["fresp3"].out[..., 1, 0, :, :], 0))
print(np.allclose(sol["fresp3"].out[..., 0, 1, :, :], 0))
```

```
True
True
```

In this example, `FrequencyResponse4` (page 227) calculates ETM mirror motion to optical fields at the back of the ITM. There are two outputs (upper and lower sidebands of the fields at the ITM) and one input (ETM motion). Here we plot ETM mirror motion to the upper and conjugate lower sidebands at the back of the ITM.

```
print("FrequencyResponse4 shape", sol["fresp4"].out.shape)
axs = bode(F_Hz, sol["fresp4"].out[..., 0, 0, 0], db=False, label="upper")
bode(F_Hz, sol["fresp4"].out[..., 1, 0, 0], axs=axs, db=False, ls="--", label="lower")
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}}$ / m]")
axs[0].set_title("Mirror motion to reflected optical field");
```

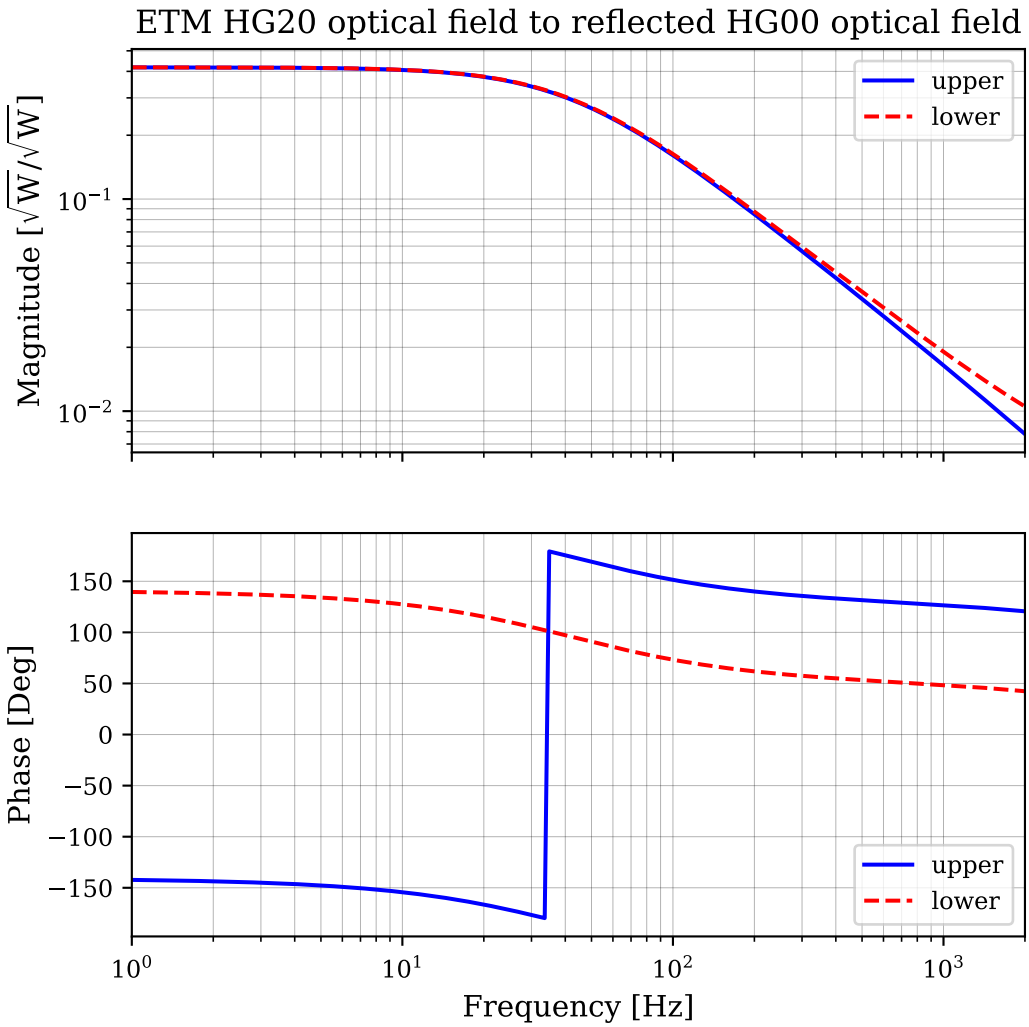
```
FrequencyResponse4 shape (200, 2, 1, 3)
```



Frequency response between HOMs

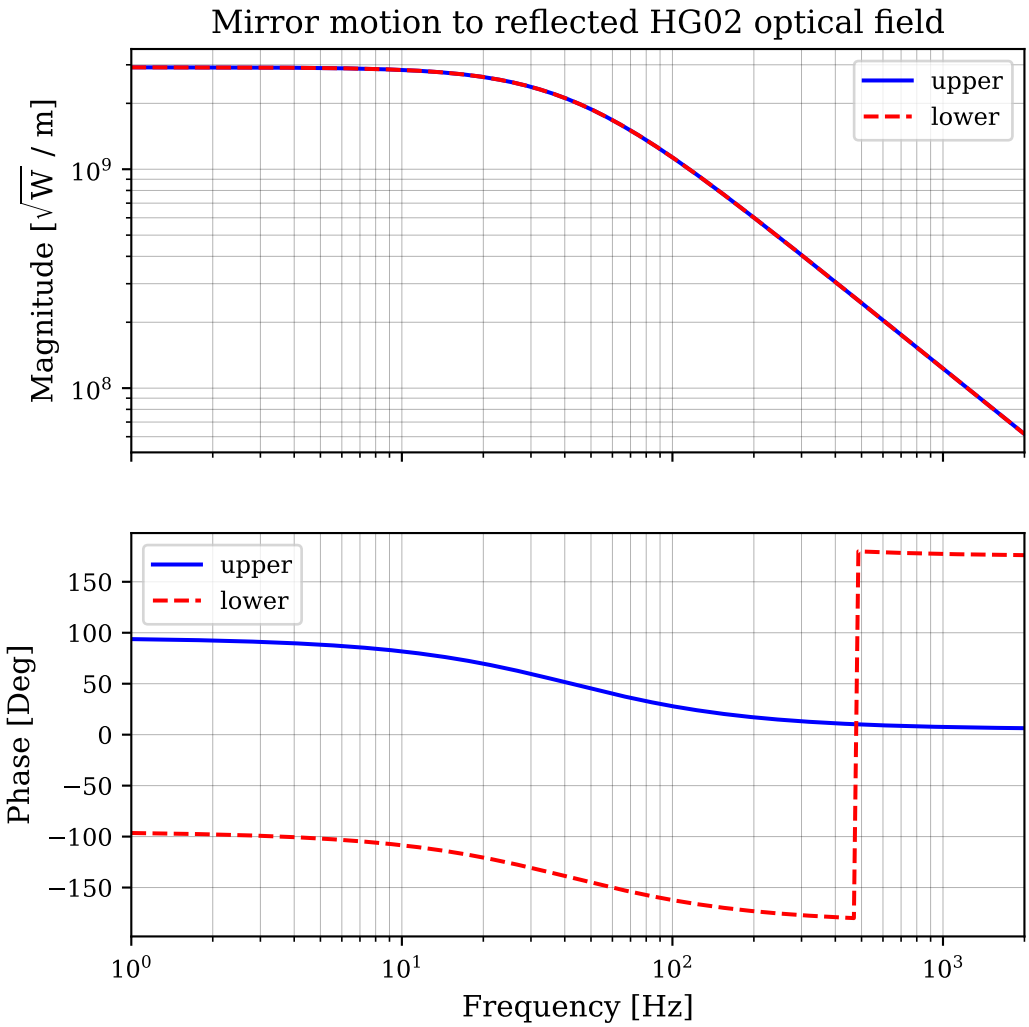
We can also look at transfer functions between HOMs. Here we plot the HG20 upper and conjugate lower sidebands at the front of the ETM to the HG00 upper and conjugate lower sidebands at the back of the ITM.

```
axs = bode(F_Hz, sol["fresp3"].out[..., 0, 0, 0, 1], db=False, label="upper")
bode(F_Hz, sol["fresp3"].out[..., 1, 1, 0, 1], axs=axs, db=False, label="lower", ls="--")
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}} / \\sqrt{\\mathrm{W}}$]")
axs[0].set_title("ETM HG20 optical field to reflected HG00 optical field");
```



Here we plot mirror motion to the HG02 upper and conjugate lower sidebands at the back of the ITM.

```
axs = bode(F_Hz, sol["fresp4"].out[..., 0, 0, 2], db=False, label="upper")
bode(F_Hz, sol["fresp4"].out[..., 1, 0, 2], axs=axs, db=False, ls="--", label="lower")
axs[0].set_ylabel("Magnitude [$\\sqrt{\\mathrm{W}}$ / m]")
axs[0].set_title("Mirror motion to reflected HG02 optical field");
```



Similarly, the response of `REFL.Q` to excitation of the conjugate lower HG02 mode at the ETM is given by `sol["fresp2"].out[..., 1, 1, 2]`, etc.

6.3.4 Closing control loops

Discuss the difference in Finesse between DC and AC closed loop control.

DC control loops - 'lock'

DC control loops are used to manually adjust the degrees of freedom (DOFs) in an optical system based on error signal measurements, until the error signals are within a specified tolerance. Several Finesse `Actions` have been developed to facilitate such locking. In particular, the `Action` `RunLocks` can be used to move a system to lock. Below we describe how this `Action` works and show an example of its use.

Methods Used for DC Locking

`RunLocks` provides two methods that the user may choose from to attempt to achieve lock in a Finesse model. Such a model must include user-defined locks specified using the `lock` command. These locks specify the error signals that will be used in locking, the unique DOF associated with each error signal, and the gain relating each error signal/DOF pair. To describe how `RunLocks` works, let us denote the set of values of the error signals as the vector $\vec{E}$, the set of values of the DOFs as the vector $\vec{F}$, and the set of gains as $\vec{G}$.

In the first method that a user may call when running `RunLocks`, henceforth named the 'proportional' method, each DOF responds proportionally to the error signal with which it was paired in the model. That is, it is assumed that each error signal E_i depends only on the DOF F_i:

$\Delta E_i = G_i \Delta F_i.$

Therefore, to attempt to lock the model with this method, the value the DOFs are iteratively updated according to the following equation, where j is the iteration step:

$F_{i,j+1} = F_{i,j} - \frac{E_{i,j}}{G_i}$.

The iteration continues until all error signals are within tolerances or until a user-defined maximum number of iterations is reached. This method may be called within `RunLocks` by using the option `method="proportional"`.

However, this method has its limitations because, in general, DOFs tend to affect multiple error signals. This can be observed by constructing a 'gain matrix,' which shows the derivatives of all error signals with respect to all DOFs. This gain matrix should be diagonal if the proportional method is to be used. For a real system, however, some off-diagonal elements may be even larger than the diagonal elements. Note that when using `RunLocks`, we assume that the model is close enough to lock that a change in any DOF induces either a linear or a negligible change in the error signals; i.e., all gains are constant or near zero throughout the locking process. The `Action CheckLinearity` may be used to check this assumption. If this assumption holds, we can write the sensing matrix as shown below:

$$\begin{bmatrix} \Delta E_1 \\ \Delta E_2 \\ \dots \\ \Delta E_n \end{bmatrix} = \begin{bmatrix} G_{11} & G_{12} & \dots & G_{1n} \\ G_{21} & G_{22} & \dots & G_{2n} \\ \dots & \dots & \dots & \dots \\ G_{n1} & G_{n2} & \dots & G_{nn} \end{bmatrix} \begin{bmatrix} \Delta F_1 \\ \Delta F_2 \\ \dots \\ \Delta F_n \end{bmatrix}.$$

In this way, the sensing matrix can be thought as a constant Jacobian matrix. Based on this intuition, the second method that `RunLocks` may use applies Newton's method for multidimensional root-finding. This method may be used with the option `method="newton"`. At each iteration step, the DOFs are changed all together according to the following equation:

$\vec{F}_{j+1} = \vec{F}_j - \mathbf{G}^{-1}\vec{E}_j$.

Theoretically, this method should achieve lock much faster and for a wider range of initial conditions than the proportional method. Below, we show this to be true in practice.

Example: Locking a Dual-Recycled Interferometer

Here, we present a simplified example for the locking of a dual-recycled interferometer, similar to Advanced Virgo. We compare the performance of the two locking methods of `RunLocks` by attempting to lock this interferometer to the dark fringe. We'll first set up the optical geometry. This geometry uses the same three modulation frequencies as used in Advanced Virgo for the construction of error signals.

```
import matplotlib.pyplot as plt
import numpy as np

import finesse
from finesse.analysis.actions import OptimiseRFReadoutPhaseDC, RunLocks, SetLockGains

finesse.init_plotting()

kat = finesse.Model()
kat.parse(
    """
    ###############################################################################
    ###   Variables
    ###############################################################################
    var Larm 2999.8
    var Lprc 12
    var Lsrc 12
    var Mtm  40
    var prmT 0.04835
    var itmT 0.01377
    var srmT 0.4
    var lmichx 5.3662
    var lmichy 5.244
    var f1 6270777
    var f2 9*f1
    var f3 4/3*f1
```

(continues on next page)

(continued from previous page)

```
    ###############################################################################
    ###   Input optics
    ###############################################################################
    l L0 125
    s l_mod1 L0.p1 eo1.p1
    mod eo1 f1 0.1 order=3
    s l_mod2 eo1.p2 eo2.p1
    mod eo2 f2 0.1 order=3
    s l_mod3 eo2.p2 eo3.p1
    mod eo3 f3 0.1 order=3
    s l_in eo3.p2 prm.p1 L=53
    #Power recycling mirror
    m prm T=prmT L=30u Rc=-1430
    s l_prc prm.p2 bs.p1 L=Lprc
    # Central beamsplitter
    bs bs T=0.5 L=0 alpha=45

    ###############################################################################
    ###   X arm
    ###############################################################################
    s lx bs.p3 itmx.p1 L=lmichx
    m itmx T=itmT L=37.5u phi=90 Rc=-1424.6
    s LX itmx.p2 etmx.p1 L=Larm
    m etmx T=5u L=37.5u phi=90 Rc=1695.0

    ###############################################################################
    ###   Y arm
    ###############################################################################
    s ly bs.p2 itmy.p1 L=lmichy
    m itmy T=itmT L=37.5u phi=0 Rc=-1424.5
    s LY itmy.p2 etmy.p1 L=Larm
    m etmy T=5u L=37.5u phi=0.0 Rc=1696.0

    ###############################################################################
    ###   Signal Recyling Cavity
    ###############################################################################
    s l_src bs.p4 srm.p1 L=Lsrc
    m srm T=srmT L=30u Rc=1430.0

    ###############################################################################
    ###   Cavities
    ###############################################################################
    # Arms
    cav cavX itmx.p2.o priority=3
    cav cavY itmy.p2.o priority=3
    # PRC
    cav cavPRW prm.p2.o via=itmx.p1.i priority=2
    cav cavPRN prm.p2.o via=itmy.p1.i priority=2
    # SRC
    cav cavSRW srm.p1.o via=itmx.p1.i priority=1
    cav cavSRN srm.p1.o via=itmy.p1.i priority=1
    """
)
```

```
<finesse.model.Model at 0x7e3b27225fd0>
```

Now we introduce the degrees of freedom (DOFs) that can be adjusted to reach lock, the error signals (measured via demodulation at the signal recycling and power recycling mirrors), and the locks relating the DOFs to the error signals. The demodulation phases of readouts are optimised before `RunLocks` is run using `OptimiseRFReadoutPhaseDC` (page 229). Since readout rd_pr_f2 is paired with two degrees of freedom, MICH and SRCL, we specify one of the

two to which to optimise the demodulation phase. In this example we choose SRCL, but in practice in this ideal model the optimal demodulation phases for MICH and SRCL are close together and the choice does affect the results. Gains are initially set at unity, but are set to appropriate values calculated with finite differences during `SetLockGains` in the proportional method. In the case of Newton's method, the entire sensing matrix will be calculated during `RunLocks` and running `SetLockGains` is not necessary.

```
kat.parse(
    """
    ###########################################################################
    ###   Degrees of Freedom, Readouts, and Locks
    ###########################################################################
    dof DARM etmx.dofs.z -1 etmy.dofs.z +1
    dof CARM etmx.dofs.z +1 etmy.dofs.z +1
    dof MICH itmx.dofs.z -1 etmx.dofs.z -1 itmy.dofs.z +1 etmy.dofs.z +1
    dof PRCL prm.dofs.z +1
    dof SRCL srm.dofs.z -1

    readout_rf rd_sr_f1 srm.p2.o f=eo1.f phase=0 output_detectors=True
    readout_rf rd_pr_f1 prm.p1.o f=eo1.f phase=0
    readout_rf rd_pr_f2 prm.p1.o f=eo2.f phase=0
    readout_rf rd_pr_f3 prm.p1.o f=eo3.f phase=0

    lock DARM_lock rd_sr_f1.outputs.I DARM.DC 1 1e-9
    lock CARM_lock rd_pr_f1.outputs.I CARM.DC 1 1e-9
    lock MICH_lock rd_pr_f2.outputs.Q MICH.DC 1 1e-9
    lock PRCL_lock rd_pr_f3.outputs.I PRCL.DC 1 1e-9
    lock SRCL_lock rd_pr_f2.outputs.I SRCL.DC 1 1e-9
    """
)
```

```
<finesse.model.Model at 0x7e3b27225fd0>
```

At this point, it would typically be necessary to perform 'pre-tuning' steps to move the model close enough to lock for the DC control loops to succeed. Here, however, because the geometry is relatively simple and higher-order modes are not considered, the model is very close to lock as defined. We can therefore kick the model away from lock by changing one or more DOFs and then observe whether the control loops succeed in bringing the model back to lock. Below, we explore simultaneous deviations in the common arm length CARM and the differential arm length DARM, varying each from their set points over the ranges $(-0.08°, 0.08°)$. Here, the units of degrees mean that we are deviating the microscopic path lengths of the optical components to reach a desired phase shift: e.g., a change of $0.01°$ for a laser wavelength of 1064 nm corresponds to a length change of 29.6 pm. For each deviation, we store the number of iterations of the locking procedure required to reach lock. If the locking does not converge within the maximum number of iterations specified (100), we return `None`. For more complex models, it is often required to set `max_iterations` to 1,000 or even 10,000. But for this simple model, if convergence is not achieved within 100 iterations, it will not be achieved within 10,000.

```
num_points = 21
DARM_limit, CARM_limit = 0.08, 0.08
DARM_values = np.linspace(-1 * DARM_limit, DARM_limit, num_points)
CARM_values = np.linspace(-1 * CARM_limit, CARM_limit, num_points)
optimise_readout_phase_pairs = [
    "DARM",
    "rd_sr_f1_I",
    "CARM",
    "rd_pr_f1_I",
    "PRCL",
    "rd_pr_f3_I",
    "SRCL",
    "rd_pr_f2_I",
]

# Array to store results.
```

(continues on next page)

(continued from previous page)

```
iter_list = np.zeros((2, num_points, num_points))
for ind_D, val_D in enumerate(DARM_values):
    for ind_C, val_C in enumerate(CARM_values):
        # Reset parameters to their set points
        kat_paired = kat.deepcopy()
        kat_newton = kat.deepcopy()
        # Deviate DARM and CARM from their setpoints
        kat_paired.DARM.DC = val_D
        kat_paired.CARM.DC = val_C
        kat_newton.DARM.DC = val_D
        kat_newton.CARM.DC = val_C

        # Try to lock with the paired method
        kat_paired.run(OptimiseRFReadoutPhaseDC(*optimise_readout_phase_pairs))
        kat_paired.run(SetLockGains())
        sol_paired = kat_paired.run(
            RunLocks(
                method="proportional",
                exception_on_fail=False,
                no_warning=True,
                max_iterations=100,
                display_progress=False,
            )
        )
        if sol_paired.iters > 99:
            iter_list[0, ind_D, ind_C] = None
        else:
            iter_list[0, ind_D, ind_C] = sol_paired.iters

        # Try to lock with Newton's method
        kat_newton.run(OptimiseRFReadoutPhaseDC(*optimise_readout_phase_pairs))
        sol_newton = kat_newton.run(
            RunLocks(
                method="newton",
                exception_on_fail=False,
                no_warning=True,
                max_iterations=100,
                display_progress=False,
            )
        )
        if sol_newton.iters > 99:
            iter_list[1, ind_D, ind_C] = None
        else:
            iter_list[1, ind_D, ind_C] = sol_newton.iters
```

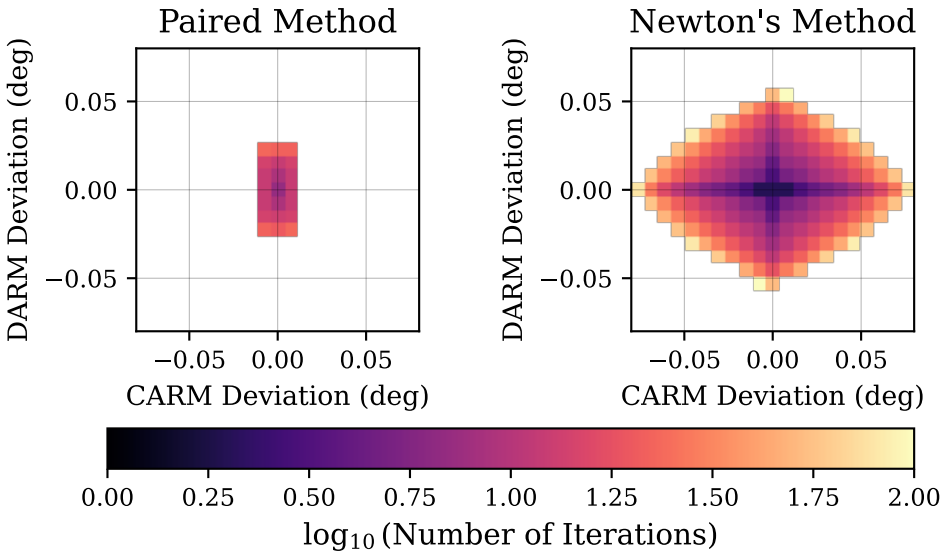
We then display the number of iterations required for each deviation. A white pixel indicates that the locking exceeded the maximum number of iterations.

```
fig, axs = plt.subplots(1, 2)
im1 = axs[0].imshow(
    np.log10(iter_list[0]),
    extent=[-1 * CARM_limit, CARM_limit, -1 * DARM_limit, DARM_limit],
    aspect=CARM_limit / DARM_limit,
    vmin=0,
    vmax=2,
)
im1 = axs[1].imshow(
    np.log10(iter_list[1]),
    extent=[-1 * CARM_limit, CARM_limit, -1 * DARM_limit, DARM_limit],
    aspect=CARM_limit / DARM_limit,
    vmin=0,
```

(continues on next page)

(continued from previous page)

```
    vmax=2,
)
axs[0].set_xlabel("CARM Deviation (deg)", fontsize=10)
axs[1].set_xlabel("CARM Deviation (deg)", fontsize=10)
axs[0].set_ylabel("DARM Deviation (deg)", fontsize=10)
axs[1].set_ylabel("DARM Deviation (deg)", fontsize=10)
axs[0].set_title("Paired Method", fontsize=12)
axs[1].set_title("Newton's Method")
fig.subplots_adjust(right=0.8, wspace=0.75)
cbar_ax = fig.add_axes([0.1, 0.1, 0.7, 0.4])
cbar_ax.axis("off")
cbar = fig.colorbar(
    im1, ax=cbar_ax, location="bottom", label=r"$\log_{10}$(Number of Iterations)"
)
```



We observe that Newton's method converges for a much larger fraction of this parameter space than the paired method. Moreover, for pixels in which the paired method converges, Newton's method requires far fewer iterations.

AC control loops

AC control loops are about shaping feedback and looking and control loop stability, etc.

6.3.5 Using degrees of freedom

- Inspecting (page 289)
- Tuning (page 289)
- Symbolic links & Locking (page 290)
  - Checking links (page 291)
  - Restoring links (page 291)
  - Locking (page 291)
- Running scans with degrees of freedom (page 292)

degree_of_freedom
dof

DegreeOfFreedom

The degree of freedom element represents degrees of freedom in your optical setup. It is used to control multiple parameters in your model in a coherent way.

Syntax

```
dof name *node_amplitude_pairs DC=0 lock_parameters=true
```

Required

`name`: Name of the degree of freedom.

`\*node_amplitude_pairs: tuple[str, float | Symbol]`: String referring to a parameter or a local degree of freedom, coupled with the amplitude with which this DOF will drive the parameter. The parameter value will be set to the product of this (symbolic) amplitude and the DOF.DC parameter. In case multiple DOFS drive the same parameter, their contributions will be summed.

Optional

`DC`: DC value for the degree of freedom, by default 0

`lock_parameters`: Whether to 'lock' the parameters driven by this degree of freedom. This prevents accidental overriding of parameter values that are meant to be controlled by a dof, by default True.

The `degree_of_freedom` (page 190) command can be useful when building control loops, but also for tuning interferometers and any other situation where you want to change multiple parameters of the model at the same time.

The typical degrees of freedom in an interferometer are [32]:

- DARM: Differential ARM length. The output channel of the detector. Generally operated with a small DC offset relative to the dark fringe conditions. $L_N - L_W$
- CARM: Common ARM length. Used to keep the arms on resonance. $\frac{L_N + L_W}{2}$
- MICH: Controls the short arms of the Michelson and determines the fringe at the dark port. $l_N - l_W$

Creating a basic version of this interferometer.

```
import finesse
import matplotlib.pyplot as plt
from finesse.analysis.actions import Xaxis, Minimize, Noxaxis

finesse.init_plotting()

base_model = finesse.Model()
base_model.parse(
    """
    laser l1 P=1
    bs bs1 R=0.5 T=0.5

    m WI R=0.9 T=0.1
    m NI R=0.9 T=0.1

    m WE R=1 T=0
    m NE R=1 T=0

    link(l1, bs1)
    link(bs1.p2, 1, WI.p1)
    link(bs1.p3, 1, NI.p1)

    link(WI.p2, 1, WE.p1)
    link(NI.p2, 1, NE.p1)

    pd out bs1.p4.o
```

(continues on next page)

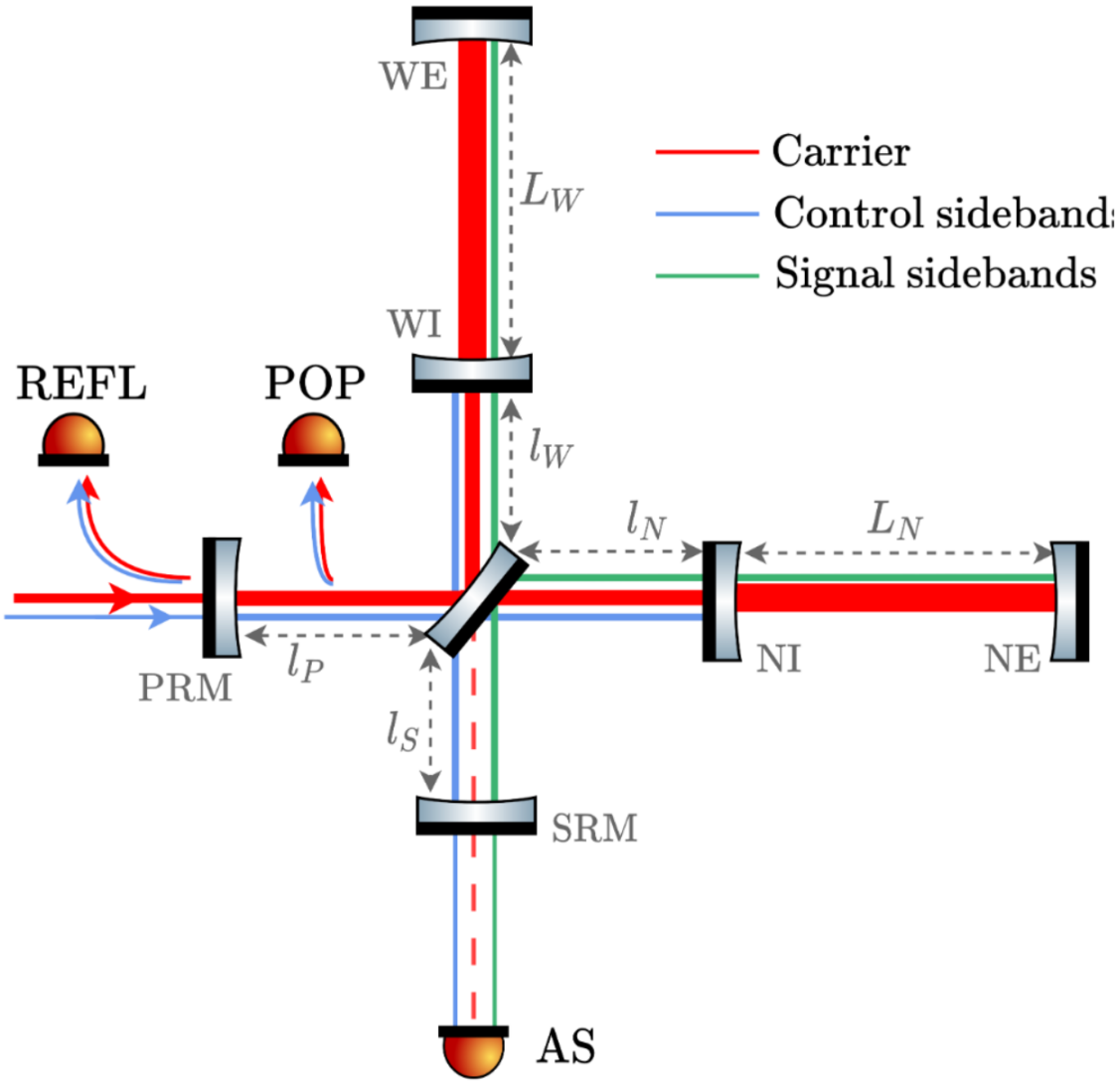


Fig. 6.9: Simplified layout of AdV+

(continued from previous page)

```
    """
)
```

```
<finesse.model.Model at 0x7f1cbb45a900>
```

Inspecting

You can inspect the available degrees of freedom for a model component by accessing their `.dofs` namespace:

```
base_model.WE.dofs
```

```
namespace(z=<'WE.dofs.z' @ 0x7f1cbb222dd0 (LocalDegreeOfFreedom) DC=0.0 degrees AC_IN=
↪<SignalNode WE.mech.z @ 0x7f1cbb0ac4d0> AC_OUT=<SignalNode WE.mech.z @ 0x7f1cbb0ac4d0>>,
          yaw=<'WE.dofs.yaw' @ 0x7f1cbb222e40 (LocalDegreeOfFreedom) DC=0.0 radians AC_IN=
↪<SignalNode WE.mech.yaw @ 0x7f1cbb0ac590> AC_OUT=<SignalNode WE.mech.yaw @ 0x7f1cbb0ac590>>
↪,
          pitch=<'WE.dofs.pitch' @ 0x7f1cbb222eb0 (LocalDegreeOfFreedom) DC=0.0 radians AC_
↪IN=<SignalNode WE.mech.pitch @ 0x7f1cbb0ac650> AC_OUT=<SignalNode WE.mech.pitch @␣
↪0x7f1cbb0ac650>>,
          F_z=<'WE.dofs.F_z' @ 0x7f1cbb222f20 (LocalDegreeOfFreedom) DC=0.0 degrees AC_IN=
↪<SignalNode WE.mech.F_z @ 0x7f1cbb0ac710> AC_OUT=<SignalNode WE.mech.z @ 0x7f1cbb0ac4d0>>,
          F_yaw=<'WE.dofs.F_yaw' @ 0x7f1cbb222f90 (LocalDegreeOfFreedom) DC=0.0 radians AC_
↪IN=<SignalNode WE.mech.F_yaw @ 0x7f1cbb0ac7d0> AC_OUT=<SignalNode WE.mech.yaw @␣
↪0x7f1cbb0ac590>>,
          F_pitch=<'WE.dofs.F_pitch' @ 0x7f1cbb223000 (LocalDegreeOfFreedom) DC=0.0 radians␣
↪AC_IN=<SignalNode WE.mech.F_pitch @ 0x7f1cbb0ac890> AC_OUT=<SignalNode WE.mech.pitch @␣
↪0x7f1cbb0ac650>>)
```

Every degree of freedom is coupled to a parameter of the component and has DC and AC attributes. For tuning the interferometer we are interested in the `z.DC` attribute.

```
base_model.WE.dofs.z.DC
```

```
<WE.phi=0.0 @ 0x7f1cbb21b970>
```

We can see it is directly coupled to the `phi` attribute of the mirror. Now we create a copy of this model with the standard degrees of freedom defined.

```
dof_model = base_model.deepcopy()
dof_model.parse(
    """
    dof DARM WE.dofs.z -1 NE.dofs.z +1 lock_parameters=True  # locks by default, just for␣
↪demonstration
    dof CARM WE.dofs.z +1 NE.dofs.z +1 lock_parameters=True  # locks by default, just for␣
↪demonstration
    dof MICH WI.dofs.z -1 NI.dofs.z +1 lock_parameters=True  # locks by default, just for␣
↪demonstration
    """
)
dof_model_2 = dof_model.deepcopy()
```

Tuning

We can reference the `MICH` degree of freedom to tune the dark port of the interferometer.

```
from finesse.analysis.actions import Minimize

out = dof_model.run(
    Minimize(dof_model.out, dof_model.MICH.DC, xatol=1e-15, fatol=1e-15)
```

(continues on next page)

(continued from previous page)

```
)
dof_model.run(Noxaxis())['out']
```

```
np.float64(1.2325951644078312e-32)
```

Inspecting the mirror tunings, we can see that they are referencing (page 238) the `MICH.DC` degree of freedom.

```
display(dof_model.WI.dofs.z.DC)
print(f"{dof_model.WI.dofs.z.DC.eval()=} degrees")
display(dof_model.NI.dofs.z.DC)
print(f"{dof_model.NI.dofs.z.DC.eval()=} degrees")
display(dof_model.MICH.DC)
```

```
<WI.phi=(-MICH.DC) @ 0x7f1cbb0bf100>
```

```
dof_model.WI.dofs.z.DC.eval()=-1.594842552110701 degrees
```

```
<NI.phi=MICH.DC @ 0x7f1cbb0bfa60>
```

```
dof_model.NI.dofs.z.DC.eval()=1.594842552110701 degrees
```

```
<MICH.DC=1.594842552110701 @ 0x7f1cb9cd74c0>
```

Symbolic links & Locking

Creating degrees of freedom creates symbolic links between the `DC` parameter of th degree of freedom and all the parameters that a degree of freedom is driving. You can check which parameters are being linked by using the `driven_parameters` property

```
for p in dof_model.MICH.driven_parameters:
    print(p.full_name, p.value)
```

```
WI.phi (-MICH.DC)
NI.phi MICH.DC
```

You see that the `MICH.DC` parameter is used in the symbolic expression for the `WI.phi` and `NI.phi` parameters. By default the degree of freedom element locks the parameters it controls, via its `lock_parameters` argument.

If you want to have the freedom to overwrite parameter values that are supposed to be controlled by a degree of freedom, you can manually unlock the parameter and overwrite its value.

```
broken_dof_model = dof_model.deepcopy()
broken_dof_model.WI.phi.unlock()
broken_dof_model.WI.phi = 10
for p in broken_dof_model.MICH.driven_parameters:
    print(p.full_name, p.value)
```

```
WI.phi 10.0
NI.phi MICH.DC
```

Alternatively, you can temporarily unlock a parameter with the `unlocked` context manager``.

```
broken_dof_model = dof_model.deepcopy()
with broken_dof_model.WI.phi.unlocked():
    broken_dof_model.WI.phi = 10
for p in broken_dof_model.MICH.driven_parameters:
    print(p.full_name, p.value)
```

```
WI.phi 10.0
NI.phi MICH.DC
```

This means that in the broken model, adjusting the `MICH` degree of freedom will now only move one of the mirrors, meaning the degree of freedom now longer works as intended. There are two ways to handle these mistakes. Firstly, we can use the `check_dof_symbolic_links` method on the `Model` object to identify broken degrees of freedom.

Checking links

```
broken_dof_model.check_dof_symbolic_links(verbose=True)
```

```
Parameter 'WI.phi' should have value '(-MICH.DC)' but has '10.0'
```

```
BrokenDOFLinkError:         (use finesse.tb() to see the full traceback)
The following parameters are not correctly linked to their degrees of freedom: 'WI.phi'␣
↪[MICH]
```

Restoring links

You can restore a model with broken degrees of freedom at any time with the `restore_dof_symbolic_links` method on the `Model` object. This also locks the parameters again (configurable via the `lock` argument).

```
broken_dof_model.restore_dof_symbolic_links(verbose=True)
broken_dof_model.check_dof_symbolic_links()  # will no longer raise an exception
broken_dof_model.WI.phi  # has original value
```

```
Restored parameter 'WI.phi'' value from '10.0' to '(-MICH.DC)', imposed by 'MICH'
```

```
<WI.phi=(-MICH.DC) @ 0x7f1cb8f44e50>
```

If you want to restore the symbolic links for just a single parameter, you can call the `restore_from_dof` method on the parameter itself.

```
broken_dof_model.WI.phi.unlock()
broken_dof_model.WI.phi = 10
broken_dof_model.WI.phi.restore_from_dof(verbose=True)
```

```
Restored parameter 'WI.phi'' value from '10.0' to '(-MICH.DC)', imposed by 'MICH'
```

Locking

You can lock any parameter in FINESSE by calling its `lock` method. This prevents the value being changed.

```
param_model = finesse.Model("l l1 P=1")
# normally allowed to change parameter values
param_model.l1.P = 2
# you can lock the parameter (and provide reason for locking)
param_model.l1.P.lock("demonstration")
# now changing the value will raise an exception
param_model.l1.P = 3
```

```
ParameterLocked:         (use finesse.tb() to see the full traceback)
Can not change the value of Parameter 'l1.P' because it is locked by 'demonstration'
You can unlock 'l1.P' using 'l1.P.unlock()'
```

Parameters can be unlocked by using their `unlock` method.

You can lock all the parameters being driven by a degree of freedom by using the `lock_driven_parameters` method. Note that this is the default behaviour when creating a new degree of freedom, unless you set `lock_parameters=False` at creation.

```
dof_model.MICH.lock_driven_parameters(verbose=True)
dof_model.WI.phi = 10
```

```
Locking parameters for DOF 'MICH'
Locking 'WI.phi'
Locking 'NI.phi'
```

```
ParameterLocked:         (use finesse.tb() to see the full traceback)
Can not change the value of Parameter 'WI.phi' because it is locked by 'MICH.lock_parameters'
You can unlock 'WI.phi' using 'WI.phi.unlock()'
```

You can also unlock all parameters:

```
dof_model_unlocked = dof_model.deepcopy()
dof_model_unlocked.MICH.unlock_driven_parameters()
dof_model_unlocked.WI.phi = 10
```

Or do the same for all degrees of freedom in a model by using the `lock_dof_parameters`.

```
dof_model.lock_dof_parameters(verbose=True)
dof_model.WI.phi = 10
```

```
Locking parameters for DOF 'DARM'
Locking 'WE.phi'
Locking 'NE.phi'
Locking parameters for DOF 'CARM'
Locking 'WE.phi'
Locking 'NE.phi'
Locking parameters for DOF 'MICH'
Locking 'WI.phi'
Locking 'NI.phi'
```

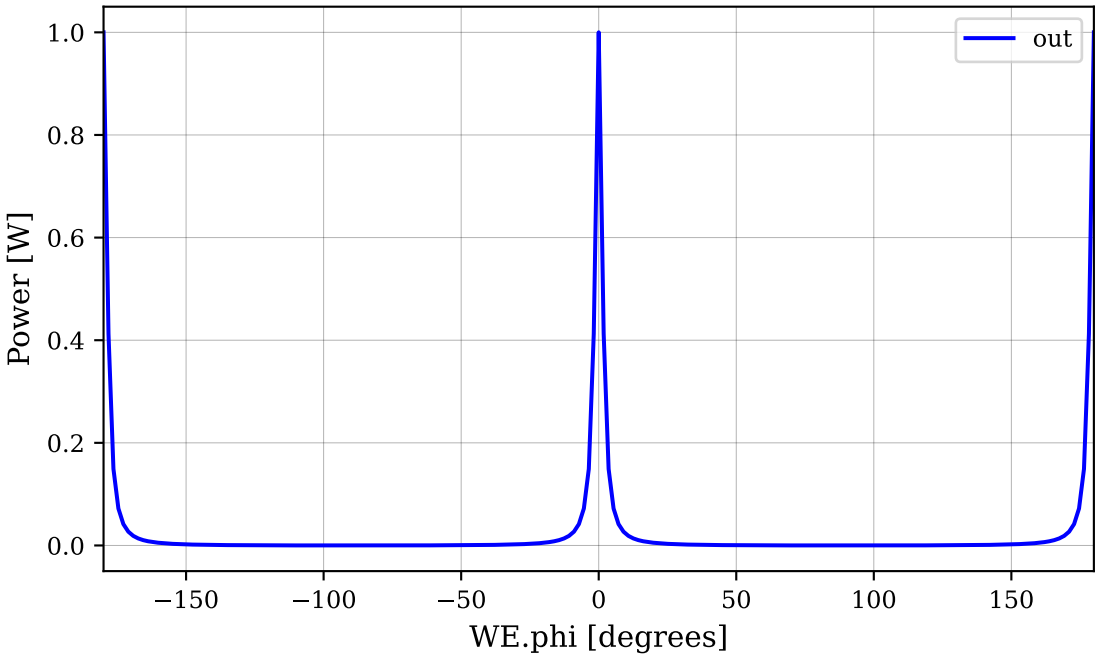
```
ParameterLocked:         (use finesse.tb() to see the full traceback)
Can not change the value of Parameter 'WI.phi' because it is locked by 'MICH.lock_parameters'
You can unlock 'WI.phi' using 'WI.phi.unlock()'
```

This will prevent you from accidentally breaking your degrees of freedom at any point in your notebook. The parameters can be unlocked again with the `unlock_dof_parameters` method.

Running scans with degrees of freedom

Note that in the base model without the degrees of freedom defined, you can still change the mirror tuning by referencing the `phi` attribute directly.

```
sol_phi = base_model.run(
    Xaxis(base_model.WE.phi, "lin", -180, 180, 200, relative=False)
)
sol_phi.plot()
```




```
{finesse.detectors.powerdetector.PowerDetector: <Figure size 576x355.968 with 1 Axes>,
 'out': <Figure size 576x355.968 with 1 Axes>}
```

But in the model with the degrees of freedom defined, you can no longer control the mirror tuning directly.

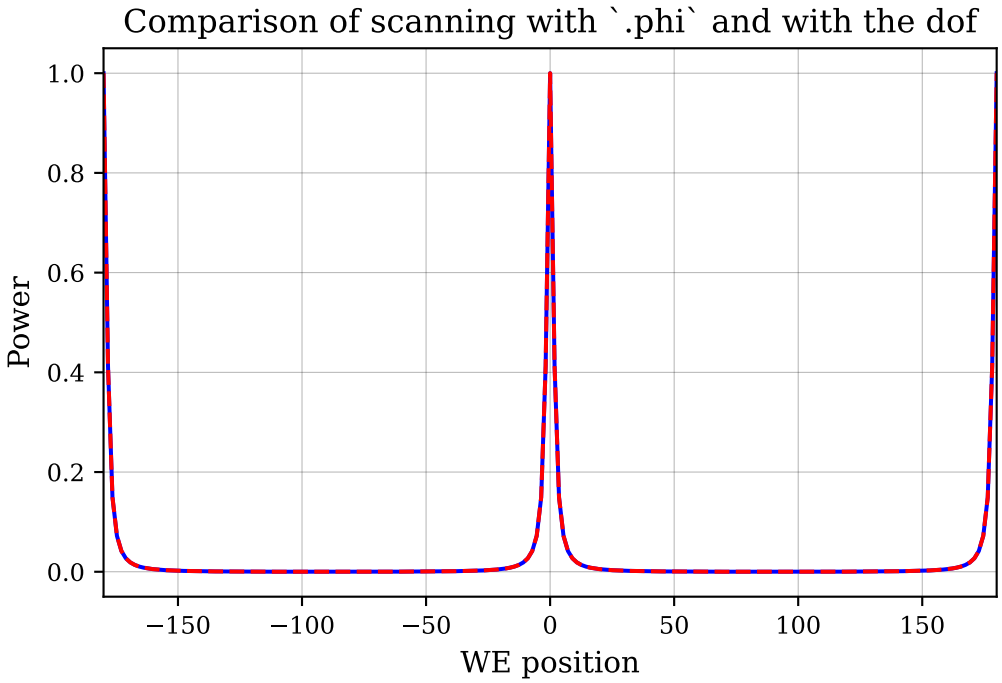
```
dof_model_2.run(Xaxis(dof_model.WE.phi, "lin", -180, 180, 200, relative=False))
```

```
ControlledByDOFException:          (use finesse.tb() to see the full traceback)
<WE.phi=(CARM.DC+(-DARM.DC)) @ 0x7f1cb9b1e110> is being controlled by (<'DARM' @␣
↪0x7f1cb9cccc30 (DegreeOfFreedom)>, <'CARM' @ 0x7f1cb9cccd60 (DegreeOfFreedom)>), so it␣
↪cannot be tuned directly.
```

It is still possible to move individual mirrors by defining additional degrees of freedom.

```
dof_model_2.parse(
    """
    dof WE_phi WE.dofs.z +1
    """
)
# reset the degree of freedom so both models are in identical state
dof_model_2.MICH.DC = 0
sol_WEDC = dof_model_2.run(
    Xaxis(dof_model_2.WE_phi.DC, "lin", -180, 180, 200, relative=False)
)

plt.plot(sol_phi.x1, sol_phi["out"], label="WE.phi")
plt.plot(sol_WEDC.x1, sol_WEDC["out"], label="WE_phi.DC", ls="--")
plt.xlabel("WE position")
plt.ylabel("Power")
plt.title("Comparison of scanning with `.phi` and with the dof")
plt.show()
```



6.3.6 Calculating transfer functions of effective systems

Sometimes it is useful to simulate a simpler effective thermo-optomechanical system that emulates a more complex physical one, but that is not strictly physical itself. In such cases it may be difficult to simulate the DC fields needed to describe this effective system. In FINESSE, the DC fields are known as the `carrier_solution`. Usually when calculating a `FrequencyResponse` (page 221), the DC fields are first computed and then used in the computation of the AC response. It is also possible to override this DC calculation by providing a `carrier_solution` where the DC fields have been manually set to their desired values. We illustrate this here with the effective three mirror coupled cavity, which describes some of the dynamics of modern and future gravitational wave detectors.

Warning

It is your responsibility to ensure that sensible DC fields are being set. FINESSE will happily return garbage if you haven't been careful when providing a `carrier_solution` to one of the `FrequencyResponse` (page 221) actions.

You can manually provide a `carrier_solution` to any of the four `FrequencyResponse` (page 221) actions, but we'll demonstrate it here only with `FrequencyResponse` (page 221).

The DARM coupled cavity

The optical topology used by all current and future detectors is known as the dual-recycled Fabry-Perot Michelson (DRFPMI) and consists of five optical cavities made up of (at an absolute minimum) seven core optics. For many purposes the dynamics of the DRFPMI can be described by *effective* three mirror coupled cavities. The differential arm motion (DARM) is equivalent to a single effective arm cavity formed by a single ITM and ETM with the addition of a signal extraction mirror (SEM) to form the signal extraction cavity (SEC). Similarly, the common arm motion (CARM) is equivalent to a single arm cavity with the addition of a power recycling mirror to form the power recycling cavity (PRC). Here we build the DARM coupled cavity equivalent to the differential arm motion of LIGO.

```
import numpy as np
import finesse
import finesse.components as fc
import finesse.analysis.actions as fa
from finesse.utilities import set_DC_fields
```

(continues on next page)

(continued from previous page)

```
from finesse.plotting import bode

finesse.init_plotting()

model = finesse.Model()
model.add(fc.Mirror("ETM", T=0, L=0, imaginary_transmission=False))
model.add(fc.Mirror("ITM", T=0.014, L=0, imaginary_transmission=False))
model.connect(model.ITM.p1, model.ETM.p1, 4e3)
model.add(fc.FreeMass("ITM_sus", model.ITM.mech, mass=40))
model.add(fc.FreeMass("ETM_sus", model.ETM.mech, mass=40))
model.add(fc.Mirror("SEM", T=0.325, L=0, imaginary_transmission=False))
model.connect(model.SEM.p1, model.ITM.p2, 55)
model.add(fc.BalancedHomodyneReadout("BHD", model.SEM.p2.o, phase=90))
model.modes("off")
```

By setting *imaginary_transmission=False*, we use the real transmission conventions for the mirrors so that the phases are intuitive. Thus, if we set the amplitude quadrature of the DC fields in the arm to be 0 deg, the `BalancedHomodyneReadout` (page 188) phase is set to 90 deg in order to sense a phase signal excited by ETM motion.

Manually setting the DC fields

The system we want is one in which the arm cavity has 750 kW circulating in the fundamental mode and no power anywhere else. This almost means that we need the four nodes of the cavity (ITM incident and reflected and ETM incident and reflected) to have fields of power 750 kW. Since the mirrors are not perfectly reflecting, a field reflected from a mirror is reduced by the amplitude reflectivity. We also have to be careful about phases. So we'll just extract the exact operators used to compute the real `carrier_solution` with an `Operator` (page 233) action so that we don't have to think about it.

```
Parm_W = 750e3
freq = 0  # carrier frequency, only 0 in this simple model
sol = model.run(fa.Series(
    fa.DCFields(name="DC"),
    fa.Operator(model.ETM.p1.i, model.ITM.p1.o, frequency=freq, name="op"),
))
DC = sol["DC"]
op = sol["op"]
DC.fields[:] = 0  # set the fields at every node to zero
```

`op` is an `OperatorSolution` which has all of the operators along the path from `ETM.p1.i` (the field incident on the front of the ETM) to `ITM.p1.o` (the field reflected from the front of the ITM) in its `matrices` attribute. You can see the entire path by

```
for idx, (fr, to) in enumerate(op.connections):
    print(f"operator {idx}: to {to} from {fr}")
```

```
operator 0: to ETM.p1.o from ETM.p1.i
operator 1: to ITM.p1.i from ETM.p1.o
operator 2: to ITM.p1.o from ITM.p1.i
```

These operators are especially simple in this example since we're not simulating HOMs, but this method is very useful in more complicated situations. `DC` is now an empty `DCFieldsSolution` ready to be manually populated by the desired DC fields. We could do this by hand like this

```
f_idx = list(DC.fields).index(freq)  # frequency index of the carrier
# set the power in the fundamental to 750 kW
Efr_i = np.zeros(len(DC.homs), dtype=complex)
Efr_i[0] = np.sqrt(Parm_W)
Ec = DC["ETM.p1.i", f_idx] = Efr_i
Ec = DC["ETM.p1.o", f_idx] = op.matrices[0] @ Ec
```

(continues on next page)

(continued from previous page)

```
Ec = DC["ITM.p1.i", f_idx] = op.matrices[1] @ Ec
Ec = DC["ITM.p1.o", f_idx] = op.matrices[2] @ Ec
```

But we can also use the `finesse.utilities.frequency_response.set_DC_fields()` helper function. This takes as arguments the initial field to be set at the first node of `op`, the frequency of the field to be set, the `DCFieldsSolution` to be populated, and the `OperatorSolution` containing the operators along the path.

```
DC[:] = 0
set_DC_fields(Efr_i, freq, DC, op)
```

You can use `finesse.utilities.frequency_response.set_DC_fields()` multiple times with different `OperatorSolution` to set the DC fields starting from multiple different nodes along different paths. You could also do more complicated things with this such as setting the power in the first eigenmode of a cavity with distortions and/or apertures by simply passing that eigenmode in instead of `Efr_i`.

Calculating the frequency response

We will compute the optomechanical plant for five cases to illustrate the dynamics of the DRFPMI using the simple DARM coupled cavity. Letting ϕ_s denote the extra one-way phase the carrier accumulates when propagating through the SEC, which is equivalent to `SEM.phi`, these cases are

- No SEM present: this is just a single Fabry-Perot cavity which is equivalent to a gravitational wave detector without an SEC, like the initial gravitational wave detectors. If the SEM is misaligned it will not form a cavity and it is effectively absent (except it will still add loss if there was any). This is accomplished in finesse by setting `SEM.misaligned=True`.
- Signal recycling (SR): when $\phi_s = 0$, the bandwidth is narrowed and the DC gain is increased relative to the single FP cavity.
- Resonant sideband extraction (RSE): when $\phi_s = \pi/2$, the bandwidth is broadened and the DC gain is reduced relative to the single FP cavity. This is how most gravitational wave detectors operate.
- Optical spring: when $\phi_s > \pi/2$, there is an anti-damped optical spring. This is how the low frequency Einstein telescope is planned to operate.
- Optical anti-spring: when $\phi_s < \pi/2$, there is a damped anti-spring.

Note that there is no standard convention and often ϕ_s is defined as `SEM.phi - \pi/2` in the literature and other codes so that all of the above cases shift by $\pi/2$.

We'll compute the same transfer function for each of these cases and so define a simple function `fresp` which returns the same frequency response giving it a different name so that it can be retrieved easily from the solution. Then we'll compute these five cases in a `Series` (page 207) using `Change` (page 235) between each to set the correct configuration. Since this will change the state of the model we do this in a `temporary_parameters` context manager so that the model reverts back to its initial state after the analysis.

```
F_Hz = np.geomspace(1, 10e3, 300)
fresp = lambda k: fa.FrequencyResponse(
    F_Hz, "ETM.mech.z", "BHD.DC", carrier_solution=DC, name=k,
)

with model.temporary_parameters():
    sol = model.run(fa.Series(
        fa.Change({"SEM.phi": 0}),
        fresp("SR"),
        fa.Change({"SEM.phi": 90}),
        fresp("RSE"),
        fa.Change({"SEM.phi": 98}),
        fresp("spring"),
        fa.Change({"SEM.phi": 82}),
        fresp("anti"),
        fa.Change({"SEM.misaligned": True}),
```

(continues on next page)

(continued from previous page)

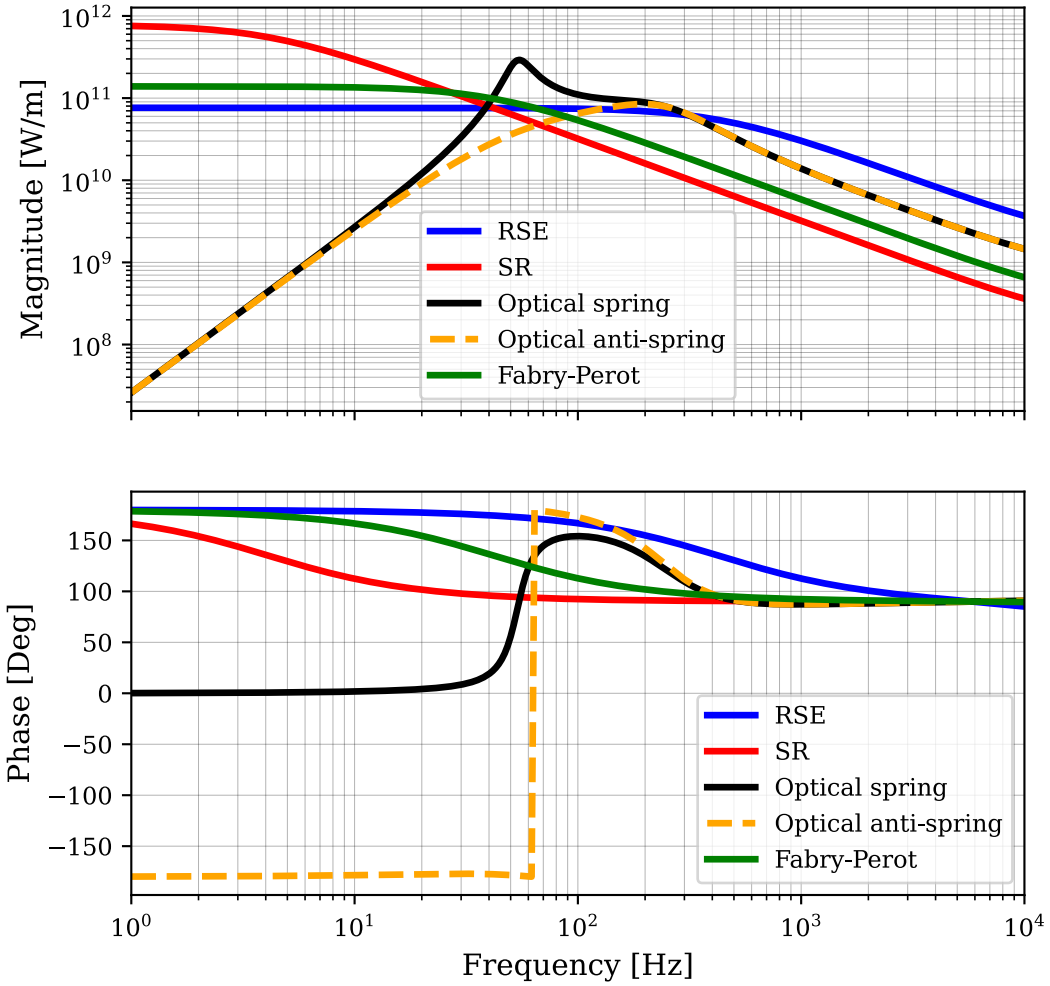
```
        fresp("arm"),
    ))
```

Note that detuning the arm is a situation where you would need to think carefully about manually setting the `carrier_solution`. There is now an extra phase introduced in the arm and so you would either need to retrieve the new operators with an `Operator` (page 233) action (the most robust) or multiply all of the fields by the appropriate phase. There are no DC fields in the SEC so we did not need to worry about that in this case where we only detuned the SEC.

The effective coupled cavity transfer functions need to be divided by $\sqrt{2}$ to account for the presence of the beamsplitter in the DRFPMI.

```
plt_fresp = lambda k: sol[k].out.squeeze() / np.sqrt(2)
lw = 2.5
axs = bode(F_Hz, plt_fresp("RSE"), db=False, label="RSE", lw=lw)
bode(F_Hz, plt_fresp("SR"), axs=axs, db=False, label="SR", lw=lw)
bode(
    F_Hz, plt_fresp("spring"), axs=axs, db=False, label="Optical spring", lw=lw,
)
bode(
    F_Hz, plt_fresp("anti"), axs=axs, db=False, label="Optical anti-spring",
    ls="--", lw=lw,
)
bode(F_Hz, plt_fresp("arm"), axs=axs, db=False, label="Fabry-Perot", lw=lw)
axs[0].set_ylabel("Magnitude [W/m]")
```

```
Text(0, 0.5, 'Magnitude [W/m]')
```



6.4 Higher-order Hermite-Gauss modes

In general, any solution $u(x, y, z)$ of the paraxial wave equation, can represent the transverse properties of a scalar electric field representing a beam-like electro-magnetic wave. Complete and countable sets of functions that are solutions to the paraxial wave equation are called *transverse electromagnetic modes (TEM)*. One such set is the Hermite Gauss modes denoted by HG_{nm}, that can be used to describe any solution of the paraxial wave equation u' by a linear superposition of Hermite–Gauss modes:

$$u'(x, y, z) = \sum_{n,m} a_{jnm} u_{nm}(x, y, z)$$

Which in turn allows us to describe any laserbeam using a sum of these modes:

$$E(t, x, y, z) = \sum_{j} \sum_{n,m} a_{jnm} u_{nm}(x, y, z) exp(i(\omega_j t - k_j z))$$

The "lowest-order" Hermite Gauss mode HG_{00} is usually called a *Gaussian Beam*. Such a beam profile (for a beam with a given wavelength λ) can be completely determined by two parameters: the size of the minimum spot size ω_0 (called the beam waist) and the position z_0 of the beam waist along the z-axis.

The Hermite-Gauss modes are usually given in their orthonormal form as:

$$u_{n,m}(x, y, z) = (2^{n+m-1} n! m! \pi)^{-1/2} \frac{1}{\omega(z)} \exp(i(n + m + 1)\psi(z))$$
$$\times H_n\left(\frac{\sqrt{2}x}{\omega(z)}\right) H_m\left(\frac{\sqrt{2}y}{\omega(z)}\right) \exp\left(-i\frac{k(x^2 + y^2)}{2R_C(z)} - \frac{x^2 + y^2}{\omega^2(z)}\right)$$

with n, m being the mode numbers (or mode indices). In this case n refers to the modes in the y-z plane (saggital) and m to the x-z plane (tangential). The following functions are used in the equation above:

- $H_n(x)$: Hermite polynomial of the order n (unnormalised),
- $\omega(z)$: beam radius or spot size,
- $R_C(z)$: radius of curvature of the phase front,
- $\psi(z)$: Gouy phase (see Accumulated Gouy Phase (page 308))

The Hermite-Gauss modes as given above are orthonormal and thus:

$$\int\int dxdy\ u_{nm}u^*_{n'm'} = \delta_{nn'}\delta_{mm'}$$

Therefore the power of a beam, being detected on a single-element photodetector (provided that the area of the detector is large with respect to the beam) can be computed as:

$$P = \sum_{n,m} a_{nm}a^*_{nm}$$

The x and y dependencies can be separated so that:

$$u_{nm}(x, y, z) = u_n(x, z)u_m(y, z)$$

For more information, see the [18].

6.4.1 Phase configuration settings

Warning

For physically correct results use the *phase 0* or *phase 2* options, which can be set with `model.phase_config(zero_k00=False, zero_tem00_gouy=False)` and `model.phase_config(zero_k00=False, zero_tem00_gouy=True)` respectively.

Note

Since version 3.0a28 the default phase setting is now `model.phase_config(zero_k00=False, zero_tem00_gouy=True)`.

The `finesse.model.Model.phase_config()` method can be used to switch on/off some simplifications when dealing with the phase of the light field. This is to aid in simulating systems more easily.

In previous versions of FINESSE this was done using the *phase* command. It was a bitflag setting that switched two options for running simple simulations quickly without having to set up a sensing and control scheme - which was laborious in previous versions of FINESSE. With the new actions available in FINESSE 3 you can more easily put optical systems at sensible operating points, therefore these phase options are less useful as models are not tuned by hand to be at a particular operating point.

These options were referred to in previous versions of FINESSE as:

- `phase 0`: No simplification. This means for example that the Gouy phase of a TEM00 is not zero and thus a space of arbitrary length is in general not resonant to the carrier field.
- `phase 1`: The phase of coupling coefficients is shifted such that the phase of k00 is 0. The phases of all coupling coefficients knm for one field coupling, for example a reflection at one side of a mirror, are changed by the same amount. To some effect that resembles the movement of the optical surface. However, since this is independently applied to all coupling coefficients of the surface (i.e. two reflections and two transmissions), this does not describe a physically realistic situation. In fact, it might invalidate energy conservation or produce other weird effects.

- `phase 2`: The phase accumulated in a `space` component is adjusted such that the phase of TEM00 is 0 (irrespective of the value of the Gouy phase). Effectively the "macroscopic length" of a space is no longer an integer multiple of the wavelength.
- `phase 3`: Both of the above (phase 1 and phase 2)

In FINESSE 3 these are now set using the `finesse.model.Model.phase_config()` method, accepting keyword arguments which are easier to remember than the bitflag input:

- `phase 3 == zero_k00=True, zero_tem00_gouy=True`
- `phase 2 == zero_k00=False, zero_tem00_gouy=True`
- `phase 1 == zero_k00=True, zero_tem00_gouy=False`
- `phase 0 == zero_k00=False, zero_tem00_gouy=False`

Users performing complex simulations should ensure they only use *zero_k00=False*. However, this will mean that values used to tune optic positions for common operating points will differ from what one would expect naively. For example, resonating a Fabry-Perot cavity for HG00 with `phase 3` will use mirrors with `phi=0`, however this is likely not the case with `phase 2` or `phase 0` where it becomes necessary to always ensure you use some locking routine or optimiser to find the correct operating point.

Removing the TEM00 Gouy phase: *phase 2*

Removing the Gouy phase using the *zero_tem00_gouy=True* does not have a detrimental effect on the accuracy of the simulations. Using this flag will ensure that the HG00 mode, when propagating along a space, will accumulate no Gouy phase. Physically this is like every space is tuned microscopically to remove the Gouy phase. The following example shows how the resonance condition of a cavity is shifted when the Gouy phase is not zeroed.

```
import finesse
import matplotlib.pyplot as plt

finesse.init_plotting()

model = finesse.script.parse(
    """
    laser l1
    m m1 R=0.99 T=0.01 Rc=100
    m m2 R=0.99 T=0.01 Rc=100
    link(l1, m1.p2, m1.p1, 80, m2.p1)
    cav cavity m1.p1.o
    pd CIRC m1.p1.o
    modes(even, maxtem=10)
    """
)

model.beam_trace()

q_mismatch = finesse.BeamParam.overlap_contour(model.l1.p1.o.q, 0.1, 0)
model.l1.p1.o.q = q_mismatch

action = 'xaxis(m2.phi, lin, -180, 180, 1000, True, name="xaxis")'

labels = [f"phase {i}" for i in range(4)]
lss = ["-", "--", "-", "--"]
model.phase_config(zero_k00=False, zero_tem00_gouy=False)
out0 = model.run(action)
model.phase_config(zero_k00=True, zero_tem00_gouy=False)
out1 = model.run(action)
model.phase_config(zero_k00=False, zero_tem00_gouy=True)
out2 = model.run(action)
model.phase_config(zero_k00=True, zero_tem00_gouy=True)
out3 = model.run(action)
```

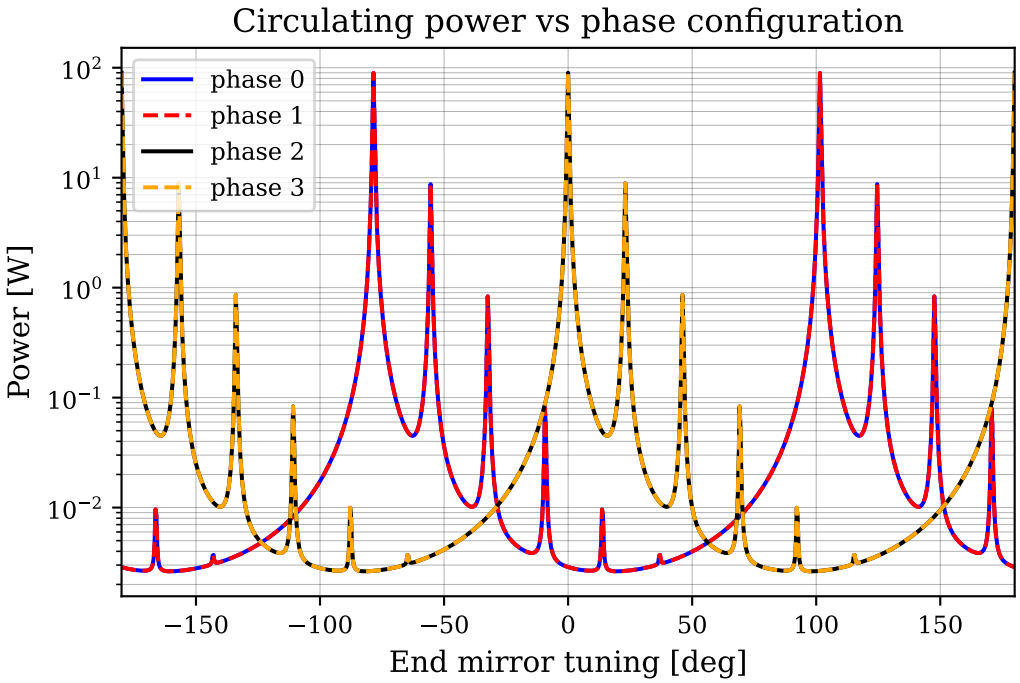
(continues on next page)

(continued from previous page)

```
for out, lbl, ls in zip([out0, out1, out2, out3], labels, lss):
    plt.semilogy(out.x[0], out["CIRC"], ls=ls, label=lbl)

plt.legend()
plt.xlabel("End mirror tuning [deg]")
plt.ylabel("Power [W]")
plt.title("Circulating power vs phase configuration")
```

```
Text(0.5, 1.0, 'Circulating power vs phase configuration')
```



Example of errors zeroing scattering matrix TEM00 coupling: *phase 1*

Below is an example where using *zero_k00=True* causes an unwanted error in the calculations. The example is a mode mismatched Fabry-Perot cavity. It demonstrates how using *zero_k00=True* returns an incorrect reflected field and total power in the simulation is no longer conserved. The *pseudo_lock_cavity* action is used to easily find the correct operating point in each case to compare the scenarios fairly.

```
model = finesse.script.parse(
    """
    laser l1
    m m1 R=0.99 T=0.01 Rc=100
    m m2 R=0.99 T=0.01 Rc=100
    link(l1, m1.p2, m1.p1, 80, m2.p1)
    cav cavity m1.p1.o
    pd REFL m1.p2.o
    pd CIRC m1.p1.o
    pd TRAN m2.p2.o
    modes(even, maxtem=10)
    """
)

model.beam_trace()

q_mismatch = finesse.BeamParam.overlap_contour(model.l1.p1.o.q, 0.1, 0)
```

(continues on next page)

(continued from previous page)

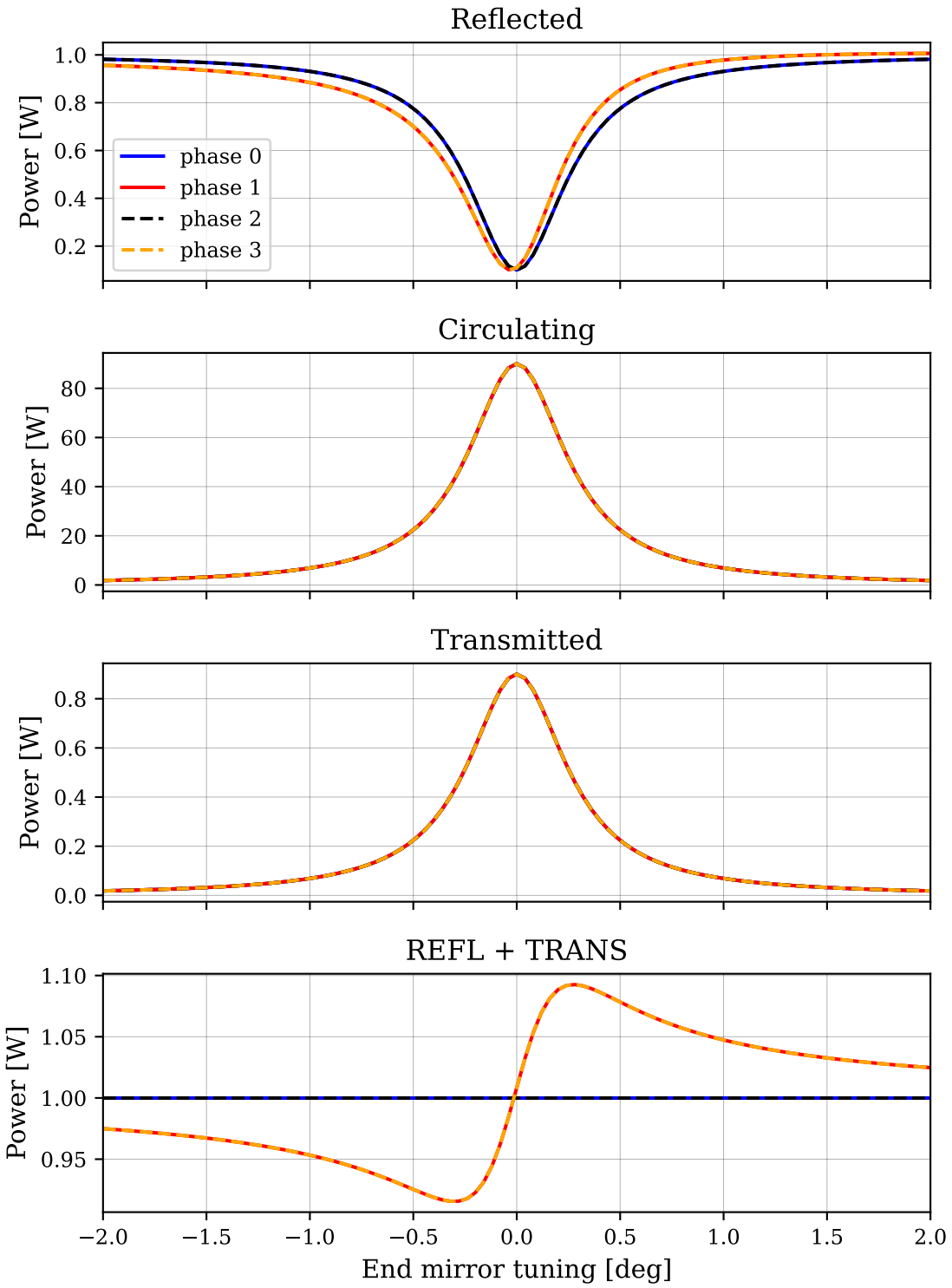
```
model.l1.p1.o.q = q_mismatch

action = 'series(pseudo_lock_cavity(cavity), xaxis(m2.phi, lin, -2, 2, 100, True, name="xaxis
↪"))'

labels = [f"phase {i}" for i in range(4)]
lss = ["-", "-", "--", "--"]
model.phase_config(zero_k00=False, zero_tem00_gouy=False)
out0 = model.run(action)
model.phase_config(zero_k00=True, zero_tem00_gouy=False)
out1 = model.run(action)
model.phase_config(zero_k00=False, zero_tem00_gouy=True)
out2 = model.run(action)
model.phase_config(zero_k00=True, zero_tem00_gouy=True)
out3 = model.run(action)


fig, axs = plt.subplots(4, 1, sharex=True, figsize=(6, 8))
for out, lbl, ls in zip([out0, out1, out2, out3], labels, lss):
    plt.sca(axs[0])
    plt.plot(out["xaxis"].x[0], out["xaxis"]["REFL"], label=lbl, ls=ls)
    plt.sca(axs[1])
    plt.plot(out["xaxis"].x[0], out["xaxis"]["CIRC"], ls=ls)
    plt.sca(axs[2])
    plt.plot(out["xaxis"].x[0], out["xaxis"]["TRAN"], ls=ls)
    plt.sca(axs[3])
    total = out["xaxis"]["REFL"] + out["xaxis"]["TRAN"]
    plt.plot(out["xaxis"].x[0], total, ls=ls)


plt.sca(axs[0])
plt.legend()
plt.title("Reflected")
plt.ylabel("Power [W]")
plt.sca(axs[1])
plt.title("Circulating")
plt.ylabel("Power [W]")
plt.sca(axs[2])
plt.title("Transmitted")
plt.ylabel("Power [W]")
plt.sca(axs[3])
plt.title("REFL + TRANS")
plt.ylabel("Power [W]")
plt.xlabel("End mirror tuning [deg]")
plt.tight_layout()
```



6.4.2 Transformation of the beam parameter

A beam parameter is, in general, transformed when a beam passes from one medium into another by interacting with some optical surface or propagated over any distance. We can use the ABCD matrix formalism [17] to treat these transformations - giving us the mechanics to describe how higher order modes couple between interfaces.

Consider Fig. 6.10, here a beam described by the beam parameter q_1 (propagating through a medium with refractive index n_1) is incident upon some interaction surface S. Through interacting with this "surface" (which could also just be a length of free space), the beam is transformed such that it is described by q_2 in the second medium (which has a refractive index n_2). The equation describing how this beam parameter is transformed through the interaction is the

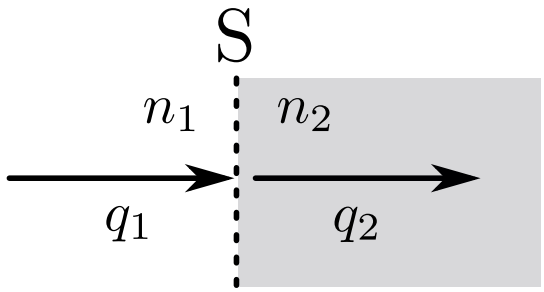


Fig. 6.10: Propagation of a beam via an interaction surface S, with a change in beam parameter across the different media.

Kogelnik transformation,

$$q = n_2 \frac{A\frac{q_1}{n_1} + B}{C\frac{q_1}{n_1} + D},$$

where A, B, C, D are the corresponding elements of the *ABCD* matrix associated with the interaction surface S. Equations detailing the ABCD matrix forms for the components in FINESSE are included in the API documentation links below. Also see `finesse.gaussian.transform_beam_param()` for the function which performs the above transformation.

Note

FINESSE uses ABCD matrices that have a determinant of 1, as defined in [17]. This effectively removes n_1/n_2 terms from the matrices themselves.

`finesse.components.beamsplitter.Beamsplitter.ABCD`(...)	Returns the ABCD matrix of the beam splitter for the specified coupling.
`finesse.components.lens.Lens.ABCD`(from_node, ...)	Returns the ABCD matrix of the lens for the specified coupling.
`finesse.components.mirror.Mirror.ABCD`(...[, ...])	Returns the ABCD matrix of the mirror for the specified coupling.
`finesse.components.space.Space.ABCD`(...[, ...])	Returns the ABCD matrix of the space for the specified coupling.

6.4.3 Cavity eigenmodes

A cavity eigenmode is defined as the optical field whose spatial properties are such that the field after one round-trip through the cavity will be exactly the same as the injected field [18]. In the case of resonators with spherical mirrors, the eigenmode will be a Gaussian mode, defined by the Gaussian beam parameter q_{cav}. For a generic cavity (an arbitrary number of spherical mirrors or lenses) a round-trip ABCD matrix M_{rt} can be defined and used to compute the cavity's eigenmode.

The change in the q parameter after one round-trip through a cavity is given by:

$$\frac{Aq_1 + B}{Cq_1 + D} = q_2 = q_1$$

where A, B, C and D are the elements of a matrix M_{rt}. If $q_1 = q_2$ then the spatial profile of the beam is recreated after each round-trip and we have identified the cavity eigenmode. We can compute the parameter $q_{\text{cav}} \equiv q_1 = q_2$ by solving:

$$Cq_{\text{cav}}^2 + (D - A)q_{\text{cav}} - B = 0,$$

An example of this is shown in Fig. 6.11 where the round trip matrix is given at the top of the figure. From this, we can compute the A, B, C and D coefficients for the round-trip matrix to solve the eigenmode equation above. This quadratic equation generally has two solutions, one being the complex conjugate of the other.

When the polynomial above has a suitable solution the optical resonator is said to be "stable". The stability requirement can be formulated using the Gaussian beam parameter: a cavity is stable only when the cavity's eigenmode, q_{cav}, has a real waist size. The waist of the beam will be a real number whenever q_{cav} has a positive non-zero imaginary part as this defines the Rayleigh range of the beam and therefore the beam waist, $\Im(q_{\text{cav}}) = \pi w_0^2/\lambda$. A complex q_{cav} is ensured if the determinant of the cavity eigenmode equation is negative.

This requirement can formulated in a compact way by defining the parameter m as:

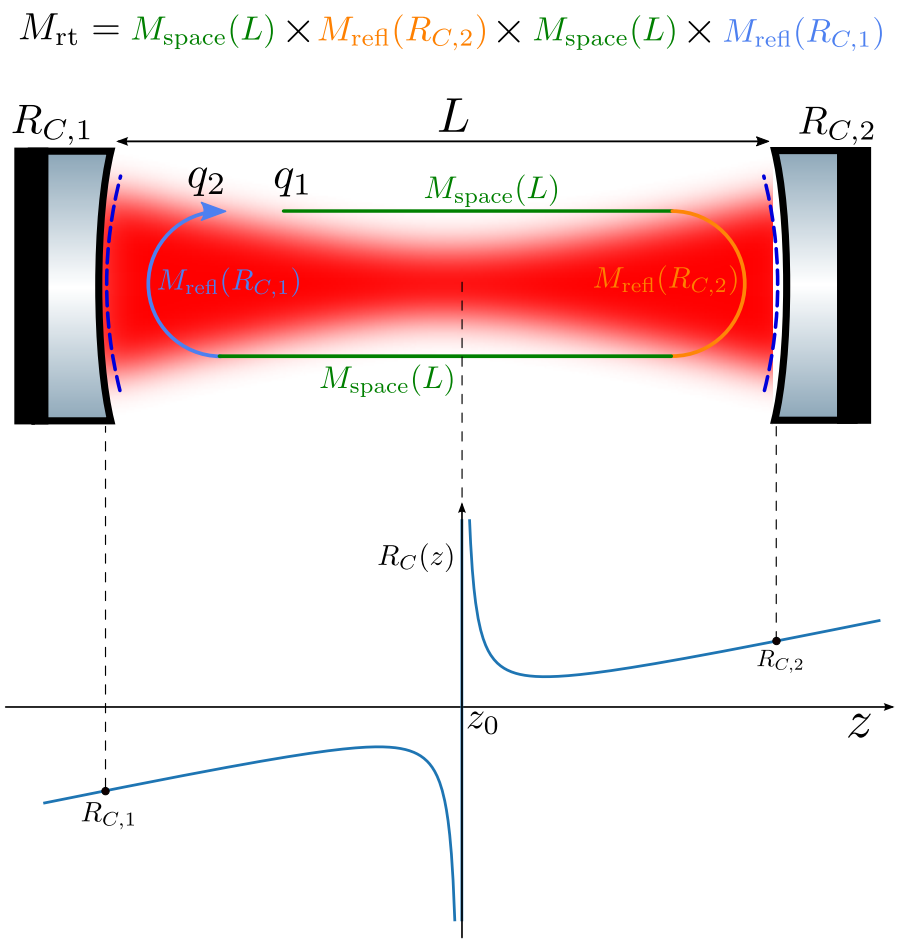


Fig. 6.11: Cavity round trip ABCD matrix for a Fabry-Perot cavity, with corresponding wavefront curvature.

The stability of simple cavities are often described using *g-factors*. These factors are simply rescaled versions of the more generic m value:

$$g \equiv \frac{m+1}{2} = \frac{A+D+2}{4},$$

Note

FINESSE uses g-factors for stability attributes in the `cavity` (page 195) class. Individual g-factors for both the tangential and sagittal planes are provided by this class.

For the cavity to be stable the g-factor must fulfil the requirement:

$$0 \leq g \leq 1$$

The closer g is to 0 or 1, the smaller the tolerances are for any change in the geometry before the cavity becomes unstable.

For a simple two-mirror cavity, such as the one given in Fig. 6.11, its g-factor is

$$\begin{aligned} g_1 &= 1 - \tfrac{L}{R_{c,1}}, \\ g_2 &= 1 - \tfrac{L}{R_{c,2}}, \\ g &= g_1 g_2. \end{aligned}$$

Where $g_{1,2}$ are the individual g-factors of the cavity mirrors and g is the g-factor of the entire cavity.

6.4.4 Tracing the beam

In general the Gaussian beam parameter of a TEM_{00} mode is changed at every optical surface (see Transformation of the beam parameter (page 303)). In other words, for each location inside the interferometer where field amplitudes are to be computed a certain beam parameter has to be set for the simulation.

A possible method to find reasonable beam parameters for every location in the interferometer (every node in FINESSE) is to first set only some specific beam parameters and then derive the remaining beam parameters from these initial ones: usually it is sensible to assume that the beam at the input can be properly described by the (hopefully known) beam parameter of the laser's output mode. In addition, in most cavities the light fields can be described safely by using cavity eigenmodes.

Algorithm details

6.4.5 Hermite-Gauss modes and coupling

FINESSE uses Hermite-Gauss modes to take the geometrical shape of the light field into account. One method of treating the transverse beam geometry is to describe the spatial properties as a sum of "spatial components" or "spatial modes" so that the electric field can be written as a sum of the different frequency components and of the different spatial modes. Of course, the concept of modes is directly related to the use of a sort of oscillator, in this case the optical cavity.

The coupling of a mode refers to how a spatial mode in one basis is represented in another; e.g. which sum of modes in the cavity basis q_2 produces the HG_{00} mode in the q_1 basis. Hermite–Gauss modes are coupled whenever a beam is not matched or aligned to a cavity or beam segment. This coupling is sometimes referred to as scattering into higher-order modes because in most cases the laser beam is a considered as a pure HG_{00} mode and any mode coupling would transfer power from the fundamental into higher-order modes. However, in general every mode with non-zero power will transfer energy into other modes whenever mismatch or misalignment occur, and this effect also includes the transfer from higher orders into a low order.

To compute the amount of coupling the beam must be projected into the base system of the cavity or beam segment it is being injected into. This is always possible, provided that the paraxial approximation holds, because each set of Hermite–Gauss modes, defined by the beam parameter at a position z, forms a complete set. Such a change of the

basis system results in a different distribution of light power in the new Hermite–Gauss modes and can be expressed by coupling coefficients that yield the change in the light amplitude and phase with respect to mode number.

Let us assume that a beam described by the beam parameter q_1 is injected into a segment described by the parameter q_2. Let the optical axis of the beam be misaligned: the coordinate system of the beam is given by (x, y, z) and the beam travels along the z-axis. The beam segment is parallel to the z'-axis and the coordinate system (x', y', z') is given by rotating the (x, y, z) system around the y-axis by the *misalignment angle* γ. The amplitude of a particular mode TEM_{nm} in the beam segment is then defined as:

$$u_{nm}(x, y;\, q_2) \exp\left(i(\omega t - kz)\right) = \sum_{n',m'} k_{n,m,n',m'} u_{n'm'}(x, y;\, q_1) \exp\left(i(\omega t - kz')\right),$$

where $u_{n'm'}(x, y;\, q_1)$ are the HG modes used to describe the injected beam, $u_{nm}(x, y;\, q_2)$ are the "new" modes that are used to describe the light in the beam segment and $k_{n,m,n',m'}$ is the *coupling coefficient* from each $\mathrm{TEM}_{n'm'}$ into TEM_{nm}.

Note that including the plane wave phase propagation within the definition of coupling coefficients is important because it results in coupling coefficients that are independent of the position on the optical axis for which the coupling coefficients are computed.

Solving the overlap integral

Using the fact that the HG modes u_{nm} are orthonormal, we can compute the coupling coefficients by the overlap integral [33]:

$$k_{n,m,n',m'} = \exp\left(i2kz' \sin^2\left(\frac{\gamma}{2}\right)\right) \iint dx'dy'\; u_{n'm'} \exp\left(ikx' \sin\gamma\right) u^*_{nm}.$$

Since the Hermite–Gauss modes can be separated with respect to x and y, the coupling coefficients can also be split into $k_{nmn'm'} = k_{nn'}k_{mm'}$. These equations are very useful in the paraxial approximation as the coupling coefficients decrease with large mode numbers. In order to be described as paraxial, the angle γ must not be larger than the diffraction angle. In addition, to obtain correct results with a finite number of modes the beam parameters q_1 and q_2 must not differ too much.

The integral can be computed directly using numerical integration methods. However, this can potentially be computationally very expensive depending on how difficult the integrand is to evaluate and complex it is. The following part of this section is based on the work of Bayer-Helms [33] and provides an analytic solution to the integral. In [33] the above integral is partly solved and the coupling coefficients are given by multiple sums as functions of γ and the mode mismatch parameter K, which is defined by

$$K = \frac{1}{2}(K_0 + iK_2),$$

where $K_0 = (z_R - z'_R)/z'_R$ and $K_2 = ((z - z_0) - (z' - z'_0))/z'_R$. This can also be written using $q = iz_\mathrm{R} + z - z_0$, as

$$K = \frac{i(q - q')^*}{2\Im q'}.$$

The coupling coefficients for misalignment and mismatch (but no lateral displacement) can then be written as

$$k_{nn'} = (-1)^{n'} E^{(x)} (n!n'!)^{1/2} (1 + K_0)^{n/2+1/4} (1 + K^*)^{-(n+n'+1)/2} \left\{S_g - S_u\right\},$$

where

$$S_g = \sum_{\mu=0}^{[n/2]} \sum_{\mu'=0}^{[n'/2]} \frac{(-1)^\mu \bar{X}^{n-2\mu} X^{n'-2\mu'}}{(n-2\mu)!(n'-2\mu')!} \sum_{\sigma=0}^{\min(\mu,\mu')} \frac{(-1)^\sigma \bar{F}^{\mu-\sigma} F^{\mu'-\sigma}}{(2\sigma)!(\mu-\sigma)!(\mu'-\sigma)!},$$

$$S_u = \sum_{\mu=0}^{[(n-1)/2]} \sum_{\mu'=0}^{[(n'-1)/2]} \frac{(-1)^\mu \bar{X}^{n-2\mu-1} X^{n'-2\mu'-1}}{(n-2\mu-1)!(n'-2\mu'-1)!} \sum_{\sigma=0}^{\min(\mu,\mu')} \frac{(-1)^\sigma \bar{F}^{\mu-\sigma} F^{\mu'-\sigma}}{(2\sigma+1)!(\mu-\sigma)!(\mu'-\sigma)!}.$$

The corresponding formula for $k_{mm'}$ can be obtained by replacing the following parameters: $n \to m$, $n' \to m'$, $X, \bar{X} \to 0$ and $E^{(x)} \to 1$ (see below). The notation $[n/2]$ means

$$\left[\frac{m}{2}\right] = \begin{cases} m/2 & \text{if } m \text{ is even,} \\ (m-1)/2 & \text{if } m \text{ is odd.} \end{cases}$$

The other abbreviations used in the above definition are

$$\bar{X} = (iz_\mathrm{R}' - z')\sin(\gamma)/(\sqrt{1+K^*}w_0),$$
$$X = (iz_\mathrm{R} + z')\sin(\gamma)/(\sqrt{1+K^*}w_0),$$
$$F = K/(2(1+K_0)),$$
$$\bar{F} = K^*/2,$$
$$E^{(x)} = \exp\left(-\frac{X\bar{X}}{2}\right).$$

Scattering matrices

Finesse computes and stores *scattering matrices* to describe how spatial modes couple with respect to each other between beam parameter bases. A scattering matrix is simply a square matrix of coupling coefficients where each coefficient represents how one mode couples into another - i.e. the scaling between the modes in terms of both the amplitude and phase of the field.

Fig. 6.12 shows the four scattering matrices present at a mirror M. For a mode-matched and aligned system, each of these matrices are just identity matrices.

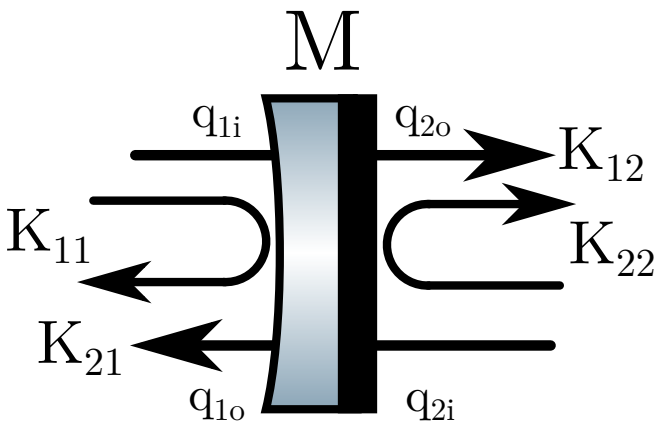


Fig. 6.12: Scattering matrices for a mirror. K11, K22 are the reflection coupling matrices whilst K12 and K21 are the transmission coupling matrices.

To compute the overall field couplings at the mirror M, Finesse multiplies the corresponding element of the mirrors' local coupling matrix by the associated scattering matrix. For example, to compute the coupling of the reflected field from the first surface, m_{11}, the equation,

$$m_{11} = rK_{11}\exp(i\varphi),$$

is applied, where $r = \sqrt{R}$ with R as the refectivity of the mirror and φ is the phase.

API for coupling coefficient calculations

Functions for computing and modifying scattering matrices can be found in this submodule:

`finesse.knm`	Library of coupling coefficient data structures and calculations.

6.4.6 Accumulated Gouy phase

Every electric field that propagates through free space of length d is delayed by a phase

$$\exp(-ikd) = \exp(-i\psi_{\text{planewave}}).$$

To help us distinguish it from other forms of phase we will call $\psi_{\text{planewave}}$ the planewave phase.

Transverse electric fields with finite extent accumulate an additional overall phase based on their transverse geometry. The manifestation of this geometric phase in Hermite-Gaussian modes is referred to as the Gouy phase. It is common to associate the Gouy phase with the Gaussian q-paramter in the following way

$$\psi_{nm} = \psi_n \psi_m$$

where

$$\psi_n = (n + 1/2) \arctan\left(\frac{\text{Real}[q]}{\text{Imag}[q]}\right)$$

which simplifies (using Mathematica) to

$$\exp(i\psi_n) = \left(\frac{iq^*}{|q|}\right)^{(n+1/2)}$$

In the case of just free space propagation the two preceeding equations produce the correct phase for all Hermite-Gaussian modes. However this is not true for propagation through a thin lens or reflecting off of a curved mirror. In both cases encoding the Gouy phase with the q-parameter leads to a discontinuous change in phase when comparing the beam before and after the lens or mirror. This is because Gouy phase is an accumulated effect that cannot be computed from an instantaneous description of a beam such as the q-parameter. This can be treated by introducing a convention of only accumulating Gouy phase across a space (As FINESSE does) and neglecting any Gouy phase across a mirror or an ideal lens as they are infinitely thin surfaces.

There are several ways to compute the accumulated Gouy phase through an ABCD optical system. One way is to use the following formula

$$\exp(i\psi_n) = \left(\frac{A + B/q^*}{|A + B/q^*|}\right)^{(n+1/2)} \tag{6.3}$$

where A and B are the ABCD parameters for a particular propagation and q is the q-parameter taken just before the propagation. This formula produces the correct accumulated Gouy phase for all basic components (spaces, lenses, mirrors). It should be noted that using the formula with a single ABCD matrix representing an overall optical system with many lenses and spaces may be off by a minus sign (particularly if the system is astigmatic). This can be avoided by only using the accumulated Gouy phase formula for simple optical components and multiplying all ψ_n's to get the overall accumulated Gouy phase of an optical system.

6.4.7 General optical cavity round trip Gouy phase

Computing the round trip Gouy phase of an optical cavity is a common task. This starts by computing the round-trip ABCD matrix in both the tangential and sagittal planes. Each plane may have a different accumulated Gouy phase due to any astigmatic effects, such as non-normal angles of incidence on mirrors. With the round-trip ABCD elements known the accumulated Gouy phase for a cavity in a particular plane is given by

$$2 \arccos\left(\text{sign}(B)\sqrt{\frac{A + D + 2}{4}}\right) \tag{6.4}$$

Equations (6.3) and (6.4) provide the same result if the correct ABCD and q-parameter are used. In this case the q-parameter can be chosen anywhere in the cavity and the source of the round-trip ABCD matrix should be at the same point the q-parameter is chosen.

6.4.8 Parity transformations on reflection

Mirrors and beamsplitters have an additional level of complexity when compared to lenses since they also appear to flip the beam left to right on reflection. In practice this is due to the mirror rotating the coordinate system of the incident beam 180 degrees in the incidence plane.

If we consider a perfectly reflective flat mirror then the reflected and incident electric field are related by the following coordinate transformation

$$\begin{aligned} E_{\text{reflected}}(x, y) &= \mathcal{P}[E_{\text{incident}}] \\ &= E_{\text{incident}}(-x, y) \end{aligned}$$

This particular operation is known as a parity transformation $\mathcal{P}$. If the electric field is composed of Hermite-Gauss modes then the parity operator has a particularly simple form

$$\mathcal{P}[u_{nm}] = \begin{cases} u_{nm} & \text{if } n \text{ even} \\ -u_{nm} & \text{if } n \text{ odd} \end{cases}$$

The parity operator can be implemented by directly applying the appropriate minus signs to the mode coefficients on reflection. A more elegant solution is to use the fact that a parity transformation is representable with an ABCD matrix and to use Eq. (6.3) to compute the appropriate minus signs. The relevant ABCD matrices that implement the parity operator $\mathcal{P}$ are

$$\mathbf{M}_{\mathcal{P},x} = \begin{bmatrix} -1 & 0 \\ 0 & -1 \end{bmatrix} \qquad \mathbf{M}_{\mathcal{P},y} = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}$$

Alternatively we can consider what the modified ABCD matrices are for reflecting off of a curved mirror

$$\mathbf{M}_x = \begin{bmatrix} -1 & 0 \\ 2/R_{c,x} & -1 \end{bmatrix} \qquad \mathbf{M}_y = \begin{bmatrix} 1 & 0 \\ -2/R_{c,y} & 1 \end{bmatrix}.$$

One may wonder if encoding the parity operator with an ABCD matrix could somehow introduce changes to the standard q-parameter propagation formulas. $\mathbf{M}_{\mathcal{P},y}$ doesn't change anything because it's the identity matrix. For $\mathbf{M}_{\mathcal{P},x}$ one can use the standard q-parameter propagation formula

$$\begin{aligned} q_2 &= \frac{Aq_1 + B}{Cq_1 + D} \\ &= \frac{-1q_1}{-1} \\ &= q_1 \end{aligned}$$

and so $\mathbf{M}_{\mathcal{P},x}$ has no effect on q-parameter propagation. Higher order scattering matrices also remain uneffected since encoding the parity transformation with ABCD matrices has no effect on q-parameters.

So now if we calculate the accumulated Gouy phase using (6.3) we get the following

$$\begin{aligned} \exp(i\psi_n) &= \left(\frac{A + B/q^*}{|A + B/q^*|}\right)^{(n+1/2)} \\ &= \left(\frac{-1}{|-1|}\right)^{(n+1/2)} \\ &= (-1)^{(n+1/2)} \\ &= (-1)^{1/2}(-1)^n \\ &= i(-1)^n \end{aligned}$$

which produces a minus sign difference between odd and even modes as desired.

6.4.9 ABCD definitions in FINESSE: extra minus sign

This parity sign is important to include. FINESSE includes the parity sign in the ABCD definitions directly. This means you can directly work with the ABCD matrices defined in `finesse.tracing.abcd` and get correct results.

Due to this, the tangential *_t* reflection ABCD methods in `finesse.tracing.abcd` will include an extra minus sign compared to canonical ABCDs you might see in textbooks or webpages elsewhere.

6.4.10 Segmented photodiodes

Segemented photodiodes are multi-element photodiodes that can resolve both frequency and spatial beats in an optical beam. The most common segmented photodiode is a quadrant photodiode, known as a QPD. This splits a photodiode into four segments, the signal generated by each segments can then be summed or subtracted from one another to measure if a beam is offset horizontally or vertically from the photodiodes central point.

Two types of measurements are possible a DC measurement and an RF measurement. The DC does as above and just simply measures the total optical power incident on each segment. The RF measurement demodulates each segment at some RF frequency and then performs the subtraction/summation. The DC scheme is used for pointing and steady state alignment of a beam, whereas the RF measurement allows us to perform differential wavefront sensing. Essentially measuring the relative tilt and alignment differences between two sidebands.

In FINESSE a segmented photodiode is represented by a matrix **K** that describes how each higher order mode beats at the photodiode. A QPD for example will cause a HG00 and HG10 to now generate a signal, whereas on a single element photodiode these two modes would be orthongal. Therefore we can understand the segmentation as breaking modal degeneracies to ensure certain modes generate a signal. A more detailed description of this process and how to mathematically generate the matrix is given in [34].

Given that we have a discrete vector of modal amplitudes, $\hat{E}$, we can compute two inner products to generate a single value output,

$$S = \hat{E}^* \cdot (\mathbf{K} \cdot \hat{E}).$$

Photodiode types - pdtype

The *pdtype* module has some common segemented photodiode types you can use.

- *pdtypes.XSPLIT* : splits the photoiode into left and right segments for measuring horizontal displacements.
- *pdtypes.YSPLIT* : splits the photoiode into up and down segments for measuring vertical displacements.

Each of the *pdtype* objects are a dictionary which contain a general statement about how different modes should beat and what scaling factor should be applied.

Printing the first few entries to XSPLIT

```
>>> pdtypes.XSPLIT

{ (0, 'x', 1, 'x'): 0.797884560802865,
(0, 'x', 3, 'x'): -0.32573500793528,
(0, 'x', 5, 'x'): 0.218509686118416,
...
```

The key of the dictionary is a tuple *(0, 'x', 1, 'x')* where the first two indicies are the n-th and m-th input mode and the latter two the output mode indices. The *'x'* is a wildcard. The value of this entry is scaling factor to use for mode beats that match this condition. There we can read the first entry as saying the 0th order mode beats with the 1st order with an extra scaling factor of 0.797. Due to the wildcard the the m index must be the same, e.g. HG01 beats with HG11, HG04 and HG14, etc. all with the same scaling factor.

These beats are often limited to order 40 or so as it becomes computational expensive to compute larger beats but also the practicality of using this many orders in a simulation is questionable.

Photodiode type example

Using KatScript we can specify the pdtype directly. Here we just vary the pointing of a beam by misaligning a component and seeing the output on a a horizontal and vertical split photoiode.

```
import finesse
finesse.init_plotting()
model = finesse.Model()
model.parse("""
l l1
bs bs1 R=1 T=0
nothing n1
pd qpdx n1.p1.i pdtype=xsplit
pd qpdy n1.p1.i pdtype=ysplit
link(l1, bs1, 10, n1)

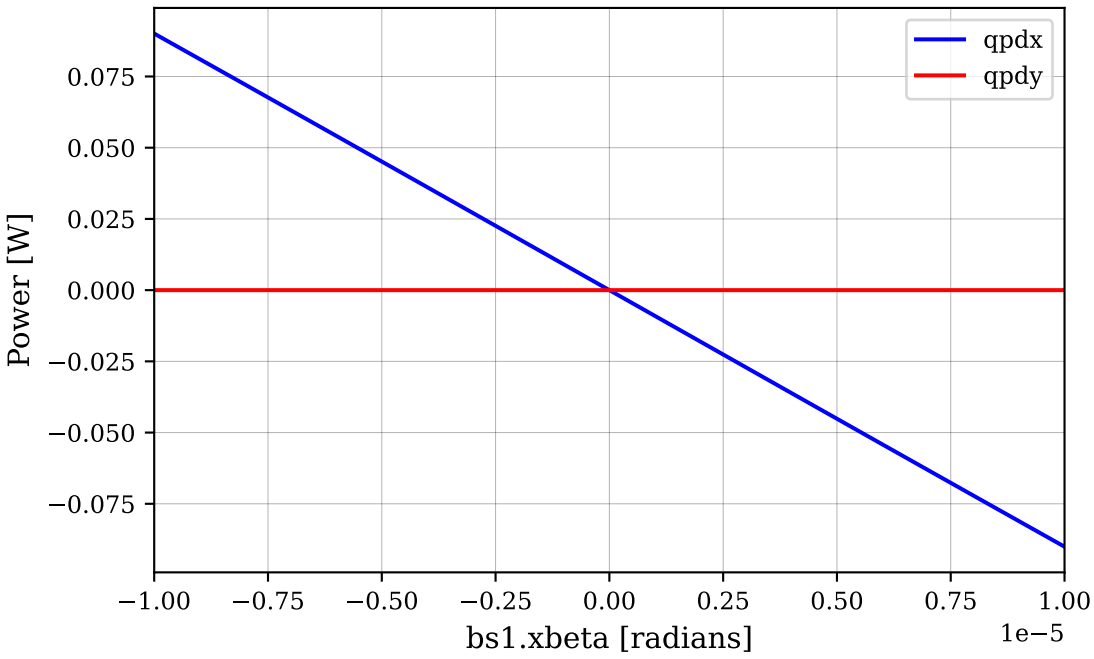
modes(maxtem=1)
```

(continues on next page)

(continued from previous page)

```
gauss g1 bs1.p2.o w0=1m z=0
""")

out = model.run("xaxis(bs1.xbeta, lin, -10u, 10u, 10)")
out.plot();
```



Higher order mode convergence

For large displacements of a beam relative to its spot size size, or misalignments relative to its divergence angle, case must be taken to use enough modes to describe the effect.

Consider a steering mirror directing some beam onto a QPD. The signal outputted from an infinitely sized QPD in the x-direction is proportional to the spot size, w and the displacement Δx

$$S = \text{erf}\left(\frac{2\sqrt{2}\Delta x}{w}\right)$$

We can use a similar model to above to compute how accurately FINESSE matches this with different number of mode orders used.

```
import finesse
import matplotlib.pyplot as plt
import numpy as np
from scipy.special import erf

finesse.init_plotting()
model = finesse.Model()
D = 10 # Distance from optic to QPD [m]

model.parse(f"""
l l1
bs bs1 R=1 T=0 # Steering mirror
nothing n1
pd qpdx n1.p1.i pdtype=xsplit
pd qpdy n1.p1.i pdtype=ysplit
link(l1, bs1, {D}, n1)
```

(continues on next page)

(continued from previous page)

```
modes(maxtem=10)
gauss g1 bs1.p2.o w0=1m z=0
""")
```

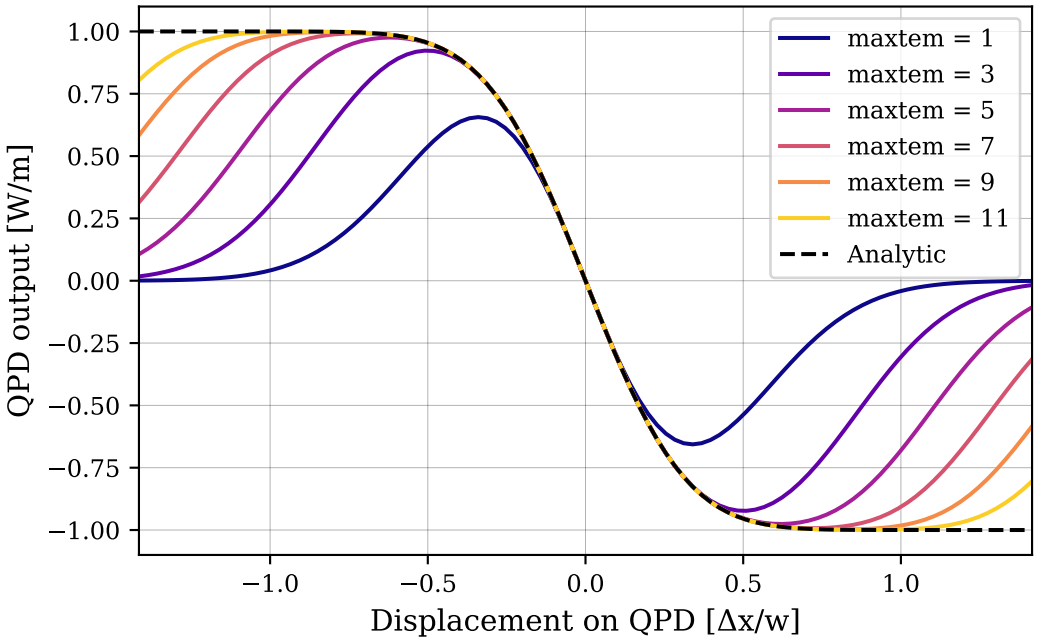
```
<finesse.model.Model at 0x7d17a7f0f4d0>
```

Lastly, we run and plot the outputs:

```
maxtems = range(1, 12, 2)
plt.gca().set_prop_cycle(
    'color', [plt.cm.plasma(i) for i in np.linspace(0, 0.9, len(maxtems))]
)
# loop over each maximum order and run the simulation
for maxtem in maxtems:
    model.modes(maxtem=maxtem)
    out = model.run("xaxis(bs1.xbeta, lin, -0.5m, 0.5m, 100)")
    delta_x = out.x1*D
    plt.plot(delta_x/model.n1.p1.i.qx.w, out['qpdx'], label=f'maxtem = {maxtem}')

w_ = model.n1.p1.i.qx.w/np.sqrt(2)
plt.plot(delta_x / model.n1.p1.i.qx.w, -erf(2*delta_x/w_), c='k', ls='--', label='Analytic')
plt.xlabel(r"Displacement on QPD [$\Delta \mathrm{x}/\mathrm{w}$]")
plt.ylabel("QPD output [W/m]")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7d17ae9a0f50>
```



The x-axis is plotted in units of displacement per spot-size - this is the important ratio to always consider when deciding how many orders are required in such scenarios. For large displacements an ever increasing number of modes are required to describe a displaced HG00, eventually this becomes too computationally expensive. However, we do see that for small displacements the first order adequately describes the system.

Note

An extra minus sign is included in the analytic formula which is due to coordinate system definitions in FINESSE. See Coordinate systems (page 146).

6.4.11 Simulating off-axis beams: calculating the shift

Due to its use of the paraxial approximation, simulating an off-axis beam in FINESSE requires the inclusion of higher order modes, see e.g. Section 9.2 in [18]. In order to study such a misaligned beam in FINESSE we can use a telescopic setup such as depicted in Fig. 6.13 with two mirrors (`m` (page 142)) or beamsplitters (`bs` (page 149)) and use the `xbeta` parameter to add a small mismatch angle β to the angle of incidence of both beamsplitters (or mirrors).

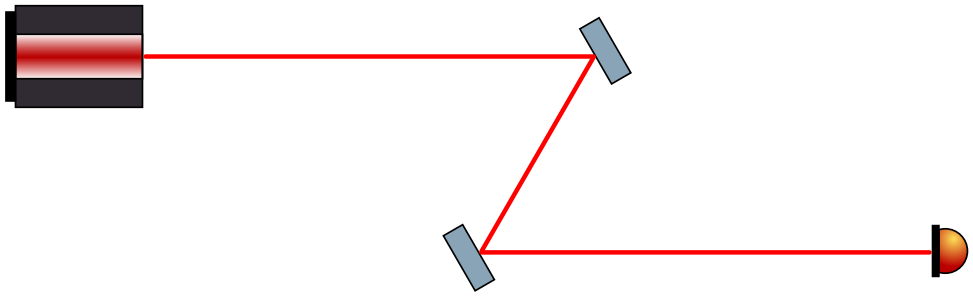


Fig. 6.13: Basic setup

We will derive here the shift in the position of the beam as a function of (among others) this angle β. In addition we will also derive the effect of this parameter on the phase of the beam, resulting from the changed path length between the laser and the detector. Both formulae will be verified using FINESSE simulations in example Aligning and translating a beam with steering mirrors (page 66).

In the next section we will discuss the required number of higher order modes to be included and look at the effects of changing the beam parameters such as waist size and waist position.

Introduction

In Fig. 6.14 we have drawn in detail the two different situations: with and without the tilt β.

The dark red part is the "untilted beam", which has an incident angle α on both of the untilted (dark blue) beamsplitters and traverses a distance s between the beamsplitters. After rotating both beamsplitters over the small angle β (light blue) the beam follows the light red path with corresponding incident angle $\alpha + \beta$ and traverses now a distance $\tilde{s}$ between the beamsplitters. The tilted beam hits the second beamsplitter at a distance d away from where the untilted beam hits it, which results in the relative vertical shift Δ of the position of the outgoing beam for which we derive Eq. (6.9) below

$$\Delta = s \cdot \sin 2\beta$$

The total difference in length between the light and dark red paths translates into a relative phaseshift $\delta\varphi$ when detecting the beam at the right of the telescope, which satisfies Eq. (6.17) below

$$\delta\varphi = \frac{2s \cdot \sin^2 \beta}{\lambda} \cdot 360^\circ$$

Calculation of shift in position Δ

Using the sine rule for the triangle enclosed by d, s and $\tilde{s}$

$$\frac{d}{\sin 2\beta} = \frac{s}{\sin \gamma} = \frac{\tilde{s}}{\sin(90^\circ + \alpha - \beta)}, \tag{6.5}$$

we can express the length d as

$$d = s \cdot \frac{\sin 2\beta}{\sin \gamma}. \tag{6.6}$$

Furthermore, from the total angle of that same triangle we can obtain from

$$(90^\circ - \alpha - \beta + 2\alpha) + 2\beta + \gamma = 180^\circ,$$

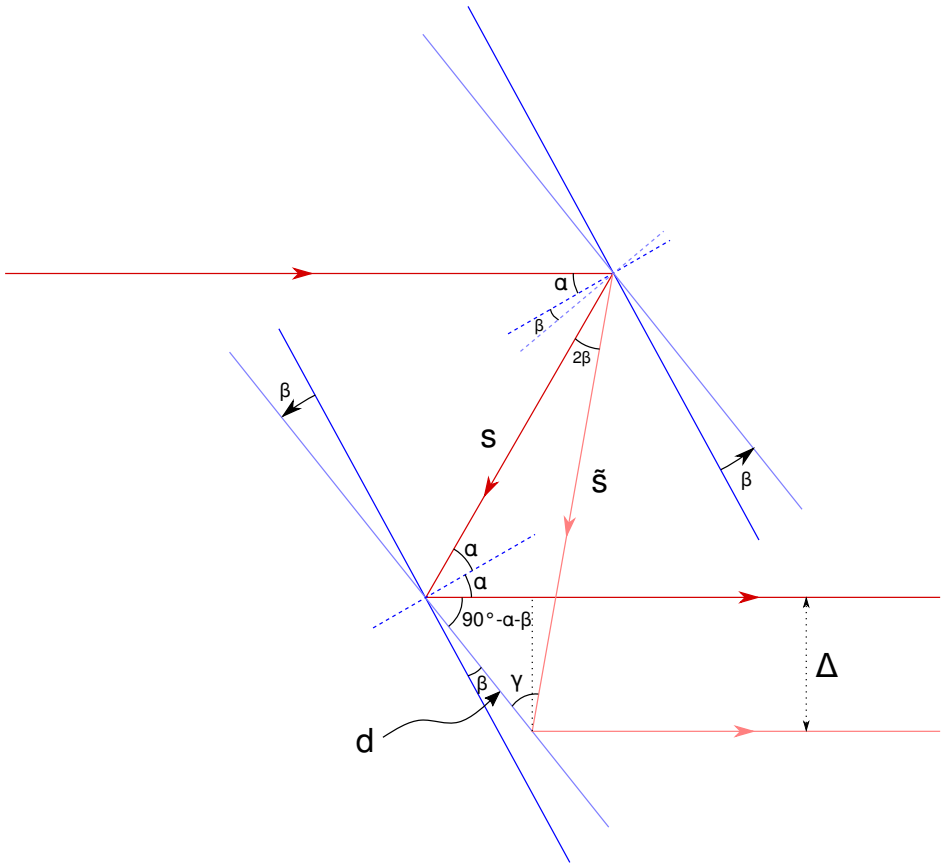


Fig. 6.14: Detailed setup

the angle γ

$$\gamma = 90^\circ - \alpha - \beta. \tag{6.7}$$

We need to calculate Δ:

$$\Delta = d \cdot \sin(90^\circ - \alpha - \beta) = d \cdot \sin\gamma, \tag{6.8}$$

where we used eq. (6.7). Inserting Eq. (6.6) we then obtain the shift

$$\Delta = s \cdot \sin 2\beta \tag{6.9}$$

We see that all α dependence cancels out. Furthermore, we see that the shift increases linearly with the distance between the two beamsplitters. Since for larger values of β the paraxial approximation requires more and more higher modes to be included in a simulation, increasing s instead of β could help simulating the same shift Δ. This will be discussed further below.

Calculation of difference in path length and $\delta\varphi$

Note that there are two parts contributing to the difference in path length:

1. p_1: the difference between $\tilde{s}$ and s
2. p_2: a horizontal part since we hit the beamsplitter further to the right

p_1 will have a positive contribution, while p_2 will have a negative contribution to the change in path length.

calculation p_1

Noting that

$$\sin\gamma = \sin(90^\circ - \alpha - \beta) = \cos(\alpha + \beta) \tag{6.10}$$

we can obtain $\tilde{s}$ from the sine rule Eq. (6.5) to get

$$\tilde{s} = s \cdot \frac{\sin(90^\circ + \alpha - \beta)}{\sin\gamma} = s \cdot \frac{\cos(\alpha - \beta)}{\cos(\alpha + \beta)} \tag{6.11}$$

so

$$p_1 = \tilde{s} - s = s \cdot \frac{\cos(\alpha - \beta) - \cos(\alpha + \beta)}{\cos(\alpha + \beta)} \tag{6.12}$$

calculation p_2

Since part p_2 shortens the total path, we add an overall minus sign:

$$p_2 = -d \cdot \cos(90^\circ - \alpha - \beta) = -d \cdot \sin(\alpha + \beta) \tag{6.13}$$

Using eqs. (6.6) and (6.10) this becomes

$$p_2 = -s \cdot \frac{\sin(\alpha + \beta)\sin 2\beta}{\cos(\alpha + \beta)} \tag{6.14}$$

combining the two

Combining eqs. (6.12) and (6.14) we get

$$s \cdot \frac{\cos(\alpha - \beta) - \cos(\alpha + \beta) - \sin(\alpha + \beta)\sin 2\beta}{\cos(\alpha + \beta)} \tag{6.15}$$

We work out the numerator:

$$\begin{aligned}
\text{numerator} &= \cos(\alpha - \beta) - \cos(\alpha + \beta) - \sin(\alpha + \beta)\sin 2\beta \\
&= \cos\alpha\cos\beta + \sin\alpha\sin\beta - \cos\alpha\cos\beta + \sin\alpha\sin\beta - \\
&\qquad (\sin\alpha\cos\beta + \cos\alpha\sin\beta) 2\sin\beta\cos\beta \\
&= 2\sin\alpha\sin\beta - 2(\sin\alpha\sin\beta\cos^2\beta + \cos\alpha\cos\beta\sin^2\beta) \\
&= 2\sin\alpha\sin\beta(1 - \cos^2\beta) - 2\cos\alpha\cos\beta\sin^2\beta \\
&= 2(\sin\alpha\sin\beta - \cos\alpha\cos\beta)\sin^2\beta \\
&= -2\cos(\alpha + \beta)\sin^2\beta
\end{aligned}$$

Putting this into eq. (6.15) we see that the $\cos(\alpha + \beta)$ factor cancels out and we find the total path difference

$$p_1 + p_2 = -2s \cdot \sin^2\beta \tag{6.16}$$

Note that again the α dependence cancels out. The relative phase shift can be calculated by dividing the path difference Eq. (6.16) by the wavelength λ and multiplying by 360°. We also pick up an extra overall minus sign, cancelling the one from Eq. (6.16), since we define a plane wave as $\cos(\omega t - kx + \varphi)$, see for example Eq. (1.4) in [18]. Putting it together we get

$$\delta\varphi = \frac{2s \cdot \sin^2\beta}{\lambda} \cdot 360^\circ \tag{6.17}$$

Summary

- Both the shift of the beam eq. (6.9) and the change in phase eq. (6.17) are independent of the angle α
- Given the square in eq. (6.16) the path length difference is typically small but not per se compared to the wavelength λ, leading to a significant phase shift.

Effect on a finite size beam

The above calculation was done for an infinitesimally narrow beam hitting the beamsplitter exactly in the midpoint (its rotation point). We show here that the result is the same for a beam hitting the beamsplitter off-axis.

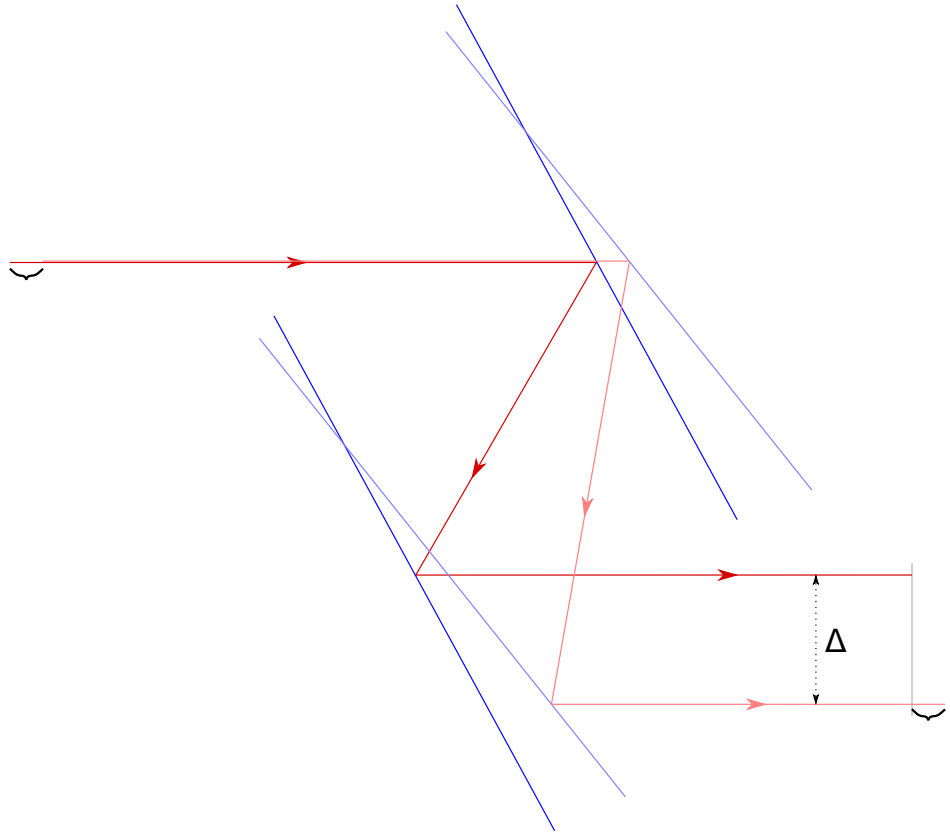


Fig. 6.15: Moving the original beamsplitter and original beam left

From Fig. 6.15 we see that – compared to the on-axis case – the effect of such an off-axis incoming beam can be described by a horizontal translation of the original beamsplitters (dark blue) together with their beams (dark red) with respect to the original tilted beamsplitters (light blue) and their beams (light red). It is clear that

- this does not affect the shift Δ of the beam on the beam detector.
- this does not affect the total path length difference and hence $\delta\varphi$ since the dark red beam gains a bit on the left but loses the same amount on the right.

6.4.12 Simulating off-axis beams: convergence and scaling

In order to properly describe off-axis beams in the paraxial approximation, a number of higher order modes will need to be taken into account. How many modes need to be included (by setting `maxtem` using the `modes` (page 197) command) will depend on the optical layout and geometry of the beam such as the waist position z_0 and the relative distance of the different optical components to each other. Typically, if we choose `maxtem` too small, we expect to miss part of the beam in simulations, resulting in missing total beam energy. In this section we will therefore study how this total beam energy - as measured using a `power_detector_dc` (page 180) detector - behaves when changing the various parameters. In general, when increasing `maxtem` we expect convergence towards the total energy put into the beam by the `laser` (page 165).

We will use the same optical layout as used in the previous section Simulating off-axis beams: calculating the shift (page 314), it is sketched again in Fig. 6.16.

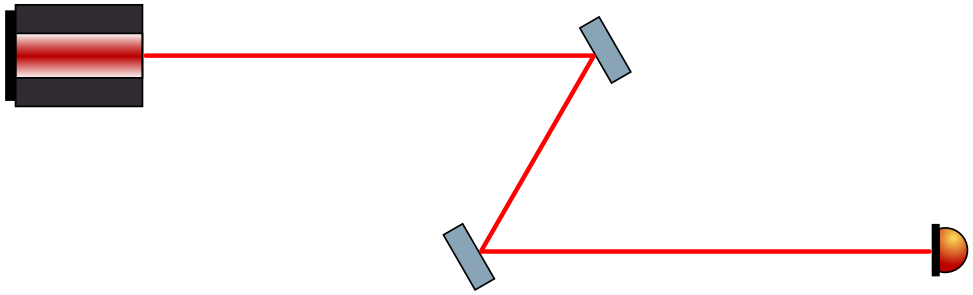


Fig. 6.16: Basic setup

Both beamsplitters are tilted over the same small angle β, resulting in a parallel off-axis beam. In order to correctly describe the physics at increasingly larger β, more and more higher order modes will need to be taken into account. We will start by investigating this behaviour for different values of `maxtem`. We will then study the effects of changing the two gaussian beam parameters waist position z_0 and waist size w_0. For z_0 we will find an optimal position within the optical layout, providing the largest valid range of β. Once we have found the optimal z_0, we will use it to study the valid β range as a function of the distance between the two beamsplitters, looking in particular at the scaling under changes of the waist size w_0 and linking it to the diffraction angle and Rayleigh range.

Base script for the different simulations

Each of the simulations will be using the following base script, consisting of a `laser` (page 165), two `beamsplitter` (page 149) elements and a `power_detector_dc` (page 180). The different (default) parameters are defined as python variables directly above the FINESSE KatScript. The layout is the same as that used in Aligning and translating a beam with steering mirrors (page 66).

```
import matplotlib.pyplot as plt
import finesse
from finesse.analysis.actions import Xaxis
finesse.configure(plotting=True)

power = 1.5  # default laser power in Watt
w0 = 10e-3   # default gaussian waist size in meter
alpha = 30   # angle of incidence for both BS in degrees
xbeta = 1e-5 # default tilt of both beam splitters in radians
s1 = 1000    # distance till 1st beamsplitter
s2 = 400     # distance between beamsplitters
s3 = 600     # default distance between 2nd beamsplitter and detector
z0 = -1200   # default waist position, 1200 meter right of laser
maxtem = 7   # default maxtem

basescript = f"""
# Define laser and gaussian beam
laser l1 P={power}
gauss g1 l1.p1.o w0={w0} z={z0}
modes(maxtem={maxtem})

# define beamsplitters and their positions
space s1 l1.p1 bs1.p1 L={s1}
beamsplitter bs1 R=1 T=0 alpha={alpha} xbeta={xbeta}
space s2 bs1.p2 bs2.p1 L={s2}
beamsplitter bs2 R=1 T=0 alpha=bs1.alpha xbeta=bs1.xbeta
space s3 bs2.p2 n1.p1 L={s3}

# use photodiode to measure integrated power
nothing n1
power_detector_dc pd1 node=n1.p1.i pdtype=none
```

(continues on next page)

(continued from previous page)

```
"""

basekat = finesse.Model()
basekat.parse(basescript)
```

```
<finesse.model.Model at 0x79998ddf1fd0>
```

Tilt angle dependency for various maxtem

We run an appropriate action `xaxis` (page 209) / `Xaxis` for the model to measure the total energy as a function β for 4 different values of `maxtem`: 3, 5, 7 and 9 (set using the `modes` (page 197) / `Model.modes()` method)

```
# Set of maxtem values (and their plot styles)
maxtem_vals   = [3, 5, 7, 9]
maxtem_styles = ['r', 'b', 'g', 'm']

# Define the model and set sweep parameter to xbeta
kat1 = finesse.Model()
kat1.parse(basescript)

out1a = []
for maxtem_val in maxtem_vals:
    kat1.modes(maxtem=maxtem_val)
    out1a.append(kat1.run(Xaxis(kat1.bs1.xbeta, "lin", 0, 4e-5, 40)))
```

In order to verify that the convergence is independent of the position of the photodiode, i.e. to verify that the (diverging) gaussian beam emerging from the second beamsplitter remains stable over larger distances, we repeat the simulation at fixed `maxtem` = 7 at two different values of s_3: 600 and 1000 meter:

```
# Set of s3 values (and their plot styles)
s3_vals   = [600, 1000]
s3_styles = ['g', 'b.']

# Reset maxtem to its default
kat1.modes(maxtem=maxtem)

out1b = []
for s3_val in s3_vals:
    kat1.spaces.s3.L = s3_val
    out1b.append(kat1.run(Xaxis(kat1.bs1.xbeta, "lin", 0, 4e-5, 40)))
```

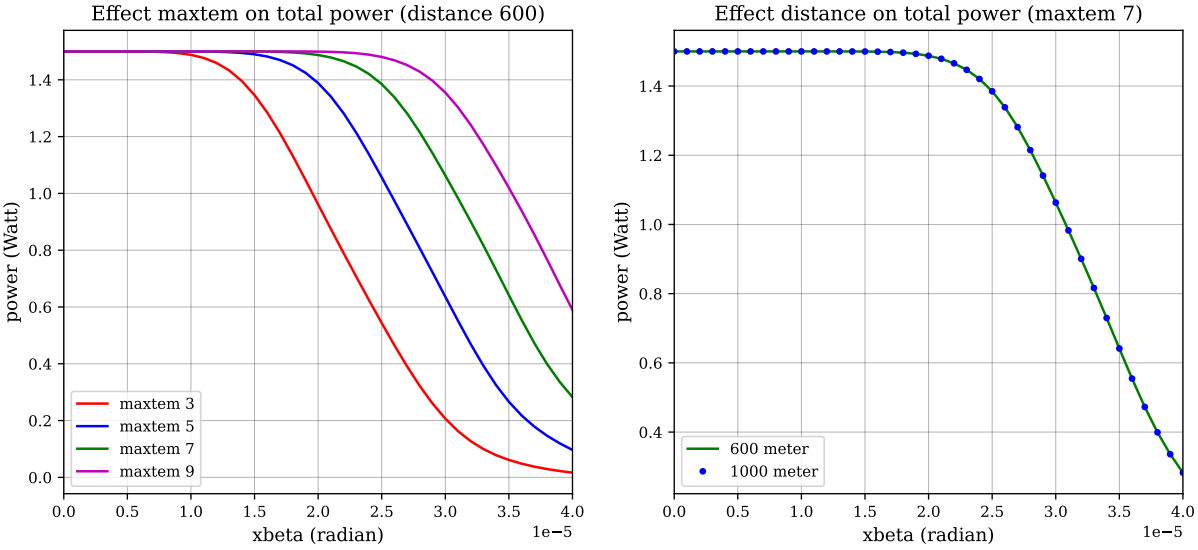
We plot both results below:

```
f,ax = plt.subplots(ncols=2, figsize=(12, 5))

ax[0].set_title(f"Effect maxtem on total power (distance {s3})")
for sim in range(len(out1a)):
    ax[0].plot(out1a[sim].x[0], out1a[sim]['pd1'],
               maxtem_styles[sim], label=f"maxtem {maxtem_vals[sim]}")

ax[1].set_title(f"Effect distance on total power (maxtem {maxtem})")
for sim in range(len(out1b)):
    ax[1].plot(out1b[sim].x[0], out1b[sim]['pd1'],
               s3_styles[sim], label=f"{s3_vals[sim]} meter")

for i in 0,1:
    ax[i].set_xlabel("xbeta (radian)")
    ax[i].set_ylabel("power (Watt)")
    ax[i].legend(loc="lower left")
```



From the left figure, it is clear that we can simulate larger and larger β if we include more and more higher order modes in the simulation, i.e. by increasing the `maxtem` value. The valid range of β scales roughly linearly with `maxtem`, meaning the number of modes to be includes roughly scales quadratically (note that for `maxtem` = N there are $\frac{1}{2}(N+1)(N+2)$ modes).

From the right figure we see that the convergence is not dependent on the distance behind the second beamsplitter at which we measure the total power in the bundle indicating a stable beam.

Waist position dependency

Above we showed that the total power in the beam can give a clear indication for the range of validity of the tilt angle β (for given `maxtem`). We will now use this dependency to find the optimal position of the waist of the Gaussian beam z_0.

We run simulations for two different ranges in z_0: one for a larger range that starts at the laser and ends at the detector, and a second for a smaller range and closer to the two beamsplitters, where z_0 varies from 600 till 1600 meter to the right of the laser (i.e from 400 meter left of the first beamsplitter till 200 meter right of the second beamsplitter).

Each simulation is run using an `x2axis` (page 213) action varying both β and z_0. Note that as sweep parameter in the `x2axis` (page 213) action we have to specify either the x- or the y-coordinate of z_0, so we need to make sure that they change simultaneously. We can easily enforce this by making one a reference to the other (see Expressions and references (page 129)).

```
# Define the model
kat2 = finesse.Model()
kat2.parse(basescript)

# Make sure zy equals zx
kat2.g1.zy = kat2.g1.zx.ref

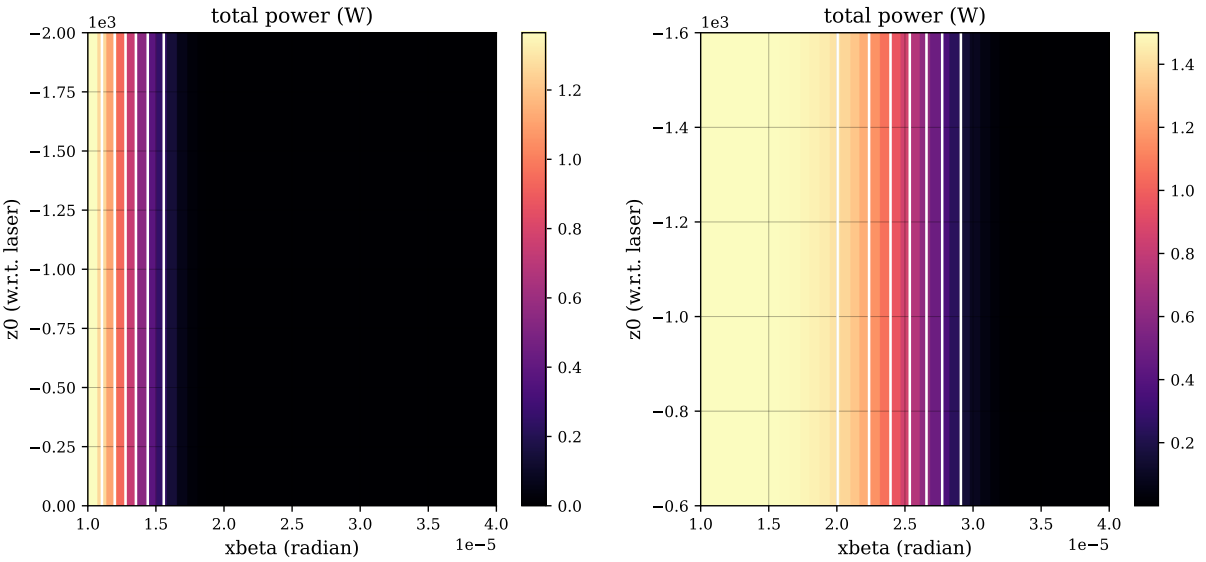
# vary beta and z0
out2 = kat2.run(f"""
    series(
        x2axis(bs1.xbeta, lin, 1e-5, 4e-5, 40, g1.zx, lin, 0, -2000, 60, name="full"),
        x2axis(bs1.xbeta, lin, 1e-5, 4e-5, 40, g1.zx, lin, -600, -1600, 60, name="zoom")
    )
""")

# Plot the full and zoom results
f,ax = plt.subplots(ncols=2, figsize=(11.8, 5))
for (i, name) in ([0, 'full'], [1, 'zoom']):
    pxy_extent = (out2[name].x[0].min(), out2[name].x[0].max(),
                  out2[name].x[1].max(), out2[name].x[1].min())
```

(continues on next page)

(continued from previous page)

```
    p = ax[i].imshow(out2[name]['pd1'].T, aspect='auto', extent = pxy_extent)
    plt.colorbar(p, ax = ax[i])
    ax[i].contour(out2[name]['pd1'].T, colors='white', extent = pxy_extent)
    ax[i].set_xlabel("xbeta (radian)")
    ax[i].set_ylabel("z0 (w.r.t. laser)")
    ax[i].set_title("total power (W)")
    ax[i].ticklabel_format(style='sci', scilimits=(0,0))
```



The left plot shows the full range from the laser till the detector, the right plot the close-up range around the beamsplitters. We see that the optimal location for the waist is around 1 km right of the laser, i.e. at the location of the first beamsplitter.

Effect of waist size

Now that we know the optimal location for the Gaussian waist, we will fix it that position and then study the effect of changing the distance between the two beamsplitters s_2 and the scaling behaviour with respect to the waist size w_0 of the beam.

We do two simulations, one at the default waist size w_0 = 10 mm and one at a slightly larger waist size w_0 = 12 mm. We will vary s_2 between 0 and 1 km for the first simulation and between 0 and about 1.4 km for the second (we will discuss below why we choose this range). We do both simulations in a single `series` (page 207), using the `change` (page 235) action to adapt the waist size. We again use a reference to ensure that the x- and y-components of both w_0 and z_0 are changing simultaneusly (see Expressions and references (page 129)).

```
# w0, beta-range and L for 10mm run
w0a = 10
beta_a = (1e-5, 4e-5)
La = 1000

# w0, beta-range and L for 12mm run
w0b = 12
beta_b = (1e-5/1.2, 4e-5/1.2)
Lb = 1000*1.2**2

# Define model
kat3 = finesse.Model()
kat3.parse(basescript)

# could set w0x and w0y separately, easier to use ref for y
kat3.g1.w0y = kat3.g1.w0x.ref
```

(continues on next page)

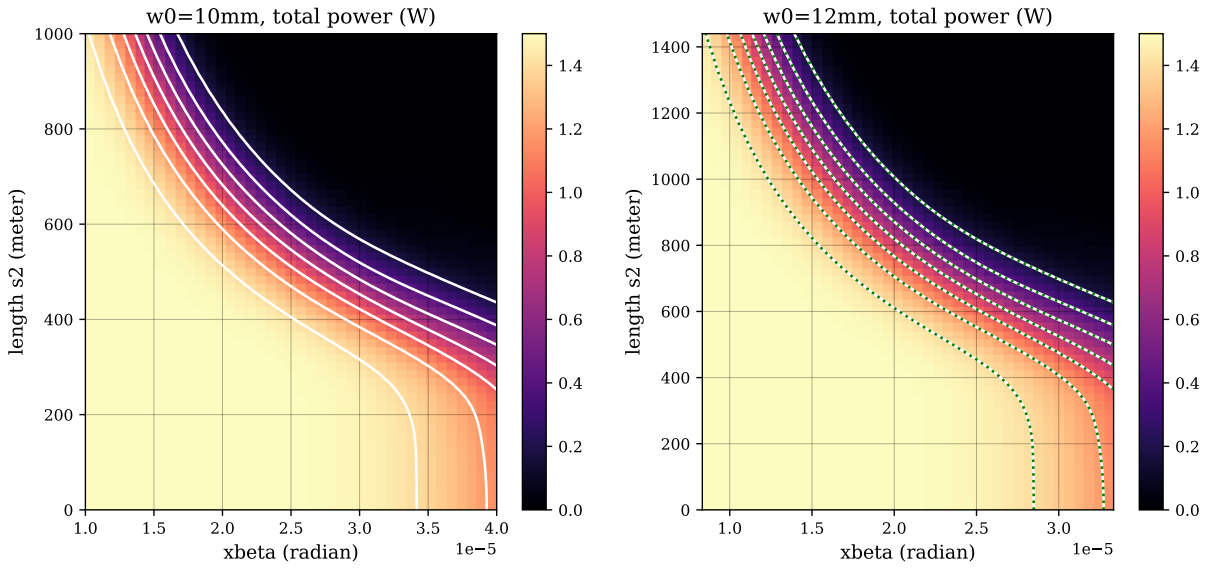
(continued from previous page)

```
# Position gauss at optimal z0: i.e. position bs1
kat3.g1.zx = -1000
kat3.g1.zy = kat3.g1.zx.ref

# run both subsimulations using series()
out3 = kat3.run(f"""
    series(
        change(g1.w0x={w0a}e-03),
        x2axis(bs1.xbeta, lin, {beta_a[0]}, {beta_a[1]}, 40, s2.L, lin, 0, {La}, 60, name="a
↪"),
        change(g1.w0x={w0b}e-03),
        x2axis(bs1.xbeta, lin, {beta_b[0]}, {beta_b[1]}, 40, s2.L, lin, 0, {Lb}, 60, name="b
↪")
    )
""")
```

And we plot the results

```
# Plot results for both waist sizes
f,ax = plt.subplots(ncols=2, figsize=(11.8, 5))
for (i, name, w0val) in ([0, 'a', w0a], [1, 'b', w0b]):
    pxy_extent = (out3[name].x[0].min(), out3[name].x[0].max(),
                  out3[name].x[1].min(), out3[name].x[1].max())
    p = ax[i].imshow(out3[name]['pd1'].T, aspect='auto', extent = pxy_extent)
    plt.colorbar(p, ax = ax[i])
    ax[i].contour(out3[name]['pd1'].T, colors='white', extent = pxy_extent)
    ax[i].set_xlabel("xbeta (radian)")
    ax[i].set_ylabel("length s2 (meter)")
    ax[i].set_title(f"w0={w0val}mm, total power (W)")
    ax[i].ticklabel_format(axis='y', style='plain', scilimits=(0,0))
# Also add rescaled contours from 'a' to 'b'
pxy_extent_scaled = (out3['a'].x[0].min()/1.2,   out3['a'].x[0].max()/1.2,
                     out3['a'].x[1].min()*1.2**2,out3['a'].x[1].max()*1.2**2)
ax[1].contour(out3['a']['pd1'].T, colors='green', linestyles='dotted',
              extent = pxy_extent_scaled);
```



Looking first at the left picture, we see that the usable range of β roughly remains constant till a distance of about 200-300 meter and then quickly decreases for larger distances. This distance of 200-300 meter corresponds roughly to the Rayleigh range z_R

$$z_R = \frac{\pi w_0^2}{\lambda} \tag{6.18}$$

which for the parameters in the left plot ($w_0 = 10\text{mm}, \lambda = 1.064\mu\text{m}$) is 295 meter. It would seem that our default value of 400 meter is not ideal, but we will discuss this further below when looking at the maximum attainable shift Δ.

Although in general the largest usable β increases as a function of `maxtem` we expect that it will be proportional to the diffraction angle Θ given by (see Eq. (9.19) in [18])

$$\Theta = \arctan\left(\frac{w_0}{z_R}\right) \approx \frac{w_0}{z_R} = \frac{\lambda}{\pi w_0} \tag{6.19}$$

which - again for the parameters used in the left plot - equals $3.4 \cdot 10^{-5}$ radians (which happens to be very close to the maximum β at the smaller distances for the `maxtem` = 7 used in this simulation).

We can now discuss the scaling behaviour as a function of the waist size w_0. From the diffraction angle Θ Eq. (6.19) we expect the useable range of β to scale as $1/w_0$, while from the Rayleigh range z_R Eq. (6.18) we expect the vertical range to scale as w_0^2. To verify these assumptions, we use - for the right plot with the results of w_0 = 12 mm - a 1.2 times smaller horizontal range and a 1.44 times larger vertical range. We also plot in the same plot the contours (dotted green) from the left plot, scaled with the same factors and see two identical pictures, confirming that z_R and Θ set the relevant scales.

From Eq. (6.9) in Simulating off-axis beams: calculating the shift (page 314) it now follows that for small angles β the largest attainable shift Δ scales (roughly) linearly with w_0, i.e. the dimensionless shift in units of w_0 is actually independent of that waist size w_0. It is therefore interesting to also look at the beam energy as a function of Δ/w_0 instead of as a function of β. This will allow us to find the optimal distance between the two beamsplitters, leading to the largest range of validity for Δ/w_0. We could refactor the results of the above simulation, but it is easier to do a simulation using Δ/w_0 directly in the `x2axis` (page 213).

```
import numpy as np

# define model
kat4 = finesse.Model()
kat4.parse(basescript)

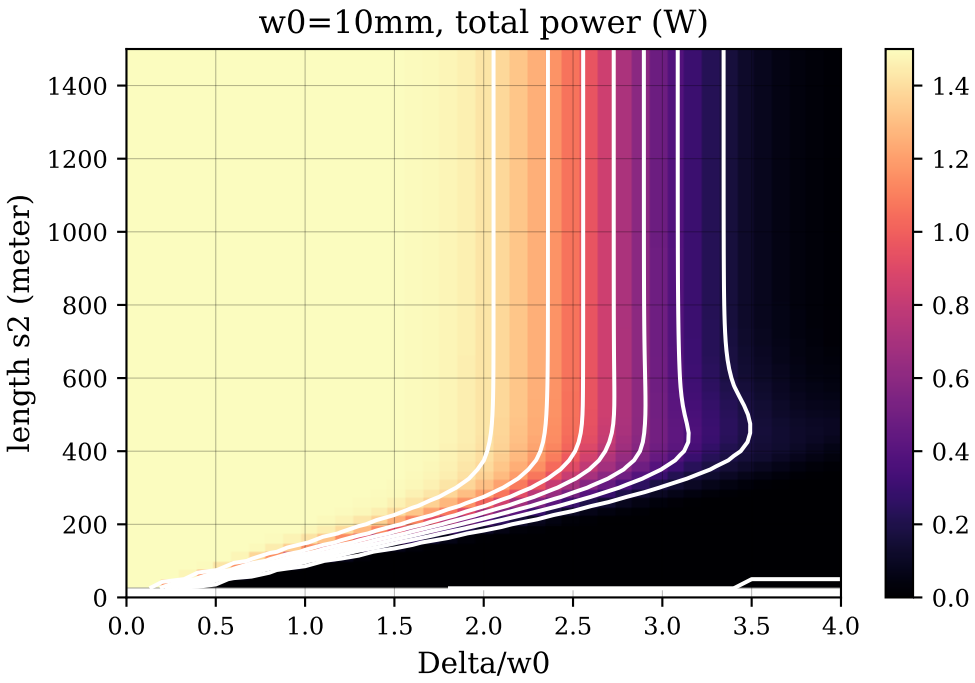
# Position gaussian waist at the optimal location
kat4.g1.zx = -1000
kat4.g1.zy = kat3.g1.zx.ref

# Introduce variable delta
kat4.parse("var Delta 1.0")
# xbeta follows from Delta/w0 = s2*sin(2*beta)
kat4.bs1.xbeta = 0.5*np.arcsin(w0*kat4.Delta.ref/kat4.spaces.s2.L.ref)

# Run actual simulation
out4 = kat4.run(f"x2axis(Delta, lin, 0.0, 4.0, 40, s2.L, lin, 0, 1500, 60)")
```

and plot the result

```
f,ax = plt.subplots()
pxy_extent = (out4.x[0].min(), out4.x[0].max(),
              out4.x[1].min(), out4.x[1].max())
p = ax.imshow(out4['pd1'].T, aspect='auto', extent = pxy_extent)
plt.colorbar(p, ax = ax)
ax.contour(out4['pd1'].T, colors='white', extent = pxy_extent)
ax.set_xlabel("Delta/w0")
ax.set_ylabel("length s2 (meter)")
ax.set_title(f"w0={w0a}mm, total power (W)")
ax.ticklabel_format(axis='y', style='plain', scilimits=(0,0))
```



We see that for larger distances between the two beamsplitters, the range of validity of Δ/w_0 becomes constant, it is then only dependent on the number of higher order modes included.

To summarize, we have seen that the best location for the first beamsplitter is at the waist of the Gaussian beam. Furthermore, the optimal position for the second beamsplitter is just beyond the Rayleigh range. Going to larger distances decreases the maximum tilt, going to smaller distances decreases the maximum (dimensionless) beamshift. The maximum value of this beamshift is not dependent on the waist size and for larger inter-beamsplitter distances depends only on the number of higher order modes included in the simulation.

6.4.13 Laguerre higher order modes

Finesse does not support Laguerre-Gaussian modes directly, but there are some tools for generating both Helical and Sinusoidal LG modes modes. These can be used for decomposition of a beam into LG modes or comparing to eigenmode solutions from Finesse, which may be in the form of Laguerre-Gaussian modes.

Here are the first few sinusoidal LG modes:

```
import finesse
from finesse.cymath import laguerre as lg
import numpy as np
import matplotlib.pyplot as plt

finesse.init_plotting()

fig, axes = plt.subplots(3, 3, figsize=(8, 8))

w0 = 1e-3
z = 0
x = np.linspace(-4 * w0, +4 * w0, 50)
y = np.linspace(-4 * w0, +4 * w0, 51)
X, Y = np.meshgrid(x, y)
helical = False  # whether to plot helical LG modes or sinusoidal LG modes

for i, p in enumerate([0, 1, 2]):
    for j, l in enumerate([0, 1, 2]):
        E = lg.compute_lg_mode(p, l, w0, z, 1064e-9, x, y, helical)
        intensity = np.abs(E) ** 2
        ax = axes[i, j]
```

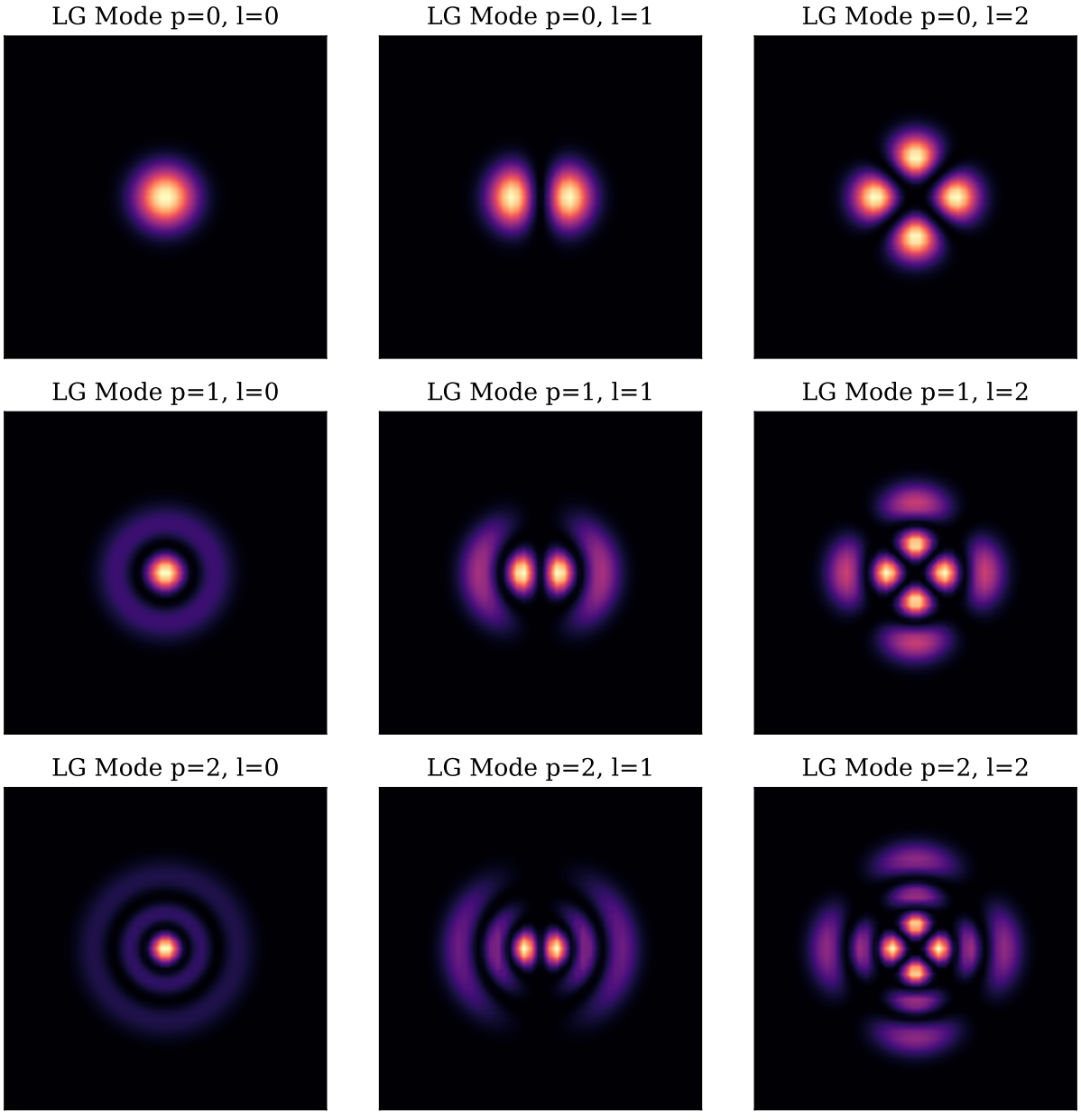
(continues on next page)

(continued from previous page)

```
        C = ax.contourf(X, Y, intensity.T, levels=100)
        C.set_edgecolor("face")
        ax.set_title(f"LG Mode p={p}, l={l}")
        ax.set_aspect("equal")

for ax in axes.flatten():
    ax.set_xticks([])
    ax.set_yticks([])

plt.tight_layout()
plt.show()
```



and the Helical modes:

```
helical = True
fig, axes = plt.subplots(3, 3, figsize=(8, 8))

for i, p in enumerate([0, 1, 2]):
```

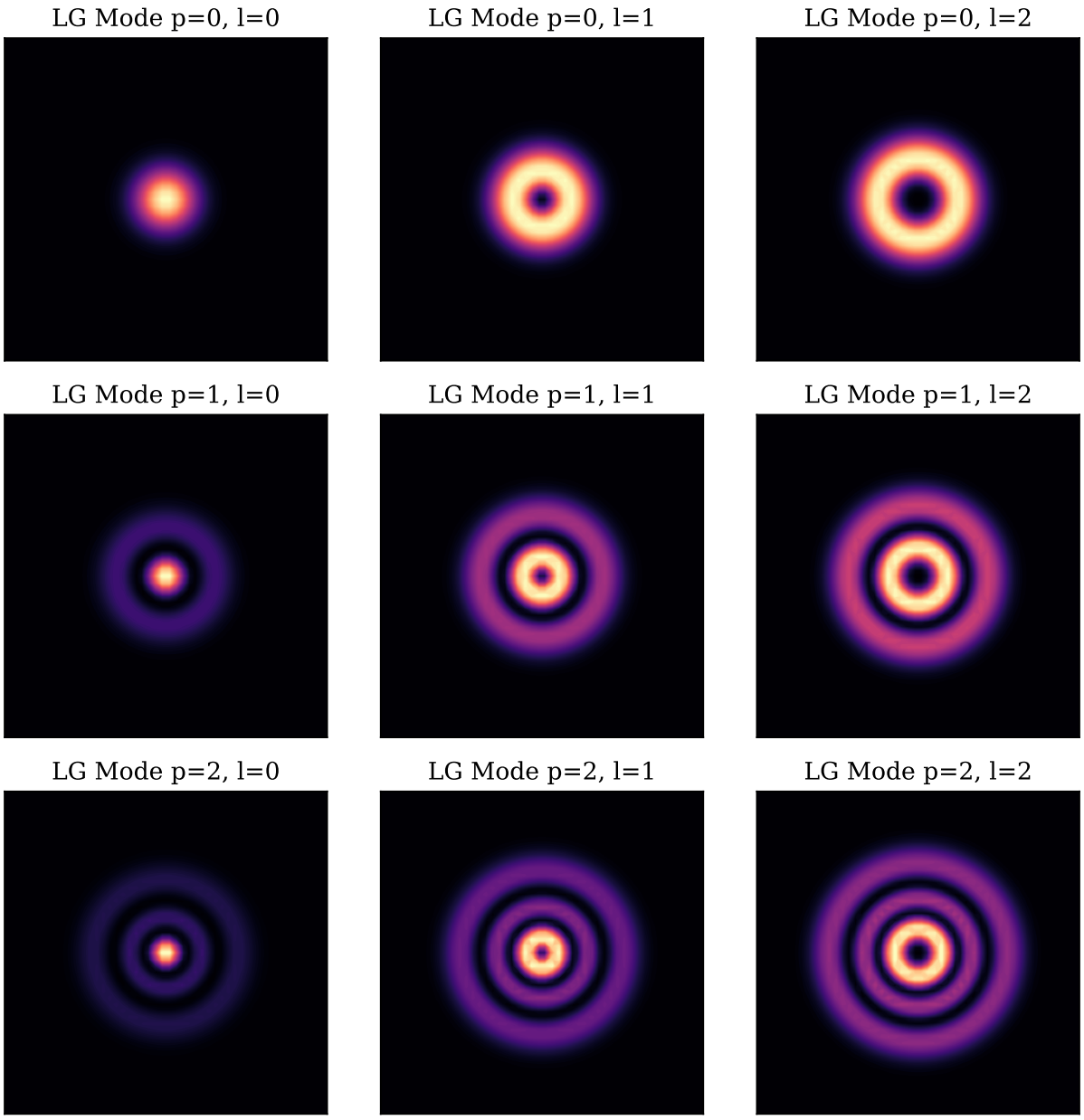
(continues on next page)

(continued from previous page)

```
    for j, l in enumerate([0, 1, 2]):
        E = lg.compute_lg_mode(p, l, w0, z, 1064e-9, x, y, helical)
        intensity = np.abs(E) ** 2
        ax = axes[i, j]
        C = ax.contourf(X, Y, intensity.T, levels=100)
        C.set_edgecolor("face")
        ax.set_title(f"LG Mode p={p}, l={l}")
        ax.set_aspect("equal")

for ax in axes.flatten():
    ax.set_xticks([])
    ax.set_yticks([])

plt.tight_layout()
plt.show()
```



6.5 Modelling with higher-order modes

If you are not familiar with spatial Higher Order Modes (HOMs) then you should read the Higher-order Hermite-Gauss modes (page 298) section of the manual. This will provide you with the background physics details for understanding the terms and examples in these pages.

These pages will focus on how to use the features of FINESSE for modelling HOMs, links to the physics behind each feature are given where appropriate.

6.5.1 Selecting the modes to model

Models in FINESSE are, by default, created as plane-wave representations of the underlying configuration. This can be switched to a modal basis by specifying the mode indices that should be modelled by the system. There are a few key ways in which this can be done, as outlined in the sections below.

Maximum TEM order

Set a value for the maximum TEM (termed *maxtem* for short) order. Users of previous version of FINESSE will be familiar with this option. By setting a value for maxtem, all of the modes up to and including this order will be included in the model. See an example below:

```
import finesse

model = finesse.Model()
# set maximum mode order (n+m) to 3
model.modes(maxtem=3)

print(model.modes())
```

```
[[0 0]
 [1 0]
 [0 1]
 [2 0]
 [1 1]
 [0 2]
 [3 0]
 [2 1]
 [1 2]
 [0 3]]
```

Note that setting `maxtem` to zero will also switch the model to a modal basis, where the only mode being modelled then is the TEM00 mode.

You can switch a model back to a plane wave representation by using `Model.switch_off_homs()` or passing "off" to `finesse.model.Model.modes()`:

```
import finesse

model = finesse.Model()
model.modes(maxtem=3)
model.modes("off")

print(model.modes())
```

```
None
```

Selection method

One of the main new features of modelling higher-order modes in FINESSE 3 is the ability to select the modes you want to model rather than including all modes up to a specific order with *maxtem*. This can be achieved via varied arguments to the `finesse.model.Model.modes()` method. The documentation for this function explains how to use it, so here are a few examples of it in practice.

Selecting even modes up to order 4:

```
model.modes("even", 4)
print(model.modes())
```

```
[[0 0]
 [2 0]
 [0 2]
 [4 0]
 [2 2]
 [0 4]]
```

Odd modes up to order 3:

```
model.modes("odd", 3)
print(model.modes())
```

```
[[0 0]
 [1 0]
 [0 1]
 [1 1]
 [3 0]
 [0 3]]
```

Tangential (x) modes up to order 5:

```
model.modes("x", 5)
print(model.modes())
```

```
[[0 0]
 [1 0]
 [2 0]
 [3 0]
 [4 0]
 [5 0]]
```

Sagittal (y) modes up to maxtem 6:

```
model.modes("y", 6)
print(model.modes())
```

```
[[0 0]
 [0 1]
 [0 2]
 [0 3]
 [0 4]
 [0 5]
 [0 6]]
```

Assigning to modes property

The `finesse.model.Model.modes()` function can be used to set the modes more to include more explicitly. A couple of examples are shown below.

Add specific modes to the model via a list of strings:

```
model.modes(["00", "11", "22"])
print(model.modes())
```

```
[[0 0]
 [1 1]
 [2 2]]
```

Or a list of tuples (any iterable of length two iterables will work):

```
model.modes([(0, 0), (2, 1), (1, 3)])
print(model.modes())
```

```
[[0 0]
 [2 1]
 [1 3]]
```

Including additional modes

The model class also provides a `finesse.model.Model.include_modes()` method for inserting additional modes (at the correct, sorted position of the mode indices array) into the existing modes.

For example, one could select even modes up to order 2 but also include the 11 mode with:

```
model.modes("even", 2)
model.include_modes("11")
print(model.modes())
```

```
[[0 0]
 [0 2]
 [1 1]
 [2 0]]
```

A note on specifying mode indices explicitly

If the *modes* argument to `Model.modes()` (or, indeed, `Model.include_modes()` and `Model.remove_modes()`) is specified as an iterable of mode indices, then care must be taken to define these modes properly. For example, if one tries to do:

```
model.modes(["010", "123"])
```

then this would result in an exception being raised due to the ambiguity in the mode indices when they are larger than 9. For these large HOM indices, one should instead use tuples (or lists), e.g:

```
model.modes([(0, 10), (12, 3)])
```

to select the HG(0,10) and HG(12,3) modes.

This principle applies equally to selecting detector masks via `MaskedDetector.select_mask()`.

6.5.2 Defining the modal basis

Using the correct modal basis (via beam parameters) at each point in the model configuration is important for computing couplings through Scattering matrices (page 308) and other quantities such as the spatial distribution u_{nm} used for, e.g., calculating beam profiles. For these reasons, it is necessary to understand how FINESSE sets beam parameters at nodes and how you can use this information to model the systems that you are investigating.

Complete details on the beam tracing algorithm of FINESSE are included in Tracing the beam (page 306), this page will focus on how to use cavities and manual setting of beam parameters.

Using cavity objects

It is important in any analysis involving cavities that HOM couplings are computed using a (propagated) cavity eigenmode as the basis. If the beam parameters at nodes inside a cavity get set to values which are not equal to the corresponding cavity eigenmode then we can get mismatches where there shouldn't be any and / or matches where there should be mismatches.

A key role of the `Cavity` (page 195) object is to prevent this from happening by influencing the beam tracing algorithm when instances of this class are present in a model. The eigenmodes of all cavities in a model are computed upon adding it to the model and these are then used to set beam parameters in the rest of the model via propagating them through the ABCD matrix formalism, see Transformation of the beam parameter (page 303) for details on this.

Adding cavities to a model

`Cavity` (page 195) objects can be added to a model in the standard way via a call to `Model.add()` using the Python API. The constructor of the Cavity class requires a *name* for the cavity and the *source* node (i.e. starting point) of the cavity; with an optional *via* node if the cavity path should traverse via a specific node.

For simple cavities where the surfaces are not bounded (on the inside of the cavity path) by AR / HR coating surfaces, you only ever need to specify the *source* node or port. The cavity path will then be determined automatically from just this node. An example of this is shown below, using a simple Fabry-Perot type cavity model:

```
import finesse
import finesse.components as components

model = finesse.Model()
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1

m ITM R=0.99 T=0.01 Rc=-2.5
s sCAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=2.5

# Add a cavity object with source node as output node
# of the second port of mirror ITM
cav FP ITM.p2
""")

# showing the cavity path
print(model.FP.draw())
```

```
o ITM.p2.o
└─o ETM.p1.i
  └─o ETM.p1.o
    └─o ITM.p2.i
```

If you have a model with AR / HR coatings at surfaces then you will need to specify the *via* node argument in order for the cavity path to include the surface itself. See the example below:

```
model = finesse.Model()
model.parse("""
l L0 P=1
s l0 L0.p1 PRM.p1 L=1

m PRM T=0.046 L=37.5u

s lpr1 PRM.p2 BS.p1 L=9

bs BS T=0.5 L=37.5u alpha=60

s ly BS.p2 ITMAR.p1 L=100
```

(continues on next page)

(continued from previous page)

```
# Y arm input mirror with AR surface
m ITMAR R=0 L=20u
s ITMsub ITMAR.p2 ITM.p1 L=0.2 nr=1.44963098985906
m ITM T=7000u L=37.5u Rc=-5580

lambda(1550n)
""")

# INCORRECT -- doesn't include ITM
model.add(components.Cavity("PRC_wrong", model.PRM.p2.o))

print(model.PRC_wrong.draw())
```

```
o PRM.p2.o
└─o BS.p1.i
  └─o BS.p2.o
    └─o ITMAR.p1.i
      └─o ITMAR.p1.o
        └─o BS.p2.i
          └─o BS.p1.o
            └─o PRM.p2.i
```

```
/usr/local/lib/python3.13/site-packages/finesse/model.py:2255:␣
↪CavityUnstableWarning: cavity 'PRC_wrong' added to the model is unstable
  warn(
```

You can see from this cavity path that the cavity "end" point is ITMAR, **not** ITM, as by default the cavity will find the shortest path back to the source node. To correct this, you must use the *via* node:

```
# CORRECT -- now we specify that the path must traverse via ITM
model.add(components.Cavity("PRC", model.PRM.p2.o, model.ITM.p1.i))

print(model.PRC.draw())
```

```
o PRM.p2.o
└─o BS.p1.i
  └─o BS.p2.o
    └─o ITMAR.p1.i
      └─o ITMAR.p2.o
        └─o ITM.p1.i
          └─o ITM.p1.o
            └─o ITMAR.p2.i
              └─o ITMAR.p1.o
                └─o BS.p2.i
                  └─o BS.p1.o
                    └─o PRM.p2.i
```

```
/usr/local/lib/python3.13/site-packages/finesse/model.py:2255:␣
↪CavityUnstableWarning: cavity 'PRC' added to the model is unstable
  warn(
```

Showing the importance of including Cavity objects

Below is an example highlighting the importance of including Cavity objects in a model for HOM modelling. In this example, the detuning of the input mirror of a Fabry-Perot cavity is scanned whilst the power on transmission is detected.

```
import finesse

model = finesse.Model()
# define a configuration with no Cavity object present
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1

m ITM R=0.99 T=0.01 Rc=-2.5
s sCAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=2.5

# set a beam parameter with w0 = 1 mm, z-z0 = -0.3 m at the laser
gauss gL0 L0.p1.o w0=1m z=-0.3

# detect the transmitted power
pd trns ETM.p2.o

modes(maxtem=4)
""")

out_no_cav = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
```

Now we add a Cavity object to the model and run it again, we also display the mismatches that are now present in the system due to the manually set beam parameter at the laser and the beam parameters automatically set in the cavity based on its eigenmode. See Creating and detecting mode mismatches (page 335) for details on the `Model.mismatches_table()` method and more.

```
# source node is output of second port of ITM
# -> cavity path is then determined automatically from this
model.add(finesse.components.Cavity("FP", model.ITM.p2.o))

print(model.mismatches_table())

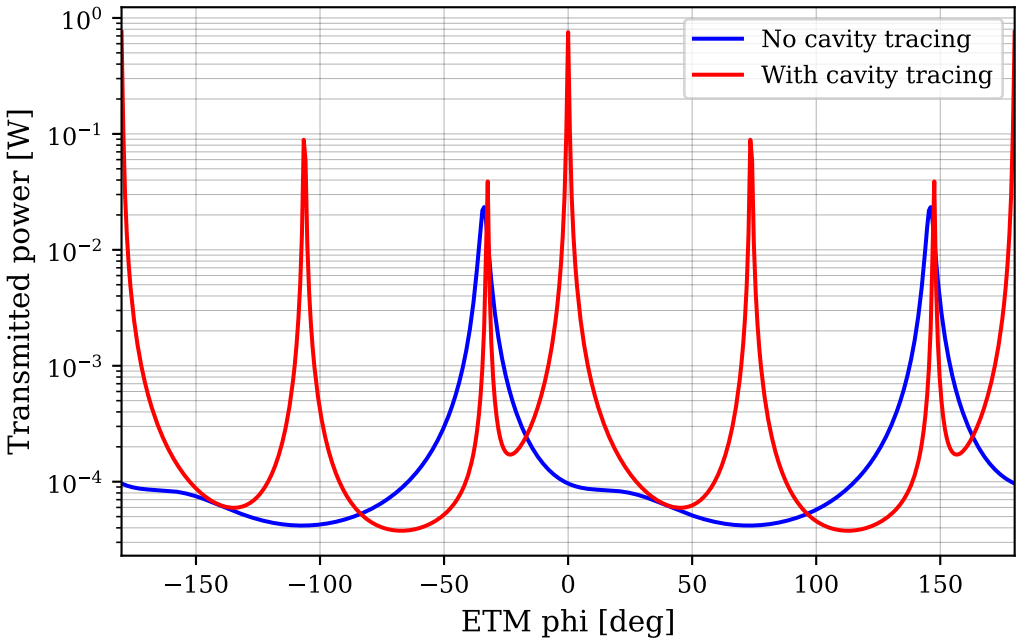
out_cav = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
```

```
┌──────────────────────┬──────────────┬──────────────┐
│       Coupling       │ Mismatch (x) │ Mismatch (y) │
╞══════════════════════╪══════════════╪══════════════╡
│ ITM.p1.i -> ITM.p2.o │       0.2460 │       0.2460 │
├──────────────────────┼──────────────┼──────────────┤
│ ITM.p1.i -> ITM.p1.o │       0.5437 │       0.5437 │
├──────────────────────┼──────────────┼──────────────┤
│ ITM.p2.i -> ITM.p1.o │       0.2460 │       0.2460 │
└──────────────────────┴──────────────┴──────────────┘
```

And then plot the results of each to show the difference:

```
import matplotlib.pyplot as plt
finesse.configure(plotting=True)

fig = plt.figure()
plt.semilogy(out_no_cav.x1, out_no_cav["trns"], label="No cavity tracing")
plt.semilogy(out_cav.x1, out_cav["trns"], label="With cavity tracing")
plt.xlabel("ETM phi [deg]")
plt.ylabel("Transmitted power [W]")
plt.legend();
```



From this plot you can see that with the cavity included we get the expected resonance peaks corresponding to the fundamental (TEM00) mode and the 02/20 and 04/40 modes; the latter arising due to the mismatch present within the system. Without the cavity object present, the transmitted power trace obtained is nonsensical.

Defining manual beam parameters

Before we demonstrated how we can use the cavity command to automatically set the beam parameters in a model. In certain cases you may also need to specify what beam parameter to use manually. For example, if you wanted the beam shape coming out of a laser to be of a particular shape and see what mismatch there is to a cavity.

Here we start with a cavity and defines the mode shape using the cavity command.

```
import finesse
import finesse.components as components
import matplotlib.pyplot as plt

finesse.init_plotting()

model = finesse.Model()
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1

m ITM R=0.99 T=0.01 Rc=-2.5
s sCAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=2.5
cav FP ITM.p2
# detect the transmitted power
pd trns ETM.p2.o
modes(maxtem=4)
""")

# run the simulation to get the ideal case
out0 = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
```

Next we use the `gauss` (page 195) command to set what shape the beam is at the output of the laser, this will add a new object to the model called *g1*.

```
model.parse("gauss g1 L0.p1.o w0=1m z=0")
out1 = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
```

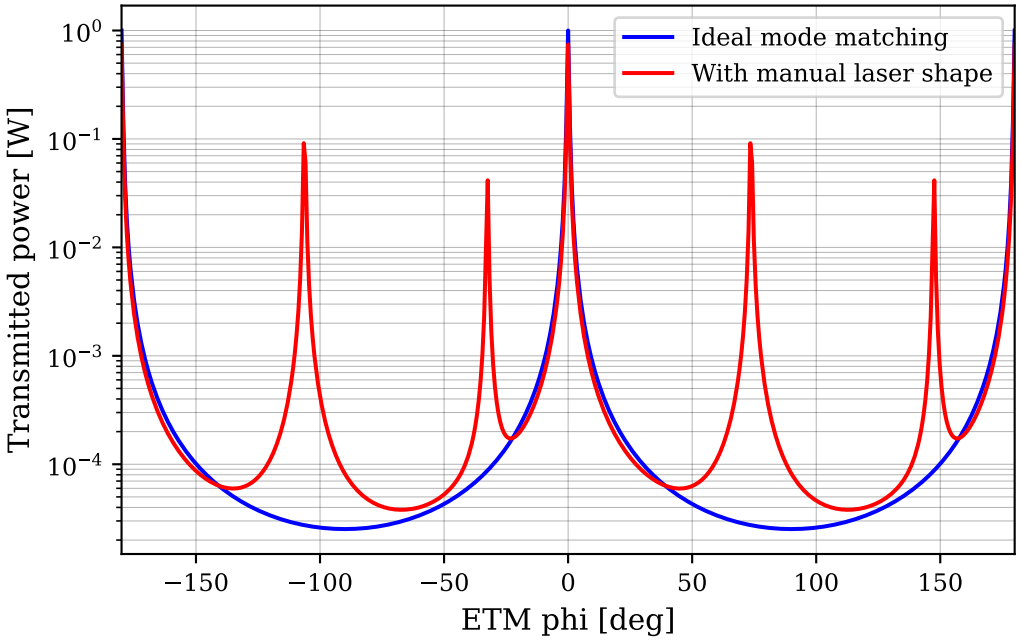
We can see what beam parameters are being set by a "gauss" command using:

```
model.gausses
```

```
{<OpticalNode L0.p1.o @ 0x78ce2ce3ef10>: <'g1' @ 0x78ce29a2a350 (Gauss)>}
```

Lastly we can plot the result and see that we have scattering into higher order modes due to the mismatch.

```
fig = plt.figure()
plt.semilogy(out0.x1, out0["trns"], label="Ideal mode matching")
plt.semilogy(out1.x1, out1["trns"], label="With manual laser shape")
plt.xlabel("ETM phi [deg]")
plt.ylabel("Transmitted power [W]")
plt.legend();
```



We can also look at the table of mismatches that are ocurring in the ABCD beam tracing calculations. With this we should see siginifcant mismatches around the ITM mirror.

```
model.mismatches_table()
```

```
<finesse.utilities.tables.NumberTable at 0x78ce296fd6d0>
```

Remove a manual beam parameter or cavity

To remove a manually set beam parameter or cavity command you can use the remove command with the name of the target:

```
model.remove('g1')
```

Defining manual beam parameters with the Python API

Above we defined the beam parameter using the KatScript command. To achieve the same result from using the Python API we need to look at the optical nodes q property. We can print these easily:

```
model = finesse.script.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1
```

(continues on next page)

(continued from previous page)

```
m ITM R=0.99 T=0.01 Rc=-2.5
s sCAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=2.5
cav FP ITM.p2
# detect the transmitted power
pd trns ETM.p2.o
modes(maxtem=4)
""")
model.beam_trace()
print(model.L0.p1.o.qx)
print(model.L0.p1.o.qy)
```

```
BeamParam(w0=581.96 um, z=-500 mm, w=650.66 um, Rc=-2.5 m)
BeamParam(w0=581.96 um, z=-500 mm, w=650.66 um, Rc=-2.5 m)
```

We can also easily set them as well:

```
# qx and qy can be set separately or you can set .q to set both at the same time
model.L0.p1.o.q = finesse.BeamParam(w0=1e-3, z=0)
model.beam_trace()
```

```
<BeamTraceSolution of beam_trace @ 0x78ce29732040 children=0>
```

Setting the beam parameter this way will not automatically perform a new beam trace through the model. If you want to query the new beam parameters due to your change you must run a `model.beam_trace()`.

Doing the above will have the same effect as using a gaussian beam command as seen before. If you check you will see a `finesse.components.gauss.Gauss` is actually added to the model with an auto-generated name and can be removed if needed.

```
model.gausses
```

```
{<OpticalNode L0.p1.o @ 0x78ce29815700>: <'gL0_p1_o' @ 0x78ce296fce10 (Gauss)>}
```

6.5.3 Creating and detecting mode mismatches

Mode mismatches in a FINESSE model arise during the beam tracing when a beam parameter set at a node is not equal to the beam parameter at a predecessor node transformed by the associated components ABCD matrix. See Transformation of the beam parameter (page 303) for the transformation formula and a table of component ABCD matrices.

These mismatches occur as a result of:

- manually set beam parameters at nodes which are mismatched from each other or any cavities in the model when propagating them,
- and / or multiple cavities in the model which are not matched to each other.

The former case here can be generated in a convenient way using `Model.create_mismatch()` and both cases can be detected with the methods `Model.mismatches_table()` and `Model.detect_mismatches()`. Before these methods are covered, however, it is important to define how FINESSE computes a figure-of-merit for a mismatch value.

Defining a figure of merit

The mismatch between two beam parameters, q_1 and q_2, is computed with `BeamParam.mismatch()` and is defined as,

$$\mathcal{M} = \frac{|q_1 - q_2|^2}{|q_1 - q_2^*|^2}.$$

This quantity gives a value $\mathcal{M} \in [0, 1]$ where 0 represents no mismatch (i.e. perfectly matched beam parameters) and 1 represents total mismatch.

Creating a custom mismatch

The `Model` class in FINESSE exposes a method `Model.create_mismatch()` which allows you to "inject" an arbitrary mode mismatch at any node in a configuration. As a beam parameter is described completely by the waist-size w_0 and distance to waist z, this method provides separate mismatch arguments for these two parameters in terms of the percentage mismatch you want to inject.

An example of this is shown below where we define an initially mode-matched cavity model and then inject a mode-mismatch at the first surface of the input mirror.

```
import finesse
finesse.configure(plotting=True)

model = finesse.Model()
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1

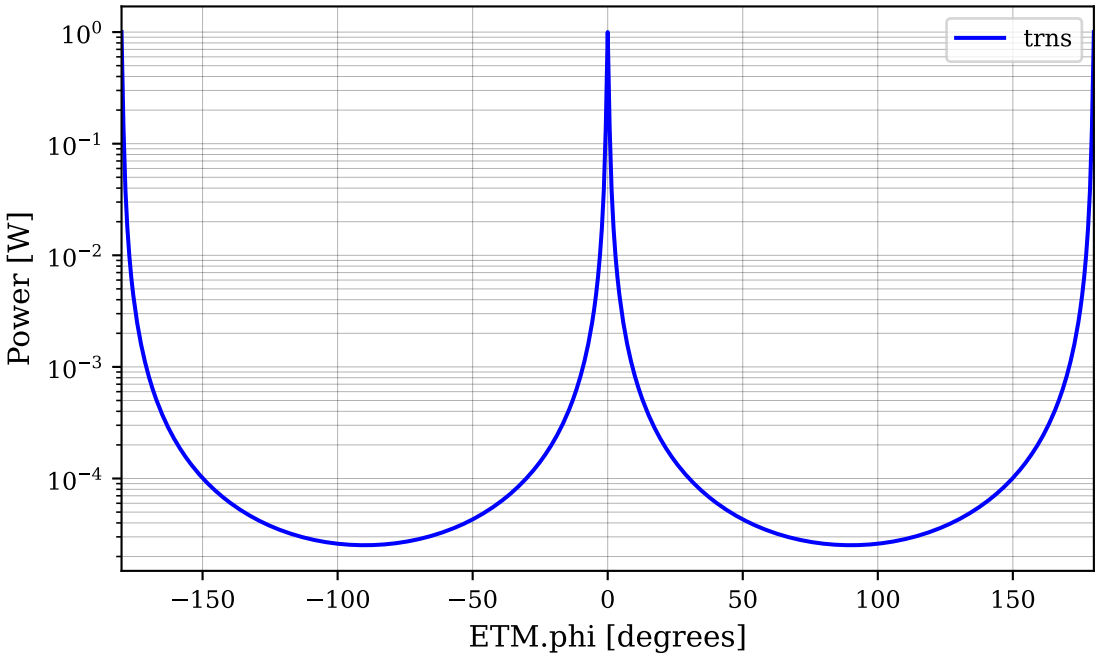
m ITM R=0.99 T=0.01 Rc=-2.5
s sCAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=2.5

# create the cavity object
cav FP ITM.p2.o

pd trns ETM.p2.o

# only care about even modes as we're just looking
# at mismatches, so select even modes up to order 4
modes(even, 4)
""")

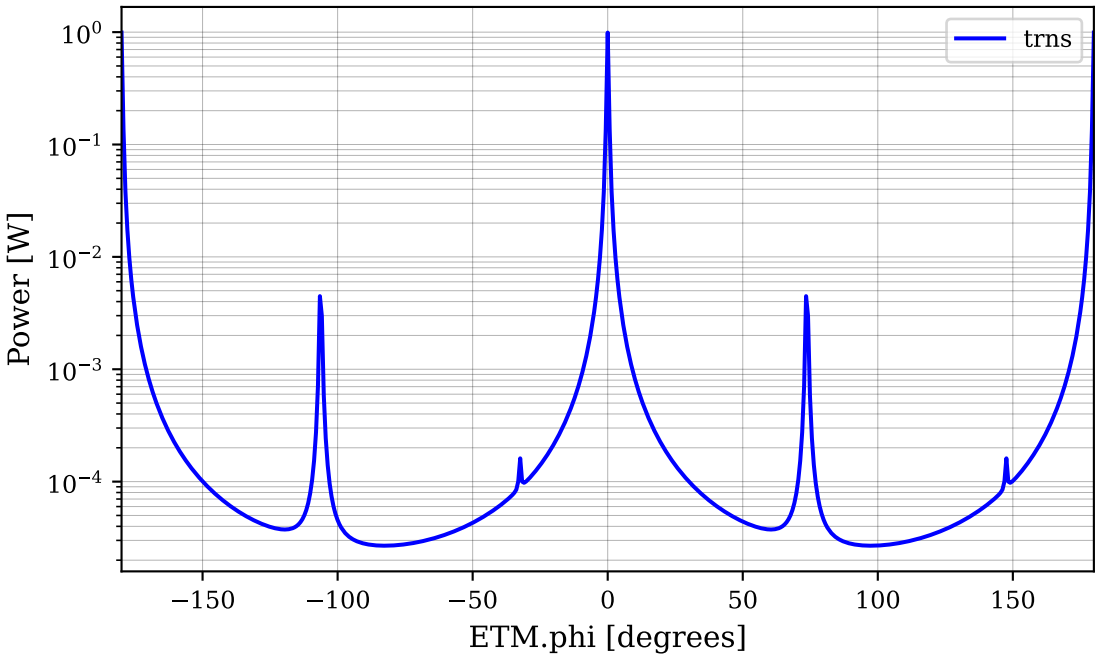
out_matched = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
out_matched.plot(logy=True);
```



The above plot shows the cavity scan with no mismatches present - as there is just a single Cavity object present with no manually set beam parameters anywhere. Now we create a mismatch of 10% in w_0 and -8% in z at ITM:

```
mm_gauss = model.create_mismatch(model.ITM.p1.i, w0_mm=10, z_mm=-8)

out_mismatched = model.run("xaxis(ETM.phi, lin, -180, 180, 500)")
out_mismatched.plot(logy=True);
```



From this plot we can see that we now get scattering into the 02 and 04 modes as we would expect from introducing a mode mismatch.

Note that the `Model.create_mismatch()` method returns the `Gauss` object (i.e. user-set beam parameter object) which was created / modified in the method. We will use the return value (*mm_gauss*) of this example later on.

Detecting mode mismatches

As mentioned previously, there are two Model methods for detecting / displaying mismatches - `Model.detect_mismatches()` and `Model.mismatches_table()`. The former is the function which performs the actual computations whilst the latter can be used to display a nicely formatted table of the mismatch values present.

Detect mismatches method

The `Model.detect_mismatches()` method is very simple to use and only takes a single optional argument which is a flag specifying whether to use the previous beam trace results (stored in `Model.last_trace`) or perform a new beam trace and compute the mismatches from that. This method will return a dictionary of surface to mismatch couplings where the mismatch couplings item is itself a dictionary of the optical node couplings mapping to their mismatch values.

This is easier to visualise using the `Model.mismatches_table()` method but this method is also exposed as a programmatic interface for accessing specific mismatches if you require them.

Print mismatches method

`Model.mismatches_table()` calls the above method and prints the resulting dictionary in a tabulated format for easier viewing of the mismatches. Taking the example above where we created mismatches in a cavity model, we can then print the mismatches to verify them:

```
print(model.mismatches_table())
```

```
╒═══════════════════════╤═══════════════╤═══════════════╕
│       Coupling        │ Mismatch (x)  │ Mismatch (y)  │
╞═══════════════════════╪═══════════════╪═══════════════╡
│ ITM.p1.i -> ITM.p2.o  │        0.0094 │        0.0094 │
├───────────────────────┼───────────────┼───────────────┤
│ ITM.p1.i -> ITM.p1.o  │        0.0293 │        0.0293 │
├───────────────────────┼───────────────┼───────────────┤
│ ITM.p2.i -> ITM.p1.o  │        0.0094 │        0.0094 │
╘═══════════════════════╧═══════════════╧═══════════════╛
```

Detecting coupling coefficients directly

Another way to detect mismatches indirectly is to probe the coupling coefficients of a scattering matrix during a simulation run. This will provide you with complex data for the specific mode coupling that you are probing, over any parameter scan you are performing.

A simple example of this is shown below, where we take the cavity model from above and scan over the waist size of the "gauss command" added by the `Model.create_mismatch()` call performed earlier.

Trace priority

6.5.4 Capturing beam images with camera objects

The `finesse.detectors.camera` module includes a number of Camera objects for probing single pixels, slices or full images of beam(s). These cameras are split into categories according to these dimensions as well as another two categories corresponding to whether it is detecting the field amplitude and phase (i.e. a sub-class of `ComplexCamera`) or the pixel intensity (ie. a sub-class of `CCDCamera`).

Equations describing the quantities computed by each of these two latter categories can be found in `finesse.detectors.compute.camera`.

Complex (field) cameras

Listed below are the camera objects which are used for probing the per-pixel field amplitude and phase for a beam at a specific frequency.

`finesse.detectors.camera.FieldCamera`(*args, ...)	Camera for detecting the full image of the beam in terms of amplitude and phase.
`finesse.detectors.camera.FieldScanLine`(...)	Camera for detecting a slice of the beam in terms of amplitude and phase.
`finesse.detectors.camera.FieldPixel`(*args, ...)	Camera for detecting a single pixel of the beam in terms of the amplitude and phase.

These are listed in order of decreasing dimensions.

CCD cameras

Listed below are the camera objects which are used for probing the per-pixel beam intensity of all (non-audio) fields present at the detection node.

`finesse.detectors.camera.CCD`(*args, **kwargs)	Camera for measuring the intensity of a beam, $I = \|E(x, y)\|^2$, where the unscaled x and y coordinate arrays used are `finesse.detectors.camera.Image.xdata` and `finesse.detectors.camera.Image.ydata`, respectively.

continues on next page

Table 6.4 – continued from previous page

`finesse.detectors.camera.CCDScanLine`(*args, ...)	Camera for measuring the intensity of a beam, $I = \|E(x, y)\|^2$, along a 1D slice.
`finesse.detectors.camera.CCDPixel`(*args, ...)	Camera for measuring the intensity of a beam, $I = \|E(x, y)\|^2$, at a single point.

These are listed in order of decreasing dimensions.

Note

All camera classes take the optional argument `w0_scaled` in their constructors. This flag is True by default and indicates that all x and y coordinates should be scaled by the waist size of the beam. Any outputs (including plots) will then be in units of the beam waist. By setting this flag to False, you can prevent this behaviour and instead have the x and y axes in units of metres.

Detecting a single pixel

A single pixel of a beam profile can be detected with `Pixel` type cameras. The return type of this detector will be a single number (complex for a field, real for a CCD). A simple example below shows how one can use a `FieldPixel` (page 177) to detect, for example, the central pixel (at $x = y = 0$) of a beam distribution on reflection from a mirror whilst tilting the mirror in the x-direction.

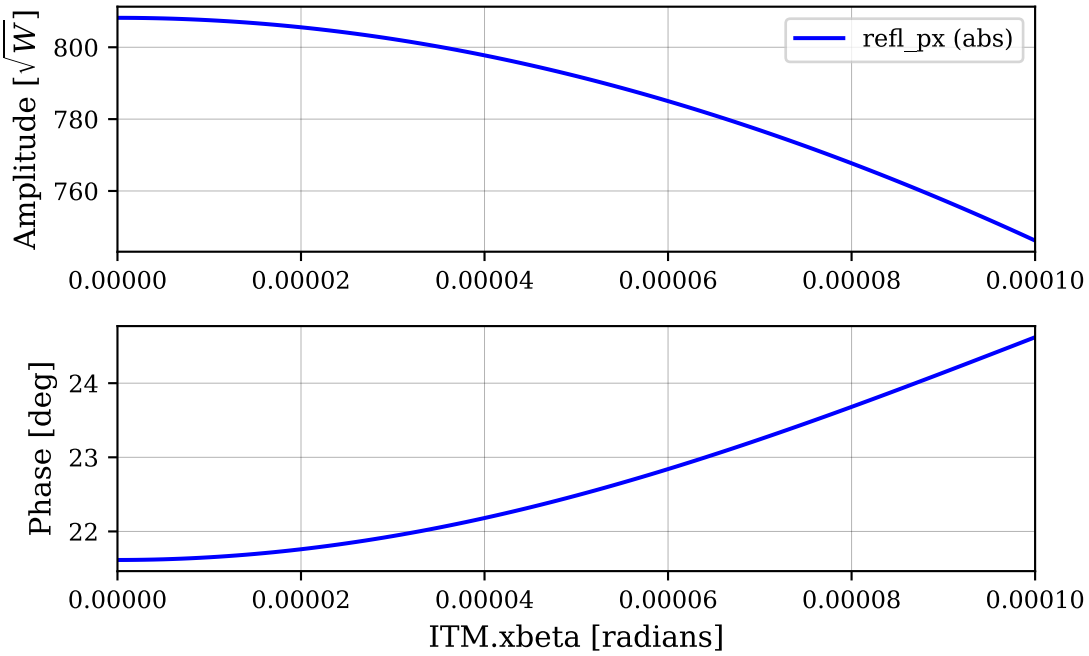
```
import finesse
finesse.configure(plotting=True)

model = finesse.Model()
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1 L=1
m ITM R=0.99 T=0.01
gauss gL0 L0.p1.o q=(-2+3j)

# a FieldPixel detector - defaults to x = y = 0 and f = 0
fpx refl_px ITM.p1.o

modes(maxtem=2)
"""
)

out = model.run("xaxis(ITM.xbeta, lin, 0, 100u, 100)")
out.plot();
```



Note that, as implied in the code above, the constructor for `FieldPixel` (page 177) uses default values of zero for x and y (so that the central pixel is the default), and it probes at a frequency offset of 0 Hz (carrier) by default too.

Probing a slice of a beam

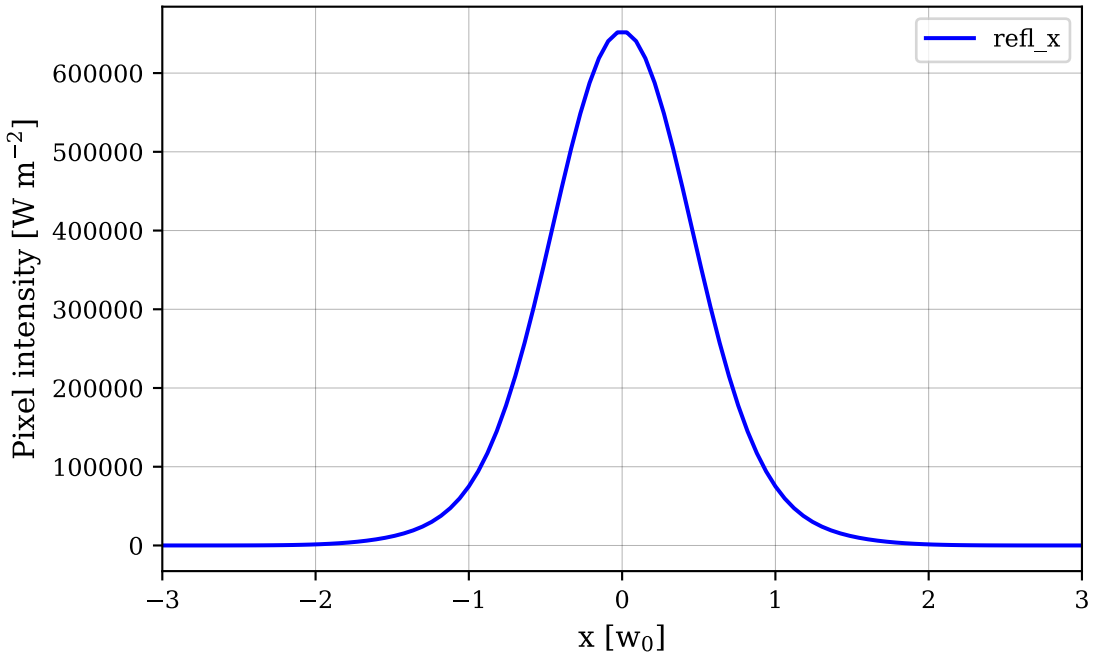
Slices of a beam profile are detected with `ScanLine` type cameras. These detectors return a 1D `numpy.ndarray` where the scanning axis is specified at construction time via the `direction` argument. An example is shown below, where the x-axis of a beam image, in terms of the intensity, is computed and plotted using a `CCDScanLine` (page 174) detector.

```
model = finesse.Model()
model.parse("""
l L0 P=1
s s0 L0.p1 ITM.p1 L=1
m ITM R=0.99 T=0.01
gauss gL0 L0.p1.o q=(-2+3j)

# A CCDScanLine detecting x in [-3, 3] with y = 0 by default
ccdline refl_x ITM.p1.o xlim=3 npts=100

modes(maxtem=2)
"""
)

out = model.run("noxaxis()")
out.plot();
```



A single value can be specified for x, y when either of these is the scanning axis - this will then create the scanning axis array ranging from, e.g., $-|x|$ to $+|x|$ if x is the axis to be scanned. The value for the non-scanned axis will take on a default of zero if not specified.

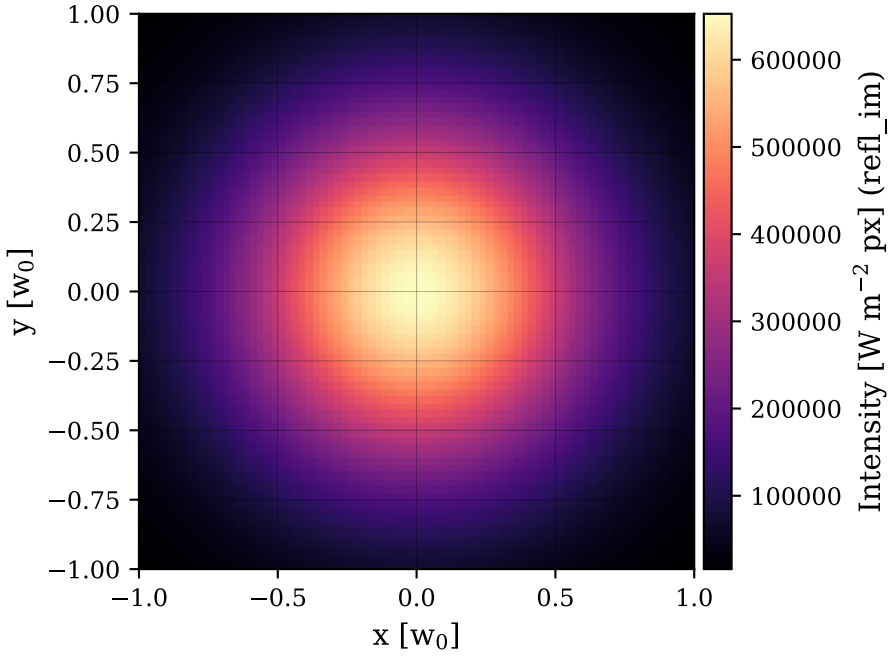
Capturing a full beam image

Full two-dimensional beam profiles can be captured with `Image` type cameras. Detectors of this type return a 2D `numpy.ndarray`. An example is shown below, using the same model as above but now taking the full beam distribution image in terms of the intensity with a `CCD` (page 174) detector.

```
# Remove the CCDScanLine detector from above
model.remove(model.refl_x)

# Add a CCD with x in [-1, 1] and y in [-1, 1] with 300 pixels per axis
model.parse("""
ccd refl_im ITM.p1.o xlim=1 ylim=1 npts=80
""")

out = model.run("noxaxis()")
out.plot();
```



Producing animated beam images

Warning

Depending on your working environment, animations may not be enabled for displaying. If you are working in a Jupyter Notebook you may need to use *%matplotlib notebook* or *%matplotlib qt* to set up your environment, other wise a single fixed image will be displayed.

Animated plots of beam distributions can be created by adding a full image detector (i.e. a `CCD` (page 174) or `FieldCamera` (page 175)) to a model, then scan over a model parameter (using the `xaxis` command for example) to produce 3D data which will be animated using the parameter scan axis when plotting via `ArraySolution.plot()`.

An example of this is shown below, where a distorted beam is injected into a Fabry-Perot cavity. A cavity scan is then performed whilst a CCD captures the beam image, at each mirror detuning, to produce a mode scan animation.

```
model = finesse.Model()
model.parse(
"""
l L0 P=1

s s0 L0.p1 ITM.p1

m ITM R=0.99 T=0.01 Rc=-10
s CAV ITM.p2 ETM.p1 L=1
m ETM R=0.99 T=0.01 Rc=10

cav FP ITM.p2.o

tem(L0, n=0, m=1, factor=0.9)
tem(L0, m=1, n=0, factor=0.9)
tem(L0, n=0, m=2, factor=0.8)
tem(L0, n=1, m=1, factor=0.4)
tem(L0, n=2, m=0, factor=0.5)

ccd trns ETM.p2.o xlim=2.5 ylim=2.5 npts=80
```

(continues on next page)

(continued from previous page)

```
modes(maxtem=2)
"""
)

out = model.run("xaxis(ITM.phi, lin, 0, 180, 60)")
figures, animations = out.plot(cmap="hot")
```

```
<IPython.core.display.HTML object>
```

6.5.5 Beam propagation

Finesse provides a tool-set of beam tracing and mode matching functions which can be used outside of a simulation context. These functions allow for arbitrary beam propagation over any optical path of a given configuration - supporting both numeric and symbolic calculations.

The primary function recommended for most beam propagation analyses is `propagate_beam()`, also callable from a model via `Model.propagate_beam()`.

The 'propagate_beam' function

As noted in the API documentation linked above, the `propagate_beam()` function can take any optical path of a defined model and propagate an arbitrary beam from the start node to the end node of the path.

Here we will use a proposed, preliminary design [35] of the Einstein Telescope Low-Frequency (ET-LF) signal recycling cavity (SRC), complete with arm telescopes, to highlight how this function can be used. The model is defined below via Finesse kat-file syntax (see KatScript (page 125)).

```
import finesse
finesse.configure(plotting=True)

model = finesse.Model()
model.parse("""
### L0 -> BS -> YARM of ET-LF
# input
l L0 P=1
s l0 L0.p1 BS.p1 L=10

# Main beam splitter
bs BS T=0.5 L=37.5u alpha=60
s BSsub1 BS.p3 BSAR1.p1 L=0.07478 nr=nsilica
s BSsub2 BS.p4 BSAR2.p1 L=0.07478 nr=nsilica
bs BSAR1 R=50u L=0 alpha=-36.6847
bs BSAR2 R=50u L=0 alpha=36.6847

# Y arm telescope
s lBS_ZM1 BS.p2 ZM1.p1 L=70
bs ZM1 T=250u L=37.5u Rc=-50
s lZM1_ZM2 ZM1.p2 ZM2.p1 L=50
bs ZM2 T=0 L=37.5u Rc=-82.5
s lZM2_ITMlens ZM2.p2 ITM_lens.p1 L=52.5

lens ITM_lens 75
s lITM_th2 ITM_lens.p2 ITMAR.p1 L=0

# Y arm input mirror
m ITMAR R=0 L=20u
s ITMsub ITMAR.p2 ITM.p1 L=0.2 nr=nsilicon
m ITM T=7000u L=37.5u Rc=-5580

# Y arm length
```

(continues on next page)

(continued from previous page)

```
s l_arm ITM.p2 ETM.p1 L=10k

# Y arm end mirror
m ETM T=6u L=37.5u Rc=5580
s ETMsub ETM.p2 ETMAR.p1 L=0.2 nr=nsilicon
m ETMAR R=0 L=500u

# SRM
s lBS_SRM BSAR2.p3 SRM.p1 L=10

m SRM T=0.2 L=0 Rc=-9410
s SRMsub SRM.p2 SRMAR.p1 L=0.0749 nr=nsilicon
m SRMAR R=0 L=50n

# cavities
cav cavARM ITM.p2
cav cavSRC SRM.p1 ITM.p1.i

var nsilica 1.44963098985906
var nsilicon 3.42009

lambda(1550n)
""")
```

```
<finesse.model.Model at 0x7e9307fce900>
```

We can call `propagate_beam()` using any two optical nodes as the targets, in this case we propagate the beam from the ITM to the SRM.

```
ps = model.propagate_beam(model.ITM.p1.o, model.SRM.p1.i)
```

Note that we can use `Port` objects as the end points too, in which case the *from_node* argument is deduced to be the output optical node of the port and *to_node* will be the input optical node of the other port. In our example, the line below is equivalent to the above.

```
ps = model.propagate_beam(model.ITM.p1, model.SRM.p1)
```

By specifying the two end nodes / ports, the optical path between these points will be determined in the function. This is generally a fast operation, however it is likely the slowest of all operations carried out by the `propagate_beam()` function as a whole. Therefore, in the rare case where this function is being called in a large loop, it makes sense to pre-compute the optical path and pass this instead. An example of this is shown below, again equivalent to the above.

```
ITM_TO_SRM = model.path(model.ITM.p1, model.SRM.p1)
ps = model.propagate_beam(path=ITM_TO_SRM)
```

Sometimes, the shortest optical path between two nodes is not the path you want to trace. For instance in the coupled cavity model below, the shortest path goes only through the first cavity. If you want to trace the path through both cavities, you can use the `via_node` argument. Note that for more complicated paths, you may also supply a sequence of nodes to the `via_node` argument.

```
from finesse import Model

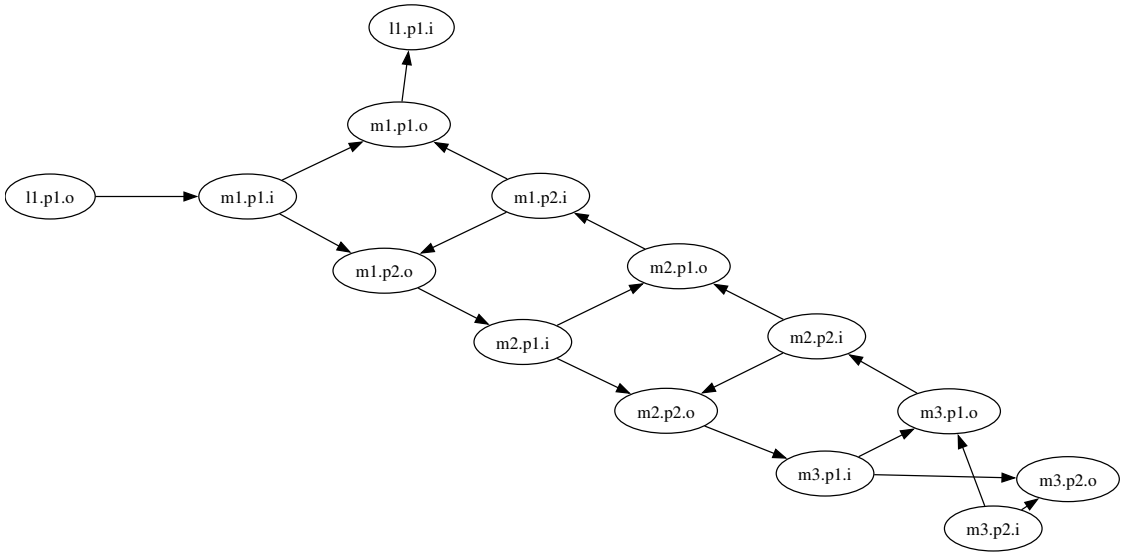
coupled_cav = Model("""
l l1
gauss g1 l1.p1.o w0=1 z=1
s s1 l1.p1 m1.p1
m m1
s cav1 m1.p2 m2.p1
m m2
s cav2 m2.p2 m3.p1
```

(continues on next page)

(continued from previous page)

```
m m3
""")

coupled_cav.plot_graph(network_type="optical")
```




```
short_sol = coupled_cav.propagate_beam(from_node="m1.p2.o", to_node="m1.p2.i")
print([n.full_name for n in short_sol.nodes])
```

```
['m1.p2.o', 'm2.p1.i', 'm2.p1.o', 'm1.p2.i']
```

```
long_sol = coupled_cav.propagate_beam(
    from_node="m1.p2.o", to_node="m1.p2.i", via_node="m3.p1.i"
)
print([n.full_name for n in long_sol.nodes])
```

```
['m1.p2.o', 'm2.p1.i', 'm2.p2.o', 'm3.p1.i', 'm3.p1.o', 'm2.p2.i', 'm2.p1.o', 'm1.p2.i']
```

One thing you may note from this type of call to `propagate_beam()` is that no input beam parameter (*q_in*) argument has been specified. This means that this argument will be automatically deduced from the model via an internal `Model.beam_trace()` call - where the beam parameter at the input node of the path is then accessed and used as the *q_in* argument. A custom beam parameter can be used by specifying a *q_in* argument explicity (this can be a complex number or a `BeamParam` instance).

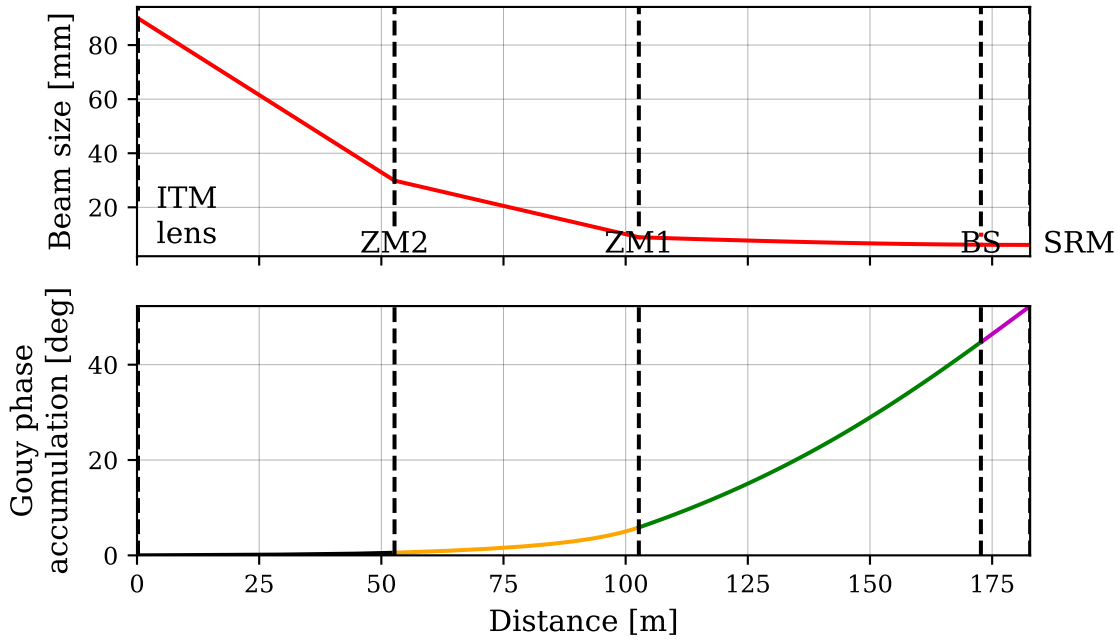
Another aspect to take into account is that `propagate_beam()` operates on a single plane (i.e. 'x' for the tangential plane, 'y' for the sagittal plane). By default, the tangential plane is used. This can be changed to sagittal by specifying `direction='y'` as an argument.

The return value of `propagate_beam()` is a `PropagationSolution` instance. We will look at the various properties and methods this class provides below, in the context of our example.

Plotting properties of the beam propagation

One of the most common use-cases is to plot a beam trace, in order to see how, in particular, the beam size and accumulated Gouy phase evolve over a path. This is as simple as calling `PropagationSolution.plot()` on the resulting solution. An example is given below for the call in the above section.

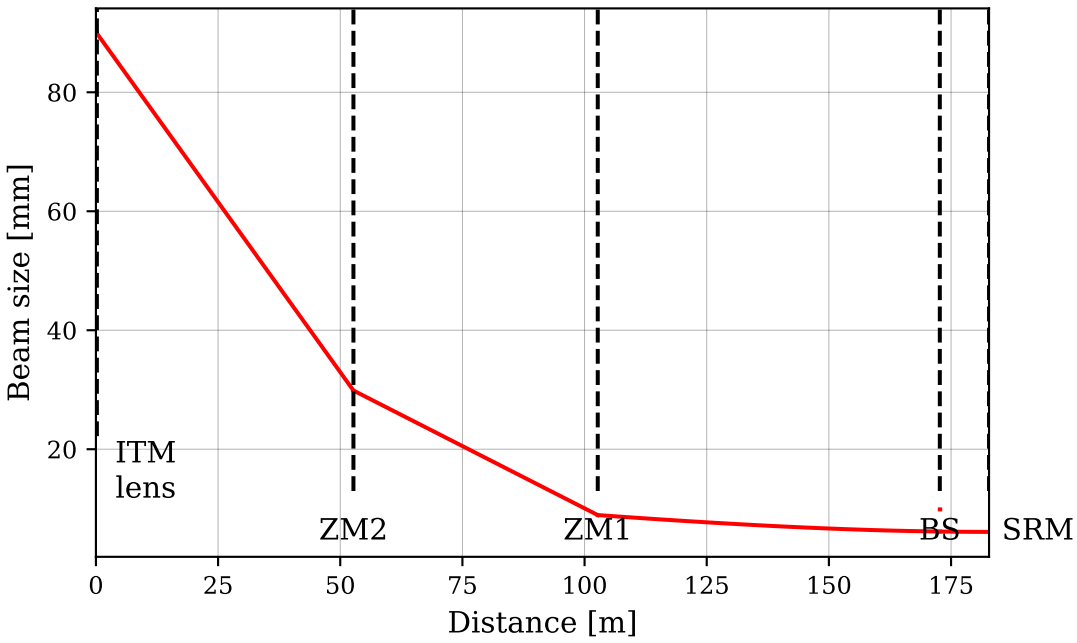
```
ps.plot(
    name_xoffsets={"ITM_lens": 10, "SRM": 10},
    name_yoffsets={"ITM_lens": 10},
);
```



The *name_xoffsets* and *name_yoffsets* arguments are optional and provide a way of shifting the positions of the component names on the figure (in terms of data coordinates) to avoid clashes.

Just the beam-sizes, for example, can also be plotted.

```
ps.plot_beamsizes(
    name_xoffsets={"ITM_lens": 10, "SRM": 10},
    name_yoffsets={"ITM_lens": 10},
);
```



The wavefront curvature is not plotted by default with a call to `PropagationSolution.plot()` but this can be plotted separately too with `PropagationSolution.plot_curvatures()` or along with the others via giving

`"all"` as the first argument to this method.

Printing propagation data

As well as plotting, one can also display useful data associated with a beam propagation by simply printing the solution object. For our example here, this results in:

```
print(ps)
```

```
┌──────────────╥──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──
│              ‖    z     │    w0    │    zr    │    w     │   RoC    │    S     │␣
↪Acc. Gouy │          q          │
╞══════════════╬══════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══
│   ITM.p1.o   ‖      0 m │ 8.9093 mm │  550.22 m │ 89.907 mm │    5.58 km │  179.21 uD │ ␣
↪     0° │ 5525.206 + 550.225j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│  ITMAR.p2.i  ‖   200 mm │ 8.9093 mm │  550.22 m │  89.91 mm │  5.5802 km │  179.21 uD │ ␣
↪204.5u° │ 5525.406 + 550.225j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│  ITMAR.p1.o  ‖   200 mm │ 8.9093 mm │  160.88 m │  89.91 mm │  1.6316 km │   612.9 uD │ ␣
↪204.5u° │ 1615.573 + 160.880j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│ ITM_lens.p2.i ‖   200 mm │ 8.9093 mm │  160.88 m │  89.91 mm │  1.6316 km │   612.9 uD │ ␣
↪204.5u° │ 1615.573 + 160.880j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│ ITM_lens.p1.o ‖   200 mm │ 431.39 um │ 377.18 mm │  89.91 mm │  -78.614 m │  -12.72 mD │ ␣
↪204.5u° │    -78.612 + 0.377j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│   ZM2.p2.i   ‖    52.7 m │ 431.39 um │ 377.18 mm │ 29.868 mm │  -26.117 m │ -38.289 mD │ ␣
↪552.87m° │    -26.112 + 0.377j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│   ZM2.p1.o   ‖    52.7 m │ 1.1751 mm │  2.7989 m │ 29.868 mm │  -71.193 m │ -14.046 mD │ ␣
↪552.87m° │    -71.083 + 2.799j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│   ZM1.p2.i   ‖   102.7 m │ 1.1751 mm │  2.7989 m │ 8.9293 mm │  -21.454 m │ -46.611 mD │ ␣
↪5.8603° │    -21.083 + 2.799j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│   ZM1.p1.o   ‖   102.7 m │ 6.1019 mm │  75.465 m │ 8.9293 mm │  -151.26 m │ -6.6111 mD │ ␣
↪5.8603° │   -80.625 + 75.465j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│    BS.p2.i   ‖   172.7 m │ 6.1019 mm │  75.465 m │ 6.1621 mm │  -546.61 m │ -1.8295 mD │ ␣
↪44.739° │   -10.625 + 75.465j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│    BS.p4.o   ‖   172.7 m │ 9.7866 mm │  281.41 m │ 9.8832 mm │ -2.0383 km │  -490.6 uD │ ␣
↪44.739° │  -39.622 + 281.411j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│  BSAR2.p1.i  ‖ 172.77 m │ 9.7866 mm │  281.41 m │ 9.8828 mm │  -2.042 km │ -489.71 uD │ ␣
↪44.754° │  -39.547 + 281.411j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│  BSAR2.p3.o  ‖ 172.77 m │ 6.1019 mm │  75.465 m │ 6.1618 mm │   -547.6 m │ -1.8262 mD │ ␣
↪44.754° │   -10.605 + 75.465j │
├──────────────╫──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──
│   SRM.p1.i   ‖ 182.77 m │ 6.1019 mm │  75.465 m │ 6.1021 mm │   -9.41 km │ -106.27 uD │ ␣
↪52.294° │    -0.605 + 75.465j │
└──────────────╨──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──
```

A matrix of distances between all the optics in the propagation can also be printed using `PropagationSolution.distances_matrix_table()`:

```
print(ps.distances_matrix_table())
```

```
┌──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┬──────────┐
│          ‖    ITM   │   ITMAR  │ ITM_lens │    ZM2   │    ZM1   │    BS    │  BSAR2   │    SRM   │
╞══════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══════════╪══════════╡
│   ITM    ‖      0 m │   200 mm │   200 mm │   52.7 m │   102.7 m │   172.7 m │ 172.77 m │ 182.77 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│  ITMAR   ‖   -200 mm │      0 m │      0 m │   52.5 m │   102.5 m │   172.5 m │ 172.57 m │ 182.57 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│ ITM_lens ‖   -200 mm │      0 m │      0 m │   52.5 m │   102.5 m │   172.5 m │ 172.57 m │ 182.57 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│   ZM2    ‖   -52.7 m │   -52.5 m │   -52.5 m │      0 m │      50 m │     120 m │ 120.07 m │ 130.07 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│   ZM1    ‖  -102.7 m │  -102.5 m │  -102.5 m │     -50 m │      0 m │      70 m │ 70.075 m │ 80.075 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│    BS    ‖  -172.7 m │  -172.5 m │  -172.5 m │    -120 m │     -70 m │      0 m │ 74.78 mm │ 10.075 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│  BSAR2   ‖ -172.77 m │ -172.57 m │ -172.57 m │ -120.07 m │ -70.075 m │ -74.78 mm │      0 m │     10 m │
├──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┼──────────┤
│   SRM    ‖ -182.77 m │ -182.57 m │ -182.57 m │ -130.07 m │ -80.075 m │ -10.075 m │    -10 m │      0 m │
└──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┴──────────┘
```

Accessing beam properties

`PropagationSolution` provides various methods for accessing physical properties of the beam at any point along the computed path. All of these relevant methods derive from `PropagationSolution.q()` (i.e. the beam parameter), so here we will explore the options available for accessing `q` at different locations.

The simplest case is to provide an `OpticalNode` instance to get the beam parameter at this node:

```
print(ps.q(model.BS.p2.i))
```

```
BeamParam(w0=6.1019 mm, z=-10.625 m, w=6.1621 mm, Rc=-546.61 m)
```

in the example above we grab the beam parameter at the input node of the second port of the beam splitter (coming from the ZM1 optic). Whilst `model.BS.p2.o` (i.e. the output node of this port) is technically not traced in the given path, we can still access the beam parameter at this node as the beam tracing assumes that opposite node beam parameters are the reverse of the forward node (i.e. $-q^*$):

```
print(ps.q(model.BS.p2.o))
```

```
BeamParam(w0=6.1019 mm, z=10.625 m, w=6.1621 mm, Rc=546.61 m)
```

One can also use the string representation of an optical node:

```
print(ps.q("BS.p2.i"))
```

```
BeamParam(w0=6.1019 mm, z=-10.625 m, w=6.1621 mm, Rc=-546.61 m)
```

Note

Any beam parameter value returned by `PropagationSolution.q()` will be a `BeamParam` instance, meaning all the various properties of this can be accessed as usual. `PropagationSolution` does provide some shortcuts for the key parameters (such as beam-size with `PropagationSolution.w()`), however, so the choice is up to the user as to whether to access like this:

```
ps.q(model.BS.p2.i).w
```

```
np.float64(0.0061620647495966045)
```

or like this:

```
ps.w(model.BS.p2.i)
```

```
np.float64(0.0061620647495966045)
```

Obtaining cumulative Gouy phases

It can be useful to see accumulated Gouy phases over segments of the traced path too, and `PropagationSolution` provides methods to obtain these. These phases are always given in degrees.

To obtain the total accumulated Gouy phase over the full traced path one can simply do:

```
ps.total_acc_gouy
```

```
52.29424393023219
```

Accumulated Gouy phase over a specific sequence of spaces can be retrieved with `PropagationSolution.acc_gouy()`, e.g:

```
ps.acc_gouy("lZM2_ITMlens", "lZM1_ZM2")
```

```
5.860050120196138
```

gives the Gouy phase accumulated from the ITM lens to ZM1 (or vice-versa). And finally, the Gouy phase accumulated up to a specific point can be obtained with `PropagationSolution.acc_gouy_up_to()`. This gets the cumulative Gouy phase from the starting node of the propagation up to the specified point. Similarly to accessing beam properties, this point can be an `OpticalNode` instance or a component or name of a component. For example, the below retrieves the accumulated Gouy phase from the ITM to BS:

```
ps.acc_gouy_up_to("BS")
```

```
44.739318811593286
```

Symbolic beam propagation

The default behaviour of `propagate_beam()` is to use the current values of each parameter, returning non-symbolic data in the `PropagationSolution` instance. Another way to use this function is to switch on the *symbolic* flag - resulting in symbolic expressions for all beam parameters, Gouy phases and ABCD matrices at each point in the solution.

Using the example file from above again, we can simply call:

```
ps_sym = model.propagate_beam(path=ITM_TO_SRM, symbolic=True)
```

to compute a symbolic propagated beam solution. Accessing beam properties (such as the beam parameter via `PropagationSolution.q()`) will now return a symbolic expression rather than just a numeric value. Note that in the case of `PropagationSolution.q()`, a `BeamParam` instance is still returned but this will be a symbolic beam parameter (as indicated by the `BeamParam.symbolic` flag).

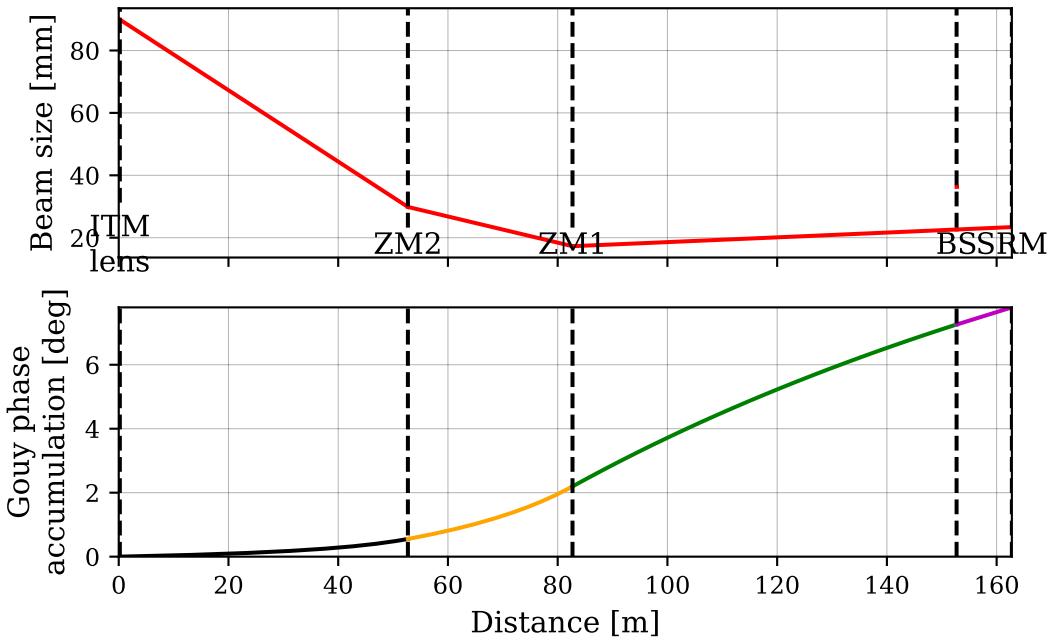
Symbolic beam propagation can result in very long symbolic expressions. Rudimentary simplification routines are provided in Finesse but for large models keeping every parameter symbolic is unnecessary. If you only need specific parameters to be kept as symbols then you can specify which to keep, for example, *symbolic=('ITMX.Rcx',)* or *symbolic=('ITMX.Rcx', 'ITMYlens.f')*. If simplification of symbolic beam propagation is required you must specify

simplify=True, it does not happen by default. This is because simplifying when using every single parameter takes too long and is to complicated to work with.

Plotting symbolic beam propagations

Plotting of the solution is supported, with the added option of substituting parameters in the `PropagationSolution.plot()` call. An example is given below, where the RoC of ZM1 is changed from $R_c = -50$ m to $R_c = -70$ m, and the distance between the two telescope mirrors (ZM1 , ZM2) is reduced to 30 metres:

```
ps_sym.plot(subs={model.ZM1.Rcx: -70, model.lZM1_ZM2.L: 30});
```



If `subs` is not given, then the current value of each parameter will be used.

Animation of a symbolic `PropagationSolution` is also supported via the method `PropagationSolution.animate()`. This method expects at least one array-like parameter substitution in the `subs` dict in order to perform the animation over this parameter scan. For example, we can see how the beam sizes and accumulated Gouy phases change as we scan over the telescope length via:

```
import numpy as np

# Vary telescope length from 20 m to 70 m
tel_zs = np.linspace(20, 70, 60)
fig, axes, anim = ps_sym.animate(
    {model.lZM1_ZM2.L: tel_zs}, interval=50,
)
```

```
<IPython.core.display.HTML object>
```

Evaluating symbolic beam properties

The true power of symbolic `PropagationSolution` objects comes from the ability to obtain symbolic expressions for any geometric property of the beam at any point in the traced path. Accessing these properties is performed in exactly the same way as detailed in the "Accessing beam properties" section above.

As an example, we can obtain a symbolic expression for the beam size on ZM1 with:

```
# Symbolic expr of w at ZM1
w_zm1_sym = ps_sym.w(model.ZM1.p2.i)
```

(continues on next page)

(continued from previous page)

```
# Evaluate this with current parameter values
print(f"w_ZM1 = {w_zm1_sym.eval() / 1e-3} mm")
```

```
w_ZM1 = 8.929276136051481 mm
```

Similarly to the plotting routines, we can pass a `subs` dict to the `eval` method of the symbolic expression to find the beam size at ZM1 using different dependent optic parameters:

```
# Evaluate w_ZM1 with ZM2 RoC changed to -70 m
print(f"w_ZM1 = {w_zm1_sym.eval(subs={model.ZM2.Rcx: -70}) / 1e-3} mm")

# Evaluate w_ZM1 with ITM lens focal length changed to 100 m
# and telescope length changed to 60 m
print(f"w_ZM1 = {w_zm1_sym.eval(subs={model.ITM_lens.f: 100, model.lZM1_ZM2.L: 60}) / 1e-3}␣
↪mm")
```

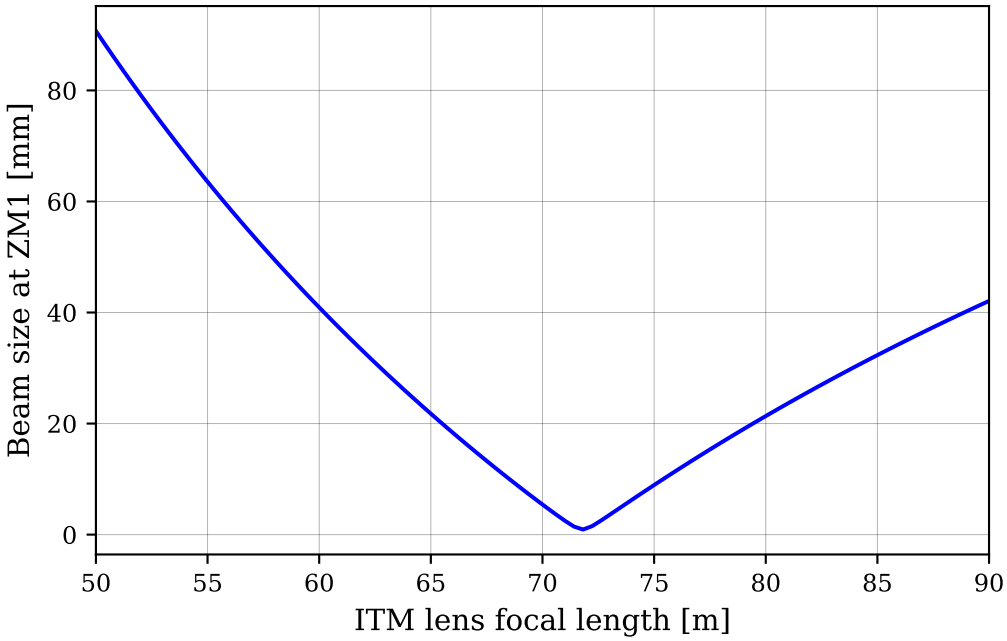
```
w_ZM1 = 15.378018812810746 mm
w_ZM1 = 61.29723873257198 mm
```

These symbolic expressions also support `numpy.ndarray` type arguments, allowing for evaluation over N-dimensional arrays. For example, we can evaluate the beam size at ZM1 over a range of ITM lens focal lengths and plot the results:

```
import matplotlib.pyplot as plt

fs = np.linspace(50, 90, 100)
w_zm1s = w_zm1_sym.eval(subs={model.ITM_lens.f: fs}) / 1e-3 # scale to mm

plt.plot(fs, w_zm1s)
plt.xlabel("ITM lens focal length [m]")
plt.ylabel("Beam size at ZM1 [mm]");
```



Or we could, for example, compute the Gouy phase accumulated up to ZM1 for the same parameter space:

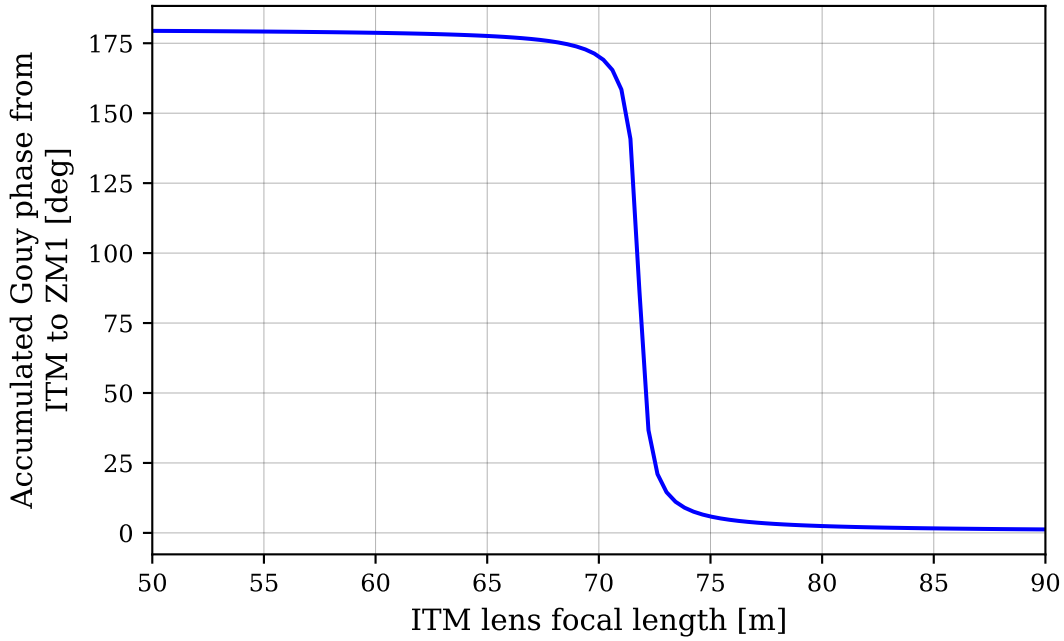
```
acc_gouy_zm1_sym = ps_sym.acc_gouy_up_to("ZM1")
```

(continues on next page)

(continued from previous page)

```
acc_gouys_zm1 = acc_gouy_zm1_sym.eval(subs={model.ITM_lens.f: fs})

plt.plot(fs, acc_gouys_zm1)
plt.xlabel("ITM lens focal length [m]")
plt.ylabel("Accumulated Gouy phase from\nITM to ZM1 [deg]");
```



This is a very powerful feature, as it leverages the speed of NumPy array calculations allowing for fast, high-dimensional grid calculations of arbitrary beam properties over any dependent model parameter.

Reverse beam propagation

Some components in a model may not have forwards and backwards couplings between ports, such as a directional beamsplitter. These elements act as a form of optical isolation, such as a faraday isolator would. This means you can only trace beams through a physically viable path, as the beam tracer only follows the connections at each component.

To get around this, it is possible to "reverse propagate" and let FINESSE know you want to get around this limitation and follow unphysical paths. Here is a simple example:

```
model = finesse.script.parse(
    """
l l1
dbs isolator
lens L1 f=1
m m1
link(l1.p1, 1, isolator.p1, isolator.p3, 2, L1, 3, m1)
gauss g1 l1.p1.o w0=1e-3 z=1e-2
"""
)

fwd = model.propagate_beam("l1.p1.o", "m1.p1.i")
rev = model.propagate_beam("m1.p1.o", "l1.p1.i", reverse_propagate=True)

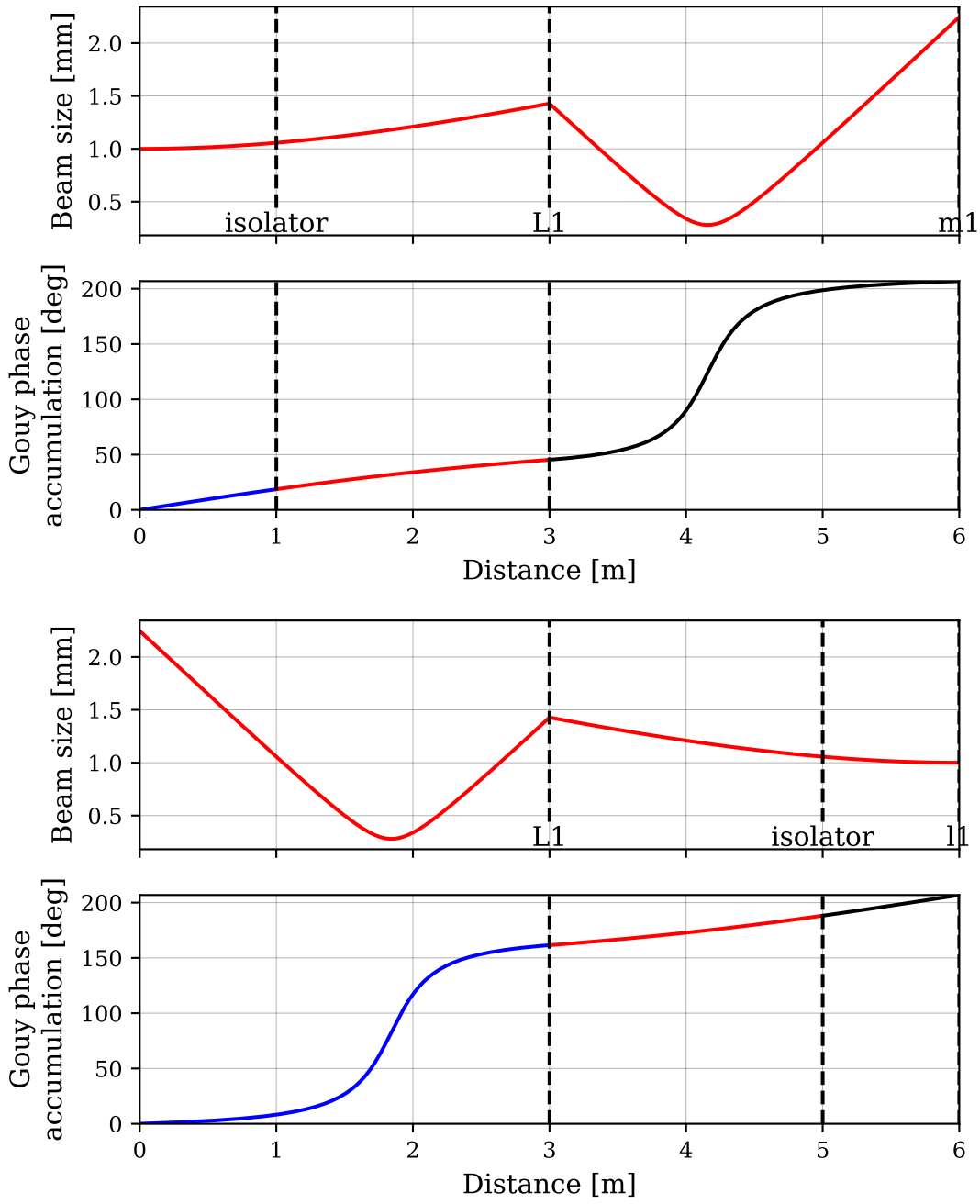
fwd.plot()
rev.plot()
```

```
(<Figure size 576x355.968 with 2 Axes>,
 array([<Axes: ylabel='Beam size [mm]'>,
```

(continues on next page)

(continued from previous page)

```
 <Axes: xlabel='Distance [m]', ylabel='Gouy phase\naccumulation [deg]'>],
dtype=object))
```



6.5.6 Computing arbitrary scattering matrices

The internal functions for computing coupling coefficient matrices during a simulation are also exposed via specific methods in the `finesse.knm` sub-module. The most-general function for computing these matrices is `make_scatter_matrix()` - this requires the matrix type as the first argument, mapping to one of the specific scattering matrix functions in this module.

As these Python-facing functions internally call the same C functions used to compute coupling coefficients during simulations, you can rest assured that these routines are highly optimised - allowing for fast computation of these matrices.

Calculating coupling matrices via the Bayer-Helms formalism

To compute a scattering matrix based on a system with mode-mismatches and tilt (angular) misalignments, one can use `make_bayerhelms_matrix()` (or `make_scatter_matrix()` with "bayerhelms" as the type, as noted above).

This function requires the input and output beam parameters, in both tangential and sagittal planes, as well as the misalignment angles in both planes. One can optionally provide a medium refractive index (defaults to $n_r = 1$) and / or a beam wavelength (defaults to $\lambda = 1064\,\text{nm}$) too.

As an example, below, we compute a scattering matrix with only mismatches present, up to a *maxtem* of 5 (see Selecting the modes to model (page 327)):

```
import finesse
finesse.configure(plotting=True)

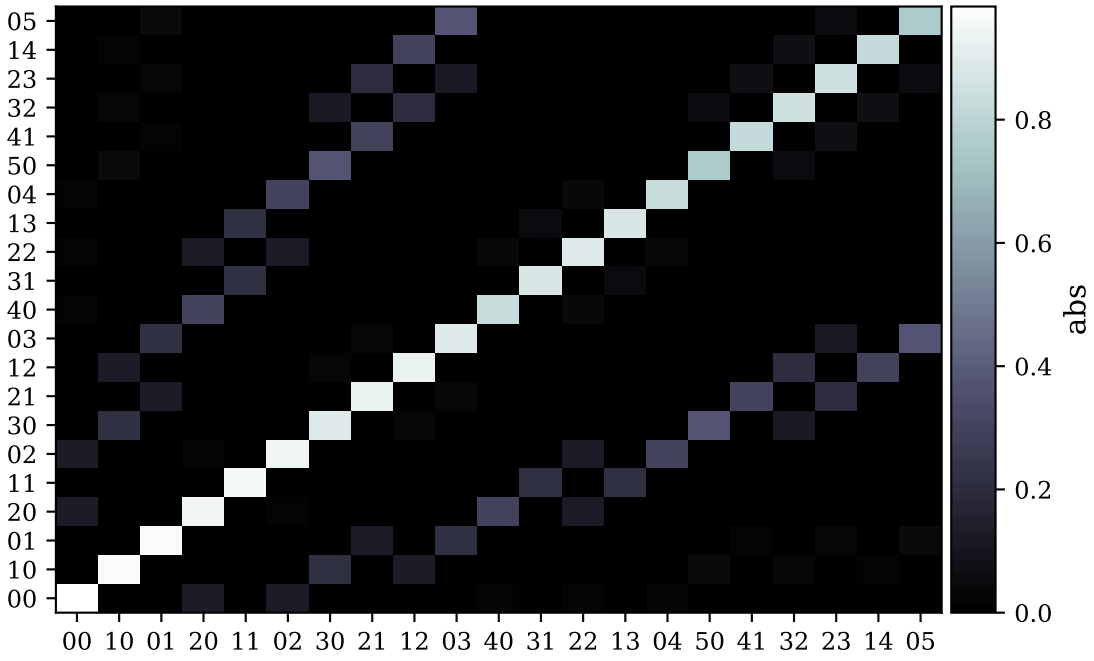
from finesse.knm.tools import make_scatter_matrix

# Non-astigmatic beam with 20% mismatch in w0
# from input to output
qx1 = qy1 = finesse.BeamParam(w0=1e-3, z=0)
qx2 = qy2 = finesse.BeamParam(w0=1.2e-3, z=0)

# No angular misalignments
xgamma = ygamma = 0

# Compute the scattering matrix, returning a KnmMatrix object
kmat = make_scatter_matrix(
    "bayerhelms",
    qx1, qx2, qy1, qy2,
    xgamma, ygamma,
    maxtem=5,
)

# Plot the absolute value of the coupling
# coefficients as a colormesh
kmat.plot(cmap="bone");
```

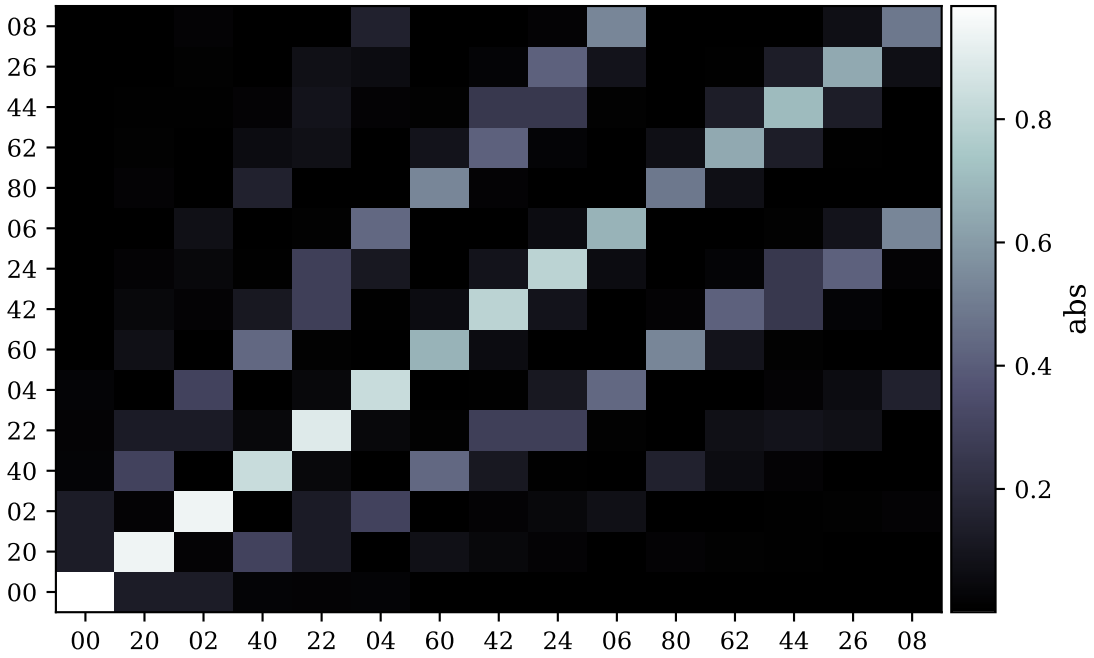


As expected, we get a plot showing non-zero couplings for even mode orders. Note that the `kmat` object here will be a `KnmMatrix` object, providing a convenient interface for accessing specific coupling coefficients, printing the matrix

and plotting the matrix. The `KnmMatrix.plot()` method computes and plots the absolute value of the coefficients by default; this can be altered as shown in the documentation for this method.

This function also accepts mode selection keyword arguments, allowing for custom mode indices to be used (see Selecting the modes to model (page 327) and `make_modes()`). For example, in this case we could select just even modes as we know that no angular misalignments are present:

```
kmat = make_scatter_matrix(
    "bayerhelms",
    qx1, qx2, qy1, qy2,
    xgamma, ygamma,
    select="even", maxtem=8,
)
kmat.plot(cmap="bone");
```



Accessing specific couplings

As implied previously, the return type of `make_scatter_matrix()` is a `KnmMatrix` object. Along with plotting and stringifying, this object provides a dict-like interface for accessing coupling coefficients in the matrix.

Here are a few convenient ways of accessing these coefficients:

```
# Using the "nm->n'm'" format
print("Coupling coeff. from 00 -> 02 = ", kmat["00->02"])

# Using the "nmn'm'" format
print("Coupling coeff. from 02 -> 40 = ", kmat["0240"])

# Using the (n, m, n', m') format
print("Coupling coeff. from 26 -> 44 = ", kmat[(2, 6, 4, 4)])
```

```
Coupling coeff. from 00 -> 02 =  (-0.1254207136745824+0j)
Coupling coeff. from 02 -> 40 =  (0.0024975293572353312+0j)
Coupling coeff. from 26 -> 44 =  (-0.12938973739354842+0j)
```

Storage convention

A `KnmMatrix` stores its coefficients in C order (row-major) with index convention `K[output_mode, input_mode]`. Applied to a mode-amplitude vector as $\vec{a}_{\text{out}} = K\,\vec{a}_{\text{in}}$, the element `K[j, i]` is therefore the coupling from input mode `i` into output mode `j`. Direct access via `kmat.data` returns the underlying numpy array with this layout.

6.5.7 Adding spatial defects with maps

Maps are, broadly speaking, 2-dimensional (x, y) information that describes how the phase and/or amplitude of a beam is affected during some operation, such as reflection or transmission. Maps are often used to describe scenarios such as surface roughness, apertures, parametric instabilities, or thermal deformations. The mathematics behind maps are described in the freely available Living Review article, and therefore will not be repeated here (See section "Scattering into higher-order modes") [18].

Warning

Currently, only mirrors and lens elements support maps - beamsplitters are a work in progress. In fact, maps in general are still a work in progress and may change in the future to fully support more options:

- https://gitlab.com/ifosim/finesse/finesse3/-/issues/488
- https://gitlab.com/ifosim/finesse/finesse3/-/issues/487

The map object

`finesse.knm.maps.Map` is the object that must be created to describe a particular deformation in phase and amplitude. Below we make a simple map using a radially quadratic term and a circular aperture.

```
import numpy as np
import finesse
from finesse.knm import Map
from finesse.utilities.maps import circular_aperture
import matplotlib.pyplot as plt

finesse.init_plotting()

# Always worth using a different x/y size to ensure you plot the
# correct variable later and do not transpose anything by accident
x = np.linspace(-0.1, 0.1, 100)
y = np.linspace(-0.1, 0.1, 101)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)

my_map = Map(
    x,
    y,
    opd=1e-6 * R**2,
    amplitude=circular_aperture(x, y, 0.1),
)

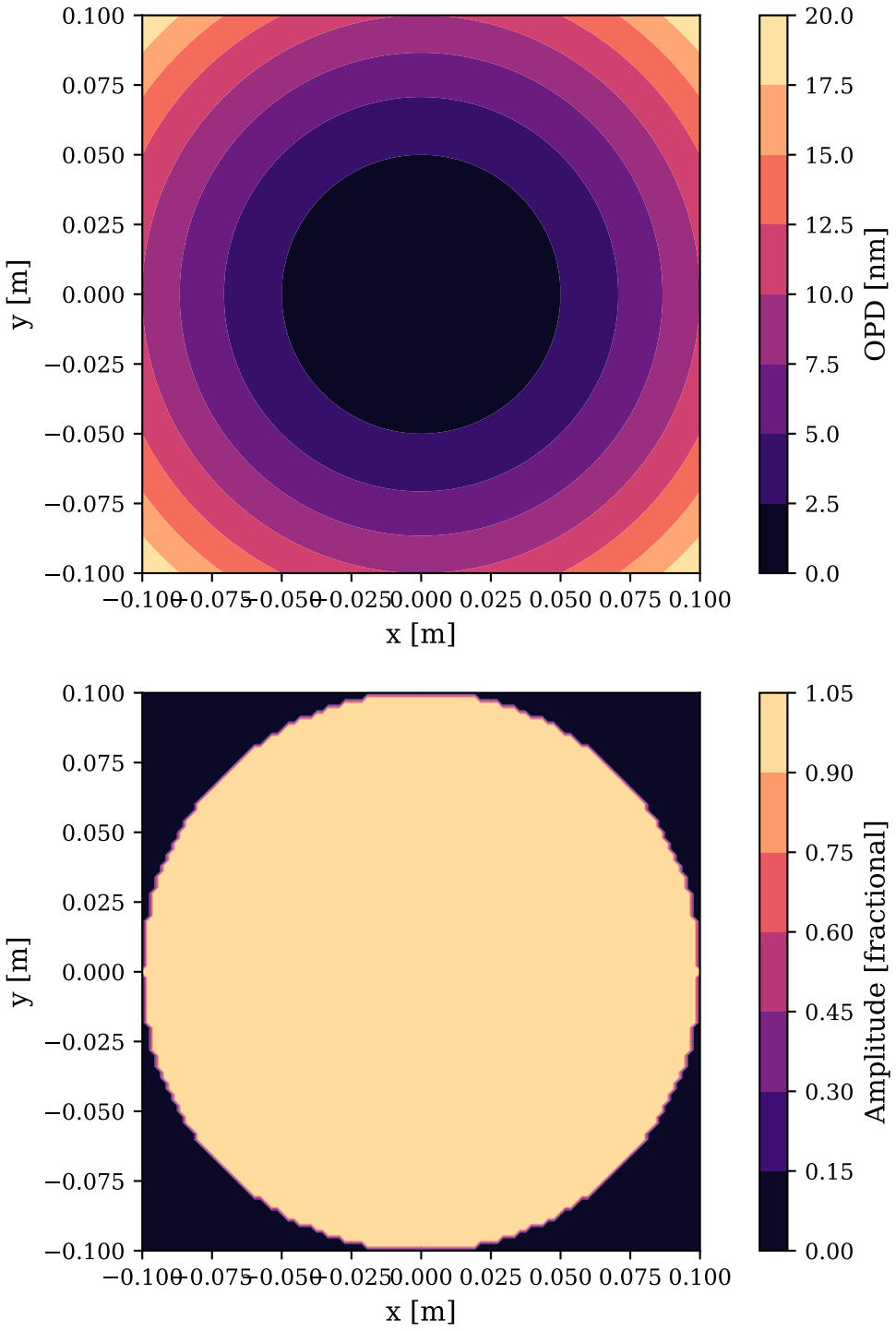
plt.figure(layout="compressed")
plt.contourf(my_map.x, my_map.y, my_map.opd / 1e-9)
plt.gca().set_aspect("equal")
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.colorbar(label="OPD [nm]")

plt.figure(layout="compressed")
plt.contourf(my_map.x, my_map.y, my_map.amplitude)
plt.gca().set_aspect("equal")
```

(continues on next page)

(continued from previous page)

```
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.colorbar(label="Amplitude [fractional]");
```



This map describes the addition of some transverse quadratic phase of the beam interacting with it and will be clipped in a circular manner.

Optical path depths

The optical path difference (OPD) is given in units of metres and is often of the order of microns or less. It should be noted that the OPD model is mostly applicable in the regime where $OPD < \lambda$, the wavelength of the light. In cases where the OPD is significantly higher the number of higher order optical modes that must be used to describe the distortion can increase exponentially. This limits the application of OPDs to small perturbations to the beam that are paraxial in nature.

Quadratic lensing with maps (What not to do)

The simplest element to consider at first is a lens. A simple thin lens imparts a purely quadratic phase shift to a beam of the form:

$$OPD(x, y) = -\frac{1}{2f} r^2$$

Where f is the focal length of the thin lens and $r = \sqrt{x^2 + y^2}$. This is what we refer to as "quadratic lensing".

A lens map describes additional deformations that a beam experiences on top of the optical path difference induced by the lens itself - such as spherical aberrations or other higher order effects.

Note

Maps only work with the Python API. There are currently no methods to interact with maps or make maps using KatScript. There are many complications in storing, inputting, and applying maps that were challenging to use in FINESSE v2. None of these have been, or plan to be, implemented in v3. The main limitation is that serialising a model that includes maps cannot be done at this time.

We can consider a case that is the opposite of what you should do but is somewhat illustrative: try and describe a large change in the beam shape with a map. Here we take an infinite focal length lens and apply a map that uses the $OPD(x, y)$ above to simulate a lens with a focal length of 500m. Normally, this 500m focal length would describe a change in the beam shape using the complex beam parameter tracing (ABCD matrices and q-parameters). However, now we have the situation where the q-parameters at each node are defined by the *q(w0=10cm, z=0)* beam through an $f = \infty$ lens, which is the basis in which the higher order modes are described. As the map now contains the quadratic lensing, the ABCD matrix knows nothing about it, so the q-parameters do not change. However, to describe the fact the beam is now lensed, we must include many higher order modes to describe the new shape of this lensed beam.

Below we plot a slice of the beam before and after the map is applied, we see that the beam has been focused - higher peak intensity and smaller spot size.

```
x = np.linspace(-0.1, 0.1, 300)
y = np.linspace(-0.1, 0.1, 301)
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
f = 500  # focal length
my_map = Map(x, y, opd=-1 / (2 * f) * R**2)

model = finesse.script.parse(
    """
    l l1
    lens L f=inf
    nothing n1
    link(l1, L, 10, n1)

    gauss g1 l1.p1.o w0=1e-2 z=0

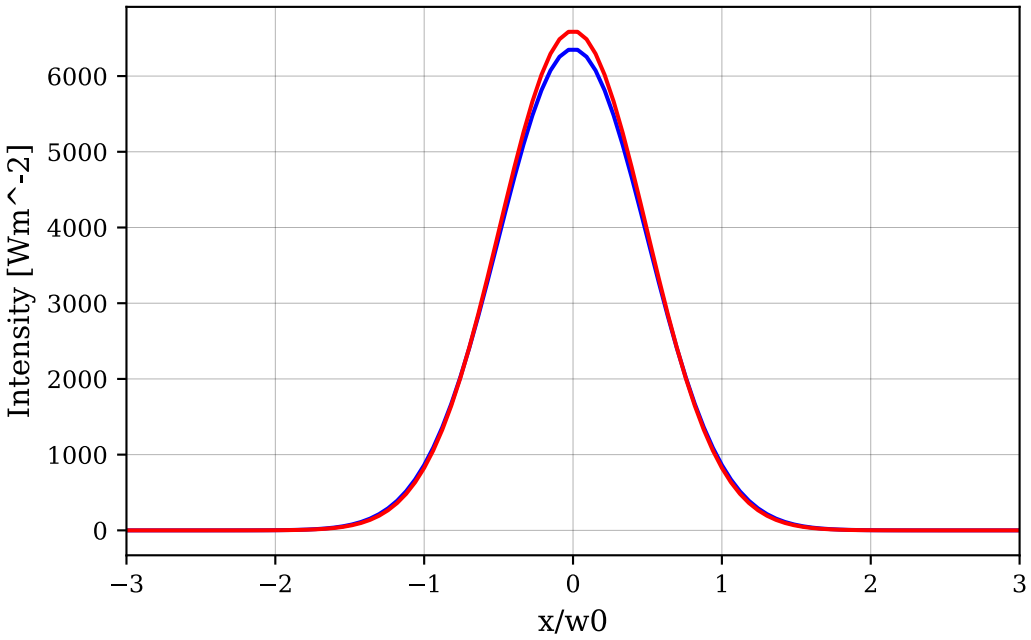
    ccdline beam n1.p1.i 100 xlim=[-3, 3]
    modes(even, maxtem=8)
    pd P n1.p1.i
    """
)
```

(continues on next page)

(continued from previous page)

```
out = model.run("noxaxis()")
plt.plot(model.beam.xdata, out["beam"])
model.L.OPD_map = my_map
out = model.run("noxaxis()")
plt.plot(model.beam.xdata, out["beam"])
plt.xlabel("x/w0")
plt.ylabel("Intensity [Wm^-2]")
print("Missing optical power:", model.l1.P-out['P'], "[W]")
```

```
Missing optical power: 3.3157007666151372e-06 [W]
```



If the map is sampling the spot size well (dx and dy are much smaller than the spot size), then you should see that using a higher `maxtem` will reduce the amount of power lost. There is some negligible amount lost due to the finite square aperture of the map but it should be close to a unitary transformation, so no power lost.

You can experiment yourself by trying stronger lenses and including enough higher order modes to ensure you have reached a converged state. It will often be the case with large lensing that you need exponentially more modes. The 100m focal length in this particular example will need `maxtem > 30` to start converging.

Removing the quadratic lens

Altogether the correct solution here is: do not use maps to describe quadratic lensing. Higher order mode optical models using OPDs should put all the beam shaping into the ABCD beam parameters. This often results in having to "remove the quadratic term" from phase maps. This quadratic term can then be represented either in a mirror curvature or a lens focal length.

Sticking with the lens, we can do this using `finesse.knm.maps.Map.get_thin_lens_f()`. This requires specifying a spot size as well. Here our beam is 1cm. Using that we can recover what focal length the map represents.

```
fx = 100.5  # focal length in x
fy = 100.1  # focal length in y
my_map = Map(x, y, opd=-1 / (2 * fx) * X**2 - 1 / (2 * fy) * Y**2)
# Compute the weighted spot size focal lengths from the map OPD
my_map.get_thin_lens_f(0.01)
```

```
(np.float64(100.5), np.float64(100.09999999999998))
```

If you start using a beam that is perhaps to large for the map, or with a map that has spherical features (non-quadratic phase terms), then the chosen spot size will result in different effective focal lengths. In this case it is because the beam is becoming clipped by the square aperture.

```
my_map.get_thin_lens_f(0.05)
```

```
(np.float64(101.04430386582891), np.float64(100.64180891647648))
```

Once you have retrieved and stored the quadratic terms parts then you can remove them:

```
original_dioptre = 1/model.L.f
new_fx, new_fy = my_map.get_thin_lens_f(0.01)
new_avg_dioptre = 1/((new_fx + new_fy)/2)
my_map.remove_curvatures(0.01, mode='average')

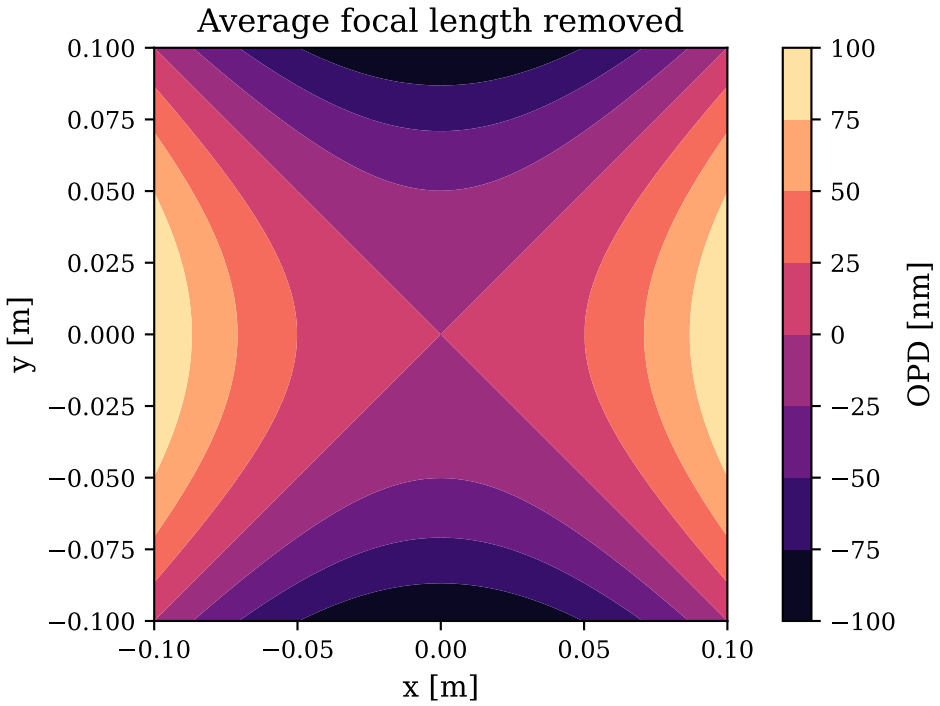
model.L.f = 1/(original_dioptre + new_avg_dioptre)
```

As the lens element only has a single average focal length (no astigmatic lenses yet) we have to get the average and set that to the element. Now we plot the leftover astigmatism which shouldn't be that much, order of 10's of kilometres.

```
print("New focal lengths in map:", my_map.get_thin_lens_f(0.01))

plt.figure(layout='compressed')
plt.contourf(my_map.x, my_map.y, my_map.opd/1e-9)
plt.gca().set_aspect('equal')
plt.xlabel("x [m]")
plt.ylabel("y [m]")
plt.colorbar(label='OPD [nm]')
plt.title("Average focal length removed");
```

```
New focal lengths in map: (np.float64(-50300.24999999735), np.float64(50300.25000000133))
```



Mirror surface maps

Mirror surface distortions are another common type of map we need to add. This represents a deformation to a reflective to refracting surface. The most common here is for applying residual polishing defects, or thermo-elastic deformations from thermal actuators or absorption.

The quadratic OPD of a reflecting mirror is given by:

$$OPD(x, y) = \frac{1}{4f} r^2$$

Which given that $R = 2f$ is also:

$$OPD(x, y) = \frac{1}{2Rc} r^2$$

Broadly similar to the lensing OPD, but there are several factors of two to consider above, due to the double pass on reflection of the optic surface, as well as the conversion from curvatures to focal powers. There is also a sign difference.

It is important to remember that the OPD of map for a mirror in FINESSE describes the mirror surface height difference, not a total OPD (This will be fixed with a name change in a later fix). You can apply mirror surface maps using:

```
model = finesse.script.parse("m m1 R=0.9 T=0.1")
model.m1.surface_map = my_map
```

where the `opd` term given to the `Map` represents the height of a surface change in the normal direction of the surface. For a mirror, a positive height change is a protrusion in the surface normal direction on the port 1 side of the element. A positive bump in the map would be a "hill" on the port 1 side, and a "valley" when seen from the port 2 side.

Quadratic mirror cuvatures terms can be calculated using `finesse.knm.maps.Map.get_radius_of_curvature_reflection()`. Remember that dioptres of focal length are $\frac{1}{f}$ whereas dioptres using curvatures are $\frac{2}{Rc}$. Mirrors have `Rcx` and `Rcy` parameters so there you can remove the quadratic astigmatic parts from maps completely.

How Bayer-Helms and map contributions are combined

On any optical connection of a `Mirror` (page 142), `Beamsplitter` (page 149) or `Lens` (page 156) that can carry a surface or optical-path map, the final scattering matrix is built from two separate contributions:

- an analytic Bayer-Helms (BH) matrix that handles mode mismatch and misalignment of the gauss bases on the two sides of the connection, and
- a numerical map-integration (MAP) matrix that handles the scattering caused by the surface amplitude and optical path difference.

The two matrices are composed by a matrix product into a single matrix that the simulation actually uses. Two settings control how the work is split between BH and MAP and in which order they are then multiplied.

The first setting is the `is_focusing_element` flag on the `Map` object (default `False`). It selects whether the integration is done with one beam base parameter (`q_from` on both sides, with BH carrying the mismatch analytically) or two (`q_from` and `q_to`, with the mismatch included in the numerical calculation). The flag is documented in the `Map` class docstring.

The second setting is the `knm_apply_bayer_helms_first` attribute on the element itself (`Mirror` (page 142), `Beamsplitter` (page 149), `Lens` (page 156); default `False`). It selects the order of the matrix product:

- `False` (default): $K = \text{BH} \cdot \text{MAP}$.
- `True`: $K = \text{MAP} \cdot \text{BH}$.

With earlier versions of Finesse we found that the ordering is not overly important. From testing we have seen that the differences are minor as the number of modes is increased. However, for ppm level computation it might be of interest to swap the ordering to see if any commutation errors exist between the matrices.

You can change this setting as follows:

```
model = finesse.Model()
model.parse('m ITM R=0.5 T=0.5')
model.ITM.knm_apply_bayer_helms_first = True  # default is False
```

To see any commutation effect, both the BH and the MAP scattering matrix must be non trivial. In the following example we apply a non-trivial surface map (linear tilt in the x-direction), combined with a beam shape mismatch (by explicitly setting the beam parameter on both sides of the optic).

```
w = 1e-3
n_grid = 301
x = y = np.linspace(-5 * w, 5 * w, n_grid)
X, _ = np.meshgrid(x, y)
opd = 1e-7 * X / w  # linear tilt, ~lambda/10 across one beam radius


model = finesse.Model()
model.parse(
    """
    l L1 P=1
    space s1 L1.p1 ITM.p1 L=0.1
    m ITM R=0.5 T=0.5 Rc=inf
    space sub portA=ITM.p2 portB=dump.p1 L=0.001 nr=1.45
    nothing dump

    gauss g_in  ITM.p1.i w0=1e-3 z=0
    gauss g_out ITM.p2.o w0=2e-3 z=0

    ad out10 ITM.p2.o f=0 n=1 m=0

    modes(maxtem=20)
    """
)
model.ITM.surface_map = Map(x=x, y=y, amplitude=np.ones_like(X), opd=opd)
print(model.ITM.knm_apply_bayer_helms_first)
```

```
False
```

Here we see the default value of `knm_apply_bayer_helms_first` is `False`. We then run the model with both the default value and with `knm_apply_bayer_helms_first` set to `True` and look at the difference in the 10 mode amplitude.

```
out_map_first = complex(np.asarray(model.run()["out10"]).squeeze())

model.ITM.knm_apply_bayer_helms_first = True  # change from the default value
out_bh_first = complex(np.asarray(model.run()["out10"]).squeeze())

abs(out_bh_first - out_map_first)
```

```
0.029327340453843515
```

6.6 Spatial distortions with maps

FINESSE provides a suite of tools for handling two-dimensional defects to simulate effects such as optical path difference variations and apertures. These 2D objects are called `maps`. This section provides an overview of their usage and the different types of maps available in FINESSE.

Note

The map functionality is only accessible via the Python API and no KatScript interface currently exists.

6.6.1 Basics of Using Maps

Introduction

In FINESSE, maps are used to model spatial imperfections on optics, such as surface figure errors or refractive index variations. These maps can be applied to certain elements to simulate real-world effects like scattering, mode mismatch, or thermal lensing. They can also be used outside of a simulation environment to visualize deformations and computer mode scattering matrices.

Most of the map functionality can be found in the `finesse.knm` and `finesse.utilities.maps` modules. A prerequisite for these sections is an understanding of the section Modelling with higher-order modes (page 327).

Note

You will often see `knm` used in the context of maps. This is a reference to the K matrices which are often used in analytics to represent a scattering operator.

What is a Map?

A *map* is a 2D array that represents a spatially varying optical property, such as:

- Surface height variations (phase maps)
- Refractive index gradients (substrate maps)
- Losses (amplitude maps)

Maps modify the behavior of optical elements by changing the phase and/or amplitude of the fields interacting with them.

Supported Maps

FINESSE currently supports the following types of maps:

- **Surface maps**: Change the optical phase on reflection; typically used to simulate surface deformations.
- **Absorption maps**: Describe spatially varying loss or transmission, e.g. from absorption or aperturing
- **Substrate maps**: Simulate index of refraction changes, e.g., from thermal lensing.

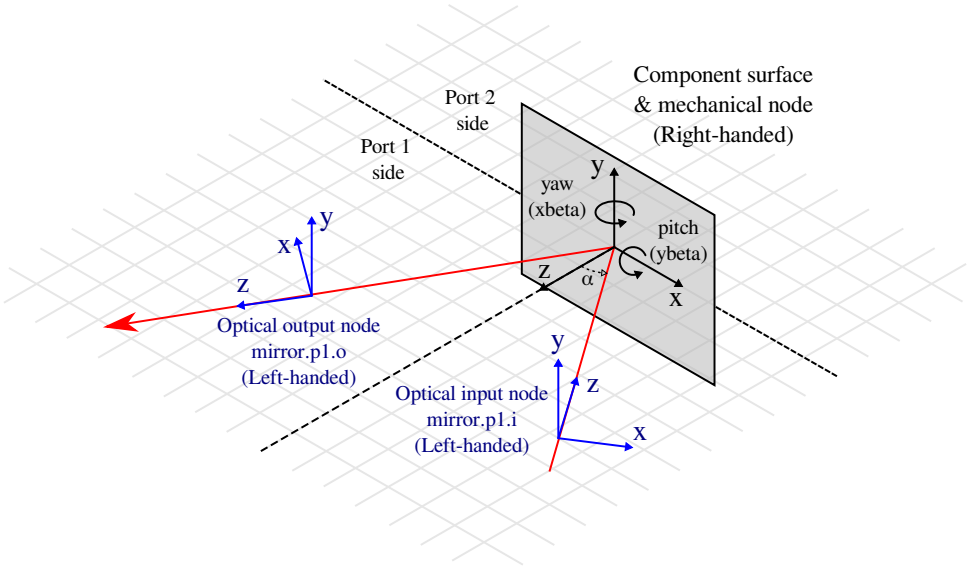
Not all optical elements support maps. The following elements can use maps:

- **Mirrors**: Apply surface maps to modify the phase of reflected light.
- **Beamsplitters**: Also apply surface maps to modify the phase of reflected light.
- **Lens**: Apply substrate maps to modify the phase of transmitted light.

You can typically combine the above elements to create more complex systems, like a thick distorted optic.

Coordinate system

Maps are defined in a 2D coordinate system, where the origin (0,0) is on the beam axis. The right-handed coordinate system used for the maps is shown here for mirrors and beamsplitters:



The surface normal vector is in the positive z direction for the component is important for surface deformations. A positive z value results in a 'hill' on the surface, a negative z value results in a 'valley'.

For lenses the z-component does not matter. The lens `OPD_map` just describes an optical delay and has no directionality associated with it. The x and y directions are the same as for mirrors and beamsplitters.

Generating maps

Unlike FINESSE 2.0, there is currently not a defined map file format. Instead, maps are generated at runtime using Python code.

Amplitude maps

In this simple example we make a 10x10cm aperture map with a 4cm radius and a value of 1 inside the aperture and 0 outside. The value of amplitude should be between 0 and 1, although there are no checks internally in FINESSE to enforce this, so be careful.

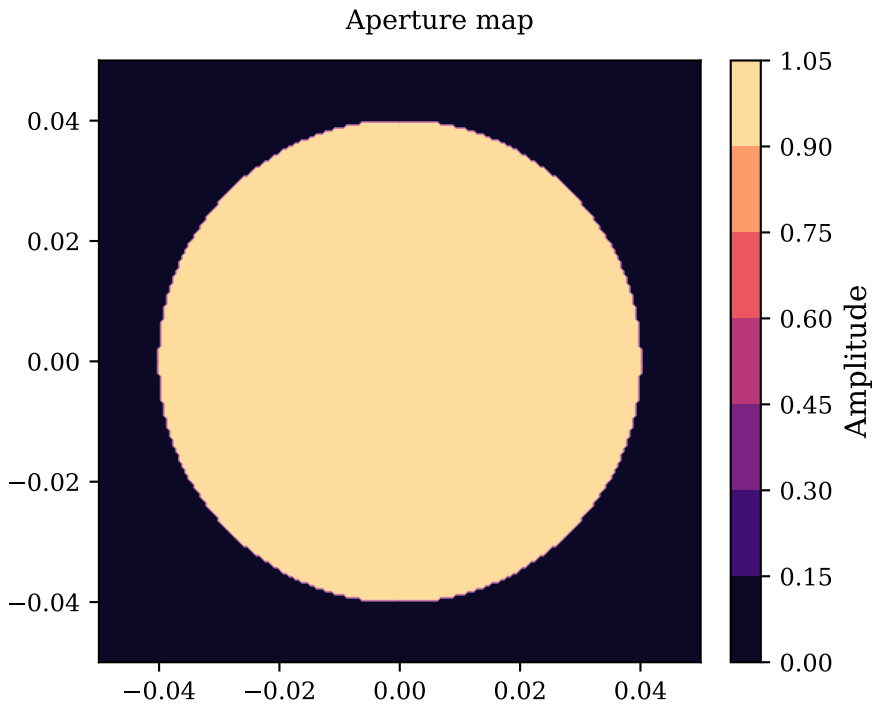
```
import finesse
from finesse.knm.maps import Map
import numpy as np
import matplotlib.pyplot as plt

finesse.init_plotting()

# Create a 10x10cm map
# We recommend using different number of X and Y points to avoid
# issues with transposing maps and data.
x = np.linspace(-5e-2, 5e-2, 200)
y = np.linspace(-5e-2, 5e-2, 201)

X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
aperture = np.zeros((y.size, x.size))
aperture[R < 4e-2] = 1
my_map = Map(x, y, amplitude=aperture)

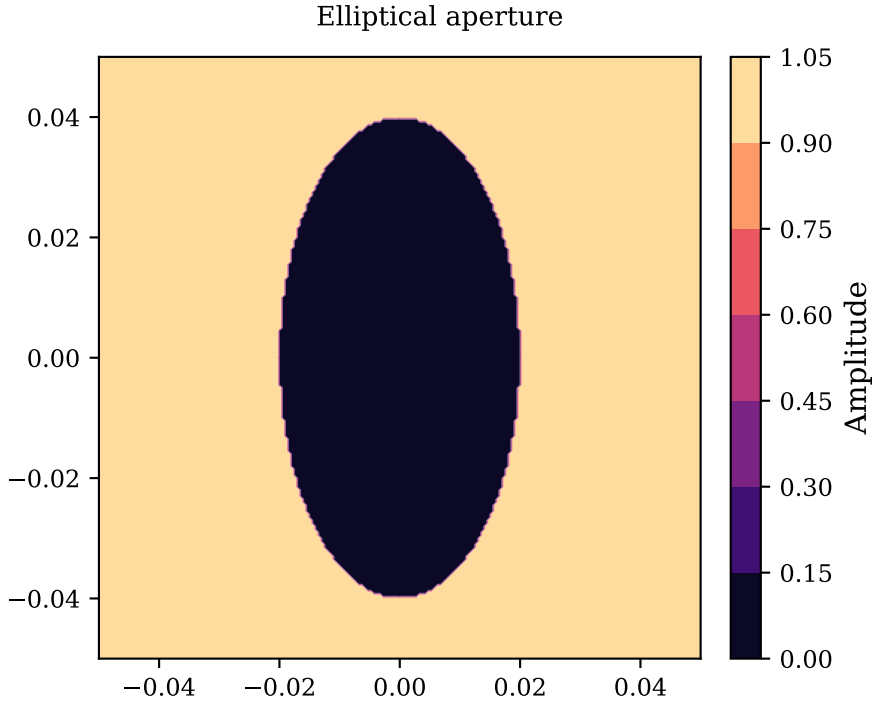
fig1, ax1 = plt.subplots(layout="compressed")
im = ax1.contourf(x, y, my_map.amplitude)
ax1.set_aspect("equal")
fig1.colorbar(im, ax=ax1, label="Amplitude")
fig1.suptitle("Aperture map");
```



There are numerous ways to achieve this but we could also make an elliptical aperture map by changing the scale of the x and y coordinates.

```
R = np.sqrt((X / 0.5)**2 + Y**2)
aperture = np.ones((y.size, x.size))
aperture[R < 4e-2] = 0
my_map = Map(x, y, amplitude=aperture)

fig1, ax1 = plt.subplots(layout="compressed")
im = ax1.contourf(x, y, my_map.amplitude)
ax1.set_aspect("equal")
fig1.colorbar(im, ax=ax1, label="Amplitude")
fig1.suptitle("Elliptical aperture");
```



Surface error maps

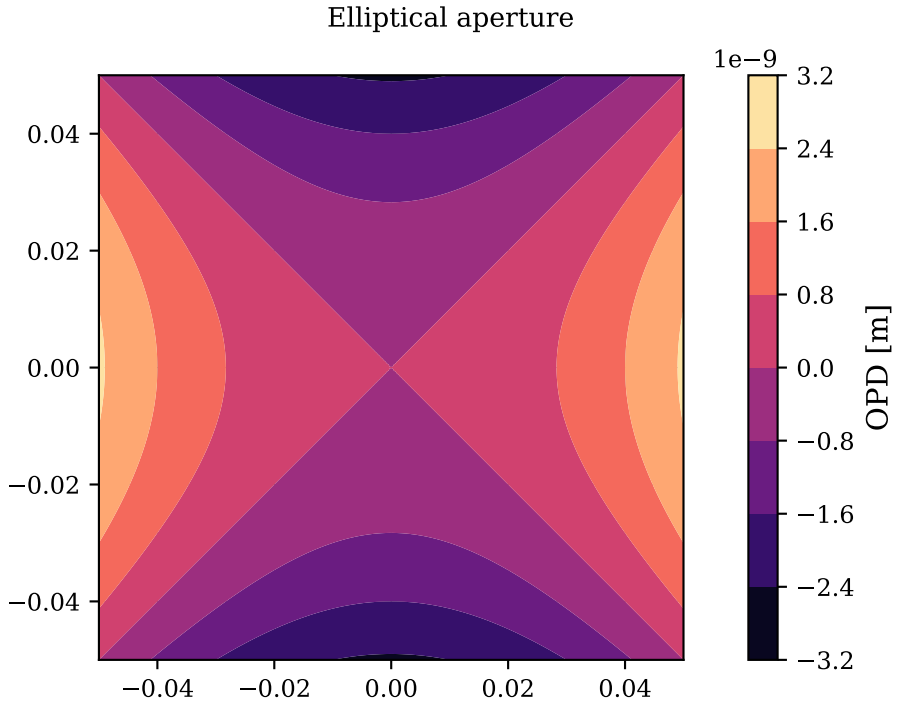
Phase maps are similar to amplitude maps, but they are used to represent optical path difference (OPD) variations relative to no map being present. The OPD must have units of meters because at this stage the map is not referenced to a particular wavelength and depends what the default model wavelength is. The phase accumulated across the wavefront is the OPD multiplied by the wavenumber k:

$$\phi(x, y) = k\mathrm{OPD}(\mathrm{x}, \mathrm{y})$$

The OPD can be set by using the `opd` keyword argument as seen here:

```
X, Y = np.meshgrid(x, y)
my_map = Map(x, y, opd=1e-6 * (X**2 - Y**2))

fig1, ax1 = plt.subplots(layout="compressed")
im = ax1.contourf(x, y, my_map.opd)
ax1.set_aspect("equal")
fig1.colorbar(im, ax=ax1, label="OPD [m]")
fig1.suptitle("Elliptical aperture");
```



Generating the complex-valued map

The complex-valued map is generated by using the `Map.get_z` method of the map object.

$$z(x, y) = \mathrm{amplitude}(\mathrm{x}, \mathrm{y}) \exp\left(\mathrm{i}\, F k \mathrm{OPD}(\mathrm{x}, \mathrm{y})\right)$$

Where F is the phase factor for a particular scenario. A phase map can be used to describe a surface height, or a refractive index change over the length of a substrate. Commonly used values for F are:

- 1: Substrate refractive index change map
- 2: Surface reflection (Factor of 2 for a double pass on reflection)
- $n_1 - n_2$: Transmission through a refractive index change at a surface

FINESSE will apply these scaling terms internally depending on what the component is using the map for. The components documentation should be referred to for exactly how the data should be provided.

Here for example is an apertured map with some surface variation, we then plot the resulting phase map that would perturb the optical field.

```
X, Y = np.meshgrid(x, y)
R = np.sqrt(X**2 + Y**2)
aperture = np.ones((y.size, x.size))
aperture[R > 4e-2] = 0
k = 2*np.pi/1064e-9 # Or use model.k0 if you have a finesse model

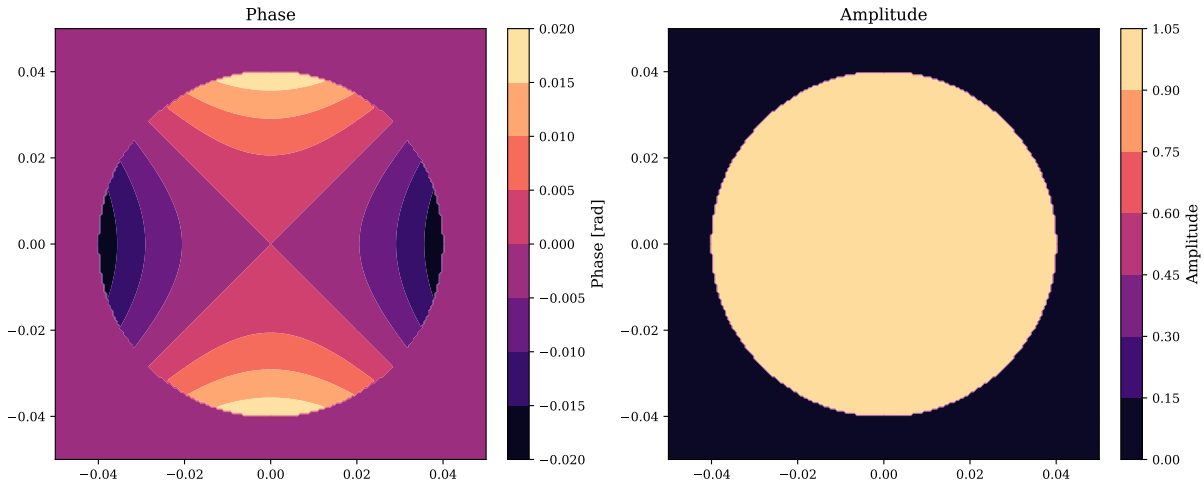
my_map = Map(x, y, opd=1e-6 * (X**2 - Y**2), amplitude=aperture)

Z = my_map.get_z(
    k,
    2, # factor of 2 for surface reflection
)

fig, axs = plt.subplots(1, 2, figsize=(12, 5), layout="compressed")

# Plot phase
contour1 = axs[0].contourf(x, y, np.angle(Z))
fig.colorbar(contour1, ax=axs[0], label='Phase [rad]')
axs[0].set_title('Phase')
axs[0].set_aspect('equal')

# Plot amplitude
contour2 = axs[1].contourf(x, y, np.abs(Z))
fig.colorbar(contour2, ax=axs[1], label='Amplitude')
axs[1].set_title('Amplitude')
axs[1].set_aspect('equal')
```



Computing scattering matrices

When performing modal model simulations the scattering matrix (or scattering operator) is typically what you are interested in. How much does one Hermite-Gaussian mode couple to another. This can be computed from the map directly for a given wavenumber, set of modes indices, and a beam parameter describing the shape of the beam at this map. Again, FINESSE will apply all this information for you when it runs simulations internally, this is simply a helper method for users to experiment and see the effect of maps on the mode scattering. Below we should see that is a larger complex scattering into from the HG00 to the HG10 due to the map representing a tilted surface.

Note

See Hermite-Gauss modes and coupling (page 306) for a more mathematical description of the scattering matrix and how to compute it.

```
X, Y = np.meshgrid(x, y)
k = 2 * np.pi / 1064e-9 # Or use model.k0 if you have a finesse model
homs = [
    (0,0), # HG00
    (1,0), # HG10
    (0,1), # HG01
]

q = finesse.BeamParam(w=2e-2, Rc=np.inf)

my_map = Map(x, y, opd=1e-9 * X)
K = my_map.scatter_matrix(
    q,
    k,
    1,
    homs,
)

display(K)
print(K)
```

```
<finesse.knm.matrix.KnmMatrix at 0x5aa93d62b140>
```

```
K[0][0]: 0 0 -> 0 0 = (0.9999989944757481-1.3806581657872145e-20j)
K[1][0]: 0 0 -> 1 0 = (-3.0724160948663404e-17-5.905165798665218e-05j)
K[2][0]: 0 0 -> 0 1 = (-1.6653345369377348e-16-3.009265538105056e-36j)
K[0][1]: 1 0 -> 0 0 = (-3.0724160948663404e-17-5.905165798665218e-05j)
K[1][1]: 1 0 -> 1 0 = (0.9999858480526327-1.5357119720985648e-20j)
K[2][1]: 1 0 -> 0 1 = (4.622231866529366e-33+6.776263578034403e-21j)
K[0][2]: 0 1 -> 0 0 = (-1.6653345369377348e-16-3.009265538105056e-36j)
K[1][2]: 0 1 -> 1 0 = (4.622231866529366e-33+6.776263578034403e-21j)
K[2][2]: 0 1 -> 0 1 = (0.9999858432596216-1.3806400084350208e-20j)
```

6.6.2 Static maps

6.6.3 Dynamic maps

6.7 Radiation Pressure

Interferometric gravitational wave detectors make use of suspension systems to decouple mirrors from ground motion. Furthermore these interferometers operate with a very high circulating light power so that the radiation pressure from the reflected photons can significantly influence the mechanical behaviour of the mirrors. This leads to a very interesting coupling between the optical fields and the mirrors as mechanical oscillators, forming so-called optical springs. Optical springs and other radiation pressure effects can change the dynamics of an interferometer significantly, especially when considering transfer functions from mechanical motions to optical outputs.

6.7.1 Radiation pressure calculation approximations

It is important, as with any simulation tool, to understand the limits of the approximations made to allow radiation pressure effects to be implemented in FINESSE. By nature, radiation pressure is proportional to the beam power, a non-linear relation in regards to the amplitude of the optical fields that FINESSE computes. To linearise the physical equations, as described in the following sections, the following assumptions are made:

- The magnitude of any motion in the mirrors surface is much smaller than the wavelength of light, by default, 1064nm.
- Therefore the optical sidebands created by such motion are also much smaller in magnitude than the magnitude of any carrier (laser or RF modulation) field.

- That the frequency difference between carrier fields, laser fields and RF sidebands created with `modulator` components, are large in comparison to the frequencies of signals and of mechanical resonances of optics. This is required so that any beating between carriers fields does not cause significant radiation pressure effects.

It also follows that the maximum signal frequency must be less than half of the minimum frequency difference between carrier fields. If this frequency limit is exceeded, it is possible for the upper sidebands of one carrier field to become the lower sideband of another. This is not a scenario that the radiation pressure implementation of FINESSE is designed to handle. Given that `modulator` frequencies are typically the order of MHz and radiation pressure effects are only of interest at low frequencies due to the mechanical susceptibility of a suspended object being $\propto 1/f^2$, this should not be an issue that comes up frequently.

Radiation Pressure at a Mirror

When light is incident on a mirror's surface, it generates a force pushing on the mirror. In the steady state, we may assume that any constant light field is accounted for by control systems, such that the only effect will be due to the beats between different frequency light fields e.g. a carrier and its signal sidebands.

The coupling relations relevant to model radiation-pressure effects at a mirror are given below.

Coupling of radiation pressure force to mirror motion

Any electromagnetic field will exert a force on a suspended optic, the frequency spectrum of such a force from one beam is given as,

$$F(f) = \frac{P(f)\cos(\alpha)}{c}$$

where $P(f)$ is the fluctuation of the power in the laser at frequency f, α is the angle of incidence and c the speed of light. FINESSE models only consider forces with $f > 0$, in other words, we assume that the DC radiation pressure force is compensated for by some control system that keeps the mirror at the intended position.

Any fluctuation in the light power at a given signal frequency f_s can be described by a positive and negative part in the frequency domain,

$$P(f_s) = P_s^+ e^{\mathrm{i}2\pi f_s t} + P_s^- e^{-\mathrm{i}2\pi f_s t}.$$

As $P_s^+ = P_s^{-*}$ there is no need to explicitly solve P_s^{-*} in any of our simulations, instead we only solve the positive frequency components. Assuming that all carrier fields possess both an upper $a^+_{s,jnm}$ and a lower $a^-_{s,jnm}$ sideband of mode TEM_{nm} and which belongs to the jth carrier field, the fluctuating power is then given by

$$P_s^+ = \sum_j \sum_{n,m} \left(a^+_{s,jnm} a^*_{c,jnm} + a^{-*}_{s,jnm} a_{c,jnm}\right).$$

It should be noted that this power fluctuation calculation depends on both the upper sideband and the conjugate of the lower sideband - conjugation is not a linear operation so cannot be represented in a matrix form of the equations. By choosing to model only the positive frequencies we must now propagate the conjugate of the lower signal sidebands in the interferometer matrices that we construct. Although internally this requires constructing the matrices differently, any output from FINESSE will always output the non-conjugated lower sideband, such as if you use the `ad` detectors.

The total force applied to a suspended mirror is then the sum of all forces due to each incoming and outgoing beam present at the optical component; for a mirror this is four beams and at a beamsplitter it is eight,

$$\begin{aligned} F_{\text{total}}(f) &= -F_{1i}(f) - F_{1o}(f) + F_{2i}(f) + F_{2o}(f) \\ &= \frac{2\cos(\alpha)}{c}\left(-P_{1i}(f) - P_{1o}(f) + P_{2i}(f) + P_{2o}(f)\right) \end{aligned}$$

The $\pm$ for each P_s is determined by which side of the mirror or beamsplitter the beam is on: The positive direction of motion, the surface normal, for the mirror is defined as the direction of the beam reflected by the side of the first node. Thus an incoming or outgoing beam imparts a negative momentum on the side of the first node and a positive on the side of the last.

As we only compute the output for a single value of f_s in each step we set $F_s = F_{\text{total}}(f)$ and $P_x(f) = P_{s,x}$ leaving,

$$F_s = \frac{2\cos(\alpha)}{c}\left(-P_{1i} - P_{1o} + P_{2i} + P_{2o}\right)$$

Variables subscripted with an s are typically frequency dependent terms which are being computed at an assumed signal frequency f_s. The longitudinal motion at the signal frequency is then found using,

$$Z_s = H_s \sum_n^{N_F} F_{s,n} \,,$$

where H_s is the transfer function for the mechanical response of the suspended optic due to a force being applied to it in the z direction, i.e. in the direction of beam propagation. The sum of forces can also contain other external forces being applied, such as from actuators.

Mirror motion to optical phase change

Evaluating the above equation for Z_s determines the amplitude and phase of the mirror oscillations. When carrier fields and RF modulation sidebands are reflected from a moving mirror, this oscillation will produce phase modulation signal sidebands, $a^{\pm}$, around these fields. The variation in height of a mirror's surface at a frequency f_s is described by $z_s(x;y)$. This function is a normalised description of the general motion; its amplitude is given by A_s, where $A_s^+ = A_s$ and $A_s^- = A_s^*$ as lower sideband computation requires the conjugate of the motion amplitude. For example, longitudinal motion is described by $z_s(x;y) = 1$ and thus $A_s = Z_s$ is in meters. Rotational motion is defined by $z_s(x;y) = x$ or y with $A_s = \Theta_{x/y}$ in radians. Making the assumption that $|A_s| \ll `1064nm, the creation of signal sidebands of a carrier field : math : `a_{c,j}$ is linearised [36],

$$a^{\pm}_{s,jnm} = irkZ^{\pm}_s \sum_{n',m'} a_{c,jn'm'} K^0_{nmn'm'}$$

where $Z_s^+ \equiv Z_s$ and $Z_s^- \equiv Z_s^*$ are the surface motion along the beam axis, and $K^0_{nmn'm'}$ is the HOM coupling due to a static mirror distortion. Note that this is for the side of the mirror defined as "positive" i.e. the port1 (*p1*) side. For the opposite side (*p2*), the longitudinal motion is negative, so an extra 180° phase is added.

In Finesse, we must also propagate the carrier field to, and the generated field away from, the mirror's surface by the mirror's static tuning ϕ. The overall couplings for a carrier field $a_{c,j}$ with frequency offset $f_{c,j}$ and signal field $a_{s,j}$ with frequency offset $f_{s,j}$ are then

$$a^{\pm}_{s,jnm} = e^{i\phi(1+\frac{f_{s,j}}{f_0}+\frac{f_{c,j}}{f_0})} irkZ^{\pm}_s \sum_{n',m'} a_{c,jn'm'} K^0_{nmn'm'}$$

for the positive side and

$$a^{\pm}_{s,jnm} = -e^{-i\phi(1+\frac{f_{s,j}}{f_0}+\frac{f_{c,j}}{f_0})} irkZ^{\pm}_s \sum_{n',m'} a_{c,jn'm'} K^0_{nmn'm'}$$

for the negative, where f_0 is the simulation's default frequency.

6.8 Thermal effects

6.8.1 Thermal lensing and deformations using Hello-Vinet

A small fraction of power is absorbed from an incident beam whenever it transmits through or reflects from a physical optical component. This absorbed power is dumped into the optic in the form of heat. Two main effects are apparent when this occurs; a thermal gradient develops within the substrate of the optic and the refractive index changes, the thermo-refractive effect; and the optic expands and deforms itself and any reflective surface, the thermo-elastic effect.

Determining how these processes affect the optical field usually requires finite element analysis tools solving the thermal diffusion and linear elastic problems for a 3D geometry. Alternatively there are also a collection of analytic solutions for on-axis cylindrically symmetric heating beams interacting with a cylindrical mirror. These formulae are known as the Hello-Vinet equations within the gravitational wave community, after the two authors who derived them. A farily comprehensive review article [37] exists which summarises the work by Vinet.

Finesse provides some of the Hello-Vinet calculations which can all be found in `finesse.thermal.hello_vinet`. In this module are functions for computing heating from fundamental HG00 Gaussian beams as well as more generic axisymmetric heating irradiance patterns. The former functions have a prefix *_HG00*.

Note

The functions provided by `finesse.thermal.hello_vinet` use a coordinate system with the axial coordinate negative to that of [37]. The front/HR surface of the mirror is at z=+h/2 and the back/AR surface is at z=-h/2. The surface normal to the HR surface is positive.

Steady state substrate temperature

The thermo-refractive effect generates a thermal lens within the substrate of an optic. The temperature change throughout the substrate can be found using the following function:

```
import finesse
import numpy as np
import matplotlib.pyplot as plt
import finesse.thermal.hello_vinet as hv
from finesse.materials import FusedSilica
finesse.init_plotting()

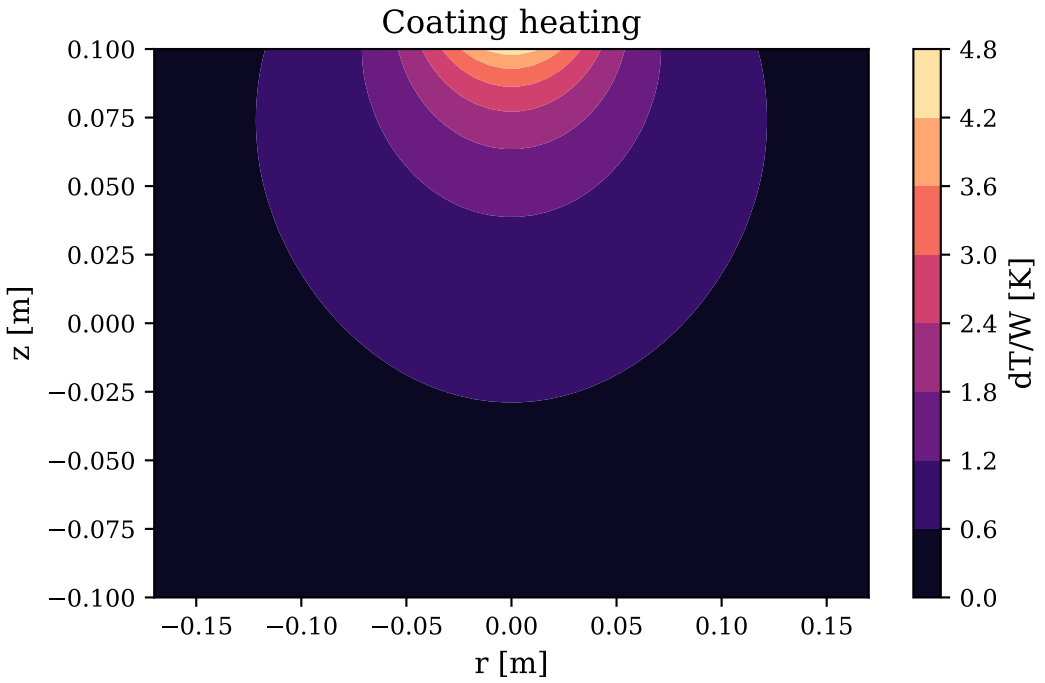
a = 0.17 # mirror radius
h = 0.2  # mirror thickness
w = 53e-3 # spot size radius
r = np.linspace(-a, a, 50) # radial points
z = np.linspace(-h/2, h/2, 100) # longitudinal points
material = FusedSilica

T_coat_per_W, T_bulk_per_W = hv.substrate_temperatures_HG00(
    r, z, a, h, w, material
)
```

The returned arrays are temperature change per Watt of optical power. Therefore to get the actual temperature changes you must multiply the results by the incident power on the coating or the power going through the substrate.

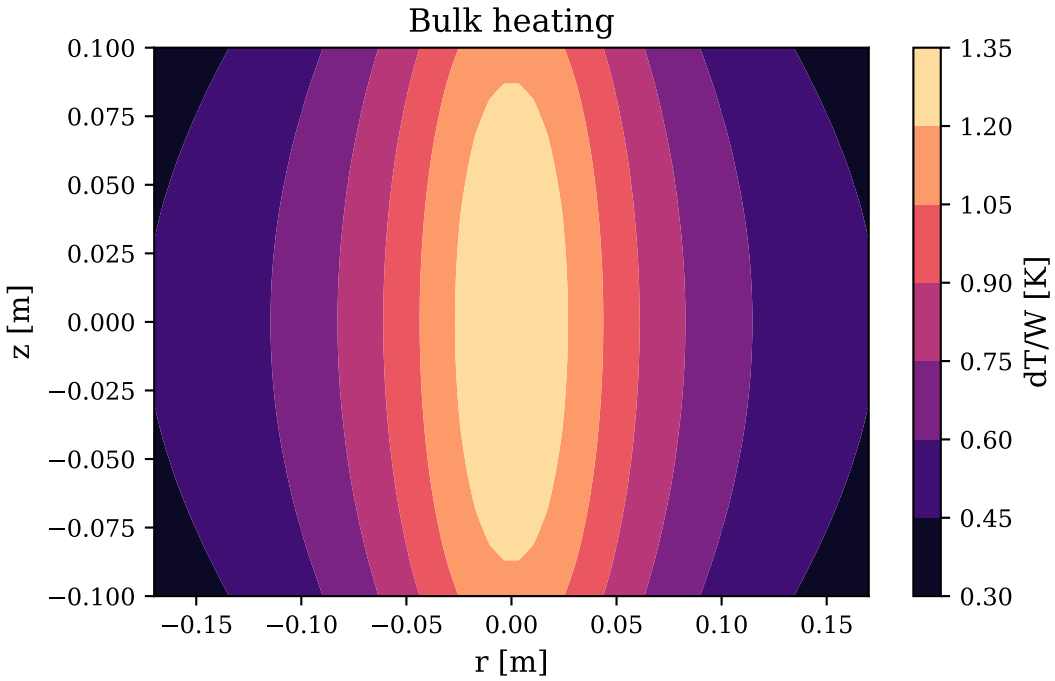
```
plt.title("Coating heating")
plt.contourf(r, z, T_coat_per_W)
plt.xlabel("r [m]")
plt.ylabel("z [m]")
plt.colorbar(label='dT/W [K]')
```

```
<matplotlib.colorbar.Colorbar at 0x7880a3b017f0>
```




```
plt.title("Bulk heating")
plt.contourf(r, z, T_bulk_per_W)
plt.xlabel("r [m]")
plt.ylabel("z [m]")
plt.colorbar(label='dT/W [K]')
```

```
<matplotlib.colorbar.Colorbar at 0x7880a199b890>
```



Currently the code only supports HG00 heating beams.

Steady state thermal lensing

The effective thermal lens due to this temperature change throughout the substrate is computed with the integral

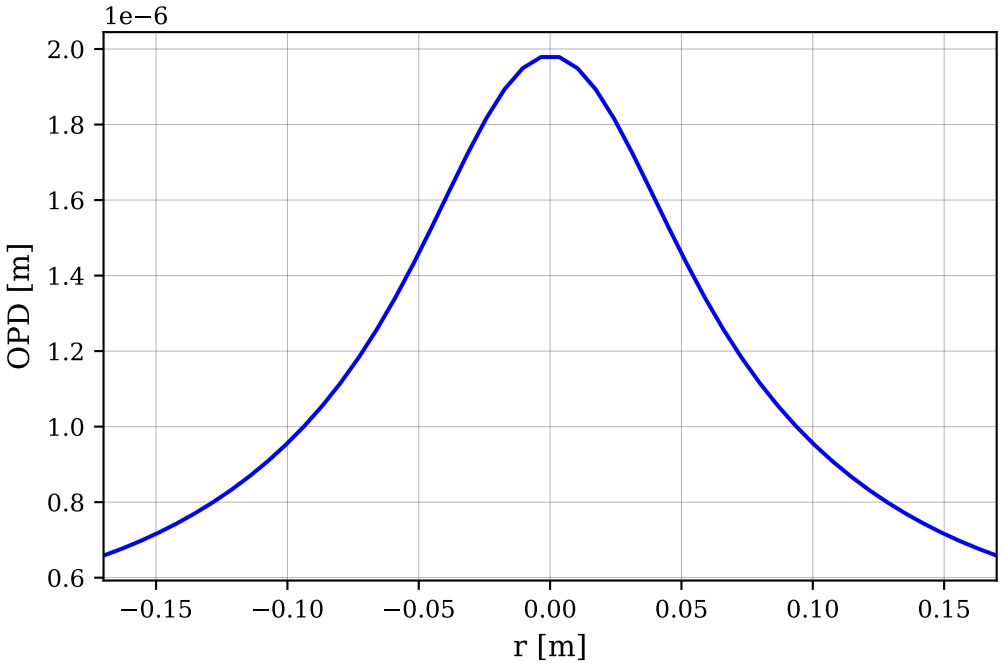
$$Z(r) = \frac{dn}{dT} \int T(r, z) dz$$

The above `finesse.thermal.hello_vinet.substrate_temperatures_HG00()` outputs can be numerically integrated, but we can also use the `finesse.thermal.hello_vinet.thermal_lenses_HG00()` method for a direct calculation. Using the same parameters from before we find the thermal lens with

```
Z_coat_per_W, Z_bulk_per_W = hv.thermal_lenses_HG00(
    r, a, h, w, material
)

plt.plot(r, Z_coat_per_W)
plt.xlabel("r [m]")
plt.ylabel("OPD [m]")
```

```
Text(0, 0.5, 'OPD [m]')
```



The returned 1D arrays are the optical path difference (OPD) in meters. This is the extra length difference the beam will experience on a single pass through the mirror. Again this has not been scaled for any incident power yet.

Steady state thermal displacements

Displacements within an optic are also caused by the thermal expansion of a material as it heats up. An Incident HG00 beam deform the surfaces of the mirrors as well as slightly expand the substrate.

Substrate displacements from coating absorption can be computed with:

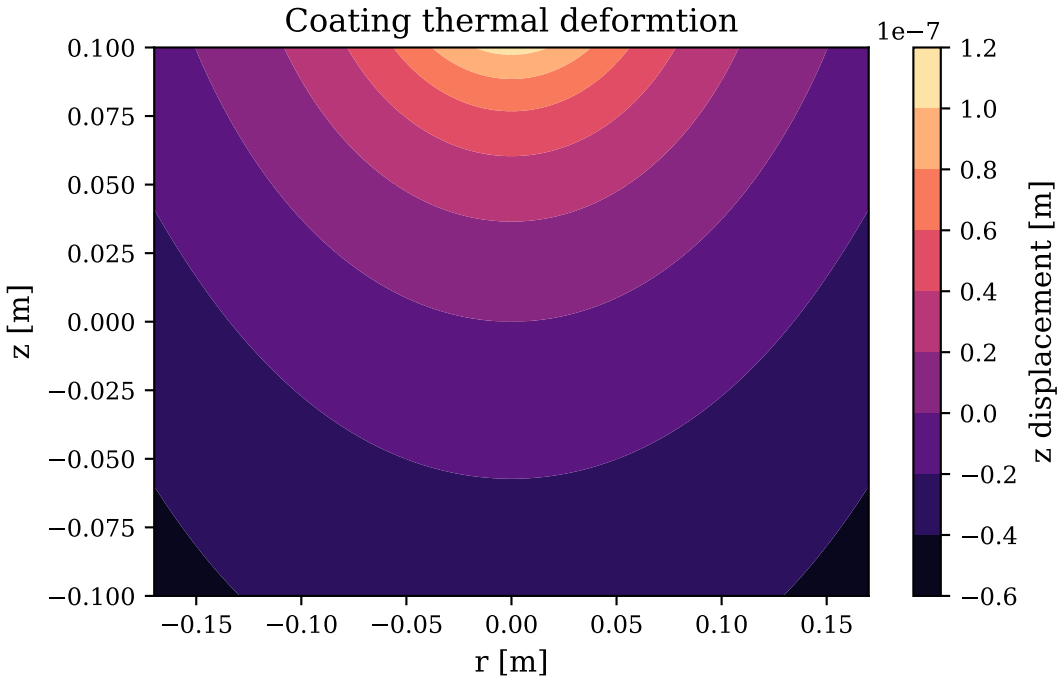
```
w = 53e-3
a = 170e-3 # radius
h = 0.2 # thickness
z = np.linspace(-h/2, h/2, 1000)
r = np.linspace(-a, a, 100)
# z displacement throughout the substrate
U_z_coat = hv.substrate_thermal_expansion_depth_HG00(r, z, a, h, w, FusedSilica)
```

(continues on next page)

(continued from previous page)

```
plt.contourf(r, z, U_z_coat)
plt.colorbar(label='z displacement [m]')
plt.xlabel("r [m]")
plt.ylabel("z [m]")
plt.title("Coating thermal deformtion")
```

```
Text(0.5, 1.0, 'Coating thermal deformtion')
```



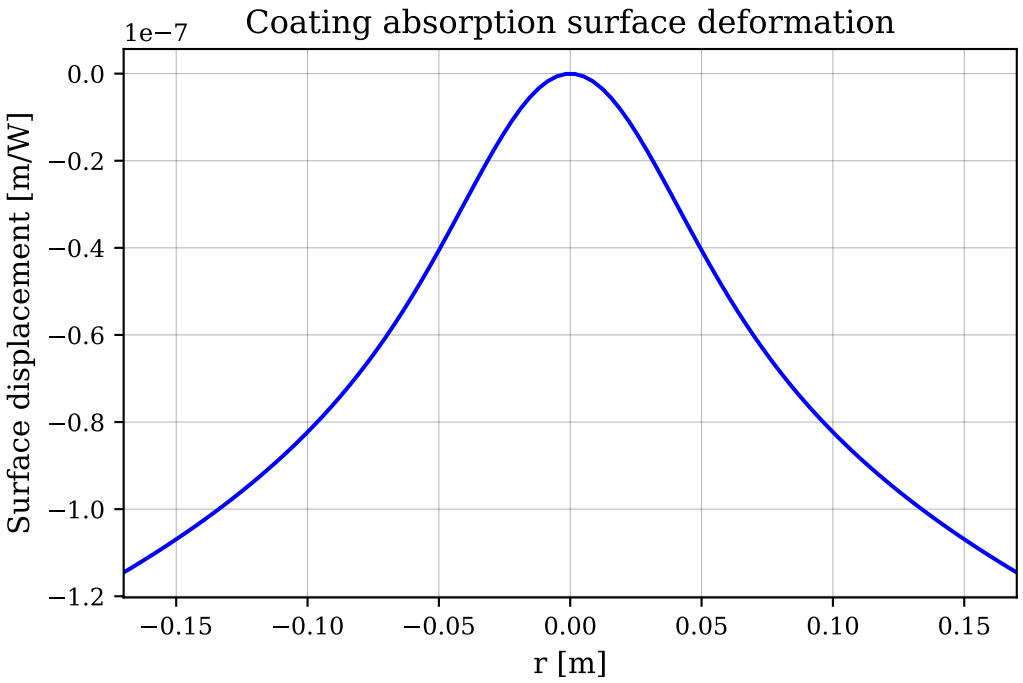
We can also just plot the surface deformation on the absorbing side.

```
w = 53e-3
a = 170e-3 # radius
h = 0.2 # thickness
r = np.linspace(-a, a, 100)

U_s_coat = hv.surface_deformation_coating_heating_HG00(r, a, h, w, FusedSilica)

plt.plot(r, U_s_coat)
plt.ylabel('Surface displacement [m/W]')
plt.xlabel("r [m]")
plt.title("Coating absorption surface deformation")
```

```
Text(0.5, 1.0, 'Coating absorption surface deformation')
```



Note

Care should be taken when using these deformation calculations in simulations to ensure that the correct coordinate system is being used. In FINESSE, the positive z direction for mirrors are for the surface normal on the port 1 side. Therefore, if the port 1 side is the front or HR surface where the power is being absorbed, the above `finesse.thermal.hello_vinet.surface_deformation_coating_heating_HG00()` gives the correct displacement which can be added directly to a mirror map. If the port 2 side is the front or HR surface however, this displacement must be negated before being added to a mirror map.

Axisymmetric heating profiles

The functions discussed above also have generic versions which compute the thermal lensing and deformation for generic axisymmetric heating profiles. The first step is to compute the Fourier-Bessel expansion of the radial heating profile. With this data you can then pass it directly to the Hello-Vinet functions to compute the resulting effects.

Here is an example computing how a more complex heating pattern is decomposed:

```
import finesse.materials
import finesse.thermal.hello_vinet as hv
import numpy as np
import matplotlib.pyplot as plt
from scipy.special import eval_hermite

finesse.init_plotting()
material = finesse.materials.FusedSilica
a = 0.17
h = 0.2
w = 53e-3
r = np.linspace(0, a, 101)

# A non-normalised 5th order hermite radial distribution
E = eval_hermite(5, np.sqrt(2)*r/w) * np.exp(-(r/w)**2)
I = E*E
plt.plot(r, I, label='I(r)')

# perform Fourier-Bessel decomposition
```

(continues on next page)

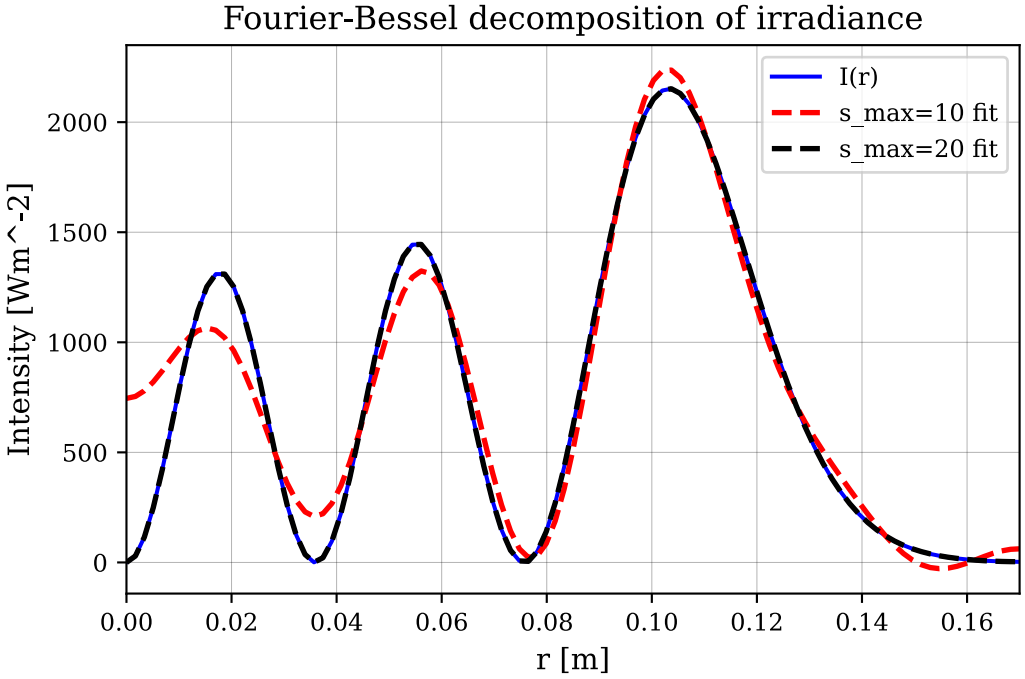
(continued from previous page)

```
data = hv.get_p_n_s_numerical(I, a, 10, material)
plt.plot(r, hv.eval_p_n_s_numerical(data), ls='--', lw=2, label='s_max=10 fit')

# perform Fourier-Bessel decomposition with
# a higher order decomposition
data = hv.get_p_n_s_numerical(I, a, 20, material)
plt.plot(r, hv.eval_p_n_s_numerical(data), ls='--', lw=2, label='s_max=20 fit')

plt.legend()
plt.xlabel("r [m]")
plt.ylabel("Intensity [Wm^-2]")
plt.title("Fourier-Bessel decomposition of irradiance")
```

```
Text(0.5, 1.0, 'Fourier-Bessel decomposition of irradiance')
```

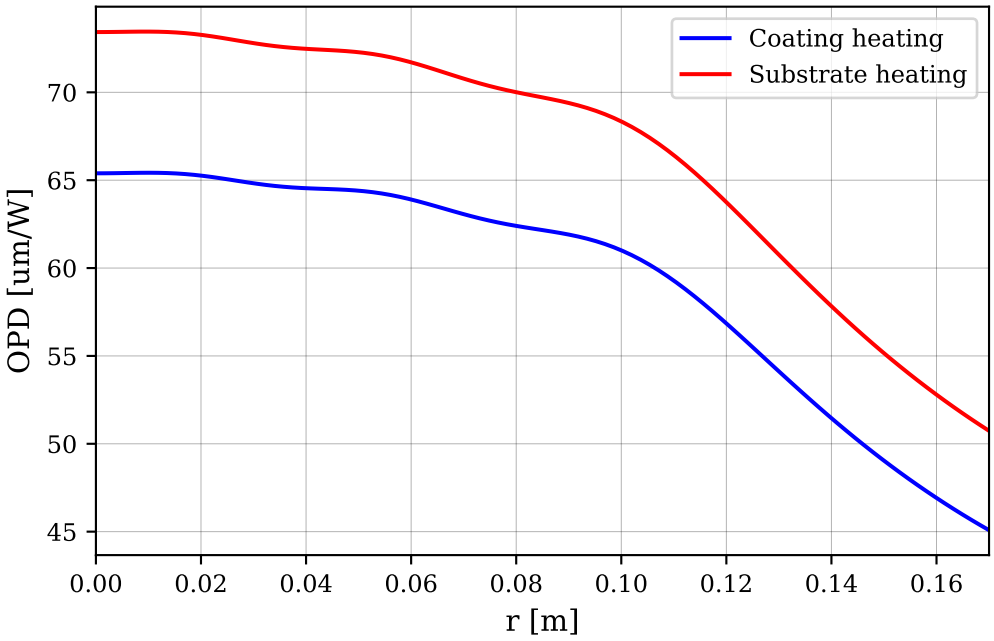


It is important to check that your heating profile is adequately described by the Fourier-Bessel decomposition. Higher spatial variance requires a larger number of roots to be found (the `s_max` parameter). When using low spatial sampling it may also be necessary to experiment with the Newton-Cotes weighting order for more accurate numerical integration. Higher spatial frequency changes in the heating profile will not decompose into bessel functions particularly well. This is not so much of an issue as naturally heating profiles tend to be smooth and slowly varying.

Next we can use this decomposition data in the generic forms of the thermal equations. For example, computing the thermal lensing in a substrate:

```
W_coat, W_bulk = hv.thermal_lenses(
    data, h
)
plt.plot(r, W_coat/1e-6, label='Coating heating')
plt.plot(r, W_bulk/1e-6, label='Substrate heating')
plt.xlabel('r [m]')
plt.ylabel('OPD [um/W]')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x78809becbb10>
```

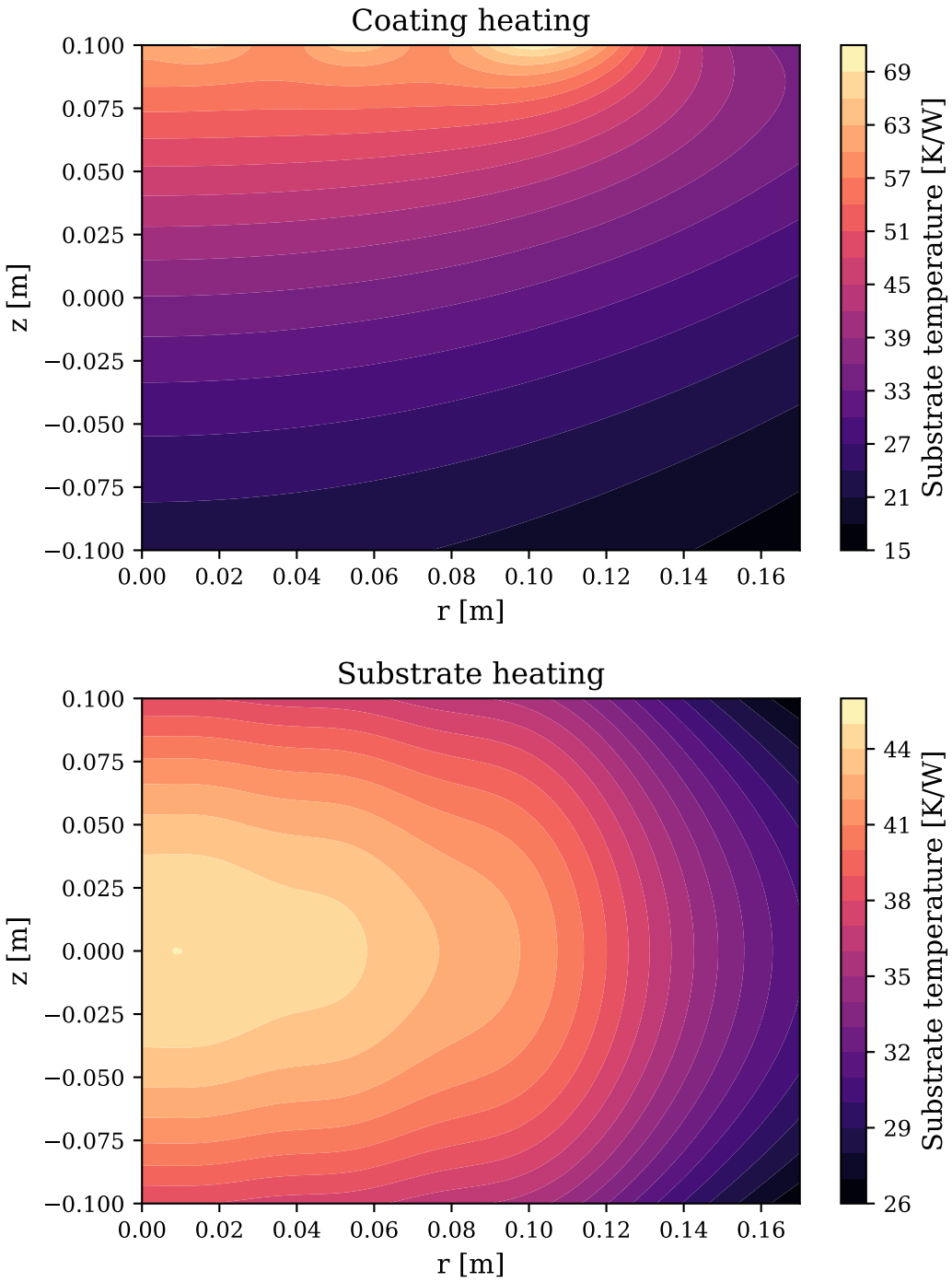


or computing the temperature distribution through the substrate:

```
z = np.linspace(-h/2, h/2, 300)
T_coat, T_bulk = hv.substrate_temperatures(
    data, z, h
)
plt.figure()
plt.contourf(r, z, T_coat, levels=20)
plt.colorbar(label='Substrate temperature [K/W]')
plt.xlabel("r [m]")
plt.ylabel("z [m]")
plt.title("Coating heating")

plt.figure()
plt.contourf(r, z, T_bulk, levels=20)
plt.colorbar(label='Substrate temperature [K/W]')
plt.xlabel("r [m]")
plt.ylabel("z [m]")
plt.title("Substrate heating")
```

```
Text(0.5, 1.0, 'Substrate heating')
```



6.8.2 Thermal lensing and deformations from ring heaters

Ring heaters are used to introduce a thermal substrate lens within an optic and also actuate on the surface curvature of the mirror. Such devices allow us to compensate for self-heating from laser beam absorption. The equations implemented in `finesse.thermal.ring_heater` are from [38] which analytically provides the substrate temperature profile through a cylindrical optic when heated around its barrel over a small area.

Note

The functions provided by `finesse.thermal.ring_heater` use a coordinate system with the axial coordinate negative to that of [38]. The front/HR surface of the mirror is at z=+h/2 and the back/AR surface is at z=-h/2. The surface normal to the HR surface is positive.

Steady state substrate temperature

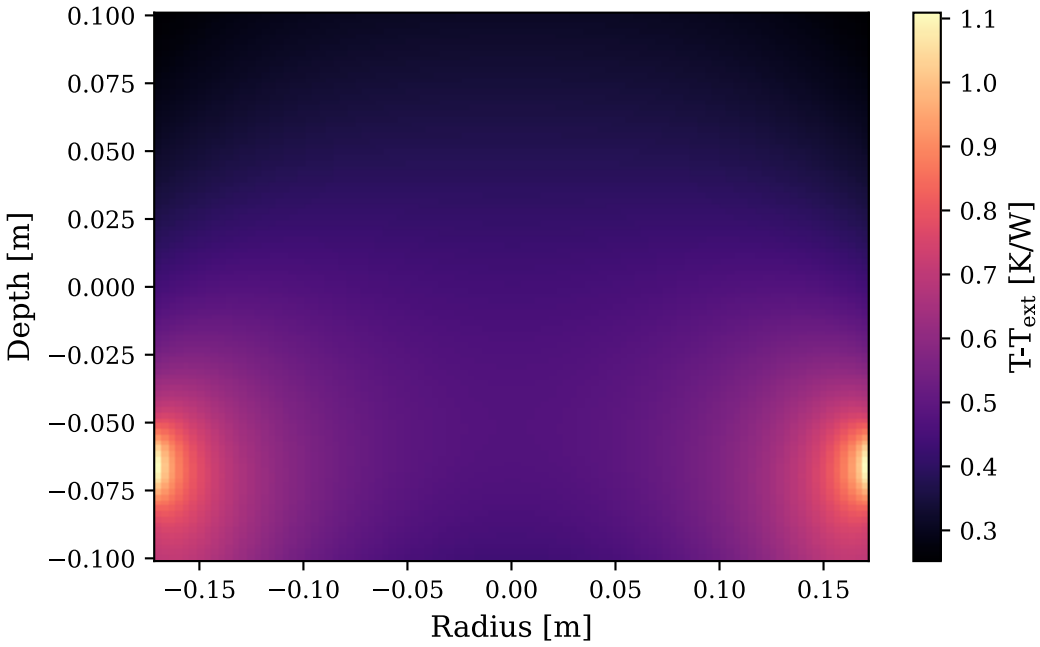
```
import finesse
import numpy as np
import matplotlib.pyplot as plt
import finesse.thermal.ring_heater as ring_heater
from finesse.materials import FusedSilica
finesse.init_plotting()

a = 0.17 # mirror radius
b = -56e-3 # LIGO ring heater position
c = -76e-3 # LIGO ring heater position
h = 0.2  # mirror thickness
w = 53e-3 # spot size radius
r = np.linspace(-a, a, 100) # radial points
z = np.linspace(-h/2, h/2, 100) # longitudinal points
material = FusedSilica

T_rh_per_W = ring_heater.substrate_temperature(r, z, a, b, c, h, material)

plt.pcolormesh(r, z, T_rh_per_W, rasterized=True, shading='auto')
plt.colorbar(label='T-$\\mathrm{T}_{\\mathrm{ext}}$ [K/W]')
plt.xlabel("Radius [m]")
plt.ylabel("Depth [m]")
```

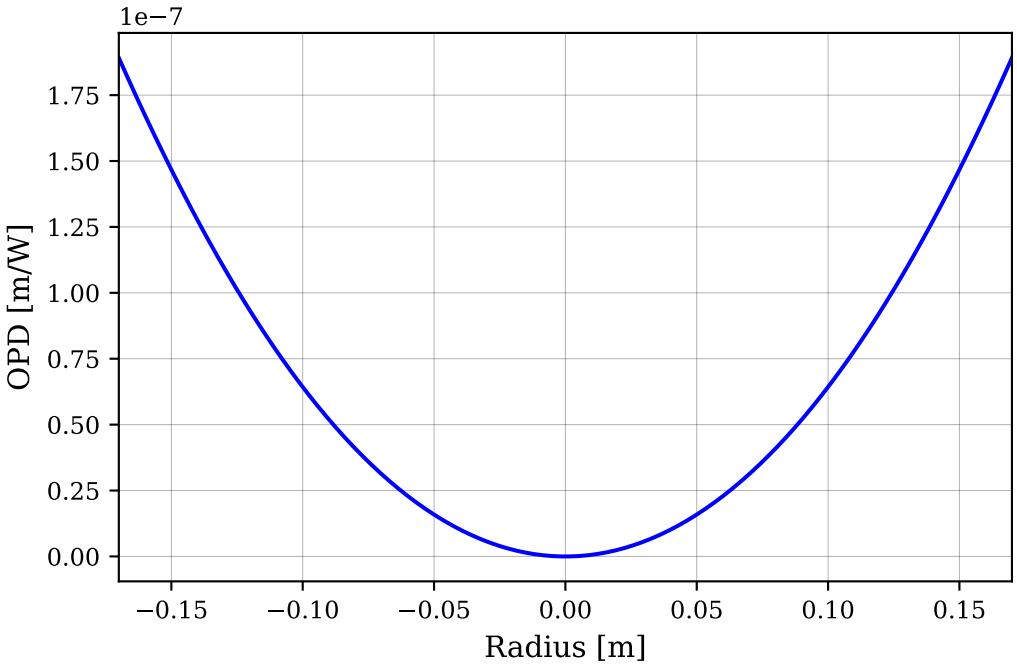
```
Text(0, 0.5, 'Depth [m]')
```



The total substrate thermal lens optical path difference can also be computed using the `finesse.thermal.ring_heater.thermal_lens()` method, again returning a distortion per Watt of power ring heater power absorbed by the mirror.

```
Z_rh_per_W = ring_heater.thermal_lens(r, a, b, c, h, material)
plt.plot(r, Z_rh_per_W)
plt.xlabel("Radius [m]")
plt.ylabel("OPD [m/W]")
```

```
Text(0, 0.5, 'OPD [m/W]')
```



Steady state Thermoelastic deformation

The note [39] describes how to calculate the thermal deformation of surfaces due to an arbitrary ring heater profile around the barrel of a mirror. `surface_deformation()`, `substrate_deformation_depth()` implement this calculation. In the examples below we plot the results for an Advanced LIGO like test mass and ring heater setup.

```
import finesse
import numpy as np
import matplotlib.pyplot as plt
import finesse.thermal.ring_heater as ring_heater
from finesse.materials import FusedSilica

finesse.init_plotting(fmts=["png"])

a = 0.17
b = -50e-3
c = -70e-3
h = 0.2
z = np.linspace(-h / 2, h / 2, 100)
material = FusedSilica
a = 170e-3  # radius of optic [m]
h = 0.2  # thickness of optic [m]

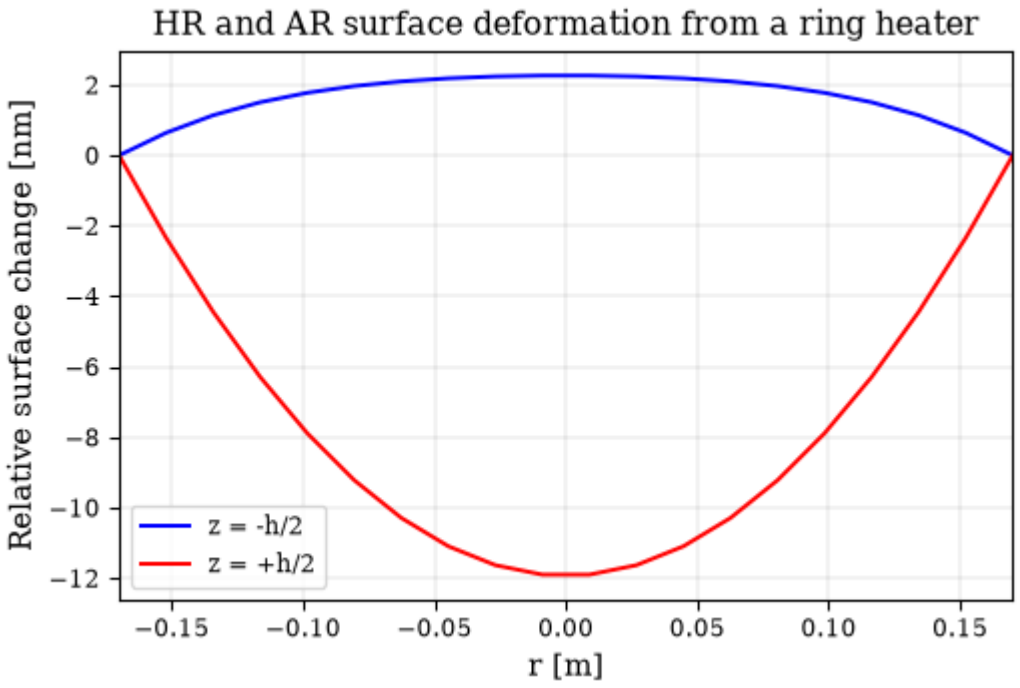
r = np.linspace(-a, a, 20)  # radial points along optic
z = np.array([-h / 2, +h / 2])  # depth points along optic

dz = ring_heater.substrate_deformation_depth(r, z, a, b, c, h, material)

AR = dz[0, :] - dz[0, :].min()
HR = dz[1, :] - dz[1, :].max()

plt.plot(r, AR / 1e-9, label="z = -h/2")
plt.plot(r, HR / 1e-9, label="z = +h/2")
plt.legend()
plt.xlabel("r [m]")
plt.ylabel("Relative surface change [nm]")
plt.title("HR and AR surface deformation from a ring heater")
```

```
Text(0.5, 1.0, 'HR and AR surface deformation from a ring heater')
```

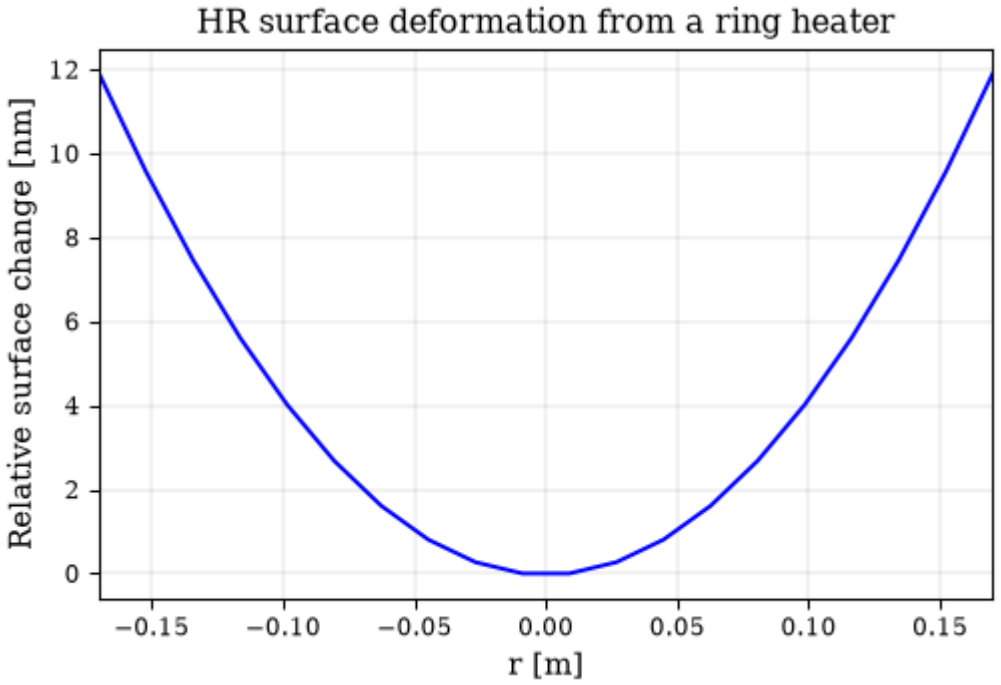


Or just the HR surface change:

```
dz = ring_heater.surface_deformation(r, a, b, c, h, material)

plt.plot(r, dz / 1e-9, label="z = 0")
plt.xlabel("r [m]")
plt.ylabel("Relative surface change [nm]")
plt.title("HR surface deformation from a ring heater")
```

```
Text(0.5, 1.0, 'HR surface deformation from a ring heater')
```



6.9 Bespoke GW Models

FINESSE can be extended with additional packages to model particular gravitational wave detectors. Two such packages are finesse-ligo and finesse-virgo. Additionally, packages exist to combine Finite Element Analysis (FEA) of the test masses, with optical simulations of the interferometer. One example is the test-mass-thermal-state package.

This example uses finesse-ligo to obtain the transfer function from Meters of test mass motion into Watts of optical power for the LIGO Livingston Observatory.

> **Warning**
>
> This example is based on the LSC example in the finesse-ligo documentation. However, the code here is not automatically tested, as this would require making *finesse-ligo* a finesse dependency.
>
> Please see the finesse-ligo documentation for up-to-date information.

6.9.1 Importing *finesse-ligo*

If it is installed, *finesse-ligo* can be imported using *import finesse_ligo*:

```
import finesse
import finesse_ligo
import finesse.analysis.actions as fac

from finesse_ligo.factory import aligo
from finesse.plotting import bode
from finesse_ligo.suspension import QUADSuspension
```

finesse-virgo works similarly. We then proceed to build the FINESSE model. A parameter file is distributed with *finesse-ligo* that contains the best estimate of all parameters in the interferometers. In this case, we will load the parameter file for the LIGO Livingston Observatory:

```
finesse.init_plotting()
factory = aligo.ALIGOFactory("llo_O4.yaml")
```

The *ALIGOFactory* function can accept either the name of a parameter file, or, a path to a parameter file, stored somewhere on your system.

6.9.2 Choosing what to model

It is computationally expensive to model the entire interferometer. Usually, one wants to model some subset of the physics and some subset of the ports. We now choose what to model using the factory function:

```
# Make a HOM model
factory.reset()
factory.options.QUAD_suspension_model = finesse_ligo.suspension.QUADSuspension
llo3 = factory.make()
llo3.modes("even", maxtem=2)
```

In this case, we have chosen to model an interferometer, with suspended optics and the HG00, HG20 and HG02 optical modes.

6.9.3 Initialising the locks

A real interferometer has a real operating point enforced by feedback control. It is critical to find this operating point using simulated locking actions. In this case, the locks and gains are known and are distributed with *finesse-ligo*:

```
llo3.run(fac.Series(aligo.InitialLock(), aligo.DARM_RF_to_DC()))
```

6.9.4 Modelling the Interferometer

We are now in a position to do some modelling. Such as plotting the the transfer function:

```
analysis = fac.FrequencyResponse(
    np.geomspace(0.1, 10e3, 300), "DARM.AC.i", "AS.DC"
)
```

For further information, please see the documentation for the extension packages, or see the Getting started (page 30) page.

6.10 Viewing logs

FINESSE contains a comprehensive logging system, issuing useful information during the parsing, building, running and output phases of a KatScript or Python script.

6.10.1 Making log messages visible

There are numerous ways to configure how and where logs are displayed or stored. Some common examples are described in the following subsections.

Option 1: use 'logging.basicConfig()'

The `logging.basicConfig()` function is a quick way to set the level, stream and format of the logs emitted by FINESSE:

```
import logging

# Anything after this call that emits logs will have them printed to the current
# shell.
logging.basicConfig(level=logging.INFO)
```

The above call sets up logging to `sys.stderr`, which usually means that log messages will appear in the shell or notebook with which you run FINESSE. It also explicitly sets the level to `logging.INFO`, which displays messages with *info* level or higher (info, warning, error, critical).

Option 2: use command line interface option

The command line interface (CLI) supports changing the log verbosity of executed scripts via option flags. See the CLI docs (page 384) for more information.

Option 3: wrap your code in a 'logs' context

The context manager `logs()` provides a quick way to turn on verbose logging for a small block of code:

```
import logging
from finesse import Model
from finesse.utilities import logs

# This sets up the root logger to write to stderr.
logging.basicConfig()

model = Model()

with logs(logging.getLogger(), level="debug"):
    # Debug messages emitted by the following call get printed to stderr.
    model.parse("laser l1 P=1")
```

The `logs()` function also accepts a `handler` argument which can for example be set to a `FinesseStreamHandler` which allows filtering of particular channels:

```
import logging
from finesse import Model
from finesse.utilities import logs
from finesse.utilities.logging import FinesseStreamHandler

# Instead of using logging.basicConfig, we configure a handler ourselves and pass it
# as an argument to `logs` below.
handler = FinesseStreamHandler()
# Exclude logs emitted in the parser channel.
handler.exclude("finesse.script.parser")

model = Model()

with logs(logging.getLogger(), level="debug", handler=handler):
    # Debug messages emitted by the following call get printed to stderr.
    model.parse("laser l1 P=1")
```

In the code above, `FinesseStreamHandler.exclude()` is used to exclude the `finesse.script.parser` log channel from being displayed. The method also has basic wildcard support, allowing whole channel groups to be excluded. This method can be useful to reduce log message "noise" when debugging complicated code blocks.

Note

If you pass a handler to `logs()`, it is added to the root logger as an *additional* handler for the duration of the context. If you also make a call to `logging.basicConfig()` earlier in your script, there will be *two* handlers printing to `sys.stderr`, resulting in log messages being printed twice. To avoid this, avoid using `logging.basicConfig()` when the `handler` argument is specified.

6.11 Command line interface

FINESSE contains a powerful command line interface (CLI), available as a program called **`kat3`** on the same system path where FINESSE was installed. The CLI's primary function is to directly run and display the results from KatScript files, but it also provides various utility and helper functionality as described below.

6.11.1 Quick start

Specifying a script path as the only argument to **`kat3`** will have the CLI parse and run the contents and show the plotted results:

```
$ kat3 myscript.kat
```

Hint

The CLI also accepts input from the standard input stream using the Unix convention of specifying a `-` in place of the path. Power users might find writing e.g. `echo "m m1 R=0.99 T=0.01" | kat3 -` to be a useful time saver.

The following sections describe how to get help, achieve finer control over simulations, and to perform other tasks with the CLI. A command reference (page 387) is also provided.

6.11.2 Getting help

To show the various subcommands available in **`kat3`**, add the `--help` (page 387) flag to the command:

```
kat3 --help
```

The `*` next to `run` indicates that it is the default subcommand assumed when no subcommand is explicitly specified (meaning `kat3 myscript.kat` is equivalent to `kat3 run myscript.kat`).

The `--help` (page 387) flag can be appended to subcommands too:

```
kat3 info --help
```

6.11.3 Running KatScript files

More control over how simulations are run and how their results are displayed is provided by the **`kat3 run`** subcommand. It can control whether to show or hide plots, trace results and other model information, and can enable legacy parsing mode:

```
kat3 run --help
```

6.11.4 Looking up KatScript syntax

The **`kat3 syntax`** subcommand shows the available KatScript elements, commands and analyses. When executed without arguments, it outputs the full list in a way that is convenient to command line tools like grep. You may optionally specify `search terms` (page 388), which can contain simple wildcards:

- `*` matches 0 or more characters
- `?` matches any single character
- `[abc]` matches any characters in abc
- `[!abc]` matches any characters not in abc

For example, all directives containing "pd" can be found with:

```
kat3 syntax "*pd*"
```

Similarly, all "pd<N>" directives can be found with:

```
kat3 syntax "pd?"
```

Warning

Most shells interpret patterns containing asterisk characters (`*`) as wildcards matching files in the current directory, and substitute these into the command passed to the program, in this case as extra search queries. Specify the search pattern(s) inside quotes (e.g. `"pd*"`) to avoid potentially unintended matches.

6.11.5 Controlling verbosity

Various flags are available in the CLI to show, filter and hide warnings, errors and debug information produced by Finesse.

Showing or hiding log messages

Various parts of the Finesse code generate messages that can be useful to inform the user of interesting decisions, events, warnings, and errors during execution. This can assist in debugging KatScript or Finesse itself, or to indicate potential pitfalls. Messages are emitted in *channels* with names that correspond to the area of the code that created them.

Messages from any channel with *WARNING*, *ERROR* or *CRITICAL* severity are shown by default when running a command via the CLI. Log messages with lower severity can be made visible through use of the `--verbose` (page 387) (or abbreviated `-v` (page 387)) flag on any command. Specifying it once will result in *INFO* messages becoming visible, and specifying it a further time (i.e. `-vv` (page 387)) will make *DEBUG* messages become visible.

In contrast, specifying `--quiet` (page 387) (or abbreviated `-q` (page 387)) one or more times will make FINESSE show only *ERROR* and *CRITICAL* messages, respectively.

Note

DEBUG logs contain a huge amount of data and are typically only useful for FINESSE developers. It is not recommended to enable debug verbosity unless asked by a developer.

Errors that result in execution halting (such as KatScript syntax errors) are always displayed regardless of the verbosity settings here.

Hint

You may wish to exclude particular log channels from being displayed by the command line interface. For instance, enabling *DEBUG* verbosity results in a large quantity of messages, some of which you may not be interested in seeing. Channels can be disabled by passing one or more `--log-exclude` (page 387) flags with exclude patterns. The same wildcards as those in Looking up KatScript syntax (page 385) are supported.

For example, to exclude channels containing *script*, specify `--log-exclude *script*`.

Showing tracebacks

By default, tracebacks generated by Python when uncaught exceptions occur are suppressed, with only the error message being displayed. This suppression can be switched off by enabling debug mode with the flag `--debug` (page 387).

Fancy error messages

When the current terminal supports it, the locations of errors in KatScript are by default displayed with red text in corresponding error messages. If this behaviour is not desired, it can be switched off with `--no-fancy-errors` (page 387), whereupon the behaviour will instead be to display markers (`^` characters) on the line *below* the part(s) of the script that contains the error(s).

6.11.6 Keeping multiple plots open using job control

You may wish to keep the plot results from a simulation open while performing another, e.g. to be able to compare them. This is possible to do using features provided in command shells on most platforms. Guidance for specific shells is provided below.

See also

Another way to keep old plots "open" is to use an interactive notebook such as JupyterLab.

Bourne-compatible shells (e.g. 'bash')

The ampersand (`&`) operator can be appended to the end of a command to execute it in a background process, which has the effect of running the command but freeing up the shell for subsequent commands rather than blocking it:

```
$ kat3 run myscript.kat &
```

A command started in the foreground (that is to say, without an `&`) can be moved to the background to free up the shell for another command by first suspending its execution using `Ctrl + Z` then executing the command `bg`. To return the job to the foreground again, the command `fg` can be used. When there are multiple commands running in the background, they can be listed with `jobs`, and the job number *N* can be brought to the foreground with `%N`.

Windows command prompt

Jobs can be started in the background by prepending the command with `START /B`:

```
C:\> START /B kat3 run myscript.kat
```

6.11.7 Command reference

Shared options

Many **`kat3`** CLI subcommands share the following common options:

`-v, --verbose`
: Increase verbosity of log output. Specify multiple times to increase verbosity further. See Controlling verbosity (page 385) for more information.

`-q, --quiet`
: Decrease verbosity of log output. Specify multiple times to decrease verbosity further. See Controlling verbosity (page 385) for more information.

`--fancy-errors, --no-fancy-errors`
: Highlight script error locations in red rather than marking them on the following line. See Controlling verbosity (page 385) for more information. On by default.

`--debug`
: Display full exception tracebacks. See Controlling verbosity (page 385) for more information.

`--log-exclude` `PATTERN`
: Ignore log records from channels matching `PATTERN`. See Controlling verbosity (page 385) for more information.

`--legacy`
: Parse `INPUT_FILE` (page 387) as Finesse 2 style KatScript. See Finesse 2 support (page 542) for more information.

`--help`
: Display help text for the current command.

kat3

Simulation program for laser interferometers.

`INPUT_FILE`
: The path to the input file to run. The destination must exist and be readable by the current user. Use – to denote the standard input stream.

kat3 info

Print information about a model parsed from a script.

`INPUT_FILE`
: The path to the input file to run. The destination must exist and be readable by the current user. Use – to denote the standard input stream.

`--summary, --no-summary`
: Print summary of parsed model. Enabled by default.

`--graph, --no-graph`
: Display graph of parsed model. Disabled by default.

-t, --type

Network type to display when --graph (page 387) is specified. Choose from "full", "components", or "optical".

--layout LAYOUT

Graph layout to use when --graph (page 387) is specified. Use --list-graph-layouts (page 388) to print a list of available layouts.

--graphviz

Use Graphviz (via pygraphviz) to display the resulting graph, rather than networkx. Graphviz and pygraphviz must be installed for this option to work.

--list-graph-layouts

Print available graph layouts. This will contain at least those of networkx. If pygraphviz is installed, additional Graphviz-based layouts will be available.

kat3 syntax

Query the kat script syntax documentation.

QUERY

Search term. May be specified zero or more times. Basic wildcard syntax is supported:

- * matches 0 or more characters
- ? matches any single character
- [abc] matches any characters in abc
- [!abc] matches any characters not in abc

--elements, --commands, --analyses

Limit the matched results to only elements, commands and/or analyses.

kat3 convert

Convert and normalize kat script to canonical form.

This can be used to convert FINESSE 2 or FINESSE 3 scripts to canonical FINESSE 3 form. The resulting script is arranged in standard order and with standard spacing.

If OUTPUT_FILE (page 388) is not specified, it defaults to standard output.

To parse FINESSE 2 files, specify --legacy (page 387).

Warning

This is an **experimental** command. It may not generate accurate nor valid KatScript!

INPUT_FILE

The path to the input file to convert. The destination must exist and be readable by the current user. Use - to denote the standard input stream.

OUTPUT_FILE

The path to the output file to write. The destination must be writable by the current user. Defaults to -, indicating the standard output stream.

6.12 Numerical precision

6.12.1 In elements

Normal integers and floats are represented in elements using Python data types. That means integers have arbitrary precision and are only limited by system memory, while floats are usually limited to 53 bits of precision, equivalent to about 16 or 17 digits (a good discussion can be found here).

Numerical arrays in Finesse elements, e.g. for the computation of ABCD matrices, are represented internally using Numpy data structures. Most integers are stored using `int64` representation. Larger numbers use `uint64` then default back to Python `int` objects.

The precision limits for integers and floats for your machine can be found in the following way using Numpy:

```
import numpy as np
```

For integers:

```
print(np.iinfo(np.uint64))
```

```
Machine parameters for uint64
---------------------------------------------------------------
min = 0
max = 18446744073709551615
---------------------------------------------------------------
```

For floats:

```
print(np.finfo(np.float64))
```

```
Machine parameters for float64
---------------------------------------------------------------
precision = 15   resolution = 1e-15
machep = -52   eps =        2.220446049250313e-16
negep =  -53   epsneg =     1.1102230246251565e-16
minexp = -1022   tiny =       2.2250738585072014e-308
maxexp = 1024   max =        1.7976931348623157e+308
nexp =   11   min =        -max
smallest_normal = 2.2250738585072014e-308   smallest_subnormal = 5e-324
---------------------------------------------------------------
```

As an example, the following shows the switch from `uint64` to `object` in Numpy:

```
# Using the maximimum value, we get a uint64.
print(repr(np.array(18446744073709551615)))

# Using one more than the maximum value, we get an object.
print(repr(np.array(18446744073709551616)))
```

```
array(18446744073709551615, dtype=uint64)
array(18446744073709551616, dtype=object)
```

Only numbers that can be represented by Numpy data types (either as scalars or arrays) are fully supported in Finesse. Finesse needs access to certain Numpy functions (e.g. `numpy.cos`) and these are only implemented for Numpy data types. *This effectively places a limit on the minimum and maximum numbers in \Finesse\ to those of Numpy.*

Chapter 7

KatScript quick reference

- This is a quick KatScript reference. For the full specification: KatScript (page 125).
- Click a row to see more information about an item.
- You can copy the syntax overview by using the button appearing on the right of the syntax.
- The full syntax can be viewed and copied in the expanded view.
- Use the input box on the right to quickly filter visible items.

7.1 Elements

Name

Syntax overview

Laser

```
l name P=1 f=0 phase=0
```

Laser producing a beam with a certain power, frequency and phase.

[ref.] (page 165)

Full syntax

```
l name P=1 f=0 phase=0 signals_only=false
```

Required arguments

Argument	Type	Description
name	str	Name of the newly created laser.

Optional arguments

Argument	Type	Description
P	float	Power of the laser (in Watts), defaults to 1 W.
f	float	Frequency-offset of the laser from the default (in Hz) or `Frequency` object. Defaults to 0 Hz offset.
phase	float	Phase-offset of the laser from the default, defaults to zero.
signals_o	bool	When True, this laser component will only inject signal sidebands. They will use the current carrier value as a scaling terms but the carrier will not be injected into the simulation. This allows a user to just inject signal sidebands into a model.

Aliases:

`l`, `laser`, `Laser`

Mirror

```
m name R T L phi=0 Rc=inf xbeta=0 ybeta=0
```

Thin dielectric surface with associated properties such as reflectivity, tuning, and radius of curvature.

[ref.] (page 142)

Full syntax

```
m name R=none T=none L=none phi=0 Rc=inf xbeta=0 ybeta=0 misaligned=false imaginary_
↪transmission=true
```

Required arguments

Argument	Type	Description
name	str	Name of newly created mirror.

Optional arguments

Argument	Type	Description
R	float	Reflectivity of the mirror; defaults to 0.5.
T	float	Transmittance of the mirror; defaults to 0.5.
L	float	Loss of the mirror; defaults to 0.0.
phi	float	Tuning of the mirror (in degrees); defaults to 0.0.
Rc	float or container of two floats	The radius of curvature of the mirror (in metres); defaults to a flat mirror (*Rc=np.inf*). Astigmatic mirrors can also be set with *Rc=(Rcx, Rcy)*. A positive value results in a concave mirror on the p1 side of the mirror.
xbeta, ybeta	float	Misalignment of the mirror in yaw and pitch in units of radians
misaligned	bool	When True the mirror will be significantly misaligned and assumes any reflected beam is dumped. Transmissions will still occur.
imaginary_tran	bool	Convention for the transmission and reflection reciprocity relations. 'True' uses imaginary transmission and equal real reflection from both sides 'False' uses real transmission, negative reflection from side 1, and positive reflection from side 2. defaults to True

Aliases:

`m`, `mirror`, `Mirror`

Beamsplitter

```
bs name R T L phi=0 alpha=0 Rc=inf xbeta=0 ybeta=0
```

Thin dielectric surface under non-normal incidence with associated properties such as reflectivity, tuning, and radius of curvature.

[ref.] (page 149)

Full syntax

```
bs name R=none T=none L=none phi=0 alpha=0 Rc=inf xbeta=0 ybeta=0 plane=0 misaligned=false␣
↪imaginary_transmission=true
```

Required arguments

Argument	Type	Description
name	str	Name of newly created beamsplitter.

Optional arguments

Argument	Type	Description
R	float	Reflectivity of the beamsplitter.
T	float	Transmissivity of the beamsplitter.
L	float	Loss of the beamsplitter.
phi	float	Microscopic tuning of the beamsplitter (in degrees).
alpha	float	Angle of incidence (in degrees)
Rc	float	Radius of curvature (in metres); defaults to `numpy.inf` to indicate a planar surface.
xbeta, ybeta	float	Angle of misalignment in the yaw plane (xbeta) and pitch (ybeta), respectively (in radians); defaults to *0*.
plane	str	Plane of incidence, either 'xz' or 'yz'. Defaults to 'xz'.
misaligned	bool	When True the beamsplitter will be significantly misaligned and assumes any reflected beam is dumped. Transmissions will still occur.

Aliases:

`bs`, `beamsplitter`, `Beamsplitter`

Modulator

```
mod name f midx order=1 mod_type=pm phase=0.0
```

Modulator optical component with associated properties such as modulation frequency, index and order.

[ref.] (page 159)

Full syntax

```
mod name f midx order=1 mod_type=pm phase=0.0 positive_only=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created modulator.
midx	float	Modulation index, >= 0.

Optional arguments

Argument	Type	Description
f	float	Frequency of the modulation (in Hz) or `Frequency` object.
order	int	Maximum order of modulations to produce. Must be 1 for amplitude modulation. Defaults to 1.
mod_t	str	Modulation type: - 'am' (amplitude modulation (page 118)) - 'pm' (phase modulation (page 117)) - 'yaw' (alignment modulation (page 119)) (EXPERIMENTAL) - 'pitch' (alignment modulation (page 119)) (EXPERIMENTAL) Defaults to 'pm'.
phase	float	Relative phase of modulation (in degrees). Defaults to 0.0.
positive_or	bool	If True, only produce positive-frequency sidebands. Defaults to False.

Aliases:

`mod`, `modulator`, `Modulator`

Space

```
s name portA portB L=0.0 nr=1.0
```

Space between two components in the interferometer configuration with a given length and index of refraction.

`[ref.]` (page 170)

Full syntax

```
s name portA portB L=0.0 nr=1.0 user_gouy_x=none user_gouy_y=none
```

Required arguments

Argument	Type	Description
portA, portB	Port	Ports to connect.

Optional arguments

Argument	Type	Description
name	str	Name of newly created space. If not specified, a name is automatically generated.
L	float	Geometric length of newly created `Space` (page 170) instance; defaults to 0.
nr	float	Index of refraction of newly created `Space` (page 170) instance; defaults to 1.0.
user_gouy_x, user_gouy_y	float	User-defined gouy phase to override the calculated value.

Aliases:

`s`, `space`, `Space`

Cavity

```
cav name source
```

Cavity in an interferometer configuration.

[ref.] (page 195)

Full syntax

```
cav name source via=none priority=0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created cavity.
source	OpticalNode or Port	Node / Port that the cavity path starts from. If no *via* node is specified, then the cavity path will be given by the shortest path from source back to the component that owns source. If a port is given then the *output* optical node of that port will be used as the source.

Optional arguments

Argument	Type	Description
via	OpticalNode	Node that the cavity path must traverse via; defaults to *None*. Note that, unlike *source*, this cannot be a `Port` object as this would be ambiguous for beamsplitter type components - i.e. determination of which node to use cannot be assumed automatically.
priority	number	Priority value for beam tracing. Beam tracing dependencies are sorted in descending order of priority - i.e. higher priority value dependencies will be traced first. Any dependency with a priority value of zero will be traced, after non-zero priority dependencies, in alphabetic order of the dependency names.

Aliases:

`cav`, `cavity`, `Cavity`

Gaussian beam

```
gauss name node
```

Gaussian beam parameters at a specific node of the model.

[ref.] (page 195)

Full syntax

```
gauss name node priority=0 **kwargs
```

Aliases:

`gauss`

Lens

```
lens name f=inf
```

Thin lens optical component with an associated focal length.

[ref.] (page 156)

Full syntax

```
lens name f=inf
```

Required arguments

Argument	Type	Description
name	str	Name of newly created lens.

Optional arguments

Argument	Type	Description
f	float	Focal length of the lens in metres; defaults to infinity.

Aliases:

`lens`, `Lens`

Nothing

```
nothing name
```

Empty point in the interferometer configuration.

[ref.] (page 171)

Full syntax

```
nothing name
```

Required arguments

Argument	Type	Description
name	str	Name of newly created nothing.

Aliases:

`nothing`, `Nothing`

Signal generator

```
sgen name node amplitude=1 phase=0
```

Signal generator which produces a signal with a given amplitude and phase.

[ref.] (page 169)

Full syntax

```
sgen name node amplitude=1 phase=0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created signal generator.
node	Node	A node to inject a signal into.

Optional arguments

Argument	Type	Description
amplitude	float	Amplitude of the signal in volts.
phase	float	Phase-offset of the signal from the default in degrees, defaults to zero.

Aliases:

`sgen`, `signal_generator`, `SignalGenerator`

Squeezer

```
sq name db f=0 angle=0
```

Squeezer producing a squeezed-light beam with a given squeezing level and angle.

[ref.] (page 168)

Full syntax

```
sq name db f=0 angle=0
```

Required arguments

Argument	Type	Description
name	str	Name of the newly created squeezer.
db	float	Squeezing factor (in amplitude decibels). 6dB gives a factor of 2 reduction in noise.

Optional arguments

Argument	Type	Description
f	float	Frequency-offset of the squeezer from the default (in Hz) or `Frequency` object. Defaults to 0 Hz offset.
angle	float	Squeezing angle (in degrees). Defaults to zero.

Aliases:

`sq`, `squeezer`, `Squeezer`

Isolator

```
isol name S=0.0
```

Isolator optical component with a specified suppression factor.

[ref.] (page 155)

Full syntax

```
isol name S=0.0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created isolator.
S	float	Power suppression in dB. Defaults to 0

Aliases:

`isol`, `isolator`, `Isolator`

Suspension ZPK

```
sus_zpk name connected_to zpk_plant
```

Suspension with multiple zeros and poles for z, pitch and yaw.

[ref.] (page 194)

Full syntax

```
sus_zpk name connected_to zpk_plant
```

Required arguments

Argument	Type	Description
name	str	Element name
connected_to	Element or mechanical port	Mechanical port or element to attach this suspension to
zpk_plant	dict	Dictionary of {(output, input):(z,p,k)}

Aliases:

`sus_zpk`, `suspension_zpk`, `SuspensionZPK`

Astigmatic lens

```
alens name fx=inf fy=inf
```

Thin astigmatic lens optical component with an associated focal lengths.

[ref.] (page 158)

Full syntax

```
alens name fx=inf fy=inf
```

Required arguments

Argument	Type	Description
name	str	Name of newly created lens.

Optional arguments

Argument	Type	Description
fx	float	Focal length in x-z plane of the lens in metres; defaults to infinity.
fy	float	Focal length of y-z plane the lens in metres; defaults to infinity.

Aliases:

`alens`, `AstigmaticLens`

Amplifier

```
amplifier name gain=1
```

Amplifier with a specified gain.

[ref.] (page 190)

Full syntax

```
amplifier name gain=1
```

Aliases:

`amplifier`, `Amplifier`

Degree of freedom

```
dof name *node_amplitude_pairs DC=0
```

Model parameter that drives multiple parameters of individual elements.

[ref.] (page 190)

Full syntax

```
dof name *node_amplitude_pairs DC=0 lock_parameters=true
```

Required arguments

Argument	Type	Description
name	str	Name of the degree of freedom.
*node_amplitu	tuple[str, float | Symbol]	String referring to a parameter or a local degree of freedom, coupled with the amplitude with which this DOF will drive the parameter. The parameter value will be set to the product of this (symbolic) amplitude and the DOF.DC parameter. In case multiple DOFS drive the same parameter, their contributions will be summed.

Optional arguments

Argument	Type	Description
DC	float	DC value for the degree of freedom, by default 0
lock_para	bool	Whether to 'lock' the parameters driven by this degree of freedom. This prevents accidental overriding of parameter values that are meant to be controlled by a dof, by default True.

Aliases:

`dof`, `degree_of_freedom`, `DegreeOfFreedom`

Directional beamsplitter

```
dbs name
```

Directional beamsplitter optical component.

[ref.] (page 153)

Full syntax

```
dbs name
```

Required arguments

Argument	Type	Description
name	str	Name of newly created directional beamsplitter.

Aliases:

`dbs`, `directional_beamsplitter`, `DirectionalBeamsplitter`

Frequency loss

```
floss name f loss=0 phase=0
```

Unphysical element which introduces a loss and/or phase for a particular frequency.

[ref.] (page 162)

Full syntax

```
floss name f loss=0 phase=0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created lens.

Optional arguments

Argument	Type	Description
f	float	Frequency to apply loss and phase to.
loss	float	Fractional loss at the frequency
phase	float	Phase change at the frequency

Aliases:

`floss`, `FrequencyLoss`

Free mass

```
free_mass name connected_to mass=inf I_yaw=inf I_pitch=inf
```

Simple free-mass suspension of an object.

[ref.] (page 192)

Full syntax

```
free_mass name connected_to mass=inf I_yaw=inf I_pitch=inf
```

Aliases:

`free_mass`, `FreeMass`

Lock

```
lock name error_signal feedback gain accuracy enabled=true offset=0
```

Compute and apply the feedback to a given parameter using an error signal.

[ref.] (page 190)

Full syntax

```
lock name error_signal feedback gain accuracy enabled=true offset=0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created lock.
error_sign	Any	An error signal parameter or an object capable of producing a real-type error signal. This is typically a demodulated `PowerDetector` instance (or the name of the instance).
feedback	Parameter	A parameter of the model to apply the locks' feedback signal to.
gain	float	Control loop gain.
accuracy	float	Threshold to decide whether the loop is locked.
enabled	boolean	If true this lock will run when the *RunLocks()* action is used. Explicitly specifying the name of the lock will override this setting, e.g. *RunLocks(name)*.
offset	float	An offset that is applied to the error signal before it is used.

Aliases:

`lock`, `Lock`

Noise

```
noise name node ASD
```

Classical noise amplitude spectral density.

[ref.] (page 170)

Full syntax

```
noise name node ASD
```

Aliases:

`noise`, `ClassicalNoise`

Pendulum

```
pendulum name connected_to mass=inf Qz=1000 fz=1 I_yaw=inf Qyaw=1000 fyaw=1 I_pitch=inf␣
↪Qpitch=1000 fpitch=1
```

Simple pendulum suspension of an object.

[ref.] (page 193)

Full syntax

```
pendulum name connected_to mass=inf Qz=1000 fz=1 I_yaw=inf Qyaw=1000 fyaw=1 I_pitch=inf␣
↪Qpitch=1000 fpitch=1
```

Aliases:

`pendulum`, `Pendulum`

Wire

```
w name nodeA nodeB delay=0 gain=1
```

Flow of information between two specific signal nodes.

[ref.] (page 172)

Full syntax

```
w name nodeA nodeB delay=0 gain=1
```

Required arguments

Argument	Type	Description
nodeA, nodeB	SignalNode	Signal nodes to connect together.

Optional arguments

Argument	Type	Description
name	str	Name of newly created wire.
delay	float	A delay time for the signal to flow from A to B in seconds
gain	float	A scaling factor that can be used to scale the output relative to the input.

Aliases:

`w`, `wire`, `Wire`

Telescope

```
tel name
```

This component represents a perfectly matching and adapting telescope.

[ref.] (page 164)

Full syntax

```
tel name
```

Required arguments

Argument	Type	Description
name	str	Name of newly created telescope

Aliases:

`tel`, `telescope`, `Telescope`

Test point

```
test name *ports
```

Test point with an arbitrary number of test nodes that can be connected to and from.

[ref.] (page 192)

Full syntax

```
test name *ports
```

Aliases:

`test`, `test_point`, `TestPoint`

Variable

```
var name value description=none units='' is_geometric=false changeable_during_simulation=true
```

Variable element in the model.

[ref.] (page 195)

Full syntax

```
var name value description=none units='' is_geometric=false changeable_during_simulation=true
```

Aliases:

`var`, `variable`

7.2 Detectors

Name

Syntax overview

Amplitude detector

```
ad name node f
```

Detector for the amplitude and phase of light fields at one frequency.

[ref.] (page 172)

Full syntax

```
ad name node f n=none m=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created detector.
node	Node	Node to read output from.
f	float	Frequency of light to detect (in Hz).

Optional arguments

Argument	Type	Description
n	int	Tangential mode index to probe. Defaults to None such that all fields of the given frequency are summed.
m	int	Sagittal mode index to probe. Defaults to None such that all fields of the given frequency are summed.

Aliases:

`ad`, `amplitude_detector`, `AmplitudeDetector`

Readout DC

```
readout_dc name optical_node
```

A Readout component which represents a photodiode measuring the intensity of some incident field.

[ref.] (page 186)

Full syntax

```
readout_dc name optical_node=none pdtype=none output_detectors=false
```

Required arguments

Argument	Type	Description
name	str	Name of the readout component

Optional arguments

Argument	Type	Description
optical_nc	Node or Port	Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `finesse.model.Model.link()` command (and then only a single readout can be connected per port).
pdtype	str or dict	A name of a pdtype definition or a dict representing a pdtype definition, by default None. See Segmented photodiodes (page 310)
output_de	bool	Whether to add a `PowerDetector` and a `QuantumShotNoiseDetector` to the solution object, by default False

Aliases:

`readout_dc`, `ReadoutDC`

Readout RF

```
readout_rf name optical_node f=0 phase=0
```

A readout component which represents a demodulated photodiode.

[ref.] (page 187)

Full syntax

```
readout_rf name optical_node=none f=0 phase=0 output_detectors=false pdtype=none
```

Required arguments

Argument	Type	Description
name	str	Name of the readout component

Optional arguments

Argument	Type	Description
optical_no	Node or Port	Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `link` (page 196) command (and then only a single readout can be connected per port).
f	float	Demodulation frequency in Hz, by default 0
phase	float	Demodulation phase in degrees, by default 0
output_de	bool	Whether to add a `PowerDetector` (DC), two `PowerDetectorDemod1` 's (I and Q) and a `QuantumShotNoiseDetectorDemod1` to the solution object, by default False
pdtype	str or dict	A name of a pdtype definition or a dict representing a pdtype definition, by default None. See Segmented photodiodes (page 310)

Aliases:

`readout_rf`, `ReadoutRF`

Balanced Homodyne Readout

```
balanced_homodyne_readout name optical_node P_lo phase
```

A readout component representing a balanced homodyne detector (BHD).

`[ref.]` (page 188)

Full syntax

```
balanced_homodyne_readout name optical_node=none P_lo=1 phase=0 output_detectors=false
```

Required arguments

Argument	Type	Description
name	str	Name of the readout component

Optional arguments

Argument	Type	Description
optical_nc	Node or Port	Optical node this readout should look at. Because the readout borrows the node, it can be connected anywhere in the model, like a detector. When passing *None*, the readout should be connected explicitly with a `Space` (page 170) or with `finesse.model.Model.link()` command (and then only a single readout can be connected per port).
P_lo	float	Power of the local oscillator in W, by default 1 W
phase	float	Phase of the local oscillator in deg, by default 0 deg
output_de	bool	Whether to add a `BalancedHomodyneDetector` to the solution object, by default False

Aliases:

`balanced_homodyne_readout`, `BalancedHomodyneReadout`

Balanced Homodyne Detector

```
bhd name node P_lo phase
```

Represents a balanced homodyne detector including the beampslitter and photocurrent subtraction of the two photodiodes.

[ref.] (page 173)

Full syntax

```
bhd name node P_lo=1 phase=0
```

Required arguments

Argument	Type	Description
name	str	Name of newly created detector.
node	Node or Port	Node to read the output from
P_lo: float, optional; default: 1 W		Power of the local oscillator [W]

Optional arguments

Argument	Type	Description
phase	float	Phase of the local oscillator [deg]

Aliases:

`bhd`, `balanced_homodyne_detector`, `BalancedHomodyneDetector`

Power detector DC

```
pd name node pdtype=none
```

Power detector that calculates the DC laser power in Watts without RF demodulations.

[ref.] (page 180)

Full syntax

```
pd name node pdtype=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created power detector.
node	Node	Node to read output from.

Aliases:

`pd`, `power_detector_dc`, `PowerDetector`

CCD, camera

```
ccd name node xlim ylim npts
```

Camera for measuring the intensity of a beam.

[ref.] (page 174)

Full syntax

```
ccd name node xlim ylim npts w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.
xlim	sequence or scalar	Limits of the x-dimension of the image. If a single number is given then this will be computed as $x_{\text{lim}} = [-\lvert x\rvert, +\lvert x\rvert]$.
ylim	sequence or scalar	Limits of the y-dimension of the image. If a single number is given then this will be computed as $y_{\text{lim}} = [-\lvert y\rvert, +\lvert y\rvert]$.
npts	int	Number of points in both axes.

Optional arguments

Argument	Type	Description
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`ccd`, `CCD`

CCD, line

```
ccdline name node npts x y xlim ylim
```

Camera for measuring the intensity of a beam along a 1D slice.

[ref.] (page 174)

Full syntax

```
ccdline name node npts x=none y=none xlim=none ylim=none w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.
npts	int	Number of points in slice axis.
x	scalar	The x coordinate of the slice. If ylim is given and this is not specified then defaults to zero. If xlim is given and this is also specified then it is ignored.
y	scalar	The y coordinate of the slice. If xlim is given and this is not specified then defaults to zero. If ylim is given and this is also specified then it is ignored.
xlim	scalar or size two sequence	The limits of the x-axis scan lines. A single number gives $x_{\text{axis}} \in [-\lvert x\rvert, +\lvert x\rvert]$, or a tuple of size two gives $x_{\text{axis}} \in [x[0], x[1]]$.
ylim	scalar or array-like	The limits of the y-axis scan lines. A single number gives $y_{\text{axis}} \in [-\lvert y\rvert, +\lvert y\rvert]$, or a tuple of size two gives $y_{\text{axis}} \in [y[0], y[1]]$.

Optional arguments

Argument	Type	Description
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`ccdline`, `CCDScanLine`

Field camera

```
fcam name node xlim ylim npts
```

Camera for detecting the full image of the beam in terms of amplitude and phase.

[ref.] (page 175)

Full syntax

```
fcam name node xlim ylim npts f=0 w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.
xlim	sequence or scalar	Limits of the x-dimension of the image. If a single number is given then this will be computed as $x_{\text{lim}} = [-\lvert x\rvert, +\lvert x\rvert]$.
ylim	sequence or scalar	Limits of the y-dimension of the image. If a single number is given then this will be computed as $y_{\text{lim}} = [-\lvert y\rvert, +\lvert y\rvert]$.
npts	int	Number of points in both axes.

Optional arguments

Argument	Type	Description
f	scalar	Field frequency offset from the carrier to detect.
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`fcam`, `FieldCamera`

Field detector

```
fd name node f
```

Detector for the higher-order-mode amplitudes at a particular node and frequency.

[ref.] (page 176)

Full syntax

```
fd name node f
```

Required arguments

Argument	Type	Description
name	str	Name of newly created detector.
node	Node	Node to read output from.
f	float	Frequency of light to detect (in Hz).

Aliases:

`fd`, `field_detector`, `FieldDetector`

Math detector

```
mathd name expression
```

Detector that performs some math operation and outputs the result.

[ref.] (page 178)

Full syntax

```
mathd name expression dtype='O' dtype_shape=[] unit='arb.' label=none
```

Required arguments

Argument	Type	Description
name	str	Name of detector
expression	Symbol	Symbolic expression to evaluate as the detectors output
dtype_sha	tuple	The expected array shape when evaluating *expression*. Defaults to an empty tuple. This must be an empty tuple if *dtype* is a scalar type or "O".
unit	str	The unit of the output for plotting. Defaults to "arb."
label	str	How to label the axis when plotting this detector. Defaults to *None*.

Optional arguments

Argument	Type	Description
dtype	numpy.dtype	The expected data type when evaluating expression. Defaults to "O".

Aliases:

`mathd`, `math_detector`, `MathDetector`

Gouy detector

```
gouy name *args from_node to_node via_node direction=x
```

Detector to measure the accumulated gouy phase across a sequence of spaces.

[ref.] (page 177)

Full syntax

```
gouy name *args from_node=none to_node=none via_node=none direction=x
```

Required arguments

Argument	Type	Description
name	str	Name of newly created gouy detector.
*args	sequence	A sequence of spaces or space names.
from_nod	node-like	An `OpticalNode` instance, or a data type which can be converted to an optical node.
to_node	node-like	An `OpticalNode` instance, or a data type which can be converted to an optical node.
via_node	node-like	An `OpticalNode` instance, or a data type which can be converted to an optical node or a sequence of the these types.
direction	str	Plane to detect in - 'x' for tangential, 'y' for sagittal. Defaults to 'x'.

Aliases:

`gouy`, `Gouy`

Power detector demod 1

```
pd1 name node f
```

Power detector that calculates the RF beat power in Watts with one RF demodulation.

[ref.] (page 180)

Full syntax

```
pd1 name node f phase=none pdtype=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created power detector.
node	Node	Node to read output from.
f	float	Demodulation frequency in Hz

Optional arguments

Argument	Type	Description
phase	float	Demodulation phase in degrees

Aliases:

`pd1`, `power_detector_demod_1`, `PowerDetectorDemod1`

Power detector demod 2

```
pd2 name node f1 phase1 f2 phase2
```

Power detector that calculates the RF beat power in Watts with two RF demodulations.

[ref.] (page 180)

Full syntax

```
pd2 name node f1 phase1 f2 phase2=none pdtype=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created power detector.
node	Node	Node to read output from.
f1	float	First demodulation frequency in Hz
phase1	float	First demodulation phase in degrees
f2	float	Second demodulation frequency in Hz

Optional arguments

Argument	Type	Description
phase2	float	Second demodulation phase in degrees

Aliases:

`pd2`, `power_detector_demod_2`, `PowerDetectorDemod2`

Astigmatism detector

```
astigd name node
```

Detector for astigmatism figure-of-merit at a given node.

[ref.] (page 173)

Full syntax

```
astigd name node
```

Required arguments

Argument	Type	Description
name	str	Name of the detector.
node	OpticalNode	Node to compute astigmatism at.

Aliases:

`astigd`, `AstigmatismDetector`

Beam property detector

```
bp name node prop
```

Probe for detecting the properties of a beam at a given node.

[ref.] (page 173)

Full syntax

```
bp name node prop direction=x q_as_bp=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created detector.
node	OpticalNode	Node to read output from.
prop	str or `BeamProperty`	The property of the beam to detect. See above for options.

Optional arguments

Argument	Type	Description
direction	str	Plane to detect in - 'x' for tangential, 'y' for sagittal.
q_as_bp	bool	If detecting q, should the detector output return `BeamParam` object instead of just a complex number.

Aliases:

`bp`, `beam_property_detector`, `BeamPropertyDetector`

Cavity property detector

```
cp name cavity prop
```

Probe for detecting the properties of a cavity.

[ref.] (page 175)

Full syntax

```
cp name cavity prop direction=x q_as_bp=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created cavity property detector.
cavity	str or Cavity	The cavity to probe. If the name is provided then the `CavityPropertyDetector.cavity` attribute will point to the corresponding `Cavity` (page 195) object when adding this detector to a `Model` instance.
prop	str or `CavityPro`	Property of the cavity to probe. See above for options.

Optional arguments

Argument	Type	Description
direction	str	Plane to detect in.
q_as_bp	bool	If detecting q, should the detector output return `BeamParam` object instead of just a complex number.

Aliases:

`cp`, `cavity_property_detector`, `CavityPropertyDetector`

CCD, pixel

```
ccdpx name node x=0 y=0
```

Camera for measuring the intensity of a beam at a single point.

[ref.] (page 174)

Full syntax

```
ccdpx name node x=0 y=0 w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.

Optional arguments

Argument	Type	Description
x	scalar	The x co-ordinate of the pixel.
y	scalar	The y co-ordinate of the pixel.
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`ccdpx`, `CCDPixel`

Field scan pixel

```
fpx name node x=0 y=0 f=0
```

Camera for detecting a single pixel of the beam in terms of the amplitude and phase.

[ref.] (page 177)

Full syntax

```
fpx name node x=0 y=0 f=0 w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.

Optional arguments

Argument	Type	Description
x	scalar	The x co-ordinate of the pixel.
y	scalar	The y co-ordinate of the pixel.
f	scalar	Field frequency offset from the carrier to detect.
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`fpx`, `FieldPixel`

Field scan line

```
fline name node npts x y xlim ylim
```

Camera for detecting a slice of the beam in terms of amplitude and phase.

[ref.] (page 176)

Full syntax

```
fline name node npts x=none y=none xlim=none ylim=none f=0 w0_scaled=true
```

Required arguments

Argument	Type	Description
name	str	Unique name of the camera.
node	OpticalNode	Node at which to detect.
npts	int	Number of points in slice axis.
x	scalar	The x coordinate of the slice. If ylim is given and this is not specified then defaults to zero. If xlim is given and this is also specified then it is ignored.
y	scalar	The y coordinate of the slice. If xlim is given and this is not specified then defaults to zero. If ylim is given and this is also specified then it is ignored.
xlim	scalar or size two sequence	The limits of the x-axis scan lines. A single number gives $x_{\text{axis}} \in [-\lvert x\rvert, +\lvert x\rvert]$, or a tuple of size two gives $x_{\text{axis}} \in [x[0], x[1]]$.
ylim	scalar or array-like	The limits of the y-axis scan lines. A single number gives $y_{\text{axis}} \in [-\lvert y\rvert, +\lvert y\rvert]$, or a tuple of size two gives $y_{\text{axis}} \in [y[0], y[1]]$.

Optional arguments

Argument	Type	Description
f	scalar	Field frequency offset from the carrier to detect.
w0_scaled	bool	Flag indicating whether the x, y axes should be scaled to the waist-size of the beam parameter at *node*.

Aliases:

`fline`, `FieldScanLine`

Knm detector

```
knmd name comp coupling n1 m1 n2 m2
```

Probe of coupling coefficients at a component.

[ref.] (page 177)

Full syntax

```
knmd name comp coupling n1=none m1=none n2=none m2=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created KnmDetector.
comp	Connector	A component which can scatter modes.
coupling	str	Coupling direction string, e.g. "11" for coupling coefficients on reflection from the front surface of a mirror.
n1, m1, n2, m2: int or None		From (n1, m1) and To (n2, m2) mode indices of the coupling coefficient(s) to retrieve. See above for the options.

Aliases:

`knmd`, `KnmDetector`

Mode mismatch detector

```
mmd name node1 node2 direction=x
```

Detector for mode mismatch figure-of-merit for a specified node coupling.

[ref.] (page 178)

Full syntax

```
mmd name node1 node2 direction=x percent=false
```

Required arguments

Argument	Type	Description
name	str	Name of the detector.
node1	OpticalNode or Port	Input node or port. If a `Port` instance is given then this node will be the input node of that port. Note that this node cannot be an output node.
node2	OpticalNode or Port	Output node or port. If a `Port` instance is given then this node will be the output node of that port. Note that this node cannot be an input node.

Optional arguments

Argument	Type	Description
direction	str	Plane of computation, defaults to "x" for the tangential plane. Changing to "y" will compute the mode mismatch in the sagittal plane.
percent	bool	Whether to calculate the mode mismatch as a fraction (default behaviour) or a percentage.

Aliases:

`mmd`, `ModeMismatchDetector`

Motion detector

```
xd name node
```

Motion detector which calculates the amplitude and phase of surface motion.

[ref.] (page 179)

Full syntax

```
xd name node
```

Required arguments

Argument	Type	Description
name	str	Name of newly created motion detector.
node	Node	Node to read output from.

Aliases:

`xd`, `motion_detector`, `MotionDetector`

Optimal q detector

```
optbp name node f
```

Detector that computes an optimal beam parameter (q) for a specified optical frequency at a node.

[ref.] (page 179)

Full syntax

```
optbp name node f fix_spot_size=false astigmatic=false accuracy=1e-09 direction=both
```

Required arguments

Argument	Type	Description
name	str	Name of the detector
node	str or Node	Node name or object to put this detector at
f	float	Frequency component tro compute the optimal beam parameter for.

Optional arguments

Argument	Type	Description
fix_spot_si	bool	When True the optimised will keep the current spot size at the node fixed and just optimise the curvature.
astigmatic	bool	When True qx and qy will be optimised separately
accuracy	float	Approximate mismatch accuracy to try and compute the optimised beam parameter to. mismatch(q_actual, q_optimal) < accuracy
direction	str	Return either *both* or just the *x* or *y* modes

Aliases:

`optbp`, `optimal_q_detector`, `OptimalQ`

7.3 Quantum

Name

Syntax overview

Quantum noise detector

```
qnoised name node
```

Quantum-noise detector that calculates the amplitude spectral density of photocurrent noise.

[ref.] (page 181)

Full syntax

```
qnoised name node nsr=false sources=none exclude_sources=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum noise detector.
node	Node	Node to read output from.

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.
sources	list of `Connector`	If given, only detect quantum noise contributions from these components.
exclude_sourc	list of `Connector`	If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

Aliases:

`qnoised`, `quantum_noise_detector`, `QuantumNoiseDetector`

Quantum noise detector demod 1

```
qnoised1 name node f phase
```

Quantum-noise detector with one RF demodulation.

[ref.] (page 181)

Full syntax

```
qnoised1 name node f phase nsr=false sources=none exclude_sources=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum noise detector.
node	Node	Node to read output from.
f	float	Demodulation frequency in Hz
phase	float	Demodulation phase in degrees

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.
sources	list of `Connector`	If given, only detect quantum noise contributions from these components.
exclude_sourc	list of `Connector`	If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

Aliases:

`qnoised1`, `quantum_noise_detector_demod_1`, `QuantumNoiseDetectorDemod1`

Quantum noise detector demod 2

```
qnoised2 name node f1 phase1 f2 phase2
```

Quantum-noise detector with two RF demodulations.

[ref.] (page 182)

Full syntax

```
qnoised2 name node f1 phase1 f2 phase2 nsr=false sources=none exclude_sources=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum noise detector.
node	Node	Node to read output from.
f1	float	First demodulation frequency in Hz
phase1	float	First demodulation phase in degrees
f2	float	Second demodulation frequency in Hz
phase2	float	Second demodulation phase in degrees

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.
sources	list of `Connector`	If given, only detect quantum noise contributions from these components.
exclude_sourc	list of `Connector`	If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

Aliases:

`qnoised2`, `quantum_noise_detector_demod_2`, `QuantumNoiseDetectorDemod2`

Quantum shot noise detector

```
qshot name node
```

Quantum shot noise detector with no RF demodulations.

[ref.] (page 182)

Full syntax

```
qshot name node nsr=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum shot noise detector.
node	Node	Node to read output from.

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.

Aliases:

`qshot`, `quantum_shot_noise_detector`, `QuantumShotNoiseDetector`

Quantum shot noise detector demod 1

```
qshot1 name node f phase
```

Quantum shot noise detector with one RF demodulation.

[ref.] (page 183)

Full syntax

```
qshot1 name node f phase nsr=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum shot noise detector.
node	Node	Node to read output from.
f	float	Demodulation frequency in Hz
phase	float	Demodulation phase in degrees

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.

Aliases:

`qshot1`, `quantum_shot_noise_detector_demod_1`, `QuantumShotNoiseDetectorDemod1`

Quantum shot noise detector demod 2

```
qshot2 name node f1 phase1 f2 phase2
```

Quantum shot noise detector with two RF demodulations.

[ref.] (page 183)

Full syntax

```
qshot2 name node f1 phase1 f2 phase2 nsr=false
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum shot noise detector.
node	Node	Node to read output from.
f1	float	First demodulation frequency in Hz
phase1	float	First demodulation phase in degrees
f2	float	Second demodulation frequency in Hz
phase2	float	Second demodulation phase in degrees

Optional arguments

Argument	Type	Description
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.

Aliases:

`qshot2`, `quantum_shot_noise_detector_demod_2`, `QuantumShotNoiseDetectorDemod2`

Quantum noise balanced homodyne detector

```
qnoisebhd name node P_lo phase
```

Represents the quantum noise of a balanced homodyne detector including the beampslitter and photocurrent subtraction of the two photodiodes.

[ref.] (page 181)

Full syntax

```
qnoisebhd name node P_lo=1 phase=0 nsr=false sources=none exclude_sources=none
```

Required arguments

Argument	Type	Description
name	str	Name of newly created quantum BHD noise detector.
node	Node	Node to read the output from.

Optional arguments

Argument	Type	Description
P_lo	float	Power of the local oscillator [W]
phase	float	Phase of the local oscillator [deg]
nsr	bool	If true, calculate the noise-to-signal ratio rather than the absolute noise value.
sources	list of `Connector`	If given, only detect quantum noise contributions from these components.
exclude_sourc	list of `Connector`	If given, this will not detect quantum noise contributions from any of these components, even if explicitly specified in *sources*.

Aliases:

`qnoisebhd`, `quantum_balanced_homodyne`, `QuantumBHD`

7.4 Filters

Name

Syntax overview

Butter filter

```
butter name order btype frequency gain=1 analog=true
```

Butterworth filter used for shaping signals.

[ref.] (page 191)

Full syntax

```
butter name order btype frequency gain=1 analog=true
```

Aliases:

`butter`, `filter_butter`, `ButterFilter`

Cheby1 filter

```
cheby1 name order rp btype frequency gain=1 analog=true
```

Chebyshev filter used for shaping signals.

[ref.] (page 192)

Full syntax

```
cheby1 name order rp btype frequency gain=1 analog=true
```

Aliases:

`cheby1`, `filter_cheby1`, `Cheby1Filter`

ZPK filter

```
zpk name z p gain=1
```

Zero-pole-gain filter used for shaping signals.

[ref.] (page 191)

Full syntax

```
zpk name z p k=none fQ=false gain=1
```

Required arguments

Argument	Type	Description
name	str	Name of element in the model
z	array_like[float or Symbols]	A 1D-array of zeros. Use *[]* if none are required. By default these are provided in units of radians/s, not Hz.
p	array_like[float or Symbols]	A 1D-array of poles. Use *[]* if none are required. By default these are provided in units of radians/s, not Hz.
gain	Parameter	Overall gain for the filter. Differs from *k* as this is a *Parameter* so can be easily switched on/off or varied during a simulation.

Optional arguments

Argument	Type	Description
k	[float or Symbol]	Gain factor for the zeros and poles. If *None* then its value is automatically set to generate a unity gain at DC.
fQ	bool	When True the zeros and poles can be specified in a tuple of (frequency, quality factor) for each pole and zero. This automatically adds both roots of each pair.

Aliases:

`zpk`, `filter_zpk`, `ZPKFilter`

Optical bandpass filter

```
obp name fc=0 bandwidth=1000
```

Idealised optical bandpass filter that transmits an optical frequency around some central frequency with a 3dB bandwidth.

[ref.] (page 161)

Full syntax

```
obp name fc=0 bandwidth=1000 filter_hom=none
```

Required arguments

Argument	Type	Description
name	str	Name of element
fc	float or symbol	Central frequency
bandwidth	float or symbol	Filter 3dB bandwidth

Optional arguments

Argument	Type	Description
filter_hor	[tuple or None]	Individual higher order modes can be filtered and transmitted by setting this to a tuple of (n, m). If `None` then no mode filtering is done. This cannot be changed during a simulation.

Aliases:

`obp`, `optical_bandpass`, `OpticalBandpassFilter`

7.5 Actions

Note

Actions use a different syntax than the other Katscript constructs. We recommend that they are used via the Python interface. They are described here for completeness and for those who may wish to use them directly in Katscript, e.g. for serialisation.

More information on the Python API for actions (page 199)

Name

Syntax overview

No x-axis

```
noxaxis(pre_step=none, post_step=none, name='noxaxis')
```

Calculate the values of all detectors without sweeping any parameter; default if no other action is provided.

[ref.] (page 208)

Full syntax

```
noxaxis(pre_step=none, post_step=none, name='noxaxis')
```

Optional arguments

Argument	Type	Description
pre_step	Action	An action to perform before the Noxaxis is run, by default None
post_step	Action	An action to perform after the Noxaxis is run, by default None
name	str	Name of the action, used to find the solution in the final output, by default “noxaxis”

Aliases:

`noxaxis`, `Noxaxis`

Plot

```
plot(name=abcd)
```

Visualize the outcome of the simulation; to be used on a solution object.

[ref.] (page 237)

Full syntax

```
plot(name=abcd)
```

Aliases:

`plot`, `Plot`

Sweep axis

```
xaxis(parameter, mode, start, stop, steps)
```

Sweep some parameter, compute all detector outputs and return a solution object with its result.

[ref.] (page 209)

Full syntax

```
xaxis(
   parameter,
   mode,
   start,
   stop,
   steps,
   relative=false,
   pre_step=none,
   post_step=none,
   name='xaxis'
)
```

Required arguments

Argument	Type	Description
parameter	Parameter or ParameterRef or str	Parameter to sweep over.
mode	'lin or 'log']	Whether to sweep linearly or logarithmically
start	float	Starting value of the parameter sweep
stop	float	Final value of the parameter sweep.
steps	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional arguments

Argument	Type	Description
relative	bool	Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False
pre_step	Action	An action to perform before each step is computed, by default None
post_step	Action	An action to perform after each step is computed, by default None
name	str	Name of the action, used to find the solution in the final output, by default "xaxis"

Aliases:

`xaxis`, `Xaxis`

Frequency response

```
freqresp(f, inputs, outputs)
```

Compute the frequency response of a signal injected at various nodes to obtain transfer functions to multiple output nodes.

[ref.] (page 221)

Full syntax

```
freqresp(f, inputs, outputs, carrier_solution=none, name='frequency_response')
```

Required arguments

Argument	Type	Description
f	array or double	Frequencies to compute the transfer functions over
inputs	iterable[str or Element]	Mechanical or electrical nodes to inject signal at
outputs	iterable[str or Element]	Mechanical or electrical nodes to measure output at

Optional arguments

Argument	Type	Description
carrier_solutio	DCFieldSolutions	When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.
name	str	Solution name

Aliases:

`freqresp`, `frequency_response`, `FrequencyResponse`

ABCD matrix

```
abcd(name=abcd, **kwargs)
```

Compute an ABCD matrix over a path.

[ref.] (page 233)

Full syntax

```
abcd(name=abcd, **kwargs)
```

Aliases:

`abcd`, `ABCD`

Beam trace

```
beam_trace(name='beam_trace', **kwargs)
```

Trace a beam through the full model.

[ref.] (page 234)

Full syntax

```
beam_trace(name='beam_trace', **kwargs)
```

Aliases:

`beam_trace`, `BeamTrace`

Eigenmodes

```
eigenmodes(cavity, frequency, name='eigenmodes')
```

Compute the roundtrip operator and calculate the eigen-values and -vectors of the cavity for a given Cavity defined in a model.

[ref.] (page 233)

Full syntax

```
eigenmodes(cavity, frequency, name='eigenmodes', method='schur_qr')
```

Required arguments

Argument	Type	Description
cavity	str or `Cavity` (page 195)	cavity name or `Cavity` (page 195) instance
frequency	float	Optical carrier or signal frequency to use for calculating the operators

Optional arguments

Argument	Type	Description
metho	str	Method used to compute the eigenvalues and eigenvectors, options are - 'eig': using numpy's eig function - 'schur_qr': using scipy's 'schur' and 'qr' decomposition for the eigenvectors - 'schur_svd': using scipy's 'schur' and 'svd' decomposition for the eigenvectors
name	str	Name of the solution generated by this action

Aliases:

`eigenmodes`, `Eigenmodes`

Maximize

```
maximize(detector, parameter, *args, name='maximize', **kwargs)
```

Maximize some detector output by applying some feedback to multiple targets in a model.

[ref.] (page 235)

Full syntax

```
maximize(detector, parameter, *args, name='maximize', **kwargs)
```

Required arguments

Argument	Type	Description
detector	str	The name of the detector output to maximize / minimize.
parameter	[`Parameter` or str or tuple]	The parameter or name of the parameter to optimize, or a tuple of parameters when using multiple targets to optimize over.
kwargs		Optional parameters passed to the Scipy optimisation routine as the *options* input. See Scipy method documentation to determine what is available.

Optional arguments

Argument	Type	Description
bounds	list	A pair of (lower, upper) bounds on the parameter value. Requires a method that uses bounds.
offset	float	An offset applied to the detector output when optimizing, defaults to 0.
kind	str	Either 'max' for maximization or 'min' for minimization, defaults to 'max'.
max_itera	int	Maximum number of solver iterations, defaults to 10000.
method	str	Optimisation method to use, see Scipy documentation for options.
name	str	The name of this action, defaults to 'maximize'.
update_map	bool	If you are changing some parameter or variable that a *Map* depends on then setting this flag to *True* will recompute the *Map* data for each iteration of the optimiser.
pre_step	Action	Action to run on each step of the optimisation.

Aliases:

`maximize`, `Maximize`

Minimize

```
minimize(detector, parameter, *args, name='minimize', **kwargs)
```

Minimize some detector output by applying some feedback to multiple targets in a model.

[ref.] (page 236)

Full syntax

```
minimize(detector, parameter, *args, name='minimize', **kwargs)
```

Required arguments

Argument	Type	Description
detector	str	The name of the detector output to maximize / minimize.
parameter	[`Parameter` or str or tuple]	The parameter or name of the parameter to optimize, or a tuple of parameters when using multiple targets to optimize over.
kwargs		Optional parameters passed to the Scipy optimisation routine as the *options* input. See Scipy method documentation to determine what is available.

Optional arguments

Argument	Type	Description
bounds	list	A pair of (lower, upper) bounds on the parameter value. Requires a method that uses bounds.
offset	float	An offset applied to the detector output when optimizing, defaults to 0.
kind	str	Either 'max' for maximization or 'min' for minimization, defaults to 'max'.
max_itera	int	Maximum number of solver iterations, defaults to 10000.
method	str	Optimisation method to use, see Scipy documentation for options.
name	str	The name of this action, defaults to 'maximize'.
update_map	bool	If you are changing some parameter or variable that a *Map* depends on then setting this flag to *True* will recompute the *Map* data for each iteration of the optimiser.
pre_step	Action	Action to run on each step of the optimisation.

Aliases:

`minimize`, `Minimize`

Propagate beam

```
propagate_beam(name='propagation', **kwargs)
```

Propagate the beam in a single plane through a given path.

[ref.] (page 234)

Full syntax

```
propagate_beam(name='propagation', **kwargs)
```

Aliases:

`propagate_beam`, `PropagateBeam`

Propagate astigmatic beam

```
propagate_beam_astig(name='astig_propagation', **kwargs)
```

Propagate the beam in both planes through a given path.

[ref.] (page 234)

Full syntax

```
propagate_beam_astig(name='astig_propagation', **kwargs)
```

Aliases:

`propagate_beam_astig`, `PropagateAstigmaticBeam`

Run locks

```
run_locks(*locks, method='proportional', scale_factor=1, sensing_matrix=none, max_
↪iterations=10000, display_progress=false, d_dof_phase=1e-09, d_dof_gain=1e-09, exception_
↪on_fail=true, no_warning=false, pre_step=none, show_progress_bar=none, name='run locks')
```

Iteratively move the system to lock.

[ref.] (page 231)

Full syntax

```
run_locks(
   *locks,
   method='proportional',
   scale_factor=1,
   sensing_matrix=none,
   max_iterations=10000,
   display_progress=false,
   d_dof_phase=1e-09,
   d_dof_gain=1e-09,
   exception_on_fail=true,
   no_warning=false,
   pre_step=none,
   show_progress_bar=none,
   name='run locks'
)
```

Required arguments

Argument	Type	Description
method	str or either "newton" or "proportional"	Which method to use in the locking iterations.
scale_fa	float	Factor by which to multiply all DOF changes. Should be set below 1 if it is desired to minimize overshooting.
sensing_ma	SensingMatrixSolution	Sensing matrix of gains used in locking, of the type that would be returned by state.apply(SensingMatrixDC(lock_dof_names, readout_names) If None, the sensing matrix is recalculated. Recommended to be None except when locking multiple times in a row, e.g. with DragLocks.
max_ite	int	The maximum number of locking steps in each execution of RunLocks.
display_pr	boolean	When true, displays the status of the error signals during locking iterations.
d_dof_	float	Step size to use when optimizing the demodulation phase for each error signal/DOF pair.
d_dof_	float	Step size to use when calculating the gain for every pair of error signals and DOFs.
exception_on	boolean	When true, raise exception if maximum iterations are surpassed.
no_war	boolean	When true, don't even raise a warning if maximum iterations are reached. Recommended to be false unless repeatedly testing locking.
pre_ste	Action	Action to apply on each step of the lock
show_p	boolean	Will enable the progress bar when true.
name	str	Name of the action.

Optional arguments

Argument	Type	Description
*locks	list	A list of locks to use in each RunLocks step. If not provided, all locks in model are used.

Aliases:

`run_locks`, `RunLocks`

Series

```
series(*actions, flatten=true, name=none)
```

Define a series of actions to perform during the simulation.

[ref.] (page 207)

Full syntax

```
series(*actions, flatten=true, name=none)
```

Required arguments

Argument	Type	Description
actions	Action	A collection of Actions to run in series
flatten	bool	When True, each action will be stored in the top level of the solution tree. When False, each action will be a child of the previous solution generated.
name	str	Optional name for the solution generated by these actions

Aliases:

`series`, `Series`

Sweep 2 axes

```
x2axis(parameter1, mode1, start1, stop1, steps1, parameter2, mode2, steps2)
```

Sweep two parameters, compute all detector outputs and return a solution object for each parameter value.

[ref.] (page 213)

Full syntax

```
x2axis(
   parameter1,
   mode1,
   start1,
   stop1,
   steps1,
   parameter2,
   mode2,
   start2,
   stop2,
   steps2,
   relative=false,
   pre_step=none,
   post_step=none,
   name='x2axis'
)
```

Required arguments

Argument	Type	Description
parameter1	Parameter or ParameterRef or str	Parameter to sweep over.
mode1	'lin or 'log']	Whether to sweep linearly or logarithmically
start1	float	Starting value of the parameter sweep
stop1	float	Final value of the parameter sweep.
steps1	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values
parameter2	Parameter or ParameterRef or str	Parameter to sweep over.
mode2	'lin or 'log']	Whether to sweep linearly or logarithmically
start2	float	Starting value of the parameter sweep
stop2	float	Final value of the parameter sweep.
steps2	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional arguments

Argument	Type	Description
relative	bool	Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False, by default False
pre_step	Action	An action to perform before each step is computed, by default None
post_ste	Action	An action to perform after each step is computed, by default None
name	str	Name of the action, used to find the solution in the final output, by default "x2axis"

Aliases:

`x2axis`, `X2axis`

Sweep 3 axes

```
x3axis(parameter1, mode1, start1, stop1, steps1, parameter2, mode2, start2, stop2, steps2,␣
↪parameter3, mode3, start3, stop3, steps3)
```

Sweep three parameters, compute all detector outputs and return a solution object for each parameter value.

[ref.] (page 215)

Full syntax

```
x3axis(
   parameter1,
   mode1,
   start1,
   stop1,
   steps1,
   parameter2,
   mode2,
   start2,
   stop2,
   steps2,
```

(continues on next page)

(continued from previous page)

```
    parameter3,
    mode3,
    start3,
    stop3,
    steps3,
    relative=false,
    pre_step=none,
    post_step=none,
    name='x3axis'
)
```

Required arguments

Argument	Type	Description
parameter1	Parameter or ParameterRef or str	Parameter to sweep over.
mode1	'lin or 'log']	Whether to sweep linearly or logarithmically
start1	float	Starting value of the parameter sweep
stop1	float	Final value of the parameter sweep.
steps1	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values
parameter2	Parameter or ParameterRef or str	Parameter to sweep over.
mode2	'lin or 'log']	Whether to sweep linearly or logarithmically
start2	float	Starting value of the parameter sweep
stop2	float	Final value of the parameter sweep.
steps2	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values
parameter3	Parameter or ParameterRef or str	Parameter to sweep over.
mode3	'lin or 'log']	Whether to sweep linearly or logarithmically
start3	float	Starting value of the parameter sweep
stop3	float	Final value of the parameter sweep.
steps3	int	Number of steps in the sweep. Solution arrays will contain 'steps+1' values

Optional arguments

Argument	Type	Description
relative	bool	Whether to apply the changes to the parameter value on top of the current value or ignore the current value of the parameter, by default False, by default False
pre_step	Action	An action to perform before each step is computed, by default None
post_ste	Action	An action to perform after each step is computed, by default None
name	str	Name of the action, used to find the solution in the final output, by default "x3axis"

Aliases:

`x3axis`, `X3axis`

for

```
for(param, values, *actions)
```

Run a set of actions for each value in the array.

[ref.] (page 207)

Full syntax

```
for(param, values, *actions)
```

Required arguments

Argument	Type	Description
param	str or Parameter	Parameter to change in the model
values	array_like	Array of values to use for the parameter
*actions	tuple(Action)	Actions to run for each parameter value.

Aliases:

`for`, `For`

Antisqueezing

```
antisqueezing(f, squeezer, readout)
```

Compute the amount of antisqueezing at the output of a squeezing element.

[ref.] (page 234)

Full syntax

```
antisqueezing(f, squeezer, readout, signal=none, name='antisqueezing')
```

Required arguments

Argument	Type	Description
f	array_like	Signal frequencies to compute the anti-squeezing over
squeezer	str	Name of squeezing component
readout	str	Name of readout port to compute squeezing at
signal	str	Name of signal drive to calculate a the signal transfer function of. This is returned in *sol.signal* and can be used to scale the noise into equivalent units of some signal.

Aliases:

`antisqueezing`, `AntiSqueezing`

DC fields

```
dc_fields(name='dcfields')
```

Save the DC (carrier) fields at all nodes, frequencies, and higher order modes for the current state of the simulation.

[ref.] (page 235)

Full syntax

```
dc_fields(name='dcfields')
```

Optional arguments

Argument	Type	Description
name	str	Name of the solution generated by this action

Aliases:

`dc_fields`, `DCFields`

Frequency response 2

```
freqresp2(f, inputs, outputs, name='frequency_response2')
```

Compute the frequency response of a signal injected at an optical port at a particular optical frequency.

[ref.] (page 223)

Full syntax

```
freqresp2(
    f,
    inputs,
    outputs,
    carrier_solution=none,
    name='frequency_response2'
)
```

Required arguments

Argument	Type	Description
f	array or double	Frequencies to compute the transfer functions over
inputs	iterable[tuple[str or Node or Frequency]]	Optical node and frequency tuple to inject at. A symbolic refence to the model's fsig.f parameter should always be used when defining a frequency to look at.
outputs	iterable[str or Element]	Mechanical or electrical (signal)nodes to measure output to

Optional arguments

Argument	Type	Description
carrier_solution	DCFieldsSolutions	When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.
name	str	Solution name

Aliases:

`freqresp2`, `frequency_response2`, `FrequencyResponse2`

Frequency response 3

```
freqresp3(f, inputs, outputs, name='frequency_response3')
```

Compute the frequency response of a signal injected at an optical port at a particular optical frequency.

[ref.] (page 225)

Full syntax

```
freqresp3(
   f,
   inputs,
   outputs,
   carrier_solution=none,
   name='frequency_response3'
)
```

Required arguments

Argument	Type	Description
f	array or double	Frequencies to compute the transfer functions over
inputs	iterable[tuple[str or Node or Frequency]]	Optical node and frequency tuple to inject at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.
outputs	iterable[tuple[str or Node or Frequency]]	Optical node and frequency tuple to measure output at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.

Optional arguments

Argument	Type	Description
carrier_solution	DCFieldsSolutions	When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.
name	str	Solution name

Aliases:

`freqresp3`, `frequency_response3`, `FrequencyResponse3`

Frequency response 4

```
freqresp4(f, inputs, outputs, name='frequency_response4')
```

Compute the frequency response of a signal injected at an electrical or mechanical port.

[ref.] (page 227)

Full syntax

```
freqresp4(
   f,
   inputs,
   outputs,
   carrier_solution=none,
   name='frequency_response4'
)
```

Required arguments

Argument	Type	Description
f	array or double	Frequencies to compute the transfer functions over
inputs	iterable[str or Element]	Mechanical or electrical node to inject signal at
outputs	iterable[tuple[str or Node or Frequency]]	Optical node and frequency tuple to measure output at. A symbolic reference to the model's fsig.f parameter should always be used when defining a frequency to look at.

Optional arguments

Argument	Type	Description
carrier_solutio	DCFieldsSolutions	When provided these DC fields will be used to compute the signal matrix with. It is up to the user to input a sensible carrier field.
name	str	Solution name

Aliases:

`freqresp4`, `frequency_response4`, `FrequencyResponse4`

Get error signals

```
get_error_signals(*locks, name='get error signals')
```

Calculate the current error signals for all or a subset of locks in a model.

[ref.] (page 230)

Full syntax

```
get_error_signals(*locks, name='get error signals')
```

Required arguments

Argument	Type	Description
name	str	Name of the action.

Optional arguments

Argument	Type	Description
*locks	list	A list of lock names to compute the error signals for. If not provided, all locks in model are used.

Aliases:

`get_error_signals`, `GetErrorSignals`

Noise projection

```
noise_projection(f, *output_nodes, scaling=none, name='loop')
```

Missing summary.

[ref.] (page 236)

Full syntax

```
noise_projection(f, *output_nodes, scaling=none, name='loop')
```

Aliases:

`noise_projection`, `NoiseProjection`

Operator

```
operator(start_node, end_node)
```

Extract operators defined by a path between two nodes in the network for external use.

[ref.] (page 233)

Full syntax

```
operator(start_node, end_node, via=none, frequency=0, name='operator')
```

Required arguments

Argument	Type	Description
start_node	str	Start node name
end_node	str	End node name

Optional arguments

Argument	Type	Description
via	str	Via node (or sequence of via nodes) to use to specify a path with multiple options
frequency	float	Optical carrier or signal frequency to use for calculating the operators
name	str	Name of the solution generated by this action

Aliases:

`operator`, `Operator`

Optimise RF readout

```
opt_rf_readout_phase(*args, d_dof=1e-10, name='optimise_demod_phases_dc')
```

Optimize the demodulation phase of ReadoutRF elements relative to some DegreeOfFreedom or driven Parameter in the model.

[ref.] (page 229)

Full syntax

```
opt_rf_readout_phase(
   *args,
   d_dof=1e-10,
   name='optimise_demod_phases_dc',
   verbose=false
)
```

Required arguments

Argument	Type	Description
args		Pairs of `DegreeOfFreedom` (page 190) or `Parameter` and `ReadoutRF` elements, or pairs of their names. If none are provided *OptimiseRFReadoutPhaseDC* will automatically search for `Lock` (page 190) elements which have `ReadoutRF` (page 187) error signal and optimise them. Every readout can only be optimised to a single `DegreeOfFreedom` (page 190) or `Parameter`. If a readout is paired with multiple drives, its phase is optimised to the first and the others are ignored.
verbose	bool	If True this will print the initial and optimised readout phases. Defaults to False

Optional arguments

Argument	Type	Description
d_dof	float	A small offset applied to the DOFs to compute the gradients of the error signals

Aliases:

`opt_rf_readout_phase`, `OptimiseRFReadoutPhaseDC`

Pseudo lock cavity

```
pseudo_lock_cavity(cavity, mode=none, lowest_loss=false, feedback=none, name='pseudo_lock_
↪cavity' )
```

Lock a cavity defined by a Cavity element to a specific mode without using any radio-frequency sensing scheme.

[ref.] (page 232)

Full syntax

```
pseudo_lock_cavity(
    cavity,
    mode=none,
    lowest_loss=false,
    feedback=none,
    name='pseudo_lock_cavity'
)
```

Required arguments

Argument	Type	Description
cavity	Cavity	Cavity element describing some Fabry-Perot like optical cavity

Optional arguments

Argument	Type	Description
mode	(int or m)	HG mode to try and lock to, default is [0,0]
lowest_loss	bool	Select the eigenmode which has the lowest loss, most likely the fundamental mode of the cavity. Using lowest loss will override the *mode* selection.
feedback	Parameter	If *None* the required cavity tuning to lock to the calculated mode will be determined from the cavity objects source node element, and the relevant *phi* parameter will be used. Alternatively you can specify which tuning parameter is used instead. Which should be a *phi* of some mirror in the cavity or a *DegreeOfFreedom* which controls the cavity length.
name	str	Name of the solution generated by this action

Aliases:

`pseudo_lock_cavity`, `PseudoLockCavity`

Pseudo lock DRFPMI

```
pseudo_lock_drfpmi(frequency=0, apply_tunings=true, name='operator_lock')
```

Find an operating point for a LIGO-like model without using RF sidebands and readouts.

[ref.] (page 232)

Full syntax

```
pseudo_lock_drfpmi(frequency=0, apply_tunings=true, name='operator_lock')
```

Optional arguments

Argument	Type	Description
frequency	float	Frequency to use for calculating the operators
apply_tunings	bool	When True the action will modify the model tunings
name	str	Name of the solution generated by this action

Aliases:

`pseudo_lock_drfpmi`, `PseudoLockDRFPMI`

Sensing matrix DC

```
sensing_matrix_dc(dofs, readouts, d_dof=1e-09, name='sensing_matrix_dc')
```

Compute the sensing matrix elements for various degrees of freedom and readouts present in the model.

[ref.] (page 230)

Full syntax

```
sensing_matrix_dc(dofs, readouts, d_dof=1e-09, name='sensing_matrix_dc')
```

Required arguments

Argument	Type	Description
dofs	iterable[str]	String names of degrees of freedom
readouts	iterable[str]	String names of readouts

Optional arguments

Argument	Type	Description
d_dof	float	Small step used to compute derivative

Aliases:

`sensing_matrix_dc`, `SensingMatrixDC`

Set lock gains

```
set_lock_gains(*locks, d_dof_gain=1e-10, gain_scale=1, name='set gains')
```

Compute the optimal lock gains using the sensing matrix found with SensingMatrixDC.

[ref.] (page 230)

Full syntax

```
set_lock_gains(
   *locks,
   d_dof_gain=1e-10,
   gain_scale=1,
   name='set gains',
   verbose=false
)
```

Required arguments

Argument	Type	Description
name	str	Name of the action.
verbose	bool	If True this will print the name of the enabled locks and their gains.

Optional arguments

Argument	Type	Description
*locks	list	A list of locks for which to set the gain. If none provided, all enabled locks in model are used. Disabled locks that are explicitly listed will have their gains set.
d_dof	float	Step size to use when calculating the gain for each error signal/DOF pair.
gain_s	float	Extra gain scaling factor applied to the gain calculation: *-gain_scale/sensing* In multiple lock models where the locks are cross coupled using a *gain_scale* < 1 can improve the stability of the locking algorithm to stop excessively large steps.

Aliases:

`set_lock_gains`, `SetLockGains`

Sweep

```
sweep( *args )
```

Sweep N parameters through values in N arrays.

[ref.] (page 218)

Full syntax

```
sweep(
   *args,
   pre_step=none,
   post_step=none,
   reset_parameter=true,
   name='sweep'
)
```

Required arguments

Argument	Type	Description
args	[Parameter or str] or array or boolean	Expects 3 arguments per axis. The first is a full name of a Parameter or a Parameter object. The second is an array of values to step this parameter over, and lastly a boolean value to say whether this is a relative step from the parameters initial value.
name	str	Name of the action, used to find the solution in the final output.

Optional arguments

Argument	Type	Description
pre_step	Action	An action to perform before the step is computed
post_step	Action	An action to perform after the step is computed
reset_parameter	boolean	When true this action will reset the all the parameters it changed to the values before it ran.

Aliases:

`sweep`, `Sweep`

Link

```
link(*args, verbose=false)
```

Connect multiple components together in one quick command.

[ref.] (page 196)

Full syntax

```
link(*args, verbose=false)
```

Required arguments

Argument	Type	Description
*args	[Components or float or Port]	Separate arguments of either components or ports. A float value will create a space or wire with the provided length or time delay.

Optional arguments

Argument	Type	Description
verbose	bool	Print out what the link command is doing

Aliases:

`link`

Signal frequency

```
fsig(f)
```

Represent the signal frequency used in a model.

[ref.] (page 196)

Full syntax

```
fsig(f)
```

Required arguments

Argument	Type	Description
f	[float or None]	Signal frequency to use in a model [Hz]. If set to `None` then no signal frequencies will be modelled in the simulation.

Aliases:

`fsig`

Default wavelength

```
lambda(value)
```

Define the wavelength of the model, default is 1064 nm.

[ref.] (page 196)

Full syntax

```
lambda(value)
```

Required arguments

Argument	Type	Description
value	float	Reference wavelength to use in a model [Hz]. If not specified, a reference wavelength of 1064 nm is used.

Aliases:

`lambda`

Modes

```
modes(modes=none, maxtem=none, include=none, remove=none)
```

Select the HOM indices to include in the model.

[ref.] (page 197)

Full syntax

```
modes(modes=none, maxtem=none, include=none, remove=none)
```

Optional arguments

Argument	Type	Description
mode	sequence or str	Identifier for the mode indices to generate. This can be: - An iterable of mode indices, where each element in the iterable must unpack to two integer convertible values. - A string identifying the type of modes to include, must be one of “off”, “even”, “odd”, “x” or “y”.
maxtem	int	Optional maximum mode order.
include	sequence or str	A single mode index pair, or an iterable of mode indices, to include. Each element must unpack to two integer convertible values.
remove	sequence or str	A single mode index pair, or an iterable of mode indices, to remove. Each element must unpack to two integer convertible values.

Aliases:

`modes`

Phase config

```
phase_config(zero_k00=false, zero_tem00_gouy=true)
```

Specify coupling coefficients and Gouy-phase scaling to change computation of higher-order modes.

[ref.] (page 197)

Full syntax

```
phase_config(zero_k00=false, zero_tem00_gouy=true)
```

Optional arguments

Argument	Type	Description
zero_k00	bool	Scale phase for k0000 (TEM00 to TEM00) coupling coefficients to 0. Defaults to True.
zero_tem00_gouy	bool	Ensure that all Gouy phases for TEM00 are 0. Defaults to True.

Aliases:

`phase_config`

Transverse electromagnetic modes

```
tem(laser, n, m, factor, phase=0.0)
```

Distribute power into the mode HGnm.

[ref.] (page 197)

Full syntax

```
tem(laser, n, m, factor, phase=0.0)
```

Required arguments

Argument	Type	Description
laser	laser	The `Laser` (page 165) to set mode power for.
n, m	int	Mode indices.
factor	float	Relative power factor, modes with equal *factor* will have equivalent power distributed to them.

Optional arguments

Argument	Type	Description
phase	float	Phase offset for the field, in degrees.

Aliases:

`tem`

Debug

```
debug(name='Debug')
```

Start an IPython debug shell.

[ref.] (page 235)

Full syntax

```
debug(name='Debug')
```

Aliases:

`debug`, `Debug`

Parallel

```
parallel(*actions)
```

Create copies of the model and restart the simulation for each action in the parallel arguments.

[ref.] (page 207)

Full syntax

```
parallel(*actions)
```

Aliases:

`parallel`, `Parallel`

Print

```
print(*args, name='printer', eval=true)
```

Print properties of the model, for example as part of a series action.

[ref.] (page 237)

Full syntax

```
print(*args, name='printer', eval=true)
```

Aliases:

`print`, `Printer`

Print model

```
print_model(name='print_model')
```

Print the model object that is used to run actions.

[ref.] (page 237)

Full syntax

```
print_model(name='print_model')
```

Aliases:

`print_model`, `PrintModel`

Print model attribute

```
print_model_attr(*args, eval=true, prefix='')
```

Print an attribute of the model.

[ref.] (page 237)

Full syntax

```
print_model_attr(*args, eval=true, prefix='')
```

Required arguments

Argument	Type	Description
args	(str or) or ((str or callable) or)	Strings input for the attribute to print. If a function is given, that funcion will be applied to the attribute before printing.

Optional arguments

Argument	Type	Description
eval	bool	When *True* symbolic expressions will be evaluated before printing. Defaults to *True*.
prefix	str	Optional string to print before the attributes

Aliases:

`print_model_attr`, `PrintModelAttr`

Save matrix

```
save_matrix(name='savematrix')
```

Save the current state of the matrix being used in the simulation.

[ref.] (page 237)

Full syntax

```
save_matrix(name='savematrix')
```

Aliases:

`save_matrix`, `SaveMatrix`

Update maps

```
update_maps(*args, name='update_maps', **kwargs)
```

Update any maps that might be changing in the simulation.

[ref.] (page 234)

Full syntax

```
update_maps(*args, name='update_maps', **kwargs)
```

Aliases:

`update_maps`, `UpdateMaps`

Change

```
change(change_dict=none, relative=false, name='change', **kwargs)
```

Change the value of a model parameter during the analysis.

[ref.] (page 235)

Full syntax

```
change(change_dict=none, relative=false, name='change', **kwargs)
```

Required arguments

Argument	Type	Description
**kwargs		Alternative method to specify parameter:value pairs to change

Optional arguments

Argument	Type	Description
change_dict	dict	Dictionary of parameter:value pairs to change.
relative	bool	Whether to increment from the parameters current value or not
name	str	Name of action

Aliases:

`change`, `Change`

Chapter 8

Developer guide

8.1 Contributing to development

FINESSE is an open source program where contributions (of any size) from members of the optical simulation community are welcome and encouraged. This page contains information on how to contribute, either with bug reports or feature requests or with direct development.

The preferred way to contribute to FINESSE development is to fork the main repository https://gitlab.com/ifosim/finesse/finesse3, work on this copy of the code then submit a "merge request".

A brief summary of the process is given below, with more details to be found in the sections later on.

1. Set up the Finesse development environment (page 21).
2. Create a new branch on your local copy to hold the changes you want to contribute:

   ```
   $ git checkout -b <feature-name> origin/develop
   ```

   The name format should follow either `feature/xyz` or `fix/xyz` where `xyz` is an informative branch slug, otherwise the branch might be rejected when pushed.

   Once you have reached this stage you can start programming!
3. Work on your branch, on your machine, using git to do the version control. Push any changes you have committed to your copy of the FINESSE repository:

   ```
   $ git push -u origin <feature-name>
   ```
4. Once you are happy with the state of your changes, submit a merge request for review.

8.2 Submitting a bug report

If you encounter a bug in the code or documentation, especially in a release version, do not hesitate to submit an issue to the Issue Tracker. If you are unsure whether something is a bug or need help submitting, please contact us in our public Matrix channel

When reporting a bug, please include the following:

- A short, top-level summary of the bug. In most cases, this should be 1-2 sentences.
- A **minimal** but *complete and verifiable* code snippet example to reproduce the bug.
- The actual result(s) of the example provided versus the expected result(s)
- The FINESSE version, Python version and platform (i.e. OS version) you are using. You can find the first two from a Python interpreter with, e.g:

```
import platform
import finesse

print(platform.python_version())
print(finesse.__version__)
```

```
3.13.2
3.0.3.dev63+gee4f5bb35
```

If the bug is Physics-related (e.g. an unexpected discrepancy in results between FINESSE and another method) then you should also provide details of the method(s) used to compute the result - in particular the equations used.

8.2.1 Creating a bug report automatically

You can automatically generate a helpful bug report in markdown format by calling `finesse.utilities.bug_report.bug_report()`. This will include some information about your system and python installation, and optionally includes the code from the file you are calling it in and the last exception that was raised.

If you want to use it in a script, you should catch the exception to report it:

```
import finesse

try:
    1 / 0
except Exception:
    report = finesse.bug_report(file="bug_report.md")
    print(report)
```

```
Bug report written to /builds/ifosim/finesse/finesse3/docs/bug_report.md

# Finesse3 bug report

## Environment

- **Finesse version:** `3.0.3.dev63+gee4f5bb35`
- **Python version:** `3.13.2 (main, Feb 25 2025, 02:24:34) [GCC 12.2.0]`
- **Platform:** `Linux x86_64`

## Entry point

`/usr/local/bin/python3`

### Arguments

`/usr/local/lib/python3.13/site-packages/ipykernel_launcher.py` `-f` `/tmp/tmpoc232b_h.json`␣
↪`--HistoryManager.hist_file=:memory:`

## Stack trace

```text
Traceback (most recent call last):
  File "/tmp/ipykernel_1575/3840209720.py", line 4, in <module>
    1 / 0
    ~~^~~
ZeroDivisionError: division by zero

```

Package versions

```text
```

(continues on next page)
```

(continued from previous page)

```
IPython == 9.17.1
babel == 2.17.0
charset_normalizer == 3.4.1
click == 8.5.0
colorama == 0.4.6
comm == 0.2.3
cycler == 0.12.1
debugpy == 1.8.22
defusedxml == 0.7.1
deprecated == 3.0.0
dill == 0.4.1
docutils == 0.21.2
finesse == 3.0.3.dev63+gee4f5bb35
fontTools == 4.67.0
h5py == 3.16.0
ipykernel == 7.4.0
ipywidgets == 8.1.9
jinja2 == 3.1.5
jupyter_client == 8.10.0
jupyter_core == 5.9.1
kiwisolver == 1.5.1
markupsafe == 3.0.2
matplotlib == 3.11.2
msgpack == 1.2.3
networkx == 3.7
numpy == 2.5.4
numpydoc == 1.11.0
packaging == 24.2
platformdirs == 4.13.5
psutil == 7.2.2
pygments == 2.19.1
pyparsing == 3.3.3
quantiphy == 2.23
scipy == 1.18.1
six == 1.17.0
sphinx == 8.2.3
sphinx_jinja2_compat == 0.4.1
sphinx_prompt == 1.10.2
sphinx_prompt == 1.10.2
tornado == 6.6
tqdm == 4.70.1
traitlets == 5.16.1
typing_extensions == 4.16.0
wrapt == 2.5.1
PIL
__future__
__main__
__main__
_abc
_ast
_asyncio
_bisect
_blake2
_bz2
_codecs
_collections
_colorize
_compat_pickle
_compression
_contextvars
_csv
```

(continues on next page)

(continued from previous page)

```
_ctypes
_curses
_cython_3_2_4
_cython_3_2_9
_cython_3_2_9limited_fastcallnofinalize
_cython_3_3_0
_datetime
_decimal
_distutils_hack
_elementtree
_functools
_hashlib
_heapq
_imp
_io
_json
_locale
_lzma
_multibytecodec
_opcode
_opcode_metadata
_operator
_pickle
_posixsubprocess
_pydev_bundle
_pydev_runfiles
_pydevd_bundle
_pydevd_frame_eval
_pydevd_sys_monitoring
_pyio
_pyrepl
_queue
_random
_signal
_sitebuiltins
_socket
_sqlite3
_sre
_ssl
_stat
_string
_strptime
_struct
_sysconfigdata__linux_x86_64-linux-gnu
_thread
_tokenize
_typing
_uuid
_warnings
_weakref
_weakrefset
_zoneinfo
abc
argparse
array
ast
asyncio
atexit
base64
bdb
binascii
```

(continues on next page)

(continued from previous page)

```
bisect
builtins
bz2
calendar
cmath
cmd
code
codecs
codeop
collections
concurrent
configparser
contextlib
contextvars
copy
copyreg
csv
ctypes
curses
cython_runtime
dataclasses
datetime
dateutil
decimal
difflib
dis
email
encodings
enum
errno
faulthandler
fcntl
filecmp
fileinput
finesse_numpydoc
fnmatch
functools
gc
genericpath
getopt
getpass
gettext
glob
grp
gzip
hashlib
heapq
hmac
html
http
importlib
inspect
io
ipaddress
itertools
json
keyword
linecache
locale
logging
lzma
```

(continues on next page)

(continued from previous page)

```
marshal
math
mimetypes
mpl_toolkits
multiprocessing
ntpath
numbers
opcode
operator
optparse
os
pathlib
pdb
pickle
pkgutil
platform
plistlib
posix
posixpath
posixpath
pprint
pwd
pydev_ipython
pydevconsole
pydevd
pydevd_file_utils
pydevd_plugins
pydevd_tracing
pydoc
pyexpat
queue
quopri
random
re
readline
reprlib
resource
rlcompleter
runpy
secrets
select
selectors
shlex
shutil
signal
site
socket
socketserver
spellchecker
sphinxcontrib
sqlite3
ssl
stat
string
stringprep
struct
subprocess
sys
sysconfig
tempfile
termios
```

(continues on next page)

(continued from previous page)

```
textwrap
threading
time
timeit
token
tokenize
traceback
types
typing
unicodedata
unittest
urllib
uuid
warnings
weakref
webbrowser
xml
xmlrpc
zipfile
zipimport
zlib
zmq
zoneinfo

```
```

In an interactive notebook environment, you can call the function from a separate cell to grab the last exception and that will also work. It will return the report as a string and save it to `bug_report.md`.

Warning

You can optionally include the source code of the file causing the exception in your report by passing `include_source=True` to `finesse.utilities.bug_report.bug_report()` , but you might unintentionally leak proprietary/confidential information by posting the report in a public space.

8.3 Contributing to the documentation

Here we describe the guidelines and conventions for contributing to the FINESSE documentation.

8.3.1 How to build the documentation

A `pdf` and an `html` version of the documentation need to be built as part of the GitLab pipeline. They need to be manually triggered after the build jobs are completed. The generated pages can be accessed in the artefacts of the documentation building jobs. However, this is often not a convenient way to iterate when writing the documentation, and a local build of the documentation can be more efficient.

The documentation system uses Sphinx to build the document(s). If FINESSE was installed for code development in the conda environment file (*./environment.yml*), everything needed should already be installed. If not, consult Installing Finesse (page 19) for how to install FINESSE.

The documentation can be built locally using the `make` commands executed in the `docs/` subdirectory. Run `make` with no arguments to print a list of available output formats. For example:

```
make html
```

This will generate the website version in the `docs/build/html` directory (with `docs/build/html/index.html` as the landing page).
```

The build process can be quite long. To speed it up, you can skip building the API documentation pages, which takes a long time:

```
make html_no_api
```

or skip also the executing of the `jupyter` code using

```
make html_no_api_no_jupyter
```

To remove temporary files created during the build process, use:

```
make clean
```

Note

To build the PDF docs (using `make latexpdf`), you must ensure that a comprehensive LaTeX package (such as `texlive`) is available on the system path.

8.3.2 Supported formats

FINESSE documentation supports two formats for creating pages:

- **reStructured Text (reST)**
  The primary format supported by Sphinx is reStructured Text. Each *.rst* file corresponds to a page in the documentation. The extension jupyter_sphinx allows executing Python code and displaying the results directly in the documentation, including in the API pages generated from the Python docstrings.
- **MyST-NB**
  It is possible to write jupyter notebooks (*.ipynb*) that can be included directly in the documentation, while still being fully compatible with jupyter. It is an easy way to create pages or code examples. MyST-NB adds some notations to the standard markdown used in the jupyter notebook to allow a seamless integration with Sphinx and reStructured Text.

  The JupyterLab plugin jupyterlab-myst can be installed using *pip* or the JupyterLab plugin manager and will allow the rendering of some MyST-NB specific formatting.

There is no preferred method to create a documentation page; the choice is left to the author. Newcomers may find it easier to write Jupyter notebooks as they are probably already familiar with the format, and it allows a live rendering of the page that closely resembles the final product (especially with the JupyterLab plugin).

See the Authoring documentation (page 460) section for more information about the two formats.

8.3.3 Directory organization

`docs`
Root directory for the documentation. Notably, it contains the `Makefile` file defining the build target.

`docs/source`
Contains the source files for the documentation and the Sphinx config file `conf.py`. Each subdirectory inside `docs/source` corresponds to a section and contains the files needed to generate it.

Each directory, including `docs/source`, contains a file named `index.rst`. Any new page needs to be added to that index file to be shown in the table of contents of the documentation.

`docs/build`
This is where the documents created using `make` will be generated, each in their subdirectory.

`docs/resources`
Contains some resource files (like images). Resource files are also present in the source subdirectories.

8.3.4 Search

If you are adding a main page for a specific topic, you might want to check Search tags (page 459) for how to boost pages in search results.

8.3.5 Updating the Changelog

In principle, every change made to FINESSE that is relevant to FINESSE users should be documented in the Changelog (page 527). Internal changes relevant only to the developers can be left out. We follow the format from keepachangelog. Special care should be taken to document changes that are not backwards compatible (i.e. will break existing code) and changes that will quietly change the scientific output of existing code.

The changelog should always have a section on top titled 'Next Release', which should be replaced with the actual version when Releasing finesse (page 494). It is encouraged to write a small summary of the changes, highlighting the most important ones so users don't have to scroll through all items to see if something is relevant. If possible, the changelog should refer to actual documentation pages, but it can include small code snippets or comparison plots highlighting the different output between the new and old versions.

8.3.6 Comments and todo items

The `.. todo::` directive will add a highlighted todo item in the manual. These should be used only as placeholders for larger sections, such as *.. todo:: This section is missing* for a missing section. For internal todo items such as 'fix this later', a simple comment can be used, for example: *.. Todo: fix this later*.

8.3.7 Writing Sphinx compatible docstrings

To produce well-formatted API documentation for FINESSE, all module, class and function docstrings will be written in the numpydoc style (follow the link for the style guide).

Note

In contrast to PEP-257, numpydoc asks for class `__init__` methods to be documented in the class docstring, not under the `__init__` method itself.

Note

There is currently no directly supported method for documenting properties in numpydoc, instead you should use reST syntax in the docstrings for a class property. This involves using docstrings only for the "getter" and passing the tags *:`getter`:* and *:`setter`:* to document both. An example is shown below for the *nr* property of the class `BeamParam`:

```
@property
def nr(self):
    """The refractive index associated with the `BeamParam`.

    :`getter`: Returns the index of refraction.
    :`setter`: Sets the index of refraction.
    """
    return self.__nr

@nr.setter
def nr(self, value):
    self.__nr = value
```

The above docstrings result in the `finesse.gaussian.BeamParam.nr` documentation page.

8.3.8 Documenting Cython extensions

See Documenting Cython extensions (page 469).

8.3.9 Style guide

FINESSE is developed by many people and maintaining consistency across the documentation is essential. The following subsections describe some style aspects to be conformed to when writing documentation.

Where style guidance is undefined, conform first to the rules enforced by doc8 then the Python documentation style guide.

Line length

Try to stick to a maximum of 88 characters. This is consistent with the rules for Python code, set by Black. This limit isn't just for aesthetics in your IDE: code blocks in documentation pages with longer lines will overflow the visible page and create irritating horizontal scroll bars.

You may wish to create a ruler in your IDE to give a visual cue for this limit.

> **Hint**
>
> In some editors you can highlight a block of text and press `Alt` + `Q` to automatically wrap the text at the configured line length. In Visual Studio Code you might wish to install the *Rewrap* extension to enable this shortcut, and set a ruler in `.vscode/settings.json` using e.g. `"editor.rulers": [88],`.

Style

Try not to write documentation with many nested subsections; documentation should be mostly prose. Only the first four section levels are displayed in the sidebar of the HTML documentation, and each level beyond the top level is hidden until its parent is navigated to. If a section makes sense to appear in the sidebar then it should be within the top four levels; use common sense.

Titles

Section titles should use sentence case, e.g. *Showing output from inline scripts*, not *Showing Output from Inline Scripts*.

Do not use backtick-style (``` ``thing`` ```) or replacement (e.g. `|Finesse|`) formatting in titles as these do not render nicely in the sidebar. Instead you could consider using single quotes, e.g. "Logging via the Python 'logging' module".

Code examples

Code examples should follow the *same style <code_style>* as FINESSE itself.

Grammar

Tone

Use an affirmative, succinct tone. Concise is better than verbose. Users typically don't read whole pages of documentation but rather arrive at particular sections via links or search results.

English

Use of US English is preferred, for consistency with the Python documentation; however, refactoring existing text just for the sake of changing English forms is discouraged - focus your efforts on something more fruitful. As with the Python documentation, consistency within a page or section is more important than global consistency.

References to Finesse

FINESSE 3 should be referred to in the documentation simply as FINESSE. Explicit inclusion of version numbers should only be made when referring to or comparing differences in behaviour.

Core documentation should focus on explaining how FINESSE 3 works, not how it's different to other tools or previous versions. Discussions of differences between FINESSE 2 and 3 should be kept in relevant sections. It's fine however to include brief "see also" directives in FINESSE 3 documentation pointing to such discussions.

Capitalisation

Various FINESSE-related terms have particular capitalisation conventions. Some are also defined as macros which format the text appropriately:

- |*Finesse*|, not e.g. *FINESSE*
- *Pykat*, not e.g. *PyKat*
- *KatScript*, not e.g. *kat script*, *katscript*, or *kat-script*

8.3.10 Linting tools

The doc8 package provides a command line tool to check documentation. Running `doc8` from the project root directory will have it automatically pick up the line length setting defined in `pyproject.toml`. For instance, you can lint the whole documentation source with:

```
$ cd /path/to/finesse/git/project/  # The directory that contains pyproject.toml.
$ doc8 docs/source
```

This will print a list of files and their corresponding issues, if any.

Search tags

The documentation includes a custom search scorer that can boost specific pages for particular search queries. Authors can control this by adding a `search-tags` directive to any RST page.

Usage

Place the directive anywhere in the page (typically near the top, after the title):

```
.. search-tags::
   :primary: mirror, mirrors
   :secondary: component, optic
```

Both options accept a comma-separated list of keywords:

primary
: Large score boost. Use this when the page is **the** main resource for a topic. For example, the mirror component page should have `:primary: mirror, mirrors`.

secondary
: Moderate score boost. Use this when the page is relevant but not the primary resource. For example, the Kat script syntax page could list many component names as secondary tags so that it appears in results but below the dedicated component pages.

Both options are optional — you can use just `:primary:` or just `:secondary:`.

The directive produces no visible output in the rendered page.

How it works

1. During the Sphinx build the `search_tags` extension (`docs/source/_ext/search_tags.py`) collects all tags and writes them to `_static/search_tags.js`.
2. The custom search scorer (`_static/search_scores.js`) reads this data and compares the user's query against the tags.
3. Matching pages receive a score boost that pushes them higher in the search results.

Because the tags live inside the RST source files, they automatically follow the page when the documentation is restructured.

Tips

- Keep the tags short, use single words or two-word phrases.
- Include common spelling variants (e.g. `mirror, mirrors`).
- Only add tags to pages that genuinely should be boosted because they are a good result for the keyword.

8.4 Authoring documentation

Extensive documentation about reStructuredText and MyST can be found online. Here are some resources to get started:

- reStructuredText Primer from the Sphinx documentation give a good overview of the format.
- reStructuredText User documentation from Docutils offers many links and information about reStructuredText.
- reStructuredText Quick reference from Docutils is a compact reference guide for reStructuredText.
- MyST-NB Documentation is the reference for using MyST-NB.
- MyST Markdown documentation is the reference for MyST Markdown, the language used by MyST-NB.

In the following, important points and guidelines for FINESSE will be presented.

8.4.1 reStructuredText (reST)

Sphinx's default file format is reStructuredText. It is a versatile and extensive language offering many possibilities for creating rich documents with internal and external linking, images, figures, tables and more. To be able to fully utilize the format, some basic concepts need to be understood, notably the concepts of role and directive. They are key mechanisms for adding semantic meaning and extended functionality to documents.

role
: A role is an inline markup element used to annotate small pieces of text with specific meaning or behavior. Roles are written using a colon syntax, such as `:role:`text`` (e.g., `:math:`E=mc^2``), and are typically used to insert things like hyperlinks, inline code, or mathematical expressions. Roles help enrich the document by adding metadata or context.

directive
: Directives are block-level constructs that extend reST's capabilities beyond basic text formatting. They begin with a double-dot prefix and include a name and optional arguments or content (e.g., `.. note::` or `.. code-block:: python`). Directives can be used to insert figures, generate tables of contents, or highlight source code.

8.4.2 Markedly Structured Text (MyST)

MyST extends standard Markdown (CommonMark) with features like directives and roles, similar to reStructuredText, making it compatible with Sphinx. MyST-NB allows the MyST language to be used inside Jupyter Notebook markdown cells. This allows authors to easily write Jupyter notebooks to be seamlessly integrated with the rest of the documentation while keeping them executable. See the Comparison between reST and MyST (page 461) for a short comparison of the notations of reST and MyST.

Table 8.1: Comparison between reST and MyST

	reStructuredText	MyST
Role	`:role-name:`role content``	`{role-name}`role content``

Directives	`.. directivename:: arguments`
	`   :key1: val1`
	`   :key2: val2`
	
	`   This is`
	`   directive content`
		```` ```{directivename} arguments ````
		`:key1: val1`
		`:key2: val2`
		
		`This is`
		`directive content`
		```` ``` ````

Label	`.. _my-label:`	`(my-label)=`

8.4.3 Special formatting of the word Finesse

As shown here, FINESSE uses a special formatting when displayed. To display it, you need to write `|Finesse|` when using reStructuredText and as `{{Finesse}}` when using MyST-NB.

Note

Do not use the special formatting in titles as it can break formatting when linked to or used in a table of contents.

8.4.4 What to add at the top of a page

At the top of a new page, include a label for internal linking and a title.

Here are simple examples of what a header of a page should look like:

reStructuredText

```
.. _my-section:

==================
My awesome section
==================

This amazing section contains ...
```

MyST-NB

```
(my-section)=

# My awesome section

This amazing section contains ...
```

8.4.5 References and links

To link a section, paragraph, figure, etc., a label must be associated with it. The label needs to be set before the corresponding element. Inside a directive, the keyword `:name:` can be used to create a label.

Table 8.2: Setting a label.

	reStructuredText	MyST
Label in text	`.. _my-label:`	`(my-label)=`
Label inside directive	`:name: my-label`	`:name: my-label`

8.4.6 Linking to internal and external objects

Sphinx enables you to cross-reference modules, classes, functions, and attributes both internal to the project and external to it. To cross-reference an internal object (or attribute, method etc.) simply use the following directives:

- `:mod:`.internal_module_name`` to create a link to an internal module,
- `:class:`.internal_class_name`` to create a link to an internal class,
- `:func:`.internal_function_name`` to create a link to an internal function or `:meth:`.internal_class_name.internal_function_name`` to create a link to an internal class method,
- `:attr:`.internal_class_name.internal_attribute_name`` to create a link to an internal class attribute.

The FINESSE documentation tooling provides additional cross-reference support for various situations:

- `:kat:command:`command_name`` to create a link to a kat-script command,
- `:kat:element:`element_name`` to create a link to a kat-script element,
- `:kat:analysis:`analysis_name`` to create a link to a kat-script analysis,
- `:issue:`123`` to create a link to an issue on the tracker,
- `:source:`path/to/script.py`` to create a link to a source code file relative to the FINESSE package root (i.e. `/src/finesse/`).

Linking to objects in external modules is just as simple. What external modules are available are defined by the `intersphinx_mapping` inside **finesse/docs/source/conf.py**. Details on how to specify new targets in this variable see the intersphinx documentation. The available module for interlinking are:

- Python
- Numpy
- Scipy
- Matplotlib
- Networkx
- Pygments
- Pygraphviz

8.4.7 Showing output from inline scripts

With MyST-NB, code execution and output can be used as usual. What you see is what you get.

For reST, the jupyter-sphinx extension is available which provides the ability to execute code embedded in the documentation in a Jupyter kernel, then show the code (with syntax highlighting) alongside the outputs of that code in the documentation. It also supports plot outputs (and others, such as LaTeX and JavaScript widgets).

Code is executed in the environment used to build the documentation, so the `finesse` package is available to import and run.

Simple code blocks can be executed with `.. jupyter-execute::` directives. More advanced usage is possible, such as running scripts within different kernels (using the `:id:` directive option), showing line numbers, importing

and showing the contents of scripts on the file system, etc. See the jupyter-sphinx documentation for more information.

Sections

With MyST-NB, the standard Markdown notation (`#`, `##`, …) need to be used to structure the document.

With reST, there are particular characters assigned to heading levels; the structure is determined from the succession of headings as defined in each `.rst` file. FINESSE documentation should nevertheless ideally follow this convention:

- = with overline, for page titles
- =, for sections
- –, for subsections
- ^, for subsubsections
- ", for paragraphs

8.4.8 Code examples

Write code examples using the same formatting rules (page 508) as Finesse source code.

Prefer the use of `.. code-block::` (provided by Sphinx) rather than `.. code::` (provided by docutils).

8.5 Guide to the Finesse source code

8.5.1 The Finesse Cython extensions

For performance reasons, all of the code which involves factoring and solving the interferometer matrix, and much of the code associated with the mathematically intensive tasks, must be written in the form of Cython extensions in order to leverage the speed of the produced C code.

If you are new to Cython, or need a refresher on some key language concepts, then see Useful resources (page 469) for details from the developers.

Structure

All Cython extensions in FINESSE must be structured in the same way as the Python sub-modules. This means that if your extension requires a definition file (*.pxd*) then it should exist in its own directory / sub-package within the code, where this directory then contains a *__init__.pxd* file if other Cython extensions need to interact with it and a *__init__.py* file if the Python code requires access to the extension.

Additionally, similarly to the Python code, if your Cython extension is comprised of multiple *.pyx* files then it should also exist in a separate directory (even if no *.pxd* files are needed).

For a brief summary on why it is important to follow the same structure as the Python package itself, see this FAQ answer on the Cython GitHub Wiki pages.

The 'finesse.cymath' extension

Using fast mathematical routines

The sub-module `finesse.cymath` is a Cython extension containing external declarations from the C headers `"math.h"` and `"complex.h"` as well as C implemented functions defining common mathematical routines required for the various parts of FINESSE. Where applicable, these methods are declared as `cpdef` such that they can be used from other Cython extensions (as a fast C function call) and Python modules (as a slower Python function call). Note that it is only the function call overhead that is different for any `cpdef` function - the code inside is implemented in pure C for all `finesse.cymath` functions and this is the same for both the generated C function and Python function.

The complex data type

One of the key elements exposed by `finesse.cymath` to other Cython extensions is the C type definition `complex_t` - defined as `np.complex128_t` which is just equivalent to the *double complex* C type.

Whenever your Cython extension requires complex numbers, always use this type definition rather than defining your own or redefining from elsewhere. To do this, simply add the following to your *.pyx* file(s):

```
from finesse.cymath cimport complex_t
```

Importing standard `"math.h"` functions

Here is the general process for using a standard C math function on non-complex types:

- By inspecting the `cdef extern from "math.h"` block at the start of `finesse/cymath/math.pxd`, check that the C function you need already exists within this.
- If it is already present, then you can simply *cimport* the function. For example, if you need the `sin` function acting on double type values then simply *cimport* as:

  ```
  from finesse.cymath.math cimport sin
  ```

- Otherwise, refer to the necessary function signature in the Common Mathematical Functions C reference and add this to the `cdef extern` block. For example, if you need the base-2 logarithm routine then simply add it as follows:

  ```
  # In this block (already present)
  cdef extern from "math.h" nogil:
      # all the existing functions...

      double log2(double x)
  ```

Importing standard `"complex.h"` functions

Similarly to above:

- By inspecting the `cdef extern from "complex.h"` block at the start of `finesse/cymath/complex.pxd`, check that the C function you need already exists within this.
- If it is already present, then you can simply *cimport* the function. For example, if you need the `csqrt` function then simply *cimport* as:

  ```
  from finesse.cymath.complex cimport csqrt
  ```

- Otherwise, refer to the necessary function signature in the Complex Number Arithmetic C reference and add this to the `cdef extern` block. For example, if you need the projection routine then simply add it as follows:

  ```
  # In this block (already present)
  cdef extern from "complex.h" nogil:
      # all the existing functions...

      double complex cproj(double complex z)
  ```

Importing custom fast arithmetic functions from `finesse.cymath`

In addition to the standard C functions, this module also provides various fast mathematical routines which are not defined in the C numerics libraries. These are typically declared in the relevant `*.pxd` file and then defined in the corresponding `.pyx` file; however some may be defined immediately in the `*.pxd` if they are declared as `inline`.

An example of such a function is the `crotate` routine which sits in `finesse.cymath.complex` and performs the rotation of a complex number (of type `complex_t`) by a given phase in radians. Importing such functions, at a C level, is as simple as the standard routines. For this example, one can just do:

```
from finesse.cymath.complex cimport crotate
```

to get access to this function.

If you need to implement any custom fast (C level) mathematics to use in some other finesse cython extension, then declare and define such functions in an identical way to those that exist within the sub-modules of `finesse.cymath`.

Optimising Cython code

Writing fast Cython code is a large topic, with many options and tricks available to users, so we will not cover everything here. See the links in Useful resources (page 469) for more details. Only the key aspects of Cython optimising with be detailed here.

Typing variables and returns

One of the major reasons why Python is slow compared to statically typed languages is that *all* variables, function arguments and function return types are `PyObject*` instances in CPython (unless otherwise stated). Python then requires that various book-keeping tasks must be performed on these objects (updating reference counters and adding tracebacks) when they are instantiated and returned from functions.

To avoid requiring these tasks to be performed if they don't need to be, then one can (and should) explicitly type any variable and/or return type where possible. Types of variables can be declared inside function bodies

```
cimport numpy as np # required for ndarray
import numpy as np # for zeros function

def foo():
    cdef:
        # declare x as an int
        int x

        # declare y and z as double-precision floating point
        double y, z

        # declare eps as a double and initialise
        double eps = 1e-10

        # declare mat as a NumPy array with 2 dimensions and initialise as a 10x10 matrix of␣
↪zeros
        np.ndarray[int, ndim=2] mat = np.zeros((10, 10), dtype=np.intc)

    # ...
```

And as arguments to any function (*note: passing arrays in this way is generally not the preferred method for |Finesse| Cython code - typed memory views are preferred, see* Typed memory-views (page 466))

```
cimport numpy as np

def bar(int x, double y, np.ndarray[double, ndim=2] mat)
```

And, finally, also as return types of `cdef` and `cpdef` functions (`def` *functions cannot have typed returns as they must always return* `PyObject*`)

```
cpdef double mean(np.ndarray[double, ndim=1] a):
    cdef:
        Py_ssize_t i
        Py_ssize_t N = a.shape[0]
        double sum = 0.0

    for i in range(N):
        sum += a[i]
```

(continues on next page)

(continued from previous page)

```
    return sum / N
```

Note the use of the type `Py_ssize_t` here - this type should always be given to variables which are used for indexing arrays.

Writing fast for-loops

In order to allow Cython to convert a for-loop into pure C, all loop incrementing and limit variables should be declared with `cdef` (if any of these are left untyped then they are assumed to be Python objects and so the loop cannot be translated to a simple, pure C loop). An example of this is shown in the `mean` function above - both `i` and `N` (which are used as the incrementing/indexing and limit variables, respectively) are declared as the type `Py_ssize_t`.

Typed memory-views

Typed memoryviews are the preferred objects for interacting with NumPy arrays in the FINESSE Cython extensions. They provide efficient access to the memory buffers underlying NumPy arrays without incurring any Python overhead, and have more features and a cleaner syntax than using `ndarray[Ty, ndim=n]` the style syntax we saw above. For more information, and a full feature list, see the Typed memoryview documentation.

The most important aspects of memoryviews are summarised here:

- Declare as

  ```
  cdef Ty[:] view_1D = exporting_object
  ```

  for a one-dimensional memoryview on some array of the same type *Ty*.

- For a higher-dimensional view use, e.g

  ```
  cdef Ty[:, :] view_2D = exporting_object
  ```

- Pass a view as an argument with

  ```
  def do_array_operation(double[:] array_view):
      ...
  ```

- All the arrays that we work with will have a contiguous C layout (the default memory structure of NumPy arrays). This means we should declare memoryviews with the

  ```
  cdef Ty[::1] contig_view = exporting_contig_object
  ```

  syntax - where this creates a 1D contiguous view on the memory buffer assigned to.

- This, of course, works for higher dimensional arrays too (here the elements in the second-dimension are one element, *sizeof(double)*, apart in memory - which is true for multi-dimensional C arrays)

  ```
  def operation_on_matrix(double[:, ::1] contig_matrix_view):
      ...
  ```

Multi-threading

Cython has support for OpenMP via the sub-module `cython.parallel`, which contains convenient wrappers such as the function prange. Since we perform a lot of elementwise operations in FINESSE, we can make use of this feature of Cython to distribute the work among multiple threads.

A simple example of using Cython parallelism is shown here, this silly example was chosen just to highlight which dependencies need to be declared *nogil*

```
cimport numpy as np
import numpy as np
from cython.parallel import prange

cdef extern from "math.h":
    # functions from C headers must also be declared with nogil
    # for use in multi-threaded Cython code
    double cos(double arg) nogil
    double sin(double arg) nogil

# declare this plain C function nogil as it is
# used within the body of a prange for loop
cdef double sin_cos_square(double x) nogil:
    return sin(x) * sin(x) + cos(x) * cos(x)

cpdef np.ndarray[double, ndim=2] transform(double[:, ::1] mat):
    cdef:
        Py_ssize_t i, j
        Py_ssize_t N = mat.shape[0]
        np.ndarray[double, ndim=2] out = np.zeros(mat.shape)

    # we use a prange here, with nogil flag set to True to indicate
    # that the loop body must be executed with the GIL released
    for i in prange(N, nogil=True):
        for j in range(N):
            out[i][j] = sin_cos_square(mat[i][j])

    return out
```

Compiler directives

Compiler directives are instructions which can be passed to Cython which modify the behaviour of certain parts of the code. Some of the available options (listed here) can be used to improve performance.

One of the key directives which we apply by default to all Cython extensions in FINESSE is the *cdivision* directive. This has the effect of translating all division operations in the Cython code to C style divisions - most importantly, this means that no zero-checking is performed on the denominator.

Other performance improving directives, which are used in certain extensions or as decorators to specific functions in FINESSE, are listed here for convenience:

- *boundscheck* = *False*: no checks on indices being out of bounds are carried out on arrays or memory-views.
- *wraparound* = *False*: disables the use of negative indices being passed to arrays and memory-views. If a negative index *is* passed then no checks are performed resulting in *Undefined Behaviour* (likely manifesting as segfaults).
- *initializedcheck* = *False*: disables checks on whether memory-views have been initialised on every access operation.

To apply a directive to a body of code within a function

```
cimport cython

def foo(double[:, ::1] mat):
    cdef:
        Py_ssize_t i, j
        Py_ssize_t N = mat.shape[0]

    with cython.boundscheck(False):
        for i in range(N):
            for j in range(N):
```

(continues on next page)

(continued from previous page)

```
                double element = mat[i][j]
                # ...
```

Or to a function as a whole

```
cimport cython

@cython.boundscheck(False)
def foo(double[:, ::1] mat):
    cdef:
        Py_ssize_t i, j
        Py_ssize_t N = mat.shape[0]

    for i in range(N):
        for j in range(N):
            double element = mat[i][j]
            # ...
```

Or to an extension (*.pyx* file) in entirety

```
#cython: boundscheck=False

def foo(double[:, ::1] mat):
    cdef:
        Py_ssize_t i, j
        Py_ssize_t N = mat.shape[0]

    for i in range(N):
        for j in range(N):
            double element = mat[i][j]
            # ...
```

And, finally, to all extensions (globally) via *setup.py*

```
# exts is a list of Cython Extension objects
cythonize(exts,
          compiler_directives={
              "boundscheck" : False
          })
```

Note that this is not recommended for arbitrary directives unless it is agreed upon by the FINESSE development team as a whole. Only directives which improve performance without having unexpected side-effects should be applied globally (e.g. *cdivision*).

Managing exceptions

Any *cdef* function in Cython which has a typed return cannot report Python extensions to its caller. This is because, when the return value is typed, it no longer returns a Python object (which is the default behaviour of all Cython, and Python, functions). As a result, in a function such as the one below:

```
cdef double inv_double(double x):
    if x == 0.0:
        raise ValueError("Division by zero.")

    return 1.0/x
```

the exception, in this case a `ValueError` instance, will be created, reported and destroyed all within the function scope. The function will then *return* the default return type value from the raised exception point - in this case, it would return a value of *0.0* if $x == 0.0$.

There are two solutions to this problem, but in the FINESSE Cython code we should only use one of these solutions - this is detailed below. The other solution is to not declare a return type for any *cdef* function which can raise an exception,

such that the return type is then *PyObject**. This is not ideal, however, as it doesn't allow Cython to fully-optimise the C source file it produces - more specifically, it results in calls to the CPython API functions: `__Pyx_XDECREF`, `__Pyx_XGIVEREF` and `__Pyx_AddTraceback` which are required when dealing with Python objects.

The solution which we use instead is documented in Cython error return values. Here, one explicitly declares that the function can raise and provides a value (of the same type as the return) which indicates that the function has raised. Following this, the general pattern we use is then shown in this pseudo-code snippet:

```
# Ty is the type of the parameter we want to compute with the function
cdef int func(Ty* result, other_args...) except -1:
    if variable in other_args is invalid:
        raise an exception

    # ...
    # do some calculations using other_args
    # ...

    # note this is equivalent to *result = ... in C, Cython uses array
    # access for pointers as *result is not valid Python syntax
    result[0] = ...

    return 0
```

This pattern informs Cython that an exception has been thrown when *func* returns -1. Cython then goes off and deals with reporting this to the caller and handles the traceback.

Checking a Cython extension for CPython interaction

When installing FINESSE, during the build process of the Cython extensions, HTML files are generated for all *.pyx* files which show the level of interaction each line has with the Python Virtual Machine.

An example of this is shown below. The yellow lines indicate Python interaction with the opaqueness of the colour giving the level of this interaction - i.e. stronger yellow means more Python interaction. These lines can be expanded to show exactly which CPython functions are being called.

When inspecting these annotated HTML files you should look out for any yellow lines present in the bodies of loops. It is these lines which are likely to be the culprits of bottlenecks in the Cython extension performance.

Documenting Cython extensions

Writing documentation for the Cython parts of FINESSE works largely the same as for the Python parts, but with a few subtleties arising due to the implicit use of types. One of the key things to note is that any *cdef* functions (regardless of whether they exist as part of a *cdef* class or not) with docstrings will not be documented during the Sphinx build process - this should be fairly obvious given that *cdef* functions are not exposed to the Python API. This is not a problem for the most part, as only FINESSE developers involved with writing Cython extensions will use *cdef* functions; everything accessible by FINESSE users (i.e. everything *but cdef* functions) is documentable.

Useful resources

The Cython documentation
: for the latest documentation pages.

Cython for NumPy users
: on the use of Cython with NumPy.

Musings on Cython
: for details on the ways of declaring functions in Cython, with profiling for real numerical problems.

Managing exceptions
: a note on raising exceptions from *cdef* functions.

```
+102: cpdef complex_t pixel_output(
 103:     BaseSimulation DC, Py_ssize_t node_id, double[::1] freqs, int[:, ::1] homs, complex_t qx, complex_t qy,
 104:     double x, double y,
 105:     double nr, double lambda0,
+106:     bint zero_tem00_gouy, f=None
 107: ):
 108:     cdef double given_freq
 109:
+110:     if f is not None:
+111:         given_freq = <double>f
    __pyx_t_3 = __pyx_PyFloat_AsDouble(__pyx_v_f); if (unlikely((__pyx_t_3 == (double)-1) && PyErr_Occurred())) __PYX_ERR(0, 111, __pyx_L1_error)
    __pyx_v_given_freq = ((double)__pyx_t_3);
+112:         return monochromatic_beam_pixel(
 113:             DC, node_id, freqs, homs, qx, qy, x, y, given_freq,
 114:             nr, lambda0, zero_tem00_gouy
 115:         )
 116:
 117:     else:
+118:         return spectral_beam_pixel(
 119:             DC, node_id, freqs, homs, qx, qy, x, y,
 120:             nr, lambda0, zero_tem00_gouy
 121:         )
 122:
 123:
+124: def ccd_output(
 125:     BaseSimulation DC, Py_ssize_t node_id, double[::1] freqs, int[:, ::1] homs, complex_t qx, complex_t qy, double nr,
 126:     double lambda0, double[::1] x, double[::1] y, bint zero_tem00_gouy, complex_t[:, ::1] out,
+127:     f=None
 128: ):
 129:     """
 130:     ccd_output(DC, node, freqs, homs, qx, qy, nr, lambda0, x, y, zero_tem00_gouy, out, f)
 131:
 132:     Computes the output image on a CCD at a specified `node` with the given `x` and `y`
 133:     co-ordinate arrays.
 134:     """
 135:     cdef:
 136:         Py_ssize_t i, j
 137:         double given_freq
 138:         double xval, yval
+139:         Py_ssize_t Npts = x.shape[0]
 140:
+141:     if f is not None:
+142:         given_freq = <double>f
+143:         for i in prange(Npts, nogil=True):
+144:             xval = x[i]
+145:             for j in range(Npts):
+146:                 yval = y[j]
 147:
+148:                 out[j][i] = monochromatic_beam_pixel(
 149:                     DC, node_id, freqs, homs, qx, qy, xval, yval, given_freq,
 150:                     nr, lambda0, zero_tem00_gouy
 151:                 )
 152:     else:
+153:         for i in prange(Npts, nogil=True):
+154:             xval = x[i]
```

8.5.2 KatScript grammar, parser and generator

Source code: script

KatScript is an application-specific, declarative language used to build FINESSE models as an alternative interface to the Python API. The KatScript parser converts KatScript from text files or the user's shell into FINESSE objects like models, components and detectors. The KatScript generator does the opposite, taking a `Model` and unparsing it back to KatScript.

Collections of tokens within KatScript represent elements such as `mirror` (page 142), detectors such as `pd` (page 180), functions such as `modes` (page 197), analyses such as `xaxis` (page 209), or comments and whitespace. Commands themselves may also contain expressions, and expressions may contain symbols. The parser has to handle all of these types of syntax.

Parsing KatScript is more complicated than many other typical small application-specific languages due to the nature of the script and the way in which it is typically used. KatScript is a declarative language, meaning that the order in which commands are defined should not influence the results; yet, because it has to create objects in Python - an imperative language - the parser must figure out the correct order in which to create commands and expressions that might rely on those defined elsewhere. Another issue to handle arises from the requirement to be able to regenerate KatScript from a parsed model in a similar form to the original script that built the model. This need arises from the way that FINESSE is typically used when building "base" scripts to represent gravitational wave detectors. The process of constructing such scripts involves optimising particular aspects of the script sequentially, where the outputs from analyses are used to update the script before another aspect is optimised. The final script is then the form that is shared with others who can then use the optimised script to study a particular behaviour of the interferometer at the operating point.

This page describes the formal KatScript grammar, and how the parsing and unparsing toolchains work on a high level.

> **Note**
>
> While the process of converting some script into machine code is typically called "parsing", parsing is strictly

just one part of the toolchain. As is typical, the FINESSE "parser" system is actually made up of a "tokeniser", a "parser" and a "compiler". Similarly, "unparsing" collectively refers to the process of converting a `Model` to KatScript, containing an "uncompiler" (itself containing an "unbuilder" and "unfiller"), an "unparser", and an "untokenizer".

Grammar

Source code: script/parser.py

KatScript is a declarative grammar supporting statements and arguments. The following sections contain a close approximation of the grammar rules in EBNF notation. When run, the parser starts at the `start` (page 471) rule and recursively reduces tokens until finished, backtracking if necessary.

Note

The declarations below are not an exact definition of KatScript; the real grammar is more complicated due to the way in which whitespace and errors are handled. It is sufficient however for the purpose of outlining the key rules.

Script

Scripts contain zero or more `script_line` (page 471) productions followed by an `ENDMARKER` (page 473) token.

```
start       ::= (script_line+)? ENDMARKER
script_line ::= (statement)? NEWLINE
```

Statements

Statements are either element-like (space delimited) or function-like (comma delimited). Both have types but only elements have names.

```
statement ::= element | function
```

Elements

Element statements are space-delimited and may not span multiple lines. They take the form `myelement myname 1 2 3 d=4 e=5`.

```
element                ::= NAME empty+ NAME (empty+ element_params)?
element_params         ::= element_value_list empty+ element_key_value_list
                         | element_value_list
                         | element_key_value_list
element_value_list     ::= positional_value (empty+ element_value_list)?
element_key_value_list ::= key_value (empty+ element_key_value_list)?
```

Functions

Function statements are comma-delimited, enclosed in brackets and may span multiple lines. They take the form `myfunction(1, 2, 3, d=4, e=5)`.

```
function                ::= NAME "(" function_params? ")"
function_params         ::= function_value_list empty* ","
                            empty* function_key_value_list
                            (empty* ',')?
                          | function_value_list (empty* ",")?
                          | function_key_value_list (empty* ",")?
function_value_list     ::= positional_value empty*
```

```
                              ("," empty* function_value_list)*
function_key_value_list ::= key_value empty*
                              ("," empty* function_key_value_list)?
```

Parameters

Parameters can be positional (simply a value) or keyword (a key, an equals sign, and a value).

```
positional_value ::= value !"="
value            ::= expr | function | array | NONE | BOOLEAN | STRING
key_value        ::= NAME "=" value
```

Expressions

Expressions can be unary (e.g. `-1`) or binary (e.g. `1+2`). Standard operator precedence is baked in to the rules by the way in which they are separated and referenced by each other.

Note

In practice, there are additional aspects to these rules to match `empty` (page 472) productions between terms of the expressions if it is in a context that allows it (e.g. a parenthesised expression or `function` (page 471)).

```
expr  ::= (expr ("+" | "-"))? expr1
expr1 ::= (expr1 ("*" | "/" | "//"))? expr2
expr2 ::= ("+" | "-") expr2 | expr3
expr3 ::= expr4 ("**" expr2)?
expr4 ::= "(" expr ")" | function | array | NUMBER | NAME | REFERENCE
```

Arrays

Arrays are comma-delimited `lists` of `values` (page 472) or `arrays` (page 472) contained within brackets (`[...]`). *Numerical* arrays are created with type `numpy.ndarray` when a standard `array` (page 472) is contained within an expression.

```
array        ::= "[" empty* array_values? empty* "]"
array_values ::= array_value (empty* "," empty* array_values)* (empty* ",")?
array_value  ::= expr | array | NONE | NUMBER | NAME | BOOLEAN
                 | REFERENCE | STRING
```

Empty

Emptiness can be represented by whitespace, comments or newlines. Emptiness is sometimes important (e.g. between parameters of an `element` (page 471)), sometimes not (e.g. between parameters of a `function` (page 471)).

```
empty ::= WHITESPACE | COMMENT | NEWLINE
```

Tokens

Tokens are matched using regular expressions, as shown below. The matching is pretty simple except for `numbers` (page 473). Numbers may be integer, float, imaginary or scientific, and may contain SI prefices such as *k*. Scientific forms cannot also use SI prefices. Number tokens are *not* tokenised with signs. Such unary operations are matched as expressions and eventually converted into functions in a later stage. Similarly, complex numbers are actually represented by two expressions (the real and imaginary parts) and a binary operator (+ or −), built by the parser into functions. The number matching regular expression is particularly huge; see `finesse.script.tokens.NUMBER` for the full form.

Note

The ENDMARKER (page 473) entry below is a *meta* token which has no representation in string form. It is used by the tokeniser to indicate the end of the current file.

```
NAME        ::= [a-zA-Z_][a-zA-Z0-9_.]*
NUMBER      ::= (see text)
STRING      ::= '[^\n'\\]*(?:\\.[^\n'\\]*)*'|"[^\n"\\]*(?:\\.[^\n"\\]*)*"
BOOLEAN     ::= true|True|false|False
NONE        ::= none|None
WHITESPACE  ::= [ \f\t]+
COMMENT     ::= #[^\r\n]*
NEWLINE     ::= \r?\n+
ENDMARKER   ::= (see note)
REFERENCE   ::= [a-zA-Z_][a-zA-Z0-9_.]*
```

Parsing

Parsing starts with the tokeniser, which converts raw characters into particular token types such as NAME (page 473), STRING (page 473), NUMBER (page 473), COMMENT (page 473), etc.

Because it must be possible to regenerate KatScript including comments and whitespace from a parsed FINESSE model, the parsing process produces a *concrete* syntax tree rather than an abstract syntax tree as is more typical. The tokeniser therefore does not ignore any token types, and the parser produces productions containing whitespace and comment tokens.

In the documentation below, tokens of a particular type are referred to using uppercase, such as NUMBER (page 473) to distinguish them from parsed productions such as expr (page 472) (expression).

Tokeniser

Source code: script/tokenizer.py

The available KatScript token types are defined in `src/finesse/script/tokens.py` using regular expressions. Text may match multiple token types, but the order in which they are defined in the overall regular expression that is built from the individual token expressions in `src/finesse/script/tokens.py` determines the matching priority.

Parser

Source code: script/parser.py

The parser reads the token stream until exhausted and attempts to reduce them to *productions*, which are containers representing particular rules in KatScript grammar. The KatScript parser implementation is a so-called *packrat* parser, using recursive descent to perform unlimited lookahead, in contrast to e.g. *LALR(1)* parsers that can only look at the next character to decide which production to use to reduce the current tokens. This algorithm uses potentially unlimited memory but is guaranteed to run in linear time due to the use of caching. In practice memory use is limited by the input script size and KatScript grammar, which are both typically small with respect to available memory on modern computers. Packrat parsers also provide other advantages over LALR(1) such as the ability to handle left-recursive rules.

The first parser rule searches for *all* tokens followed by an ENDMARKER (page 473). The ENDMARKER (page 473) is a special token yielded by the tokeniser to indicate the end of the file, and there should only be one, which indicates the parsing should stop. Within the tokens it sees, the parser then looks for *lines* (groups of tokens ending with a NEWLINE (page 473)) containing statement (page 471) productions, which can themselves be either element (page 471) or function (page 471) productions that accept one or more arguments. These productions accept the same types of arguments but differ in the way they require them to be delimited: in the case of element (page 471) arguments, the delimiters are spaces (on a single line), whereas for function arguments the delimiters are commas (and whitespace and newlines are optional).

Arguments can be numbers, null, booleans, expressions, references and strings. Expressions can be arbitrarily nested and take more care to parse in a way that obeys operator precedence; multiple rules are therefore required.

Containers

Some grammar rules in the parser have equivalent **containers**, which are objects that hold the tokens that represent the relevant production. Rules without equivalent containers usually provide data to a higher order rule that has a container. These containers are primarily used by the compiler but also for error handling, so they inherit from `TokenContainer` which defines some useful attributes and properties for figuring out where in the input the production appears. There are also `ArgumentContainer` and `ExtraTokenContainer` mixins which containers can inherit to conveniently add support for additional types of data. The attributes these mixins define are also recognised and supported by the compiler, so container-specific rules in the compiler are mostly not needed.

Arrays

Arrays are sequences of numbers, expressions, strings, or other arrays. This data structure is provided to allow such sequences to be passed to elements and commands. There can be some special behaviour depending on the location in which the array appears. Stand-alone array arguments are passed as `lists`, but arrays embedded inside expressions are converted to `ndarray` so as to support element-wise operations such as `2 * [1, 2]`. This context-specific behaviour is employed so that lists of strings can be passed to e.g. plot commands in KatScript without having these converted to `ndarray`, which are most suited to containing numbers.

Note that in order to support the default `list` type behaviour for arrays not used in expressions, the `array` (page 472) rule has to come before `expr` (page 472) in rules that can match both productions (otherwise all arrays regardless of contents would be converted to `ndarray`).

Error handling

Source code: script/exceptions.py

Error handling in the parser is not "free". Due to the nature of the recursive descent algorithm, if the syntax is incorrect the parser will back-track to the last successful production and try the next rule. If all available alternative rules are exhausted, only then would a syntax error normally be raised. This normally leads to unhelpful error messages, without any indication as to the syntax that actually caused the error, because the parser back-tracked all the way to the first rule. It is therefore also important to build a number of special rules into the grammar that match common error conditions, which can then trigger more useful error messages.

Handled errors usually raise a `KatScriptError`, which uses the provided error item (a token or container) and the original script to figure out where to determine where in the script the error(s) occurred. Depending on how FINESSE is being run, these error locations may be shown with markers or in a special colour. Here is an example of an error message:

```
from finesse.script import parse

parse("laser l1 P=1 freq=1M")  # "freq" is not a valid argument.
```

```
KatDirectiveBuildError:         (use finesse.tb() to see the full traceback)
line 1: 'laser' got an unexpected keyword argument 'freq'
-->1: laser l1 P=1 freq=1M
                   ^^^^
Syntax: l name P=1 f=0 phase=0 signals_only=false
```

Compiler

Source code: script/compiler.py

This section frequently shows the results of building KatScript into graphs, so we'll define a convenience function to let us see them:

```
import matplotlib.pyplot as plt
from finesse.script.compiler import KatCompiler
from finesse.plotting import plot_graph

def plot_and_show(script, **kwargs):
    """Compile script and show the syntax graph."""
    compiler = KatCompiler()
    compiler.compile(script, **kwargs)
    # Plot the graph, colouring nodes by their type.
    plot_graph(
        compiler.graph,
        layout="planar",
        node_labels=False,  # Reduce clutter.
        node_attrs=True,
        edge_attrs=True,
        attr_font_size=12,
        edge_font_size=8,
        ax=plt.figure(figsize=(12, 16)).gca(),
        node_color_key=lambda node, data: data["type"],
        bounding_ellipses=False,
        graphviz=False,
    )
```

The compiler's job is to take the parsed productions and turn them into a `model` and `elements` containing the relevant parameter values and references. There are three sub-steps to compilation: *filling*, *resolving* and *building*. These all involve writing to or reading from a graph that gets constructed to represent the KatScript's syntax.

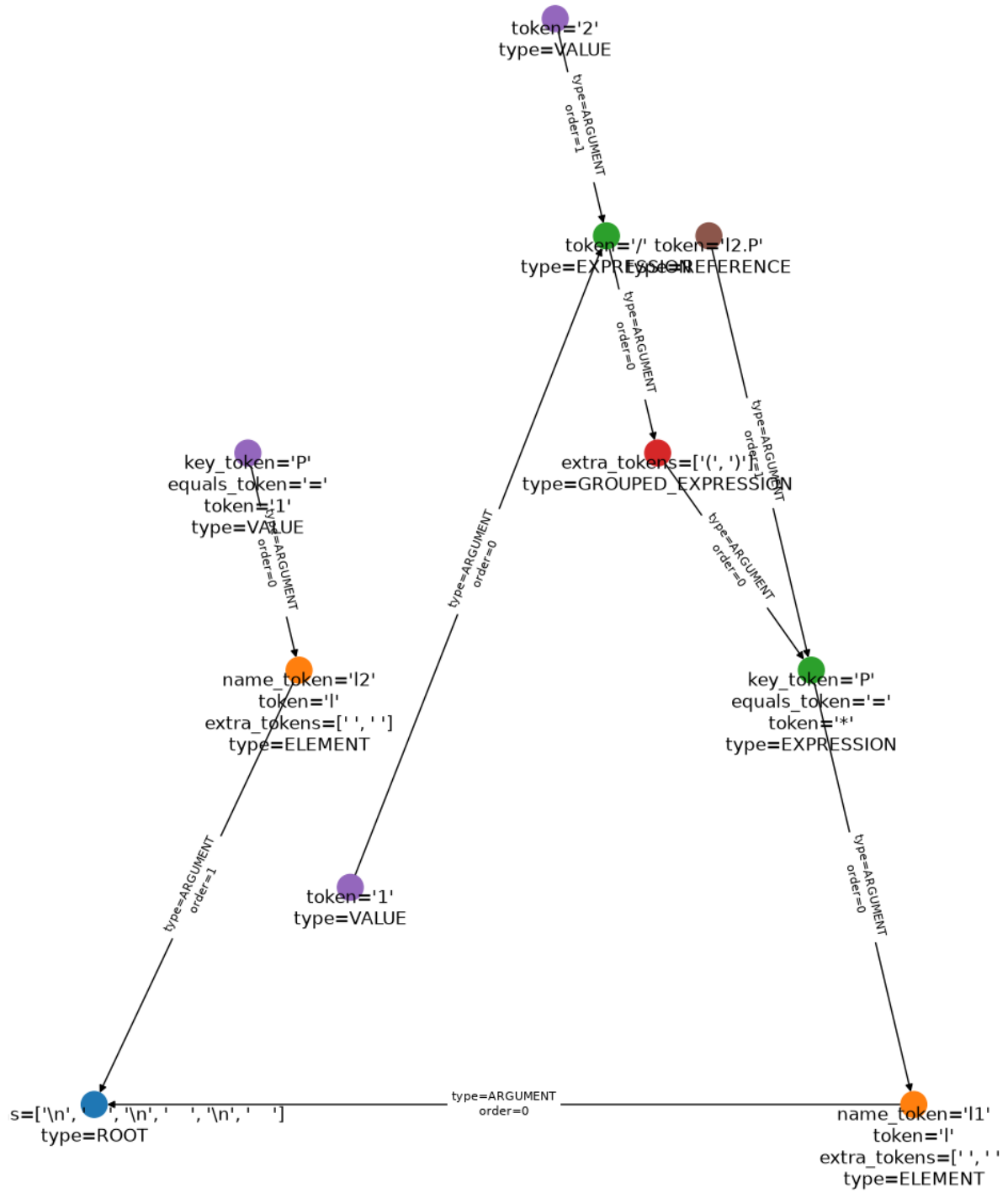
Filling

The filler matches different types of production yielded from the parsing step and creates corresponding nodes in the graph. Productions that support arguments have these arguments recursively filled as well, and edges of type `ARGUMENT` are drawn to each from the owning node.

As mentioned in the introduction, a *concrete* syntax tree is constructed because Finesse cannot ignore (and therefore must parse) whitespace, comments and other meta-tokens so that it can still reconstruct something resembling the original kat script later. In practice, the tokens stored in the container parsed from a particular part of the KatScript are stored as node attributes. All containers have a primary token, which is stored as the field `token`. Named containers (elements and functions) also store a `name_token` field, and extra characters such as the `=` of a keyword parameter are stored in `extra_tokens`.

After filling, the graph is strictly a tree because references between parameters of different elements have not yet been resolved. Everything descends from the `ROOT` node, which represents the model:

```
# Filled tree for a model containing a simple reference.
plot_and_show(
    """
    l l1 P=(1/2)*l2.P
    l l2 P=1
    """,
    resolve=False,
    build=False,
)
```



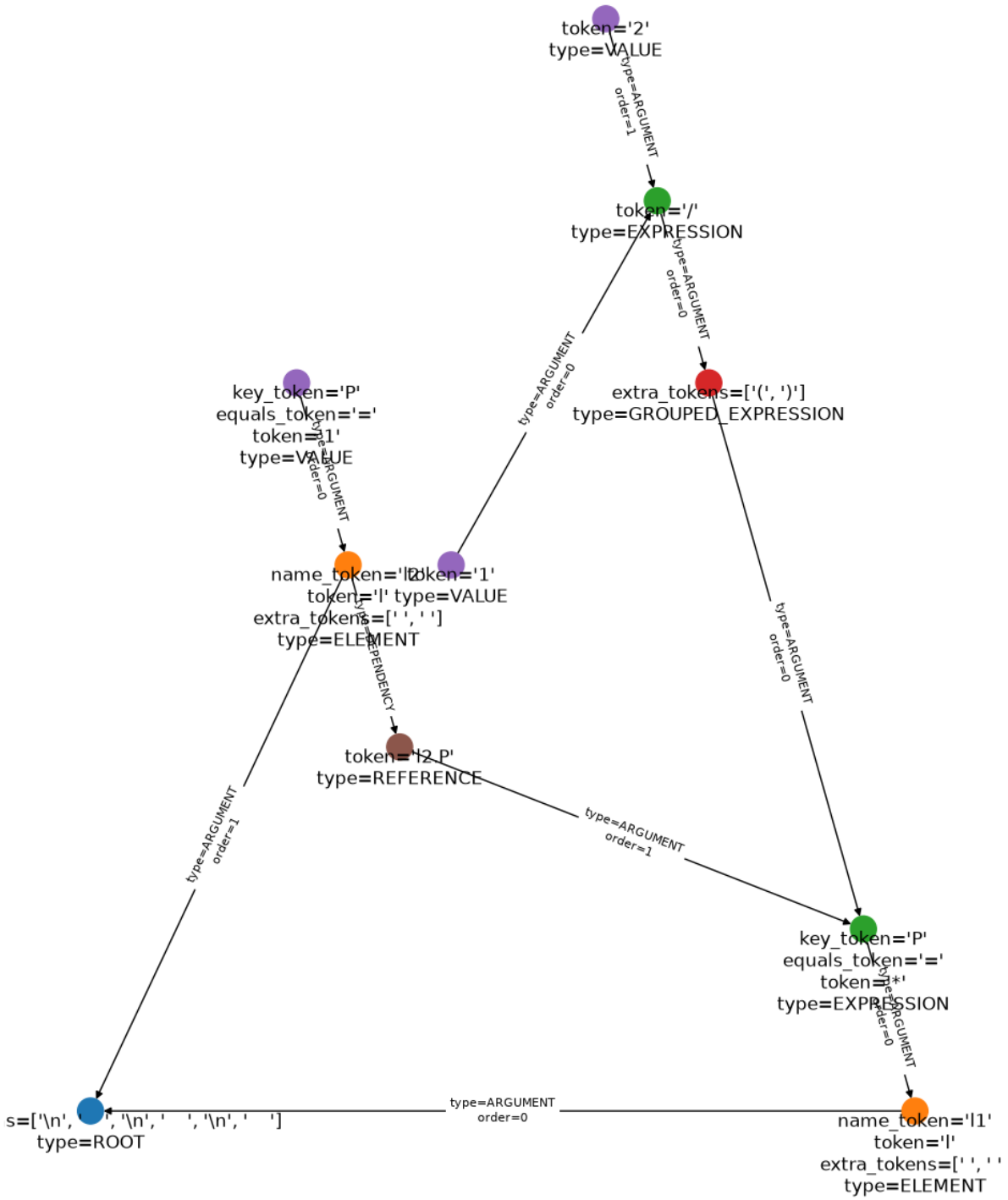

```
<Figure size 640x480 with 0 Axes>
```

Resolving

The resolver takes the graph and draws edges between dependencies created by references, i.e. arguments containing tokens of type NAME (page 473) or REFERENCE (page 473). It also performs some sanity checks on the graph, such as to check for invalid components, duplicate commands, and cycles between references (i.e. that which would be created by e.g. `m m1 R=m2.R T=0` and `m m2 R=m1.R T=0`).

After resolving, the graph looks something like this:

```
# Resolved graph for a model containing a simple reference.
plot_and_show(
    """
    l l1 P=(1/2)*l2.P
    l l2 P=1
    """,
    resolve=True,
    build=False,
)
```




```
<Figure size 640x480 with 0 Axes>
```

Note that the *l2.P* node is now connected to `l2` by a `DEPENDENCY` edge. The `l2.P` node is the first statement (`l l1 P=0.5*l2.P`)'s first argument (`P`)'s second argument. As a reference, this parameter depends on `l2` existing by the time it is being compiled, so an edge is drawn to the target to represen this dependency. We don't draw an edge to the target parameter node (`P`) itself because the target parameter is created at the same time as its owning element anyway (this however doesn't give us a way to handle references between parameters of the same element, but we'll

get to that in the *build* step).

The resolver also figures out the correct order in which the elements must be built given the dependencies created by references. If you look closely at the `ARGUMENT` edges from the root node, you will see that the `order` attribute has changed since the filling step. This is because `l1` depends on `l2` (via its `P=0.5*l2.P` argument) even though `l1` was defined earlier in the script.

Building

The builder starts at the root of the resolved graph and recursively follows each argument in order, building everything into relevant objects. It's in this step that the model is created (if one wasn't provided), each element node is built into a relevant `ModelElement`. Arguments are passed to the `ModelElement` constructors. Statement-level functions behave similarly, with the arguments being passed to a function instead of a model element.

The graph is used to determine whether a given expression can be eagerly evaluated. Eager evaluation is possible for any expression node that represents a closed branch of the graph, i.e. one without nested references. Such expressions are evaluated at compile time, and only the resulting value is passed to the Python object or function. Compound expressions containing references have only the closed sub-branches eagerly evaluated. As an example, take the graph shown in the previous section. The expression for the `P` parameter, `(1/2)*l2.P`, contains two top level clauses: `(1/2)` and `l2.P`. The latter is a reference and must be left unevaluated at parse time, but the former can be evaluated immediately. The builder therefore converts this expression to `0.5*l2.P` before it gets passed to `Laser.__init__()`.

The final step in building is handling *self-references*, which are parameters that reference another parameter in the same component. Such references create cycles in the graph during the resolving step, but these are intentionally ignored at that time. These references cannot be resolved in a single pass because element parameters are only created when the element itself is created, via its `__init__` method, and that method requires initial values for the parameters. Instead, parameters that contain self-references are built with an initial value set to a `Resolving` symbol, which, if evaluated (e.g. by code running in an element's `__init__`), will throw an error. Once a component gets built, any parameters with `Resolving` symbols then are fully resolved by calling the relevant build routine once again. This time, because the target parameters now exist (because the element was since created), it is possible to make real references to them using FINESSE symbols.

The conversion between parsed, resolved arguments and FINESSE objects is handled by `KatSpec`, which contains a series of `finesse.script.adapter.ItemAdapter` instances representing the available statements. This way, only the language constructs need to be hard-coded into the parser stack and the available commands, elements and analyses can be changed, e.g. to register a custom component.

Continuing with the example used so far, shown below is the result of the build step. Since this is also the last step in the parsing process, this is also what gets returned to the user.

```
from finesse.script.compiler import KatCompiler

compiler = KatCompiler()
model = compiler.compile(
    """
    l l1 P=(1/2)*l2.P
    l l2 P=1
    """
)
```

The `Model` contains the relevant items:

```
print(model.info())
```

```
<finesse.model.Model object at 0x7cdef03242d0>

1 optical mode
==============

    - [0 0]
```

(continues on next page)

(continued from previous page)

```
2 components
============

    - Laser('l2', signals_only=False, f=0.0, phase=0.0, P=1.0)
    - Laser('l1', signals_only=False, f=0.0, phase=0.0, P=(0.5*l2.P))

0 detectors
===========

0 cavities
==========

0 locking loops
===============
```

Unparsing

Source code: script/generator.py

The job of the unparser is to transform a FINESSE `Model` and other objects into KatScript. Currently this is an experimental feature, though it is already quite powerful - for example, combined with the legacy parser it is capable of converting FINESSE 2 style KatScript to that of FINESSE 3.

The unparser can be used on models originally built with KatScript or via the Python API; it does not matter. The original KatScript form is currently not retained during model compilation, so unparsing does not retain the original form: script order may be different, comments are lost and whitespace is normalised. Future improvements to the unparser are planned to allow the original parsed form of a model to be retained as much as practical (see **#200**).

The unparser is a toolchain containing an *uncompiler* (itself containing an *unbuilder* and *unfiller*), an *unparser*, and an *untokenizer*. The majority of the computation takes place in the uncompiler stage, with unparsing and untokenizing relatively trivial operations. These are explained in the sections below.

Uncompiler

The uncompiler transforms a `Model` or other FINESSE object into KatScript by first building a `KatGraph` and then tokenising it. The graph filling is performed by the *unbuilder* and the tokenising is performed by the *unfiller*.

Unbuilder

The graph is built using a functional programming dynamic dispatch paradigm, applying polymorphic behaviour depending on type (using `functools.singledispatch()`). The dispatcher is called recursively, with the code for each matched type extracting and dispatching contained objects, adding nodes for each intermediate step, until types that have direct `KatToken` analogues are reached. Edges are drawn connecting related nodes, such as the name, parentheses, delimiters and parameters of `functions` (page 471).

For example, a model gets built by passing it to the dispatcher. The dispatcher matches and runs the code for `Model`, which extracts its elements, commands and analysis, further calling the dispatcher on each. In the case of the model's elements, the dispatcher matches and runs the code for `ModelElement`, which extracts, among other things, the `model parameters` and once more passes these to the dispatcher. Parameters containing terminal types such as numbers are again dispatched, with the dispatcher this time matching code that creates a `KatNumberToken`. The graph filled by this sequence is a tree (unlike the graph built by the parser, where references create edges between branches), with each branch (representing elements, commands and analyses and their constituents therein) ultimately terminating with nodes corresponding to `tokens`.

Unfiller

The graph created by the unbuilder is converted into a series of `production` objects by using the type attribute defined for each node to determine which production constructor to call. Connected nodes are themselves recursively converted to appropriate productions.

The productions are added to a `KatScript` object. This is then passed to the *unparser*.

Unparser

The unparser receives a `KatScript` object, loops through its productions, extracts its tokens in order and calls the *untokenizer*.

Untokenizer

The untokenizer takes `KatToken` objects as input and converts them to their corresponding KatScript string forms. In practice, the token's raw value is simply extracted and returned.

8.5.3 Requirements for building, running and developing Finesse

FINESSE requires Python 3.9 or higher. Python 3.8 is a hard requirement due to the use of positional-only arguments, so-called walrus operators, the use of `functools.singledispatchmethod` and the requirement for `dict` to be reversible in the code. The requirement for Python 3.9 comes from using Cython 3 with SciPy.

FINESSE additionally requires some system dependencies:

- Python development header files
- SuiteSparse 4.0 or higher
- A C compiler (typically `gcc`)

> **Note**
>
> These packages are only required to build FINESSE, not to run it. Users who merely wish to use FINESSE, not develop it, do not need to install them and can instead rely on the various pre-built packages (page 19) that are available.

An optional system runtime dependency is GraphViz, used to visualise graphs of interferometer networks. If you don't plan to use such features, you don't need to install GraphViz.

> **Note**
>
> If used, the Conda development (page 21) environment installs GraphViz.

FINESSE also depends on various Python packages to be able to build and run, as discussed in the next section.

Python package requirements for Finesse

Source code: setup.py, pyproject.toml, environment.yml, environment-win.yml, generate_environment_files.py

As a short summary, requirements should be manually added to **pyproject.toml**. There is a pre-commit hook that automatically regenerates the **environment.yml** and **environment-win.yml** files.

This ensures these three requirements files are kept in sync.

There is another pre-commit hook that ensures that requirements that are listed multiple times in **pyproject.toml** are identical (NumPy is listed as a build requirement and a runtime requirement for instance)

The following section describes the logic behind this redundancy.

Defining dependencies for Finesse

FINESSE's Python requirements are defined for maximum convenience to users and developers using a hybrid approach: they are listed using standards defined by **PEP 517** and in Conda environment files (page 21).

In theory, with the **PEP 517** configuration in **pyproject.toml**, a user or developer can simply run `pip install finesse` or `pip install .` to build and install FINESSE either from PyPI or from the local directory, respectively. This is the case for most Linux and OSX systems on x86 architecture. Due to a lack of availability of dependencies on PyPI for some systems (such as for AMD64 architecture), environment files for Conda (using the conda-forge channel which provides the required PyPI packages across a wider variety of systems) are provided. When used, Conda takes over the responsibility for providing the required runtime and build dependencies for FINESSE, allowing users to use the program on a wider variety of systems.

Note

Nomenclature

Conda is the package manager, *conda-forge* is the name of the community-driven package channel that is used to install the dependencies for FINESSE.

In practice, the hybrid approach means that any changes to FINESSE's runtime or build dependencies require changes to *both* **pyproject.toml** (PEP 517 runtime and build dependencies, respectively), *and* **environment.yml** and **environment-win.yml** (Conda runtime and build dependencies on Linux/OSX and Windows, respectively).

PEP 517 builds

PEP 517 requires a Python function available on the current path to act as a build backend (responsible for compiling the package and adding it to the path). This is currently set in FINESSE to `setuptools.build_meta`, though this can be changed later if a tool offering more useful features becomes available.

The `setuptools.build_meta` backend requires *runtime* requirements, i.e. those required to be installed in the environment for FINESSE to run, to be listed in `pyproject.toml` and/or `setup.py` files in the project root. It's not strictly necessary to use *both* files with `setuptools` (indeed it's now recommended to use only the declarative `pyproject.toml` where possible), but because FINESSE needs Cython extensions with relatively complex configurations to be built by the backend it uses a `setup.py` to define the required logic. It may be possible to remove `setup.py` later (see **#367**).

Build requirements, on the other hand, are defined in `pyproject.toml`.

Note

PEP 517 recommends (but does not strictly require) that build frontends by default set up an isolated environment in which the backend can build the project. This means that build requirements are installed in a temporary location used only for the building of the package, then deleted when the newly built package is moved to the user's environment and the temporary environment is removed. This behaviour is usually beneficial because it helps to catch missing dependencies and removes the need for build requirements (e.g. `Cython`) to be available in the local environment, but when debugging build issues it can be useful to switch this behaviour off. In pip this can be done by adding the `--no-build-isolation` flag to `pip install`. Note that you'll need to ensure the `inplacebuild` extras listed in `pyproject.toml` are installed.

When FINESSE is to be built, by default `pip` will create a temporary, isolated environment with only the standard library available, then install the build dependencies listed in `pyproject.toml` using PyPI, before building using the specified build backend. This isolation is intentional to ensure that the intended build dependencies are the (only) ones used to build the package. Build isolation is useful to determine whether an undeclared (Python) dependency is is involved in the build, which could lead to trouble installing the package on other machines.

When FINESSE is to be run, it simply uses the Python packages available on the current path. If the user installed

FINESSE using `pip install finesse` or `pip install .` these packages will have been installed from PyPI and added to the path.

Conda builds

Conda provides packages for a wide variety of systems, and, unlike pip, takes care of installing system dependencies on behalf of the user. For example, Conda can install the FINESSE system requirement *SuiteSparse* without the user having to do it themselves.

Note

Conda commands can be notoriously slow to execute. Mamba is a much faster drop-in replacement for all conda commands (e.g. `conda install` can be replaced with `mamba install`). Alternatively, you can add the `--solver libmamba` argument when using conda commands.

Requirements for FINESSE within a Conda environment are defined in **environment.yml** and **environment-win.yml**.

When building FINESSE within a Conda environment, it is not necessary and often not desirable to use pip's PEP 517 isolated build system. As described above, part of the PEP 517 default build process is to install build requirements using PyPI, but these requirements are not always available from PyPI for the target platform. It is therefore typically desirable to build FINESSE on such platforms *in-tree* using `pip install . --no-build-isolation`, which tells pip to use the build dependencies from the existing (Conda) environment. It's for this reason that build dependencies, not just runtime dependencies, are also defined in `environment.yml` and `environment-win.yml`.

Warning

Dependencies in the Conda environment files refer to their name in the conda-forge channel, not their name on PyPI. Most Conda packages use the same name as their PyPI counterparts, but some do not. For example, interchanging of hyphens and underscores is allowed by pip but not by Conda. A good mapping of non-standard conda-forge package names can be found as part of the `grayskull` package here.

The *extra* dependencies for development (such as `sphinx` for building the documentation) are also defined in `environment.yml` and `environment-win.yml`, and therefore get installed when the user sets up their development environment.

Warning

The conda command-line interface is (at times of writing) a confusing, badly documented mess. They are working to improve it, but progress seems to be slow. The main confusion is the presence of `conda env`. It is an older, completely separate program that was merged into the conda codebase. It supports the environment.yml file syntax and is always aware of the environment it is running in, allowing for a simple `conda env update/create environment.yml` to either update or create an environment.

It unfortunately does not support any of the newer conda features, like specifying the channel with the `-c` argument or specifying multiple requirement files. The `install` subcommand is also missing.

The default conda command supports all of the above and adds a few useful commands like `search` and `clean`. However, it is not aware of the current environment and always needs to specify the environment with the `-n` command. On top of that it **does not** support the *environment.yml* file syntax at all!

Version pinning

The versions of the FINESSE runtime requirements are set by a compromise between the desire to use the latest (and usually greatest) versions and the need to help package managers include FINESSE alongside other tools with their own, possibly conflicting, requirements. It is therefore best to set only minimum required versions in `pyproject.toml`,

`environment.yml`, and `environment-win.yml`, leaving the maximum version open unless there are known incompatibilities.

We rely on the regular running of the tests in the continuous integration pipeline to catch issues created by the latest versions of requirements. In cases where a particular dependency version is identified to cause problems with FINESSE, it can be forbidden in the version string for the respective dependency in the relevant file (see next section). It is also useful to leave a note next to the requirement to inform others (including package maintainers, who may not be FINESSE developers) why this is the case.

Modifying requirements

Modification of requirements should only be done in the `pyproject.toml` file, the conda files should update automatically with the pre-commit hook:

- Runtime requirements are listed within the `dependencies` key of the `[project]` section of `pyproject.toml`.
- Build requirements are defined in the `requires` key within the `[build-system]` section and the `inplacebuild` key of the `[project.optional-dependencies]` of `pyproject.toml`.
- Extra requirements are defined in the `[project.optional-dependencies]` section of `pyproject.toml` and can be used to specify optional requirements for users or developers. These are not installed by most frontends by default. For example, the key `docs` lists requirements for building the documentation. Extras are duplicated in `environment.yml` / `environment-win.yml`. Note that conda does not support optional dependencies, so the extra dependencies always get installed when using conda.

PEP 508 specifies the format for requirements strings.

> **Warning**
>
> As described in a previous section, the *building* of FINESSE usually takes place in an isolated environment, so most build requirements these can be pinned in `pyproject.toml` to whatever version is desired without causing trouble elsewhere. It is important, however, that at least the `numpy` requirement are set to the same versions in `pyproject.toml` because parts of the FINESSE code are compiled and linked against the `numpy` C ABI. Using different versions of `numpy` for building and running FINESSE frequently results in numerous warnings issued by `numpy`, or even errors. It's also very useful from the point of view of debugging to maintain binary compatibility between FINESSE built in an isolated environment (i.e. via the full build process) and in-place (e.g. via `make`).

After modifying requirements, verify that the tests still pass and build process still operates as expected. Also consider making the changes in a branch, then push to GitLab to allow the continuous integration (page 526) pipeline to build and run using the updated dependencies too, before merging into the `develop` branch.

8.6 Finesse internal code structure

8.6.1 Interferometer Matrix Equation

$$\begin{pmatrix}\text{interferometer}\\ \text{matrix}\end{pmatrix} \times \begin{pmatrix}\vec{x}_{\text{sol}}\end{pmatrix} = \begin{pmatrix}\vec{x}_{\text{RHS}}\end{pmatrix} \tag{8.1}$$

The whole of Finesse (excluding things like the parser, and beamtracing/propagation code) revolves around setting up and solving the matrix equation (8.1) by inverting the interferometer matrix and multiplying it with $\vec{x}_{\text{RHS}}$

There is a separate matrix for the carrier and signal calculations (page 492)

Most of this page mentions code currently implementing the sparse solver, which uses Suitesparse KLU sparse matrix solver under the hood. In principle the code architecture supports implementing different solvers, like a possible graph-based symbolic solver.

Solution Vector

The desired output is the solution vector $\vec{x}_{\text{sol}}$. In the codebase this vector is named `out_view` in **simulations/homsolver.pxd**. It contains the field (light amplitude and phase) at every Node (page 492), for every frequency and for every higher order mode.

The index into this vector is calculated in the `finesse.simulations.homsolver.HOMSolver.field()` method in **simulations/homsolver.pyx**, which is used in the `finesse.simulations.homsolver.HOMSolver.get_out()` to get a single value and `finesse.simulations.homsolver.HOMSolver.node_field_vector()` to get a pointer to the HOM vector.

Most of the detectors end up calling one of these functions (indirectly, via their workspaces) to get the field at a certain node and do some further calculations. See for instance `c_ad_single_field_output` in **detectors/compute/amplitude.pyx**.

Right hand side vector

The right hand side vector $\vec{x}_{\text{RHS}}$ has the same structure as the solution vector, but all the values are initially set to zero, except for source nodes such as lasers and modulators (described in more detail in Input fields or the 'right hand side' vector (page 120)).

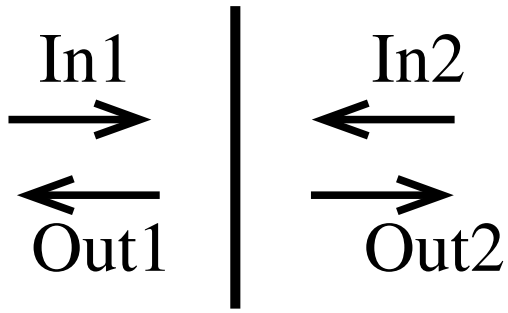
Source values are set using the `finesse.simulations.sparse.solver.SparseSolver.set_source()` and `set_source_fast` functions defined in **simulations/sparse/solver.pyx**. You can see it is being indexed with the same `field` and `field_fast` functions mentioned above.

You can see an example in `c_laser_carrier_fill_rhs` in **components/modal/laser.pyx**, where the laser field values are being set.

Interferometer Matrix

The interferometer matrix contains the linear equations that couple the field values of different nodes together.

The following simple example describes the field couplings for a mirror (also described here Mirror (page 114)):



$$\begin{pmatrix} \text{Out1} \\ \text{Out2} \end{pmatrix} = \begin{pmatrix} a_{11} & a_{21} \\ a_{12} & a_{22} \end{pmatrix} \begin{pmatrix} \text{In1} \\ \text{In2} \end{pmatrix} \tag{8.2}$$

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ -a_{11} & 1 & -a_{21} & 0 \\ 0 & 0 & 1 & 0 \\ -a_{12} & 0 & -a_{22} & 1 \end{pmatrix} \times \begin{pmatrix} \text{In1} \\ \text{Out1} \\ \text{In2} \\ \text{Out2} \end{pmatrix} = \begin{pmatrix} \text{In1} \\ 0 \\ \text{In2} \\ 0 \end{pmatrix}. \tag{8.3}$$

In a typical interferometer, most components only couple with components directly attached to them. Therefore the interferometer matrix will be 'sparse' (most entries are zero).

CCSMatrix

For memory efficiency and performance, finesse stores the IFO matrix in compressed column storage format (CCS/CSC), which is implemented in the `finesse.cymath.cmatrix.CCSMatrix` class in **cymath/cmatrix.pyx**. This resource has a nice explanation of the format.

The most important methods are:

- `finesse.cymath.cmatrix.CCSMatrix.declare_equations()`
- `finesse.cymath.cmatrix.CCSMatrix.declare_submatrix_view()`
- `finesse.cymath.cmatrix.CCSMatrix.construct()`

Multiple calls to `declare_equations` define the structure and the order of the sparse matrix. Since the sparse matrix represents a linear system of equations, every row relates to an 'equation' coupling the corresponding field to other fields in the system of equations, with the values in the row describing the coupling coefficients. Equation (8.3) describes the following 4 equations:

$$\begin{aligned} \mathrm{In1} &= \mathrm{In1} \\ -a_{11}\cdot\mathrm{In1} + \mathrm{Out1} - a_{21}\cdot\mathrm{In2} &= 0 \\ \mathrm{In2} &= \mathrm{In2} \\ -a_{12}\cdot\mathrm{In1} + \mathrm{Out2} - a_{22}\cdot\mathrm{In2} &= 0 \end{aligned}$$

We can set up this problem using `finesse.cymath.cmatrix.KLUMatrix` (which derives from `finesse.cymath.cmatrix.CCSMatrix`, but interfaces to KLU as well so we can solve the matrix at the end):

```
import numpy as np
from finesse.cymath.cmatrix import KLUMatrix

mat = KLUMatrix(name="mirror_example")
In1 = mat.declare_equations(Neqs=1, index=0, name="I_In1", is_diagonal=True)
Out1 = mat.declare_equations(Neqs=1, index=1, name="I_Out1", is_diagonal=True)
In2 = mat.declare_equations(Neqs=1, index=2, name="I_In2", is_diagonal=True)
Out2 = mat.declare_equations(Neqs=1, index=3, name="I_Out2", is_diagonal=True)

mat.construct()
mat.print_matrix()
```

```
Matrix mirror_example: nnz=4 neqs=4
    (col, row) = value
    (0, 0) = (1+0j) : I_In1 mode=0 -> I_In1 mode=0
    (1, 1) = (1+0j) : I_Out1 mode=0 -> I_Out1 mode=0
    (2, 2) = (1+0j) : I_In2 mode=0 -> I_In2 mode=0
    (3, 3) = (1+0j) : I_Out2 mode=0 -> I_Out2 mode=0
```

Here we also see the function of `finesse.cymath.cmatrix.CCSMatrix.construct()`, which needs to be called after declaring the equations to allocate the memory and fill in the values.

So far we have only set up a system of 4 equations where every field only directly couples to itself (setting `is_diagonal=True` fills in a `1` in the diagonal positions). Note that by setting `Neqs=1` we only declare a single equations per index, but we can also declare a block of equations (representing higher order modes for example).

To set up coupling between the different fields, we use `finesse.cymath.cmatrix.CCSMatrix.declare_submatrix_view()`:

Note

To make the matrix printing work correctly, we have to comply with the convention that the names of submatrices created by `finesse.cymath.cmatrix.CCSMatrix.declare_equations()` start with an `I`, while submatrices created by `finesse.cymath.cmatrix.CCSMatrix.declare_submatrix_view()` should not start with an `I`.

```
a11 = mat.declare_submatrix_view(In1.from_idx, Out1.from_idx, "_In1->Out1", False)
a22 = mat.declare_submatrix_view(In2.from_idx, Out2.from_idx, "_In2->Out2", False)
```

(continues on next page)

(continued from previous page)

```
a12 = mat.declare_submatrix_view(In1.from_idx, Out2.from_idx, "_In1->Out2", False)
a21 = mat.declare_submatrix_view(In2.from_idx, Out1.from_idx, "_In2->Out1", False)

mat.construct()
mat.print_matrix()

print(f"{a11.shape=}")
```

```
Matrix mirror_example: nnz=8 neqs=4
    (col, row) = value
    (0, 0) = (1+0j) : I_In1 mode=0 -> I_In1 mode=0
    (0, 1) = 0j     : I_In1 mode=0 -> I_Out1 mode=0
    (0, 3) = 0j     : I_In1 mode=0 -> I_Out2 mode=0
    (1, 1) = (1+0j) : I_Out1 mode=0 -> I_Out1 mode=0
    (2, 1) = 0j     : I_In2 mode=0 -> I_Out1 mode=0
    (2, 2) = (1+0j) : I_In2 mode=0 -> I_In2 mode=0
    (2, 3) = 0j     : I_In2 mode=0 -> I_Out2 mode=0
    (3, 3) = (1+0j) : I_Out2 mode=0 -> I_Out2 mode=0

a11.shape=(1, 1)
```

The first two arguments are the indices of the nodes between which we want to declare a coupling. In this case, since there is only 1 equations per node, we are creating a memory view to a 1x1 matrix, which represents a single coupling coefficient. The power of this abstraction is more apparent when you have multiple equations per node, for which `declare_submatrix_view` returns a view of a dense matrix of which the elements are located at the appropriate locations in the overall sparse matrix.

We can now 'fill' the sparse matrix by overwriting the values in the submatrix views using the NumPy array slicing syntax:

```
r = np.sqrt(0.9)
t = np.sqrt(0.1)

a11[:] = r
a22[:] = r
a12[:] = 1j * t
a21[:] = 1j * t

mat.print_matrix()

# print in dense format
np.set_printoptions(precision=1, linewidth=250)
print(mat.to_scipy_csc().todense())
```

```
Matrix mirror_example: nnz=8 neqs=4
    (col, row) = value
    (0, 0) = (1+0j)                     : I_In1 mode=0 -> I_In1 mode=0
    (0, 1) = (-0.9486832980505138+0j)   : I_In1 mode=0 -> I_Out1 mode=0
    (0, 3) = (-0-0.31622776601683794j)  : I_In1 mode=0 -> I_Out2 mode=0
    (1, 1) = (1+0j)                     : I_Out1 mode=0 -> I_Out1 mode=0
    (2, 1) = (-0-0.31622776601683794j)  : I_In2 mode=0 -> I_Out1 mode=0
    (2, 2) = (1+0j)                     : I_In2 mode=0 -> I_In2 mode=0
    (2, 3) = (-0.9486832980505138+0j)   : I_In2 mode=0 -> I_Out2 mode=0
    (3, 3) = (1+0j)                     : I_Out2 mode=0 -> I_Out2 mode=0

[[ 1. +0.j   0. +0.j   0. +0.j   0. +0.j ]
 [-0.9+0.j   1. +0.j   0. -0.3j  0. +0.j ]
 [ 0. +0.j   0. +0.j   1. +0.j   0. +0.j ]
 [ 0. -0.3j  0. +0.j  -0.9+0.j   1. +0.j ]]
```

Note

The submatrix views deriving from `finesse.cymath.cmatrix.SubCCSView` also provide optimised c-functions for filling values. The `z` in the naming refers to complex values, `a` to a scalar and `M` to a matrix.

- `fill_za` fills the entire submatrix with the same value `a`
- `fill_zm` copies over values from matrix `M` with the same shape
- `fill_za_zm` fills the submatrix with the scalar-matrix product of `a * M`

The reason they all exist as separate functions is for optimization only.

These submatrix view objects are being kept track of in 'connections' objects like `finesse.components.modal.mirror.MirrorOpticalConnections`, which are being stored in Workspace (page 492) (like `finesse.components.modal.mirror.MirrorWorkspace`)so eventually the workspaces can call their fill functions to fill in couplings in the matrix. See for example the `mirror_fill_optical_2_optical` in **components/modal/mirror.pyx**, which ends up calling the `fill_negative_za_zm_2` method of `SubCCSView`.

We have now created a matrix identical to (8.3), and we can get to 'solving' it. This consists of three steps:

- Factoring the matrix into LU form with `finesse.cymath.cmatrix.KLUMatrix.factor()`
- Inverting the matrix with `finesse.cymath.cmatrix.KLUMatrix.solve()`
- Multiplying the inverse matrix with $\vec{x}_{\text{RHS}}$ to get $\vec{x}_{\text{sol}}$. This is also done by `finesse.cymath.cmatrix.KLUMatrix.solve()`, which returns a view of $\vec{x}_{\text{RHS}}$ which has been overwritten with the solution values

We will first inject an initial value for the incoming field:

```
mat.set_rhs(In1.from_idx, 1.0)
```

And finally do the solving:

```
mat.print_rhs()

mat.factor()
mat.solve()

mat.print_rhs()
```

```
Vector mirror_example: neqs=4
    (row) = value
    (0) = (1+0j) : I_In1 mode=0
    (1) = 0j     : I_Out1 mode=0
    (2) = 0j     : I_In2 mode=0
    (3) = 0j     : I_Out2 mode=0


Vector mirror_example: neqs=4
    (row) = value
    (0) = (1+0j)                   : I_In1 mode=0
    (1) = (0.9486832980505138+0j) : I_Out1 mode=0
    (2) = 0j                       : I_In2 mode=0
    (3) = 0.31622776601683794j     : I_Out2 mode=0
```

You can see the matrix in action by passing `debug_mode=True` via the `simulation_options` dict of `finesse.model.Model.run`

```
import finesse
import numpy as np
```

(continues on next page)

(continued from previous page)

```
model = finesse.Model()

np.set_printoptions(linewidth=200, precision=1)

model.parse(
    """
        l l1
        s s1 l1.p1 m1.p1
        m m1 R=0.9 T=0.1
    """
)
model.run(simulation_options={"debug_mode": True});
```

```
Matrix carrier: nnz=12 neqs=6
    (col, row) = value
    (0, 0) = (1+0j)                    : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0 -> I,node=l1.
↪p1.i,f=0,fidx=0,Neq=1 mode=0
    (1, 1) = (1+0j)                    : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=l1.
↪p1.o,f=0,fidx=1,Neq=1 mode=0
    (1, 2) = (-1-0j)                   : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=m1.
↪p1.i,f=0,fidx=2,Neq=1 mode=0
    (2, 2) = (1+0j)                    : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p1.i,f=0,fidx=2,Neq=1 mode=0
    (2, 3) = (-0.9486832980505138-0j)  : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (2, 5) = -0.31622776601683794j     : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0
    (3, 0) = (-1-0j)                   : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=l1.
↪p1.i,f=0,fidx=0,Neq=1 mode=0
    (3, 3) = (1+0j)                    : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (4, 3) = -0.31622776601683794j     : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (4, 4) = (1+0j)                    : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p2.i,f=0,fidx=4,Neq=1 mode=0
    (4, 5) = (-0.9486832980505138+0j)  : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0
    (5, 5) = (1+0j)                    : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0


Matrix in dense format:

[[ 1. +0.j   0. +0.j   0. +0.j  -1. +0.j   0. +0.j   0. +0.j ]
 [ 0. +0.j   1. +0.j   0. +0.j   0. +0.j   0. +0.j   0. +0.j ]
 [ 0. +0.j  -1. +0.j   1. +0.j   0. +0.j   0. +0.j   0. +0.j ]
 [ 0. +0.j   0. +0.j  -0.9+0.j   1. +0.j   0. -0.3j  0. +0.j ]
 [ 0. +0.j   0. +0.j   0. +0.j   0. +0.j   1. +0.j   0. +0.j ]
 [ 0. +0.j   0. +0.j   0. -0.3j  0. +0.j  -0.9+0.j   1. +0.j ]]

Right hand side vector:

Vector carrier: neqs=6
    (row) = value
    (0) = (1.3416407864998738+0j) : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0
    (1) = (1.4142135623730951+0j) : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0
    (2) = (1.4142135623730951+0j) : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0
    (3) = (1.3416407864998738+0j) : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0
    (4) = 0j                      : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0
```

(continues on next page)

(continued from previous page)

```
    (5) = 0.447213595499958j      : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0
```

Note that you can have direct access to the `finesse.simulations.sparse.simulation.SparseMatrixSimulation` instance by using the context manager `finesse.model.Model.built()`, and thereby to the `finesse.simulations.sparse.KLU.KLUSolver` and eventually the `finesse.cymath.cmatrix.KLUMatrix` instance:

```
with model.built() as sim:
    carrier_solver = sim.carrier
    klu_mat = carrier_solver.M()
    klu_mat.print_matrix()
    klu_mat.print_rhs()
```

```
Matrix carrier: nnz=12 neqs=6
    (col, row) = value
    (0, 0) = (1+0j)                   : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0 -> I,node=l1.
↪p1.i,f=0,fidx=0,Neq=1 mode=0
    (1, 1) = (1+0j)                   : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=l1.
↪p1.o,f=0,fidx=1,Neq=1 mode=0
    (1, 2) = (-1-0j)                  : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0 -> I,node=m1.
↪p1.i,f=0,fidx=2,Neq=1 mode=0
    (2, 2) = (1+0j)                   : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p1.i,f=0,fidx=2,Neq=1 mode=0
    (2, 3) = (-0.9486832980505138-0j) : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (2, 5) = -0.31622776601683794j    : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0
    (3, 0) = (-1-0j)                  : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=l1.
↪p1.i,f=0,fidx=0,Neq=1 mode=0
    (3, 3) = (1+0j)                   : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (4, 3) = -0.31622776601683794j    : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p1.o,f=0,fidx=3,Neq=1 mode=0
    (4, 4) = (1+0j)                   : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p2.i,f=0,fidx=4,Neq=1 mode=0
    (4, 5) = (-0.9486832980505138+0j) : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0
    (5, 5) = (1+0j)                   : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0 -> I,node=m1.
↪p2.o,f=0,fidx=5,Neq=1 mode=0


Vector carrier: neqs=6
    (row) = value
    (0) = (1.3416407864998738+0j) : I,node=l1.p1.i,f=0,fidx=0,Neq=1 mode=0
    (1) = (1.4142135623730951+0j) : I,node=l1.p1.o,f=0,fidx=1,Neq=1 mode=0
    (2) = (1.4142135623730951+0j) : I,node=m1.p1.i,f=0,fidx=2,Neq=1 mode=0
    (3) = (1.3416407864998738+0j) : I,node=m1.p1.o,f=0,fidx=3,Neq=1 mode=0
    (4) = 0j                      : I,node=m1.p2.i,f=0,fidx=4,Neq=1 mode=0
    (5) = 0.447213595499958j      : I,node=m1.p2.o,f=0,fidx=5,Neq=1 mode=0
```

KLUMatrix

As seen above, the `KLUMatrix` class, implements methods to factor and solve sparse matrices by linking into KLU. KLU is a library included in SuiteSparse which stand for "Clark Kent" LU factorization algorithm (as opposed to "SuperLU").

This is also the source of the infamous "KLU_singular error", which can easily be replicated by passing a singular matrix:

```
from finesse.cymath.cmatrix import KLUMatrix

mat = KLUMatrix(name="singular_example")
In1 = mat.declare_equations(Neqs=1, index=0, name="I_In1", is_diagonal=True)
Out1 = mat.declare_equations(Neqs=1, index=1, name="I_Out1", is_diagonal=True)
a12 = mat.declare_submatrix_view(In1.from_idx, Out1.from_idx, "_In1->Out1", False)
a21 = mat.declare_submatrix_view(Out1.from_idx, In1.from_idx, "_Out1->In1", False)

mat.construct()

a12[:] = 1
a21[:] = 1

mat.factor()
```

```
---------------------------------------------------------------------------
Exception                                 Traceback (most recent call last)
Cell In[8], line 14
     10
     11 a12[:] = 1
     12 a21[:] = 1
     13
---> 14 mat.factor()

File src/finesse/cymath/cmatrix.pyx:1740, in finesse.cymath.cmatrix.KLUMatrix.factor()
-> 1740 'Could not get source, probably due dynamically evaluated source code.'

File src/finesse/cymath/cmatrix.pyx:1775, in finesse.cymath.cmatrix.KLUMatrix.factor()
-> 1775 'Could not get source, probably due dynamically evaluated source code.'

Exception: An error occurred in KLU during factoring: STATUS=1 (KLU_SINGULAR)
```

As you can see, at the core the matrix solving is rather straightforward. Most of the complexity of Finesse arises from keeping track of all the state: the components in the model, their changing parameters, the different frequencies and higher order modes and the connections between the components.

8.6.2 Simulation

The simulation class takes the Model (page 492) object and 'builds' a simulation from it. This includes the following:

- Beam Tracing functionality (this can be done without solving the matrix equation)
- Storing common simulation settings
- Checks is a signal simulation is required
- Checks which Parameters (page 492) are changing and update symbolic expressions
- Determines is the simulation is modal
- Keep track of the detector workspaces

The `BaseSimulation` class is found in **simulations/simulation.pyx** and the child class `SparseMatrixSimulation` in **simulations/sparse/simulation.pyx**. The separation is such that Finesse might support different types of solvers (like a graph-based solver) in the feature.

Most importantly, it builds the carrier Solver (page 492) and (if necessary) the signal Solver (page 492) objects. The simulation object is saved in the `finesse.analysis.actions.base.AnalysisState` object, which in turn gets passed to the `finesse.analysis.actions.base.Action._do` method of the actions.

The actions then usually call `state.sim.run_carrier` or `state.sim.run_signal`, depending on what the action needs for it's output.

You can manually create a `SparseMatrixSimulation` object using the `finesse.model.Model.built()` context manager:

Warning

Manually running simulations like this is only for demonstration the internals of FINESSE!

```
import numpy as np
import finesse


model = finesse.Model()

model.parse(
    """
    l l1 P=1
    s s1 l1.p1 m1.p1
    m m1 R=0.1 T=0.9
    """
)
with model.built() as sim:
    sim.run_carrier()
    for node, val in zip(
        sim.carrier.nodes.keys(), np.array(sim.carrier.M().rhs_view[0, :]),
        strict=True
    ):
        print(node, val)
```

```
l1.p1.i (0.447213595499958+0j)
l1.p1.o (1.4142135623730951+0j)
m1.p1.i (1.4142135623730951+0j)
m1.p1.o (0.447213595499958+0j)
m1.p2.i 0j
m1.p2.o 1.3416407864998738j
```

This example is loosely based on the `DCFields` (page 235) class.

8.6.3 Solver

The solver object couples the nodes (page 492) from the Model (page 492) to the underlying matrix solver. In principle it would support different kinds of matrix solvers, but currently only the `KLUMatrix` class is used.

The inheritance tree is `BaseSolver->HOMSolver->SparseSolver->KLUSolver`. The classes are implemented in **simulations** and **simulations/sparse**.

8.6.4 ModelElement

8.6.5 Detector

8.6.6 Port

8.6.7 Node

8.6.8 Parameters

8.6.9 Workspace

8.6.10 Model

8.6.11 Signal Matrix

`Click to download example as python script`

`Click to download example as Jupyter notebook`

8.7 Developer tools

The FINESSE repository and optional extra `lint` requirements define configurations for various helpful development tools. These tools help to maintain code quality in a collaborative environment. This section provides some additional information these tools and their configuration.

8.7.1 Tool configuration

Settings for most tools can be found in their own sections in **pyproject.toml**. One exception is `flake8` which doesn't support `pyproject.toml` and instead has its configuration in **tox.ini**.

8.8 Packaging Finesse

FINESSE is currently packaged for pip and conda. FINESSE requires some system dependencies in addition to those defined in the `setuptools` configuration. See Requirements for building, running and developing Finesse (page 481) for more information on FINESSE dependencies.

8.8.1 Conda

We expect that conda packages will be the main way that users get hold of FINESSE. The standard Anaconda channel does not contain all the packages we need. Instead we use Conda-Forge, which incidentally is also what IGWN as well as LIGO laboratory also uses to distribute and maintain their packages.

Conda-Forge packaging can be relatively straight forward however with the complicated FINESSE compilation it can throw various issues. All the Conda-Forge package management happens on GitHub. When making a new package you have to work with the staging repository first to get the package initially made and signed off. Once that is done a "feedstock" is made. The main FINESSE feedstock is also on Github. You must be made a maintainer to work with it and push new fixes.

The workflow with the feedstock is

- Make a new release tag and push it to the FINESSE
- Push a new source to PyPi
- The feedstock bot will see this sometime in the next day or so and a new pull-request will be made
- Hopefully, this pull-request will just work and all the tests will pass
- If new packages or build changes have been made the feedstock *recipe/meta.yaml* might need updating
- If there are new package requirements this must be reflected in the *meta.yaml* file

It is very important you read the Conda-Forge packaging instructions first before doing much, as there are various ways things should be done. Overall the process isn't too painful.

If a new PyPi package dependency is added to FINESSE we must also ensure that the same package is available in Conda-Forge. It will not use PyPi as a dependency. For simple python packages you can use Grayskull to quickly make a conda package and get it approved.

8.8.2 Wheels

Platform-specific wheels are built for most Linux platforms (so-called *manylinux* wheels) and macOS. This procedure is automated as part of the continuous integration (page 526) pipeline, but it is quite simple. The Red Hat based Docker image `manylinux2014` provided by PyPA is used. The only upstream dependency is currently `suitesparse-devel` which provides the KLU C headers used by FINESSE's simulation module.

Building for 'manylinux'

The procedure for building a `bdist_wheel` for the `manylinux` target is:

```
$ yum install -y suitesparse-devel
$ pip wheel . --no-deps -w wheelhouse
$ auditwheel repair wheelhouse/*.whl
```

These commands should be run from the source code root on the Docker image. The `/path/to/pip` parts should be replaced by the path to pip corresponding to the Python version being targeted.

Building for Mac OSX

The procedure to make a wheel for OSX is similar. The wheel must of course be built using OSX, and SuiteSparse must be installed by some means other than `yum` (e.g. with `brew install suite-sparse`). The tool for delocalising the produced wheel is also not `auditwheel`, but rather `delocate-wheel` provided by delocate:

```
$ pip wheel . --no-deps -w wheelhouse
$ delocate-wheel wheelhouse/*.whl
```

8.9 Releasing finesse

FINESSE is released on two channels, pypi and conda-forge.

Before releasing, you should always make sure the changelog (page 457) is up to date!

Since the conda-forge release is based on the pypi release, we will describe the PyPi release first.

8.9.1 PyPi

The PyPi release is configured via the gitlab pipelines **/.gitlab-ci.yml**, in the `deploy` stage. You can trigger a release at any time by creating a tag on any branch, but preferably we tag a commit on develop that has already passed the tests and following pep 440. We follow semantic versioning, e.g. `3.feature.patch`.

The tag should be accompanied by a short description of user-facing changes since the last release.

The `pypy` job currently pushes a source distribution (a compressed `tar.gz` archive) and a binary distributions (wheels) for manylinux and MacOS arm64.

The source distribution is important because it is the basis for the conda-forge release and a fallback for platforms that we do not build wheels for. Installing finesse from PyPi on these platforms will trigger a build of the cython modules, requiring Cython and Suitesparse to be available on the system. This is similar to a Developer installation (page 21).

The PyPi release is usually fairly straightforward, since binary wheels are being created in every pipeline, catching any issues early during development. As of 3.0a23 Windows wheels are not being published.

8.9.2 Conda-forge

The conda-forge release is in principle automated, but can be more difficult to debug when issues arise. Much of the process is also documented by conda forge.

The conda-forge feedstock repository is where the release process is configured. To able to release finesse, your github account needs to be added to the maintainer list. New maintainers can be added by opening an issue in the feedstock with the title `@conda-forge-admin, please add user @username`. This creates a pull request which can be merged by existing maintainers.

Almost all behaviour is configured in the meta.yaml , see conda forge docs.

Automatic releases

It runs a bot which watches the finesse PyPi repository and automatically creates a pull request when a new PyPi version is released. See an example pull request. The creation of the pull request triggers a series of builds on the conda-forge platform, building binary conda packages for Finesse on a variety of platforms and python versions. See an example run. This process is called 'rendering'.

When issues inevitably arise, you will need to fix the configuration by committing to the branch the bot created, and adding a comment in the PR with the text `@conda-forge-admin please rerender`. When the builds for all platforms are successful the pull request should be merged to finalize the release.

Manual releases

If you do not want to wait for the bot to create the pull request, you can do so yourself. You will need to fork the feedstock and make the following changes in the `meta.yaml` file:

- Reset the `build->number` to 0 (when not patching an existing release)
- Update the `source->sha256` to the hash of the corresponding source release on PyPi
- Check if the requirements defined in the `meta.yaml` still match those defined in the **pyproject.toml**
- See the current list of `source->patches` are functioning and whether any new patches are required.

Afterwards, you create a pull request of a branch in your fork into the original feedstock repository. You can then follow the flow described above.

Patches

Releasing for conda-forge unfortunately requires some patching of the finesse repository, mostly to the different files specifying versions. They are listed under `source->patches` and defined in the various `.patch` files in the `recipe` folder.

Patches failing usually results in a `hunk fail` in the logs of the rendering.

Updating requirements

The main source of problems is the different specifications for the package requirements. The `meta.yaml` file can conflict with the finesse `pyproject.toml`. For automatic releases the bot will perform a 'Dependency Analysis' that is meant to flag mismatches between the two specifications. Example issue

The requirements are spread between `build`, `host` and `run` sections. See the docs for a brief explanation.

Another possible source of failure is our dependencies themselves having misconfigured their conda feedstock, leading to version conflicts.

Chat channel

When getting stuck or confused, you can always try the asking questions in the chat channel

8.10 How to add a notebook to the documentation

So, you have a Jupyter notebook with beautiful Finesse examples, physics explanations, tutorials or anything that could improve the Finesse website, and you would like to add it to the documentation. Great! Here's how to do it.

8.10.1 Before starting, some info

Important

This guide is intended for people with developer access to the Finesse GitLab repository. If this is not the case, please contact us to see if we can take care of adding the page ourselves. You can also fork the Finesse repository and create your merge request from there. In all cases, please adapt the following workflow to your situation and needs.

Attention

I don't have a development environment for Finesse …

Contributing to development (page 449)

- Finesse uses Sphinx to generate its documentation. The MyST-NB extension enables Jupyter notebooks to be used as a documentation page. The notebooks need to be written in the MyST language, but no worries, MyST is a superset of CommonMark, the Markdown variant used in the Jupyter notebook markdown cells. This means that MyST-NB will understand everything that you would write in a standard Jupyter notebook, and more; no need to change your text or code. MyST also gives you access to new ways to display information that are not available with CommonMark. A quick syntax guide to MyST is provided at the end of this page in Annexe: Quick MyST reference (page 499).
- The notebook that generates this page is located in the Finesse source code directory at docs/source/developer/how-to-notebook.ipynb. You can make a copy to use as a template or look at the source code to understand how it works.
- To ensure quality and consistency of the documentation, your notebook will be reviewed before being added to the documentation.
- MyST-NB provides a plugin for JupyterLab that activates some MyST formatting. Search "jupyterlab-myst" in the extension manager (in the left sidebar or in the `View` menu of JupyterLab).

8.10.2 Step 0: Discuss your plan

Before starting the endeavour of creating a new page for the documentation, discuss with the Finesse team to see if, and where, your page could be added to the documentation, the scope of the document, and if we can help at any step of the process. To communicate with us, you can:

- Join The Finesse room on Matrix (preferred way)
- Send an email to `finesse-support AT nikhef.nl`

8.10.3 Step 1. Write your notebook

Tip

JupyterLab and the jupyterlab-myst extension are installed by default in the Finesse environment named `finesse3-env` (see: Contributing to development (page 449)).

1. Make sure your notebook is complete and correctly formatted:
   - Use sections and subsections, defined by headings of appropriate levels, to structure your document.
   - Add descriptions and explanations in Markdown cells.
   - Make sure your figures are correctly labelled and easy to read.
   - Make your code understandable, with comments and docstrings when relevant.
   - Remove any bugs in your code. The whole notebook should run without exception (or at least without uncaught exceptions).
2. Modify the notebook so that it is coherent and can be integrated with the rest of the documentation:
   - **Write a title**: If it is not already done, add a title to your page using the markdown notation "# title". That will be the name displayed in the table of contents and other references.
   - **Write a label**: To allow your document to be referenced, you need to add a label before the title (or before any sections that you want to be able to reference). Use a label that will be unique and descriptive

(the name of the file could be a valid option). To write a label, write something like `(my-label)=` in a Markdown cell, before the section title.

- **Add authors and date**: Write your name(s) as author(s) and the date. This will allow any questions to be directed to the right person and determine if the information may be outdated.
- **Add internal references**: Add/rewrite links to internal Finesse resources using references. see: Labels and references (page 499)
- **Replace the word "Finesse" with "{{Finesse}}"**. This will activate the standard Finesse word formatting.
- **If relevant, add download links**: If you want to allow the page to be downloaded as a notebook, you need to write in a markdown cell something like (write the proper notebook filename), ideally at the bottom of the page:

```
{jupyter-download-nb}`Download this page as Jupyter notebook <how-to-notebook>`
```

resulting in

`Download this page as Jupyter notebook`

- **Be careful with the size and runtime of your notebook**: For practical reasons, we cannot add notebooks that take too long to run (high maxtem for example) or are too big (plot with really high dpi for example).

8.10.4 Step 2. Create your git branch from an up-to-date development branch

Note

We assume the current setup

- You have access and know how to use a command line interface(CLI)
- You are either under Linux, MacOS, or Windows Subsystem for Linux (WSL)
- Your copy of the git repository is located at: `$HOME/finesse3`
- `git` and `mamba` software are accessible from your CLI
- Your mamba environment is named `finesse3-env`

Please adapt the instructions if this does not correspond to your situation.

1. **Move to the Finesse docs directory.** `cd $HOME/finesse3/docs`
2. **Update the `develop` branch.** `git switch develop && git pull`
3. **Activate the development environment.** `mamba activate finesse3-env`
4. (Optional - see the note below) **Build the documentation.** `make html`.

   You can consult the documentation by opening with a browser the file `$HOME/finesse3/docs/build/html/index.html`.
5. **Create and switch to a new branch.** The new branch should begin with "doc/". `git switch -c <doc/a-relevant-name-for-my-branch>`

You should now have:

- ✓ An active `finesse3-env` environment,
- ✓ A checked out new git branch named `doc/<a-relevant-name-for-my-branch>` based on the develop branch (use `git status` to know the active branch).
- ✓ Your working directory set to `$HOME/finesse3/docs/`.

- ✓ (Optional) An up-to-date version of the documentation (located in `$HOME/finesse3/docs/build/html`),

Note

The initial local build of the documentation can take several minutes to complete. The subsequent builds should be faster. You may need to build from a clean state (after a `make clean`) from time to time, when the build doesn't complete properly.

If you choose not to build the documentation locally, you can inspect your page after creating a merge request at step 4 inside the artefacts of the merge request pipeline on GitLab. Please note that the pipeline can take over 40 minutes to complete and that the documentation building process must be initiated manually.

More about the build process there: How to build the documentation (page 455).

8.10.5 Step 3. Add your notebook to the documentation

1. **Copy your notebook to its final destination in `$HOME/finesse3/docs/source/*`.** The directory hierarchy follows the same hierarchy as the documentation table of contents.
2. **Add your page to the table of contents.** Write the relative path to your notebook, without the extension (*i.e.* no `.ipynb`), in the relevant `index.rst` file. This will add your page to the table of contents of this section. If you don't add it to an index, your page will only be reachable via the search function.
3. (Optional) **Build the documentation.** Run `make html` and look at the results by opening with a browser the file `$HOME/finesse3/docs/build/html/index.html`.

You should now have:

- ✓ a version of the documentation that includes your new page.

8.10.6 Step 4. Create a merge request

1. **Add your document(s) to your `git` branch.** `git add path/to/my.ipynb` (don't forget also to add any images or files needed to generate your page).
2. **Commit your work.** `git commit`. Write a short commit message explaining the commit.
3. **Push your branch to GitLab.** `git push origin <doc/a-relevant-name-for-my-branch>`
4. **Create a new merge request targeting the develop branch.** If the push was successful, a link to create a new merge request will be present in the server response. You can also make a merge request via the GitLab website.
5. **Fill in the information for the merge request**.

You should now have

- ✓ an active merge request waiting for review.

Tip

- You can commit to your local branch as often as you want to track your changes, save your work or if you need to change branches.
- You can use `git commit -a` to automatically commit all modifications on tracked files without needing to use `git add` on each file at every commit. Be careful not to add unwanted modifications.
- Use `git status` to see which branch is active, the modified files, and more.
- Use `git diff` to see the modified lines since your last commit.

8.10.7 Step 5. Your merge request is reviewed

You can find and consult your merge request at Finesse merge requests. You should also receive emails about comments and modifications of your merge request.

We may ask that you correct some things on your page. If so

1. **Make sure your branch is active.** `git switch <doc/a-relevant-name-for-my-branch>`
2. **Modify your file.**
3. (Optional) **Build the documentation and inspect the result.** `make html` and open with a browser `$HOME/finesse3/docs/build/html/index.html`
4. **Commit your changes.** `git commit -a`. Write a short commit message explaining the commit.
5. **Push the modification.** `git push origin <doc/a-relevant-name-for-my-branch>`
6. **Answer the threads on your merge request.** All threads (merge request discussions/questions/corrections) need to be resolved before merging.

8.10.8 Step 6. Your page is accepted and merged

✓ Done. **Thank you for contributing to the Finesse documentation!**

8.10.9 Annexe: Quick MyST reference

In this section, we quickly present some features that you can utilise with MyST. The intended use is a quick reference when writing. Some examples are extracted from the JupyterBook cheatsheet.

For more information about writing your document with MyST consults the MyST documentation

Headers

```
# Heading level 1
## Heading level 2
### Heading level 3
#### Heading level 4
##### Heading level 5
###### Heading level 6
```

Labels and references

Label sections with `(target_label)=` before the header to enable referencing them.

Reference with:

Syntax	Result
`{ref}`sec:labels-and-references``	Labels and references (page 499)
`{ref}`Custom text <sec:labels-and-references>``	Custom text (page 499)

Additional cross-reference support

Examples

```
- Module: {mod}`finesse.components`
- Class: {class}`finesse.components.cavity.Cavity`
- Function: {func}`finesse.model.Model.path`
- Attribute: {attr}`finesse.model.Model.detectors`
- kat-script command: {kat:command}`link`
- kat-script element: {kat:element}`mirror`
- kat-script analysis: {kat:analysis}`xaxis`
- GitLab issue: {issue}`601`
- Source file: {source}`docs/source/developer/how-to-notebook.ipynb`
```

Results

- Module: `finesse.components`
- Class: `finesse.components.cavity.Cavity`
- Function: `finesse.model.Model.path()`
- Attribute: `finesse.model.Model.detectors`
- kat-script command: `link` (page 196)
- kat-script element: `mirror` (page 142)
- kat-script analysis: `xaxis` (page 209)
- GitLab issue: **#601**
- Source file: **docs/source/developer/how-to-notebook.ipynb**

Citations

Important

Add the publication that you want to cite, if new, in the .bib file docs/source/zbiblio/refs.bib.

Example	Result
`A good paper to read {cite}`amaldi03_freise`.`	A good paper to read [40].

Horizontal lines

Write a horizontal line with `---`.

Links

Example	Result
`[Finesse 3](https://finesse.ifosim.org)`	Finesse 3
`<https://finesse.ifosim.org/>`	https://finesse.ifosim.org/

Tables

More info: MyST documentation - Tables

Example

```
|    Training    |   Validation    |
| :------------- | --------------: |
|        0       |        5        |
|     13720      |      2744       |
```

```
```{list-table} This table title
:header-rows: 1
:name: example-table

* - Training
  - Validation
* - 0
  - 5
* - 13720
  - 2744
```
```

Result

Training	Validation
0	5
13720	2744

Table 8.3: This table title

Training	Validation
0	5
13720	2744

Figures and images

More info: Myst documentation - Images and figures

Example

```
```{figure} ../images/andreas_daniel_left_handed_small.jpg
:alt: Left-handed coordinate system
:height: 150px
:name: andreas_daniel_left_handed
:target: coordinate-systems
:align: center

Andreas and Daniel showing left-handed coordinate systems.
```
```

Result



Fig. 8.1: Andreas and Daniel showing left-handed coordinate systems.

Mathematical equations

Syntax	Example	Result
Inline	`This is an example of an inline equation $z=\sqrt{x^2+y^2}$.`	This is an example of an inline equation $z = \sqrt{x^2 + y^2}$.

Math blocks	`This is an example of a math block`
	
	`$$`
	`z=\sqrt{x^2+y^2}`
	`$$`
		This is an example of a math block $$z = \sqrt{x^2 + y^2}$$

Math blocks with labels	`This is an example of a math block with a label`
	
	`$$`
	`z=\sqrt{x^2+y^2}`
	`$$ (mylabel)`
		This is an example of a math block with a label $$z = \sqrt{x^2 + y^2} \tag{8.4}$$

Math directives	`This is an example of a math directive with a label`
	```` ```{math} ````
	`:label: eq-label`
	
	`z=\sqrt{x^2+y^2}`
	```` ``` ````
		This is an example of a math directive with a label $$z = \sqrt{x^2 + y^2} \tag{8.5}$$

Referencing math directives

Example	Result
`Check out equation {eq}`eq-label`.`	Check out equation (8.5).

Executing code

What is in a `Code` cell will be executed, and any output displayed.

```
# This is an example of some code


def do_some_calculation(a: float) -> float:
    """Using typing and docstrings helps readers to understand what
    your code does."""
    return a * a * a


n = 12
print(f"The cube of {n} is {do_some_calculation(n)}.")
```

```
The cube of 12 is 1728.
```

Tip

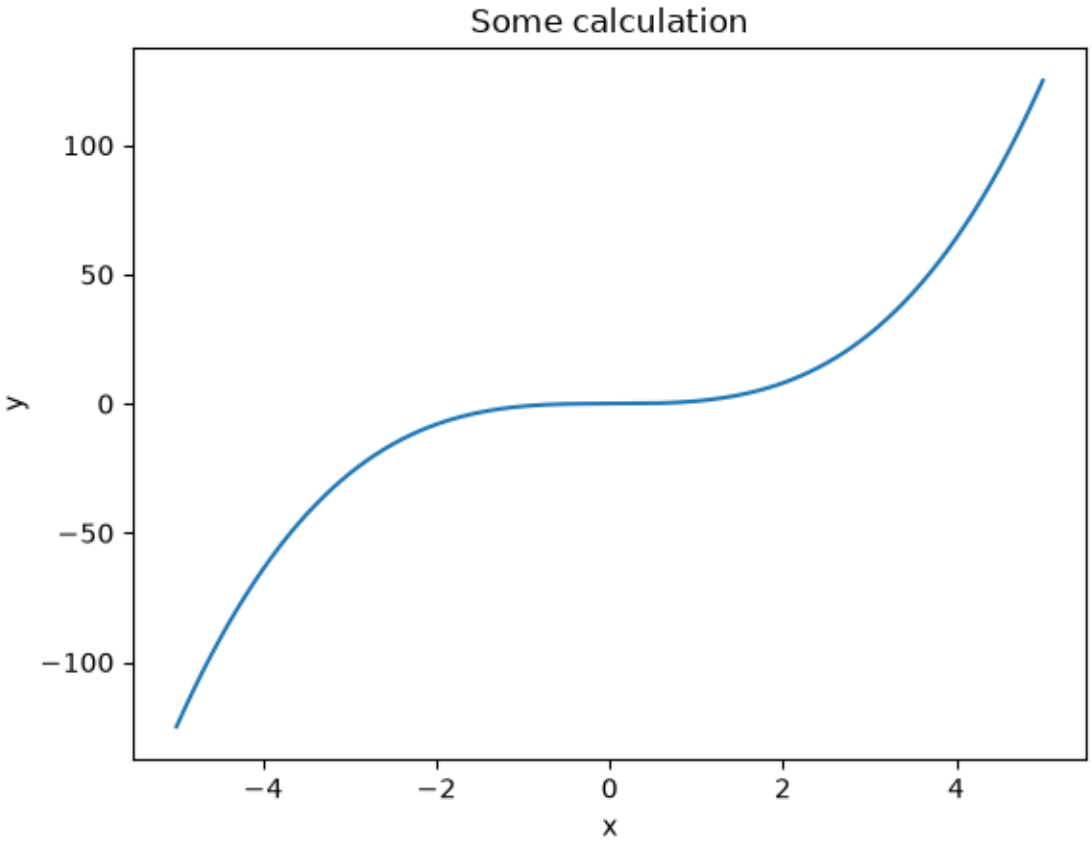
All the code cells of a page are run in the same Python instance, one after the other, meaning that you can use previous code in the following cells.

```
# Plotting example

import matplotlib.pyplot as plt
import numpy as np

fig, ax = plt.subplots()
x = np.linspace(-5, 5, 201)
y = do_some_calculation(x)
ax.plot(x, y)
ax.set_xlabel("x")
ax.set_ylabel("y")
ax.set_title("Some calculation")
# The ';' will prevent the display of the object returned by the function
```

```
Text(0.5, 1.0, 'Some calculation')
```



Admonitions

Example	Result
```` ```{note} This is an example of a note. ``` ````	**Note** This is an example of a note.

Admonitions available

Note

…

Warning

…

Tip

…

Caution

…

Attention

…

Danger

…

Hint

…

Tip

…

Important

...

Code

In-line code

Example:

```
Wrap in-line code blocks in backticks: `boolean example = true;`.
```

Result:

Wrap in-line code blocks in backticks: `boolean example = true;`.

Code and syntax highlighting

Example:

````
```python
note = "Python syntax highlighting"
print(node)
```
````

Result:

```python
note = "Python syntax highlighting"
print(node)
```

Lists

Ordered list

Example	Result

`1. First item`
`2. Second item`
`    1. First sub-item`
	1. First item
	2. Second item
	    1. First sub-item

`1. First item`
`2. Second item`
`    * First sub-item`
	1. First item
	2. Second item
	    • First subitem

Unordered list

Example	Result

`* First item`
`* Second item`
`  * First subitem`
	• First item
	• Second item
	  – First subitem

`* First item`
`  1. First subitem`
`  2. Second subitem`
	• First item
	  1. First subitem
	  2. Second subitem

Footnotes

Example

```
This is an example of a footnote.[^footnote1]

[^footnote1]: The definition for referencing footnotes is generally placed at the bottom of␣
↪the document.
```

Result

This is an example of a footnote.[1]

8.11 Contributing to the editor extensions

The various editor extensions have different structures and packaging requirements. Where applicable, instructions for contributing can be found in each project's README. Specifics for packaging and distribution can be found here.

Table of Contents

- JupyterLab (page 506)
- Visual Studio Code (page 507)
- Vim (page 508)
- Pygments (page 508)

8.11.1 JupyterLab

Repository: https://gitlab.com/ifosim/finesse/katscript-jupyterlab

The JupyterLab extension is distributed in two places; via pip and npm. New versions should be published at both locations. You need to have nodejs installed to work with the extension.

pip

First, make sure you have an account on https://pypi.org. An existing collaborator can then invite you to be a maintainer, by going to the project's collaboration page and entering your username.

To publish an update, run the following from the repository:

[1] The definition for referencing footnotes is generally placed at the bottom of the document.

```
$ pip wheel . -w dist
```

This will create a folder named `dist` and build a wheel of the extension there. You can then publish this to PyPI with twine:

```
$ twine upload dist/katscript_jupyterlab-*.whl
```

After following the prompt to login, your update will be published.

npm

The `npm` package is published under the gwoptics organisation. An existing collaborator can invite you to join the organisation from the members page, either by email or username. Either way, you'll need to create an account on https://www.npmjs.com.

To publish an update, you'll first need to log in to `npm` from the command-line:

```
$ npm login
```

Next, from the repository, run:

```
$ npm run build
$ npm publish
```

Your update should then be immediately available from the JupyterLab extension manager.

> **Warning**
>
> Files published to https://npmjs.com are **permanent**; you cannot unpublish a package more than 24 hours after it was published, so take care.

8.11.2 Visual Studio Code

Repository: https://gitlab.com/ifosim/finesse/katscript-vscode

The Visual Studio Code extension is published in two places: the Visual Studio Marketplace, which is used by Visual Studio Code, and the open-source Open VSX, which is used by VSCodium and other editors. New versions should be published at both locations. To publish at either location, you'll need to have nodejs installed. The initial set-up for both locations is fairly involved.

Visual Studio Marketplace

The Visual Studio Marketplace extension is published under the gwoptics publisher, which is tied to the corresponding gwoptics Azure DevOps organisation. First, an existing member must invite you to the Azure organisation from the organisation settings page. You then need to obtain a Personal Access Token, as described in the extension guide.

Next, you need to be invited to the gwoptics publisher, which an existing member can do from the publisher management page. An "Invalid Domain" error may occur when adding a new user, in which case the existing member will have to contact Microsoft support from the "Contact Microsoft" link in the top-right. They will need the ID of the new user, which can be obtained as described in this stackoverflow answer, so you may as well send that as part of the request for help.

To actually publish an update, follow the steps in the extension guide. You'll need your Personal Access Token to log in. These basically amount to running the following from the repository:

```
$ npx vsce login gwoptics
$ npx vsce publish
```

Open VSX

The Open VSX extension is again under the gwoptics namespace. The following steps are the same as those in the publishing guide.

First, create a GitHub account if you don't have one. Next, create an Eclipse account, filling in the *GitHub Username* field with the correct name. You can then log in via https://open-vsx.org/ using this GitHub account. From your profile page, click *Log in with Eclipse* to authorize Eclipse to access your account. Next, click the newly-added *Show Publisher Agreement* button on your profile page, and click *Agree* if you do. An existing member can then add your username to the namespace from their namespaces settings page.

Once you have an account, you'll have to generate an Access Token from your settings page. You can then publish the extension by running the following from the repository, replacing `<token>` with your Access Token:

```
$ npx ovsx publish -p <token>
```

8.11.3 Vim

Repository: https://gitlab.com/ifosim/finesse/katscript-vim

The Vim extension can be installed in the usual ways, such as with vim-plug or Vim's built-in package management. No packaging work is required, as it's installed straight from the git repository.

8.11.4 Pygments

The Pygments extension is installed along with FINESSE by defining the `pygments.lexers` entry point in **setup.py**, so no further packaging work needs to be done. The actual extension is defined in **script/highlighter.py**, and is described there.

8.12 Finesse code style

In general, try to follow Flake8 and Black code styles.

8.12.1 Specific notes

'__init__.py' files

To avoid Flake8 flagging imports in `__init__.py` files, the imports intended to be available to other modules importing the package via `from package import *` should be explicitly listed in `__all__` as *strings*.

Absolute and relative imports

Absolute imports like `from finesse.script import parse` are preferred in FINESSE over relative imports like `from .script import parse`. Absolute imports are clear, straightforward, easy to track and will be valid even if the importing file moves. This is the general recommendation, including by PEP8. Only use relative import for long import statement.

Indentation of multi-line quotes

The Black formatter treats multi-line quotes (i.e. `"""`) as parentheses. This occasionally leads to seemingly unintuitive reformatting, such as when embedding kat script inside Python. For example, this code:

```
parse("""
    # some KatScript
""")
```

gets reformatted to:

```
parse(
    """
    # some KatScript
"""
)
```

In this regard, Black is being pretty reasonable. It treats multi-line quotes the same way it treats parentheses that enclose text that is too long for one line, by moving the contents onto the next line and indenting by one more level. It cannot, however, assume that it can reformat the contents of the multi-line string itself, so it leaves that alone. The result is that the opening multi-line quotes get indented by one, but the contents and the closing multi-line quotes get kept as they are.

To avoid this somewhat ugly reformatting, the solution to this is to manually indent the contents and closing multi-line quotes as well, if appropriate. Since KatScript ignores indents, the following would work for the above example:

```
parse(
    """
    # some KatScript
    """
)
```

Singletons

The direct use of singletons (single-instance classes) in Finesse is discouraged because they create and modify global state in a way that is difficult to automatically reset. Improperly reset global state can cause problems for unit testing since tests may implicitly rely on a certain default global state. One solution would be for each test that touches code containing singletons to fully reset any global state modified by such singletons, but this is difficult because each test must know about and how to reset every one that was used. Where global state is not fully reset by a test, a seemingly unrelated test may fail elsewhere because it implicitly relied on a particular default state. Improperly reset state can also lead to so-called *flaky* tests, i.e. tests that sometimes but don't always fail, depending on the order in which tests are run.

The single-instance class pattern is nevertheless sometimes useful and can still be employed using a safer mechanism. Some Finesse objects makes use of a single, global "datastore" that stores references to single instances of what would otherwise have been singletons. This single datastore is then reset between unit tests by a single tear-down function shared by all tests, ensuring the state is returned to the default for the next test. Two examples of this are *_FinesseConfig* and *_TracebackHandler*.

An alternative approach could be to use so-called *dependency injection*, but this requires a little more boilerplate code to set up and maintain so may be overkill depending on the application. Where reasonable, re-use of the "datastore" concept above is recommended since the infrastructure for resetting state between tests is already implemented.

8.13 Testing

8.13.1 Running and writing tests

Source code: tests

Finesse contains a large suite of tests configured to use Pytest. These are run automatically when changes are made to the code and pushed to the git repository, but can also be run manually with the console command `pytest`. Pytest is installed automatically as part of the development dependencies (page 449) for Finesse.

Pytest

Pytest is a framework for writing tests and provides a number of useful features over the standard library's `unittest`. Pytest strongly encourages the use of "fixtures" with tests, which are functions which set up test resources to be provided to tests themselves. Fixtures are often used in Finesse testing to set up models, which individual test functions then check.

Refer to the pytest documentation for more information.

Running the tests provided with Finesse

Pytest provides a command-line interface (CLI) for executing tests. To run all of the tests, simply run Pytest (you must have installed Finesse in developer mode (page 21)):

```
$ pytest
```

Note

Pytest is configured in **pyproject.toml** to run tests in the **tests** directory by default.

Pytest's CLI can run individual or combinations of tests by specifying specific subdirectories, modules or even particular functions with particular inputs (see Test IDs (page 511)).

Displaying output and logs during tests

Output from `print` statements is normally ignored by pytest unless a particular test fails. It can be forced to collect and display this output as it occurs by passing the `-s` flag.

Log display is ignored for all tests, even those that fail, unless the `--log-level` argument to the `pytest` command is set to the desired log level (e.g. `--log-level=debug`). As with standard output, though, this only displays log messages for tests that fail. To display live log messages, set `--log-cli-level` instead.

Types of test

Finesse shuns the traditional labelling of tests as "unit" or "integration" tests. Instead there are two broad categories of test in the Finesse test directory: *functional* and *validation*. These are described in the following sections.

Functional tests

These tests check the behaviour of individual parts of the Finesse code, such as functions, classes and modules. They do not check the physical predictions of Finesse, just that the building blocks are doing what they are supposed to do.

Functional tests may check the behaviour of atomic pieces of of code like functions, methods, classes, etc. (usually called *unit* tests) or check behaviour of multiple units together (usually called *integration* tests). These types of test sometimes use mock objects (page 511) to mimic the behaviour of context-dependent parts of the system in which the code runs, such as the network, user input, or databases.

Validation tests

Validation tests are to test the correctness of the high level outputs from Finesse, such as its predicted interferometer behaviour. It may be useful to use validation tests to e.g. compare scripts using Finesse to analytical models, or to compare the results of two separate Finesse scripts, or to other simulation tools.

The validation test directory also contains IPython notebooks. These define more complex validation tests which check behaviour against analytical models, and can be run manually to verify expected behaviour.

Writing tests

Before starting, it is useful to take a look at existing tests and the Pytest documentation to understand how to write good (and avoid writing bad) tests. More guidance for particular features of Pytest used in the Finesse test suite is given in the following sections.

Fixtures

A powerful feature of Pytest is so-called *fixtures* which allow you to define reusable code to test in test functions. Fixtures are by default generated for every test input, but can also be scoped for whole modules or packages if they are expensive to run. Fixtures can be inherited too, allowing fixtures to build upon other fixtures.

Mock objects

In order to test the functionality of a single unit, it is sometimes necessary to use mock objects to mimic the behaviour of other functions/methods/classes required for the operation of the function under test. Pytest provides tools for managing mock objects in the form of the monkeypatch fixture. This allows you to change attributes or fields of Python objects passed to each test, with the old state being restored automatically after the test passes. This can be used for example to patch the return values of functions used in tests, so that you can quickly create a particular program state for the test to check without including lots of setup code.

Parameterising tests

Pytest provides a parametrize decorator that lets you define sets of values to pass to the test function across multiple calls. This lets you quickly perform the same test with multiple inputs. Furthermore, when you define multiple parametrize decorators, Pytest automatically computes and runs all possible combinations of inputs, allowing you to concisely define dozens or hundreds of inputs.

Property-based testing

Sometimes it's useful not to hard-code specific inputs for tests, but to define what *types* of input the test should allow, and let some external code generate the inputs. The external code can then generate example inputs, run the test, and try to find edge cases that fail. This is called property-based testing or "fuzzing", and Finesse uses the Hypothesis library to provide this functionality.

Hypothesis provides a set of standard inputs such as integers, floats, lists, etc., and the ability to define custom types. A set of custom types for Finesse tests can be found in **/tests/testutils/fuzzing.py**.

Hypothesis will run for a fixed time per test before giving up. The default is set low enough such that running the whole test suite does not take too long, but to increase the time (and thus the number of inputs hypothesis tests), specify the `--hypothesis-max-examples` argument to `pytest`.

Test IDs

See also

The Selecting tests based on their node ID section of the Pytest documumentation

Particular tests can be run using the pytest CLI:

```
$ pytest /path/to/test.py::test_function
```

If the test is parametrised, you can even run a particular input to the test by referring to its parameters:

```
# Assume test_function is parametrised with a "cos(pi)" input...
$ pytest /path/to/test.py::test_function[cos(pi)]
```

Tests with multiple parametrised inputs can generate long and ugly test IDs in the CLI. These are especially true when test inputs are multiline strings. Test IDs can be assigned to input parameters using *pytest.param()* and assigning the `id` argument to a descriptive string:

```
@pytest.mark.parametrize(
    "input_a,input_b",
    (
```

(continues on next page)

(continued from previous page)

```
        pytest.param("a", "b", id="a-and-b"),  # The ID shouldn't contain whitespace.
    )
)
def test_function(input_a, input_b):
    ...
```

This string is then shown in the CLI by the test runner if that particular test input fails, and can be referred to to run only that particular test:

```
$ pytest /path/to/test.py::test_function[a-and-b]
```

Test markers

Tests of a particular category can be given "markers", which allows them to be included or excluded from a particular test suite. Some tests, such as those that use hypothesis (page 511), are automatically marked.

Available markers can be listed with `pytest --markers`

Tests with a particular marker can be run or *not* run using e.g.:

```
# Run hypothesis tests.
$ pytest -m hypothesis
# Don't run hypothesis tests (run everything else).
$ pytest -m "not hypothesis"
```

8.13.2 Test suite guide

Plane-wave radiation pressure effects tests

This section contains derivations of analytical formulas for radiation pressure induced modulation of a plane-wave laser beam, in various setups. Test cases simulating these setups can be found in the file `plane_wave/test_radiation_pressure.py`, in the FINESSE validation tests folder.

- Mirrors (page 512)
  - Single-Sided Free Mass (page 512)
  - Double-Sided Free Mass, One Modulation (page 515)
  - Double-Sided Free Mass, Two Modulations (page 517)
- Beamsplitters (page 519)
  - Single-Sided Free Mass (page 519)
  - Double-Sided Free Mass, One Modulation (page 521)
  - Double-Sided Free Mass, Two Modulations (page 523)

Mirrors

Single-Sided Free Mass

Analytics

We start with an amplitude modulated input field, such as produced by connecting a `SignalGenerator` (page 169) to a `Laser` (page 165)'s `amp` port in FINESSE,

$$E_\mathrm{i} = E_0 \cos(\omega_0 t) \left(1 - \frac{A}{2}\left(1 - \cos(\Omega t)\right)\right) \tag{8.6}$$

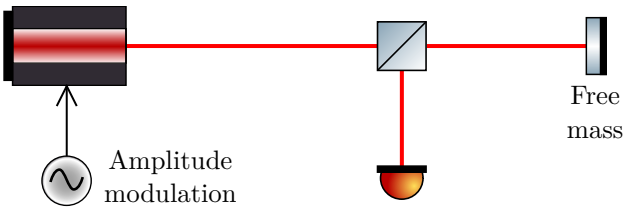


Fig. 8.2: Setup for the first test case, an amplitude modulated laser with two sidebands reflecting off a free mass.

where A is the modulation index. Writing this in the complex notation used in FINESSE gives

$$\begin{aligned} E_\mathrm{i} &= E_0 e^{i\omega_0 t}\left(1 - \frac{A}{2}\left(1 - \cos(\Omega t)\right)\right) \\ &= E_0 e^{i\omega_0 t}\left(1 - \frac{A}{2} + \frac{A}{4}\left(e^{i\Omega t} + e^{-i\Omega t}\right)\right) \end{aligned}$$

We can then separate this into three fields; the carrier field a_0, and the upper and lower sidebands $a_\pm$, such that

$$E_\mathrm{i} = a_0 + a_+ + a_-, \tag{8.7}$$

where

$$\begin{aligned} a_0 &= \left(1 - \frac{A}{2}\right) E_0 e^{i\omega_0 t} \\ a_+ &= \frac{A}{4} E_0 e^{i(\omega_0+\Omega)t} \\ a_- &= \frac{A}{4} E_0 e^{i(\omega_0-\Omega)t}. \end{aligned}$$

Next, we consider this field when incident on a free-floating mirror. For a free mass, we have

$$F(t) = m\ddot{x}(t).$$

Taking the Fourier transform, we see that

$$\begin{aligned} \tilde{F}(\omega) &= -m\omega^2 \tilde{x}(\omega) \\ \therefore \tilde{\chi}(\omega) \equiv \frac{\tilde{x}(\omega)}{\tilde{F}(\omega)} &= -\frac{1}{m\omega^2}. \end{aligned}$$

For a force of the form $F_0 \cos(\Omega t)$, we have

$$\tilde{F}(\omega) = \pi F_0(\delta(\omega + \Omega) + \delta(\omega - \Omega)),$$

from which we can find $x(t)$:

$$\begin{aligned} x(t) &= \frac{1}{2\pi}\int_{-\infty}^{+\infty} \tilde{\chi}(\omega)\tilde{F}(\omega)e^{i\omega t}\mathrm{d}\omega \\ &= -\frac{\pi F_0}{2\pi}\int_{-\infty}^{+\infty} \frac{1}{m\omega^2} e^{i\omega t}(\delta(\omega+\Omega) + \delta(\omega-\Omega))\mathrm{d}\omega \\ &= -\frac{F_0}{m\Omega^2}\frac{e^{i\Omega t} + e^{-i\Omega t}}{2} \\ &= -\frac{1}{m\Omega^2} F_0 \cos(\Omega t). \end{aligned}$$

A force of this form arises from the beating of the two amplitude modulation sidebands with the carrier. The force due to radiation pressure of light interacting with a mirror is given by

$$F_\mathrm{rad} = \cos\alpha \frac{P_\mathrm{1i} + P_\mathrm{1o} - P_\mathrm{2i} - P_\mathrm{2o}}{c}, \tag{8.8}$$

where $P_{\{1,2\}\{\mathrm{i,o}\}}$ is the power of the light on the negative and positive x sides of the mirror incoming and outgoing respectively, and α is the angle of incidence. In this simple case, $\alpha = 0$, and there is no incoming beam from the positive x direction. The force can then be calculated to leading order in A from Equation (8.6) as

$$\begin{aligned} F_{\mathrm{rad}} &= \frac{|E_{\mathrm{i}}|^2 + |E_{\mathrm{r}}|^2 - |E_{\mathrm{t}}|^2}{c} \\ &= \frac{(1 + R - T)}{c}|E_{\mathrm{i}}|^2 \\ &= \frac{(1 + R - T)}{c}\left({E_0}^2(1 - A) + A{E_0}^2\cos(\Omega t)\right), \end{aligned}$$

where R is the coefficient of reflectivity of the mirror. Here we are uninterested in the D.C. term as this will not contribute to the mirror's oscillation, and so have

$$\begin{aligned} F_0 &= \frac{(1 + R - T)A{E_0}^2}{c}, \\ \therefore x(t) &= -\frac{(1 + R - T)A{E_0}^2}{mc\Omega^2}\cos(\Omega t). \end{aligned}$$

Fig. 8.3 shows how an incoming field changes upon reflection from the mirror.

$$\begin{aligned} E'_{\mathrm{i}} &= E_{\mathrm{i}}e^{-i\phi} \\ E'_{\mathrm{r}} &= rE'_{\mathrm{i}} \\ E_{\mathrm{r}} &= E'_{\mathrm{r}}e^{-i\phi} \end{aligned}$$

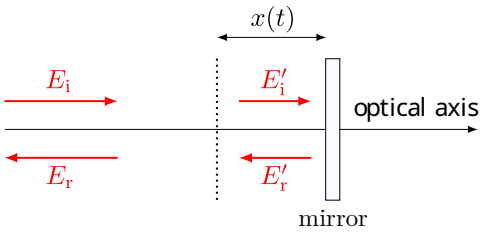


Fig. 8.3: Fields at a mirror's surface.

The incident field will be multiplied by the reflection coefficient of the mirror, r. The reflected light will also be phase shifted due to propagation through space by a total amount $e^{-2i\phi}$, where

$$\begin{aligned} \phi &= \frac{2\pi x(t)}{\lambda} \\ &= -\frac{2\pi(1 + R - T)A{E_0}^2}{\lambda mc\Omega^2}\cos(\Omega t) \\ &= -\frac{(1 + R - T)A{E_0}^2\omega_0}{mc^2\Omega^2}\cos(\Omega t). \end{aligned}$$

This results in the incoming beam being phase modulated by the mirror motion,

$$E_{\mathrm{r}} = rE_{\mathrm{i}}\exp(iB\cos(\Omega t)),$$

where

$$B = \frac{2(1 + R - T)A{E_0}^2\omega_0}{mc^2\Omega^2},$$

which to first order gives

$$\begin{aligned} E_{\mathrm{r}} &= rE_{\mathrm{i}}\left(1 + i\frac{B}{2}\left(e^{i\Omega t} + e^{-i\Omega t}\right)\right) \\ &= r(b_0 + b_+ + b_-), \end{aligned}$$

where

$$\begin{aligned} b_0 &= E_{\mathrm{i}} \\ b_+ &= i\frac{B}{2}E_{\mathrm{i}}e^{i\Omega t} \\ b_- &= i\frac{B}{2}E_{\mathrm{i}}e^{-i\Omega t}. \end{aligned}$$

Putting Equation (8.7) into this gives

$$b_0 = a_0 + a_+ + a_-$$
$$b_+ = i\frac{B}{2}E_0 e^{i(\omega_0+\Omega)t}$$
$$b_- = i\frac{B}{2}E_0 e^{i(\omega_0-\Omega)t},$$

to leading order, where we can once again see an input field and an upper and lower sideband. The full expression for the reflected field can then be written as

$$\begin{aligned} E_\mathrm{r} &= r(a_0 + a_+ + a_- + b_+ + b_-) \\ &= rE_0 e^{i\omega_0 t}\left(1 + \left(\frac{A+2iB}{4}\right)\left(e^{i\Omega t} + e^{-i\Omega t}\right)\right) \\ &= rE_0 e^{i\omega_0 t}\left(1 + \left(\frac{A}{4} + i\frac{(1+R-T)A{E_0}^2\omega_0}{mc^2\Omega^2}\right)\left(e^{i\Omega t} + e^{-i\Omega t}\right)\right) \end{aligned}$$

Finesse

The analytics above are compared to the sidebands produced by the following FINESSE file:

```
l l1 P=3
s s1 l1.p1 m1.p1 L=5
m m1 R=0.5 T=0.4
free_mass m1_sus m1.mech 0.5e-3

ad upper s1.p1.o fsig.f
ad lower s1.p1.o -fsig.f

sgen sig l1.amp.i 0.1
fsig(0.1)

xaxis(sig.f, log, 0.1, 1e7, 400)
```

The results are shown in Fig. 8.4.

Double-Sided Free Mass, One Modulation

Analytics

This is similar to the previous test, however there are no amplitude modulation sidebands present in the input field. As a result, we only expect to see the output containing the phase modulation sidebands. The second laser will not contribute to the oscillation of the mirror, as it is a purely D.C. force. The sidebands will also be $180°$ out of phase, as the mirror's motion is inverted when looking from the other side. In addition, we assume that the mirror is opaque i.e. $T = 0$. For a given input field

$$E_\mathrm{i} = E_1 \cos(\omega_0 t),$$

we therefore have

$$E_\mathrm{r} = rE_1\left(1 - i\frac{(1+R)A{E_0}^2\omega_0}{mc^2\Omega^2}\left(e^{i\Omega t} + e^{-i\Omega t}\right)\right).$$

Finesse

The analytics above are compared to the sidebands produced by the following FINESSE file:

```
l l1 P=3
s s1 l1.p1 m1.p1 L=5
```

(continues on next page)

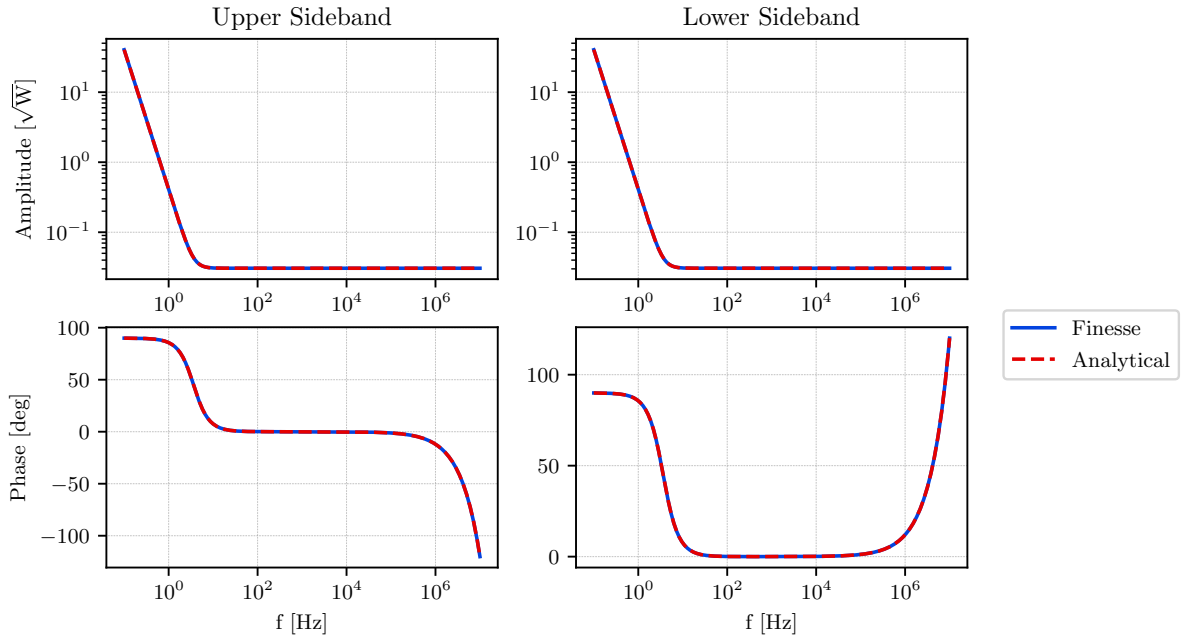


Fig. 8.4: Comparison of the upper and lower output sidebands from FINESSE with analytics.

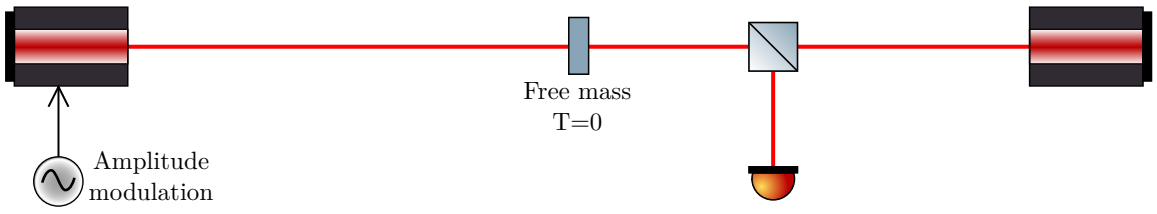


Fig. 8.5: Setup for the second test case, an amplitude modulated laser with two sidebands reflecting off a free mass causing sidebands to be generated in an unmodulated laser incident on the other side.

(continued from previous page)

```
m m1 R=1 T=0
s s2 m1.p2 l2.p1 L=5
l l2 P=2

free_mass m1_sus m1.mech 0.5e-3

ad upper s2.p2.o fsig.f
ad lower s2.p2.o -fsig.f

sgen sig l1.amp.i 0.1
fsig(0.1)

xaxis(sig.f, log, 0.1, 1e7, 400)
```

The results are shown in Fig. 8.6.

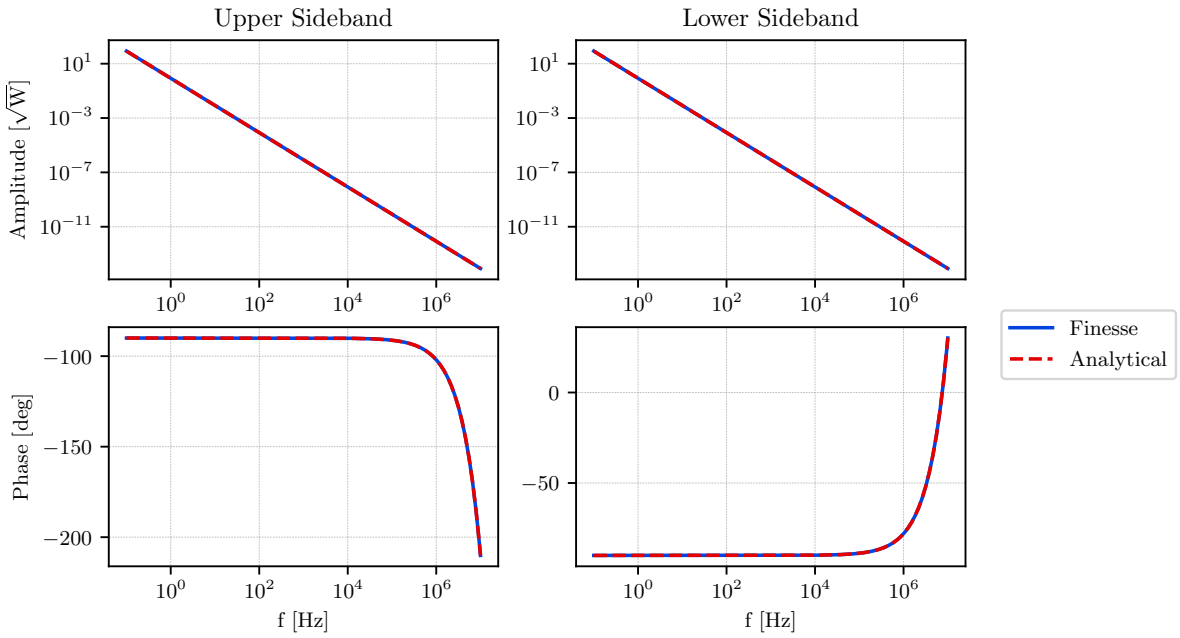


Fig. 8.6: Comparison of the upper and lower output sidebands from FINESSE with analytics.

Double-Sided Free Mass, Two Modulations

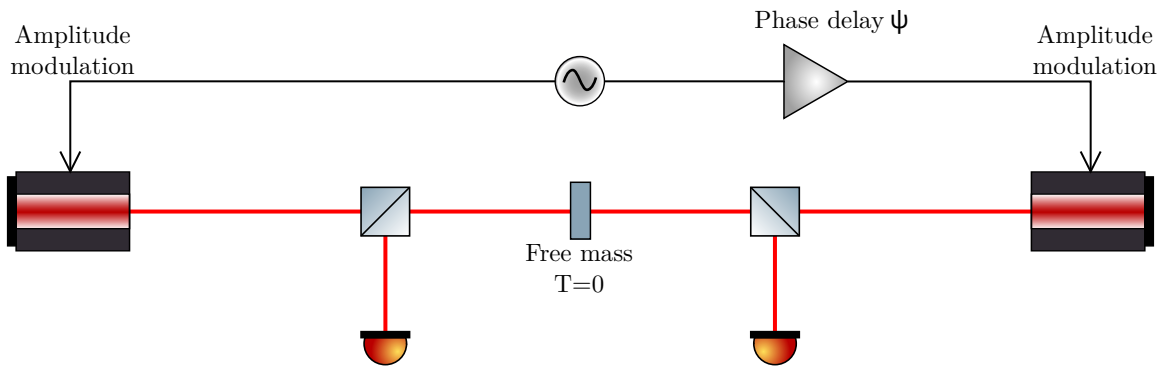


Fig. 8.7: Setup for the third test case, two amplitude modulated lasers with two sidebands each incident on each side of a mirror, with some arbitrary phase difference.

Analytics

As the second beam is now also amplitude modulated, it will contribute to the mirror's motion, so we must recalculate the radiation pressure force from Equation (8.8). For the left and right input fields,

$$E_{\text{Left}} = E_0 \cos(\omega_0 t)\left(1 - \frac{A_0}{2}\left(1 - \cos(\Omega t)\right)\right)$$
$$E_{\text{Right}} = E_1 \cos(\omega_0 t)\left(1 - \frac{A_1}{2}\left(1 - \cos(\Omega t - \psi)\right)\right),$$

we then have

$$F_{\text{rad}} = \frac{1+R}{c}\left({E_0}^2(1 - A_0) + A_0{E_0}^2\cos(\Omega t) - {E_1}^2(1 - A_1) - A_1{E_1}^2\cos(\Omega t - \psi)\right),$$

or ignoring the D.C. terms,

$$F_{\text{rad}} = \frac{1+R}{c}\left(A_0{E_0}^2\cos(\Omega t) - A_1{E_1}^2\cos(\Omega t - \psi)\right).$$

To write this as a single cosine, we use some trigonometry:

$$\begin{aligned} F_{\text{rad}} &= \frac{1+R}{c}\left[A_0{E_0}^2\cos(\Omega t) - A_1{E_1}^2(\cos(\Omega t)\cos(\psi) + \sin(\Omega t)\sin(\psi))\right] \\ &= \frac{1+R}{c}\left[(A_0{E_0}^2 - A_1{E_1}^2\cos(\psi))\cos(\Omega t) - A_1{E_1}^2\sin(\psi)\sin(\Omega t)\right]. \end{aligned}$$

Writing the term in brackets in the form $D\cos(x+\theta)$, we get

$$D^2 = \left(A_0{E_0}^2 - A_1{E_1}^2\cos(\psi)\right)^2 + \left(A_1{E_1}^2\sin(\psi)\right)^2$$
$$D = \sqrt{{A_0}^2{E_0}^4 - 2A_0A_1{E_0}^2{E_1}^2\cos(\psi) + {A_1}^2{E_1}^4}$$

and

$$\theta = \tan^{-1}\left(\frac{A_1{E_1}^2\sin(\psi)}{A_0{E_0}^2 - A_1{E_1}^2\cos(\psi)}\right).$$

We can now proceed in the same way as in Single-Sided Free Mass (page 512), to obtain an expression for the reflected field on the left hand side of the mirror,

$$E_{\text{r}} = rE_0e^{i\omega_0 t}\left[1 + \left(\frac{A_0}{4}\left(e^{i\Omega t} + e^{-i\Omega t}\right) + i\frac{(1+R)D\omega_0}{mc^2\Omega^2}\left(e^{i(\Omega t+\theta)} + e^{-i(\Omega t+\theta)}\right)\right)\right].$$

To get the field on the right hand side of the mirror, we can apply the following (remembering that θ will also be affected):

$$E_0 \iff E_1$$
$$A_0 \iff A_1$$
$$\psi \implies -\psi.$$

Finesse

The analytics above are compared to the sidebands produced by the following FINESSE file:

```
l l1 P=3
s s1 l1.p1 m1.p1 L=5
m m1 R=1 T=0
s s2 m1.p2 l2.p1 L=5
l l2 P=4

free_mass m1_sus m1.mech 0.5e-3
```

(continues on next page)

(continued from previous page)

```
ad upper_left s1.p1.o fsig.f
ad lower_left s1.p1.o -fsig.f

ad upper_right s2.p2.o fsig.f
ad lower_right s2.p2.o -fsig.f

fsig(0.1)
sgen sig1 l1.amp.i 0.1
sgen sig2 l2.amp.i 0.1 phase=29

xaxis(fsig.f, log, 0.1, 1e7, 400)
```

For comparison with the analytical results, we must also include an extra factor of $e^{\pm i\psi}$ in the upper / lower sidebands respectively when looking at the right hand side field. This is because FINESSE measures phase relative to the first laser in the file, which in this case is the left hand side laser. The results are shown in Fig. 8.8 and Fig. 8.9.

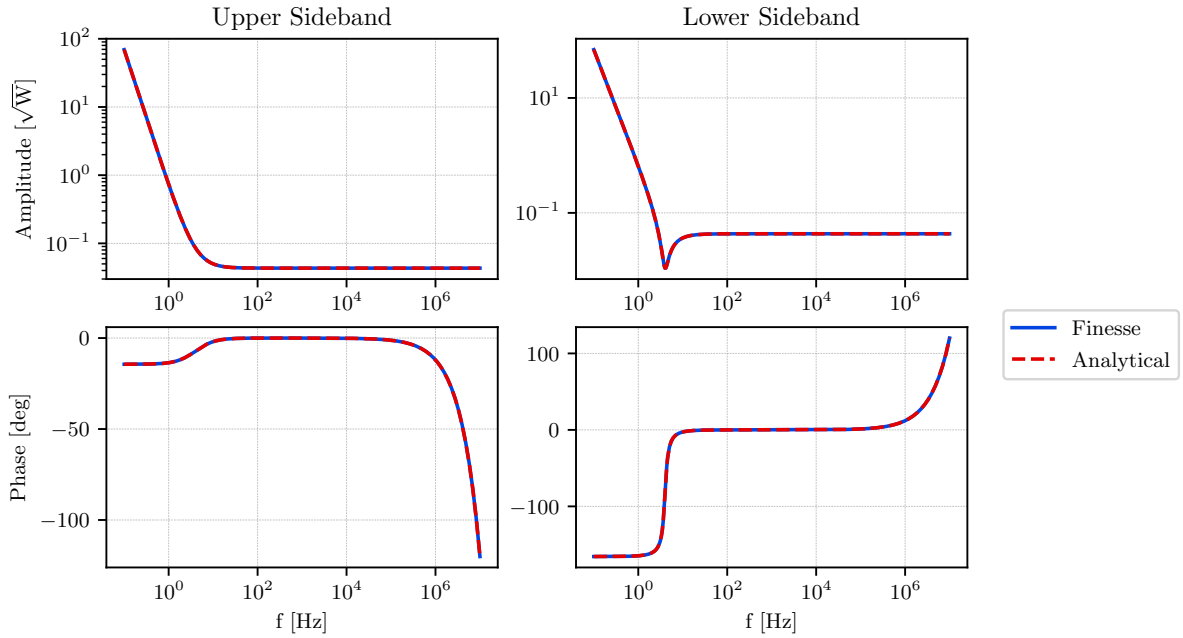


Fig. 8.8: Comparison of the upper and lower output sidebands from FINESSE and analytics for the reflected field on the left of the mirror.

Beamsplitters

The beamsplitter behaves similarly to the mirror, with two key differences. Firstly, the angle of incidence α in Equation (8.8) is no longer necessarily 0. This also affects the phase change on reflection due to the beamsplitter tuning, ϕ. Secondly, there are now 4 ports at which light can enter and exit, rather than two. Here we repeat similar experiments to the previous section, taking these factors into account.

Single-Sided Free Mass

Analytics

The steps here are much the same as in the equivalent mirror case (page 512), with the only changes being the inclusion of α in the radiation pressure force calculation, and a change in the phase shift due to mirror tuning ϕ — the total phase shift of reflected light will now be $e^{-2i\phi\cos\alpha}$, as the tuning ϕ is defined relative to the surface normal.

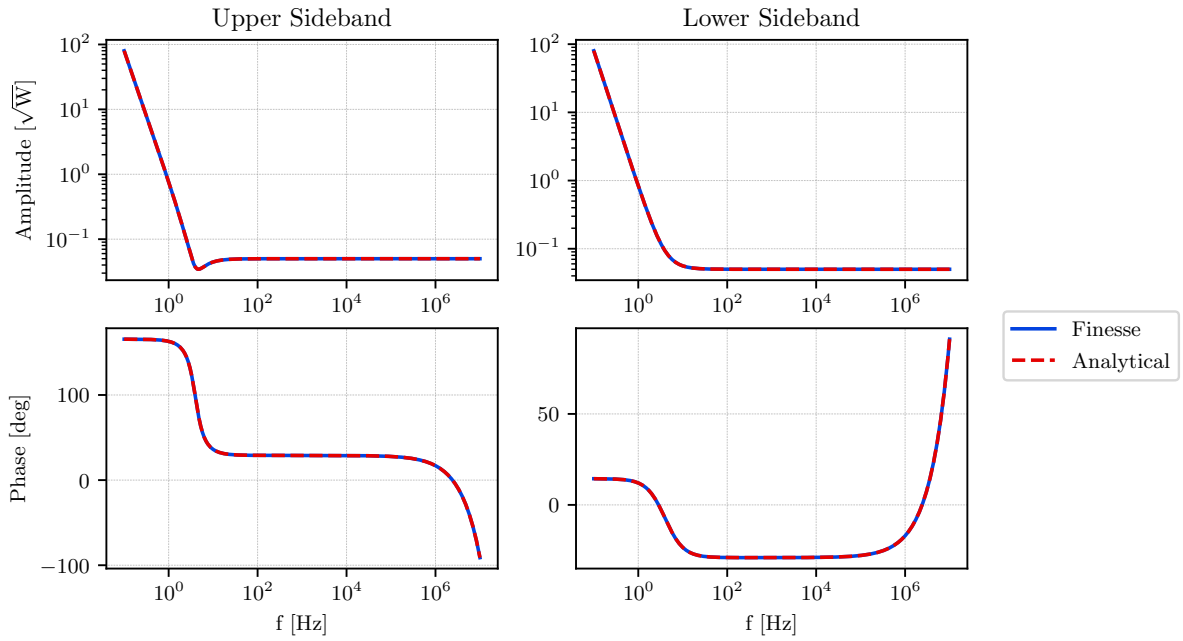


Fig. 8.9: Comparison of the upper and lower output sidebands from FINESSE and analytics for the reflected field on the right of the mirror.

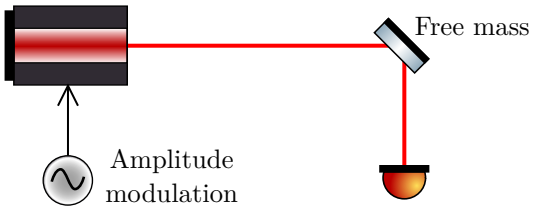


Fig. 8.10: Setup for the first beamsplitter test case, an amplitude modulated laser with two sidebands reflecting off a free mass at some non-zero angle.

We therefore have

$$x(t) = -\frac{(1+R-T){AE_0}^2}{mc\Omega^2}\cos(\Omega t)\cos\alpha.$$
$$\therefore \phi = -\frac{(1+R-T){AE_0}^2\omega_0}{mc^2\Omega^2}\cos(\Omega t)\cos\alpha.$$

This results in the incoming beam being phase modulated by the mirror motion,

$$E_\mathrm{r} = rE_\mathrm{i}\exp(iB\cos(\Omega t)),$$

where

$$B = \frac{2(1+R-T){AE_0}^2\omega_0}{mc^2\Omega^2}\cos^2\alpha,$$

Following the same steps as in the mirror case (page 512), the full expression for the reflected field can then be written as

$$\begin{aligned} E_\mathrm{r} &= rE_0e^{i\omega_0 t}\left(1+\left(\frac{A+2iB}{4}\right)\left(e^{i\Omega t}+e^{-i\Omega t}\right)\right) \\ &= rE_0e^{i\omega_0 t}\left(1+\left(\frac{A}{4}+i\frac{(1+R-T){AE_0}^2\omega_0}{mc^2\Omega^2}\cos^2\alpha\right)\left(e^{i\Omega t}+e^{-i\Omega t}\right)\right) \end{aligned}$$

Finesse

The analytics above are compared to the sidebands produced by the following simple FINESSE file:

```
l l1 P=3
s s1 l1.p1 bs1.p1 L=5
bs bs1 R=0.5 T=0.4 alpha=37
s s2 bs1.p2 out.p1 L=5
nothing out

free_mass bs1_sus bs1.mech 0.5e-3

ad upper s2.p2.o fsig.f
ad lower s2.p2.o -fsig.f

fsig(0.1)
sgen sig l1.amp.i 0.1

xaxis(fsig.f, log, 0.1, 1e7, 400)
```

The results are shown in Fig. 8.11.

Double-Sided Free Mass, One Modulation

Analytics

This is similar to the equivalent mirror case (page 515), with the inclusion of the $\cos^2\alpha$ term from the previous section. For a given input field

$$E_\mathrm{i} = E_1\cos(\omega_0 t),$$

we therefore have

$$E_\mathrm{r} = rE_1\left(1-i\frac{(1+R){AE_0}^2\omega_0}{mc^2\Omega^2}\cos^2\alpha\left(e^{i\Omega t}+e^{-i\Omega t}\right)\right).$$

Finesse

The analytics above are compared to the sidebands produced by the following FINESSE file:

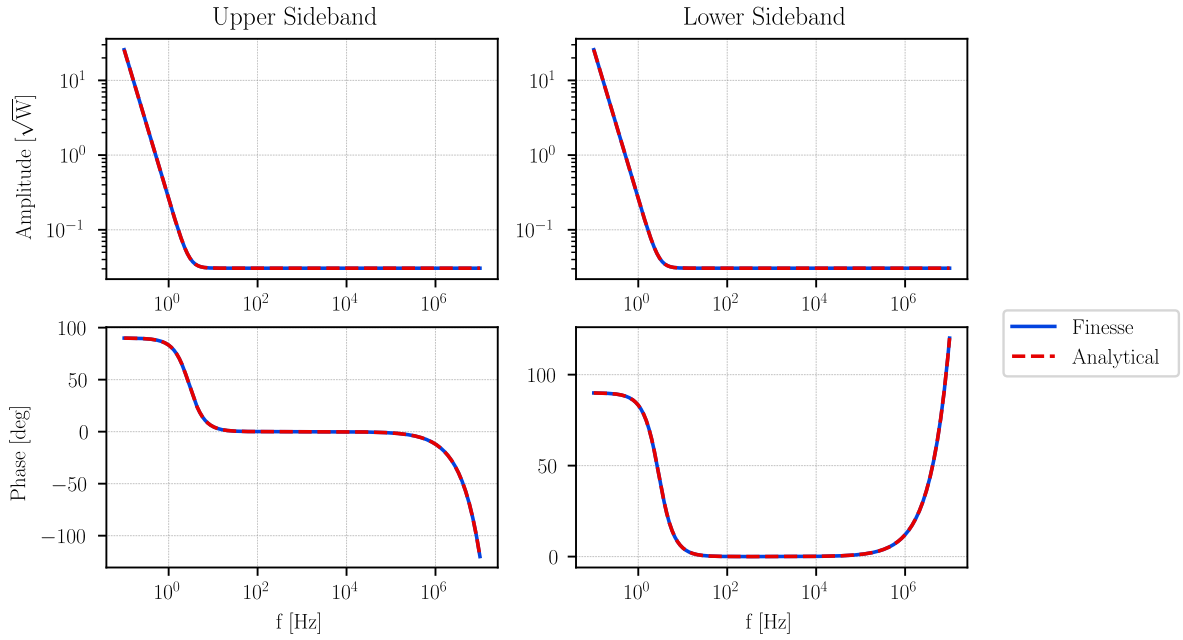


Fig. 8.11: Comparison of the upper and lower output sidebands from FINESSE with analytics.

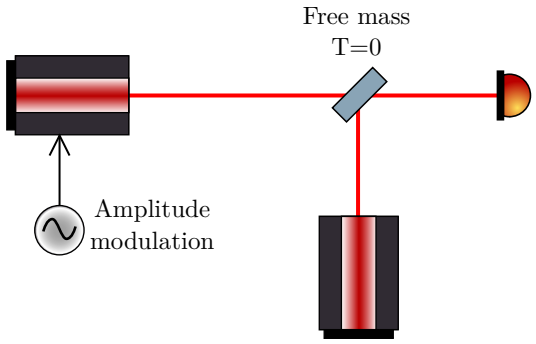


Fig. 8.12: Setup for the second beamsplitter test case, an amplitude modulated laser with two sidebands reflecting off a free mass causing sidebands to be generated in an unmodulated laser incident on the other side.

```
l l1 P=3
s s1 l1.p1 bs1.p1 L=5

l l2 P=2
s s2 l2.p1 bs1.p4 L=5

bs bs1 R=1 T=0 alpha=37
s sout bs1.p3 out.p1 L=5
nothing out

free_mass bs1_sus bs1.mech 0.5e-3

ad upper sout.p2.o fsig.f
ad lower sout.p2.o -fsig.f

fsig(0.1)
sgen sig l1.amp.i 0.1

xaxis(fsig.f, log, 0.1, 1e7, 400)
```

The results are shown in Fig. 8.13

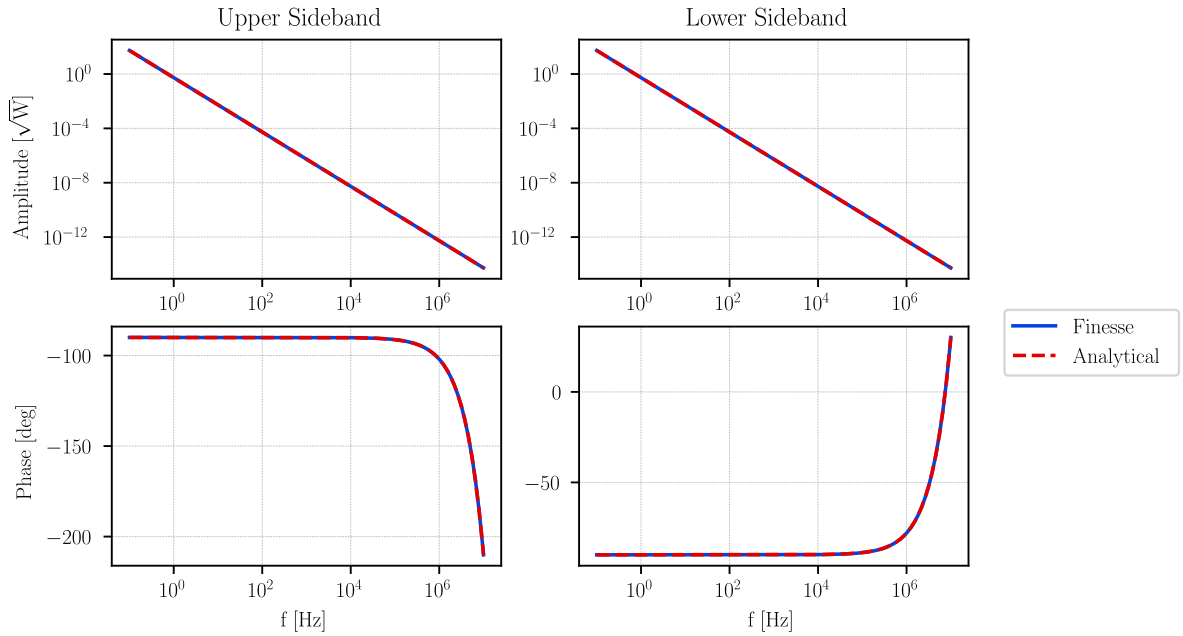


Fig. 8.13: Comparison of the upper and lower output sidebands from FINESSE with analytics.

Double-Sided Free Mass, Two Modulations

Analytics

Once again, this is similar to the procedure in the equivalent mirror case (page 517), with the addition of a $\cos^2\alpha$ term on the amplitude of the generated sidebands. The output at the top detector will therefore be

$$E_\mathrm{r} = rE_0 e^{i\omega_0 t}\left[1 + \left(\frac{A_0}{4}\left(e^{i\Omega t} + e^{-i\Omega t}\right) + i\frac{(1+R)D\omega_0}{mc^2\Omega^2}\cos^2\alpha\left(e^{i(\Omega t+\theta)} + e^{-i(\Omega t+\theta)}\right)\right)\right],$$

with D defined as in the mirror case (page 517). The results for the right detector can be obtained with the same transformations as before.

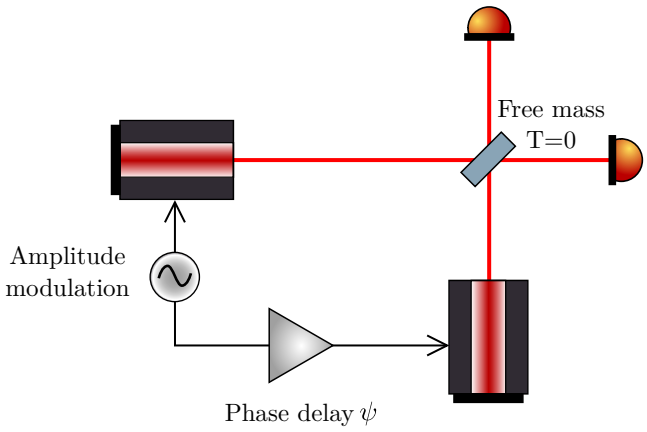


Fig. 8.14: Setup for the third beamplitter test case, two amplitude modulated lasers with two sidebands each incident on each side of a beamsplitter, with some arbitrary phase difference.

Finesse

The analytics above were compared to the sidebands produced by the following FINESSE file:

```
l l1 P=3
s s1 l1.p1 bs1.p1 L=5

l l2 P=4
s s2 l2.p1 bs1.p4 L=5

bs bs1 R=1 T=0 alpha=37

s sout_left bs1.p2 out_left.p1 L=5
nothing out_left

s sout_right bs1.p3 out_right.p1 L=5
nothing out_right

free_mass bs1_sus bs1.mech 0.5e-3

ad upper_left sout_left.p2.o fsig.f
ad lower_left sout_left.p2.o -fsig.f

ad upper_right sout_right.p2.o fsig.f
ad lower_right sout_right.p2.o -fsig.f

fsig(0.1)
sgen sig1 l1.amp.i 0.1
sgen sig2 l2.amp.i 0.1 phase=29

xaxis(fsig.f, log, 0.1, 1e7, 400)
```

As in the mirror case (page 517), for comparison with the analytical results, we must also include an extra factor of $e^{\pm i\psi}$ in the upper / lower sidebands respectively when looking at the right hand side field. This is because FINESSE measures phase relative to the first laser in the file, which in this case is the left hand side laser. The results are shown in Fig. 8.15 & Fig. 8.16.

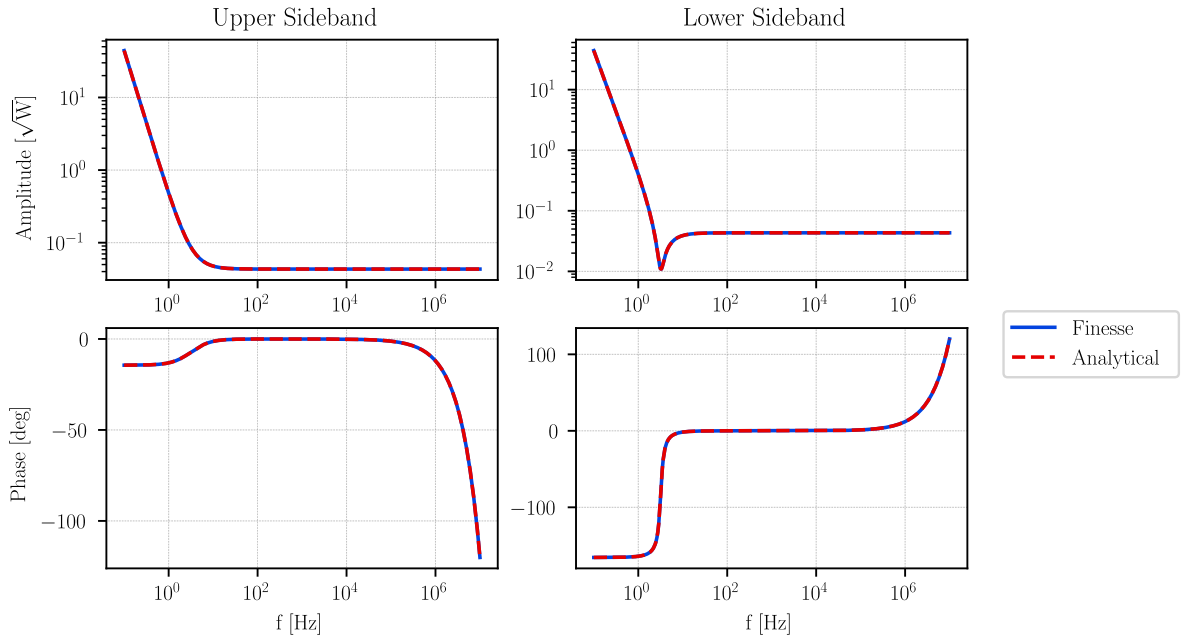


Fig. 8.15: Comparison of the upper and lower output sidebands from FINESSE and analytics for the reflected field on the left of the beamsplitter.

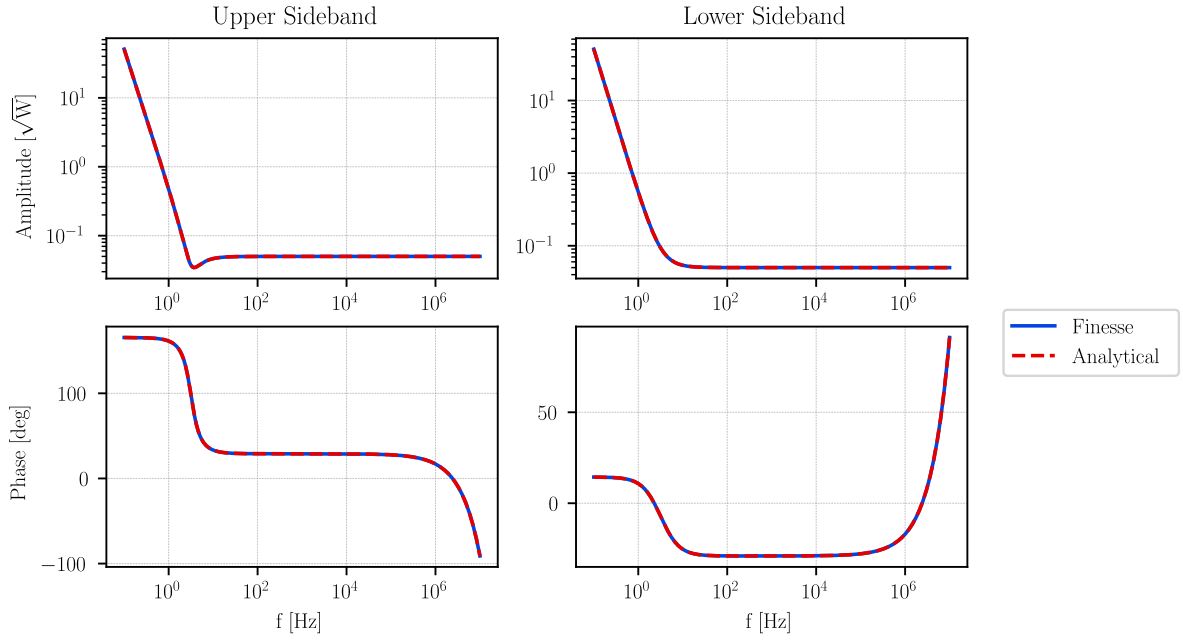


Fig. 8.16: Comparison of the upper and lower output sidebands from FINESSE and analytics for the reflected field on the right of the beamsplitter.

8.14 Todo List

8.15 Continuous integration

The FINESSE project makes use of GitLab's continuous integration feature to automatically build, test and deploy FINESSE on various environments.

The primary way in which continuous integration is used in the FINESSE project is to run tests on different target platforms to catch bugs there, especially those that don't appear in a developer's editable install (page 21). Other uses are to build and deploy the documentation, and to generate statistics regarding the project such as its test coverage. The pipeline run history is also useful to help determine which changes to FINESSE or its dependencies introduced some bug or change in behaviour.

The continuous integration pipeline runs on GitLab automatically whenever code is pushed to the upstream FINESSE repository. If you have configured your account for it, GitLab will send an email whenever a pipeline triggered by your push fails. You can inspect the console output from the failed job by clicking the link in the email, or browsing to the *CI/CD* section of the GitLab project.

8.15.1 Configuring the pipeline

The **.gitlab-ci.yml** file in the project root defines the continuous integration pipeline. See the syntax reference for more information on how this must be formatted. This file can also be edited using the GitLab web interface.

A linting tool is provided as part of the GitLab project to ensure there are no syntax errors or issues before committing changes to this file.

8.15.2 Pushing code without triggering the pipeline

GitLab supports skipping the pipeline via git's push options flag. This can be used to help save needless computation for minor changes that don't need the full build and test suite to run:

```
$ git push -o=ci.skip
```

The pipeline will then appear in GitLab as "skipped".

8.15.3 Wheels built by the pipeline

The pipeline contains build jobs that produce so-called *wheels* which make installation of FINESSE straightforward on Linux and Mac OSX by bundling compiled C extensions. The process follows that outlined in the section on packaging (page 493) but some caveats apply in certain situations, listed below.

Mac OSX on AMD64

Due to FINESSE's use of modern **PEP 517** build isolation, `pip wheel` must normally be able to install all required build dependencies via PyPI. This is unfortunately not possible on OSX 11 (Big Sur) and AMD64 (also referred to as "M1" or "Apple Silicon") as of writing because wheels are not available for `scipy` and `h5py`. This triggers pip to try to build the wheels locally, and the OSX runner then encounters errors due to lack of required build dependencies for *those* packages. The solution until wheels are available for these dependencies is to switch off isolated builds and instead build within a conda distribution. That's why the CI job for OSX on AMD64 adds the `--no-build-isolation` flag to `pip wheel`.

8.16 Debugging Finesse

Errors originating from C code compiled via Cython often do not yield useful error messages. Instead you may encounter segmentation faults without any additional information to point to the error origin. Setting up a debugger for cython code is quite complicated, so there is a dockerfile included in the repo that sets up a debugging environment. See the readme in the docker folder for instructions.

Chapter 9

Appendix

9.1 Glossary

API
: Application programming interface. In computing, this term generally refers to interfaces that bridge one computing domain to another. When used in relation to FINESSE, this typically refers to the Python API, which is the mechanism by which users interact with FINESSE via Python by creating classes, making calls to functions, and otherwise using the FINESSE library.

model
: The Python object that contains the elements that represent some optical setup: components and detectors, and the connections between them. Alternatively, the kat-script that defines the model may sometimes also be referred to as a model, but strictly this is just a set of instructions to define a model.

9.2 Changelog

This changelog describes the changes made between the different alpha versions of FINESSE. When seeing unexpected behavior with existing finesse scripts after updating, you should read the entries between the last working version and the version introducing the issues, to see if there are any entries that are relevant to your code. See Updating an environment (page 29) for instructions on how to update to the latest version.

> **Note**
>
> You are currently looking at the documentation for FINESSE 3.0.2-63-gee4f5bb3. Some of the changes listed for older versions may no longer be relevant, or contain broken links to non-existing documentation pages.

9.2.1 3.0.3 (YYYY-MM-DD)

Added

- Add a link to the PDF documentation in the left-hand menu, the version matches the html version (i.e. either develop or release).
- Add the version on the title page of the PDF documentation.
- Added `Parameter.add_dependency()` to declare that a parameter depends on another one without a symbolic expression. The dependency will be include in `Model.symbolic_network`. The model use that network to find the complete set of parameters that are changing.
- A `RigidBody` component which allows multiple components to share the same mechanical degrees of freedom and thus acting as one combined rigid body. Note that this is a building block for future composite components

that is currently only accessible via the Python API.

- `finesse.utilities.control.fQ_roots()` to compute the roots of (frequency, Q) pairs.

Fixed

- The API docs link in the menu should be skipped for PDF and point directly to a note for the html (without _api) target.
- Update remaining obsolete references to gwoptics as home for Finesse3.
- Prevent potential locale problems in tinyexpr by using a fixed C locale for the is* functions.
- Fixed several issues in tinyexpr, including out of array bounds, checks for malloc failures, maximum nesting to prevent stack overflow. This also fixes compiler warnings.
- Fixed a memory leak where a model with a mirror surface map was never released after running, because the map workspace held strong references to the simulation and the model that the garbage collector could not break. This fixes **#865**.
- Added missing factor of 2π in `Wire` delay.
- Fixed bug where the DC fields are sometimes not recomputed after a map update.
- Removed top-level IPython import in `finesse.utilities.text`, since IPython is not a Finesse dependency
- Passing an empty list to the `sources` argument of a `quantum_noise_detector` (page 181) resulting in non-deterministic return values.
- Don't call `colorama.init()` but `colorama.just_fix_windows_console()`. This fixes **#859** and makes ANSI escape sequences work again in JupyterLab and VSCode.
- The elapsed time of an action, `finesse.solutions.base.BaseSolution.time`, is now measured with a monotonic clock (`time.perf_counter_ns()`) instead of the wall clock (`time.time_ns()`).
- In `SetLockGains` (page 230), set the gains for disabled locks if they are requested explicitly, as promised by the docstring.
- `CavityPropertyDetector` (page 175) reporting the value of `tau` when requesting the `pole` value.
- `MapOPDDetector` and `MapAmplitudeDetector` work without raising exceptions and have testcases.
- `Space` (page 170) not refilling its couplings in the signal matrix when its length was being swept.
- Fixed `fQ = True` method of specifying roots in `ZPKFilter` and incorrect calculation of over-damped roots. This fixes **#859**.
- Parameters locks survive simulation runs.
- Changing the `amplitudes` of a `DegreeOfFreedom` whose parameters are locked no longer raises a `ParameterLocked` error.
- Removing `DegreeOfFreedom` parameter locks only removes the locks from that DOF.
- Fixed the four quantum noise demod detectors overcounting HOM vacuum. This fixes **#863**.

Changed

- next_token in tinyexpr is defined static, since it's for private use only.
- Use a more stable URL for the SciPy intersphinx mapping
- `lock` now takes an *exclusive* kwarg which is True by default which allows it to be locked by a single owner. Multiple DOFs can still lock a single parameter by calling `lock(by, exclusive=True)()`.
- Added logic to reduce the number of unnecessary scattering matrix calculations, thus speeding up certain kinds of simulations.

9.2.2 3.0.2 (2026-08-26)

Fixed

- Working initialisation of COMPLEX_0 on Windows

9.2.3 3.0.1 (2026-08-25)

Changed

- The sign convention for the axial coordinates in the functions defined in the *finesse.thermal* package was switched so that the front/HR surface of the test mass is at z=+h/2, the back/AR surface is at z=-h/2, and surface maps calculated by these functions are added to, rather than subtracted from, a mirror's surface map. See **#760**
- Removed the `set_gains` option from `RunLocks` (page 231). This option was not used by `RunLocks` (page 231). It is recommended to run `SetLockGains` (page 230) before running `RunLocks` (page 231) when using `method=proportional`.
- Updated Sphinx to version 9
- **API docs:**
  - use separate pages for modules
  - ignore `__all__` imports, that also results in some previously undocumented methods now being included in the API docs.
- `finesse.tracing.tools.propagate_beam()` now also accepts a list of multiple nodes in the `via_node` argument.

Fixed

- **#835** 2D plots incorrect without calling `finesse.init_plotting`, solved by explicitly setting origin="lower" to the relevant imshow calls.
- **#833** bug for matrix multiplication order when combining a map scattering matrix and the normal bayer-helms scattering matrix for mode mismatches. See How Bayer-Helms and map contributions are combined (page 361)
- **#840** incorrect amplitude of gradients in `OptimiseRFReadoutPhaseDC` (page 229).
- **#839** incorrect behaviour of `OptimiseRFReadoutPhaseDC` (page 229) when a readout is paired with multiple drives.
- **#846** The `plot` functions of the solution returned by `FrequencyResponse` (page 221) failing when a custom matplotlib ax is input.
- **#844** `SensingMatrixDC` (page 230) and `OptimiseRFReadoutPhaseDC` (page 229) changed the model state by numerical precision error.
- Fixed a wrong assignment to a const complex (`cymath/complex.pxd:11`) now detected by new Cython release.
- Operator action was returning operators that were not updated for the current parameter values. This was only an issue if parameters were varied without running any simulation. Extracting operators without running something like a Lock, Xaxis, Noxaxis, etc could have returned incorrect operators.
- `finesse.solutions.ArraySolution.expand()` wasn't working correctly as the flat iterators that the detector workspaces use were not being updated correctly for the newly expanded data array.
- `finesse.frequency.FrequencyContainer.get_frequency_index()` not raising an exception when a frequency can't be found
- Bug where Parameters in in pure python workspaces weren't updated.
- `floss` (page 162) not refilling the matrix when its parameters were changing

- In `finesse.plotting.plot.ws_phase_space()` replace `matplotlib.axes.Axes.set_xticklabels()` with `matplotlib.axes.Axes.set_xticks()` to prevent a warning. Showed up in the example in the docstring of `finesse.gaussian.ws_overlap_grid()` (which previously didn't get built because of the *__all__* import setting).
- Memory leak in `finesse.analysis.actions.base.AnalysisState.action_workspaces`
- Two minor beam tracing bugs: `Model.detect_mismatches()` now works for symbolic reflectivities; and a numerical ABCD matrix caching bug was fixed.
- Bug where KatScript variable names with shared prefixes could cause errors during simulations.
- Bug where KatScript variable names with similar names could cause errors during simulations when used in a symbolic expression

Added

- Symbolic version of the standard error function *erf*
- Type hints for the `finesse.frequency` cython module
- Verbose argument for `OptimiseRFReadoutPhaseDC` (page 229). See OptimiseRFReadoutPhaseDC (page 74) for an example.
- `finesse.analysis.actions.base.AnalysisState.set_parameter_values()` to replace copy-pasted code snippets used to clean up parameter values after certain actions were ran.
- Support for balanced homodyne detection: a balanced homodyne readout component `BalancedHomodyneReadout` (page 188), a balanced homodyne detector `BalancedHomodyneDetector` (page 173), and a quantum balanced homodyne detector `QuantumBHD` (page 181).
- `name` keyword argument for `StoreModelAttr` (page 237) and option to supply a conversion function for each argument of `PrintModelAttr` (page 237)
- Reintroduce linking to sources in the API docs, now via direct links to gitlab (incl. cython methods, enums etc.)

9.2.4 3.0.0 (2026-06-10)

This is the first stable release of Finesse 3.0. We will now use semantic versioning, e.g. `major.minor.path`, to name the versions.

Changed

- Final documentation improvements and polishing for the 3.0.0 release
- Updated contributor list with new contributors
- Deprecation mechanism (now use `deprecated` library)

Fixed

- **#782**, **#813** bug in mirror map utility methods
  - Two of the map overlap integral utility functions (RMS wavefront error and piston coefficient) were wrongly weighted.
  - All overlap integral functions are now normalized consistently and take into account the finite domains and numerical integration.
  - All overlap integral functions verify the dimensions of the input arrays instead of silently producing the wrong results.
- Deprecation problems because of 3.0.0 release

Removed

- Removed unused functions and leftovers from unfinished features

9.2.5 3.0rc1 (2026-06-02)

Changed

- No longer recommended the *sgen*/detector/*xaxis* flow for transfer functions, all examples use `frequency_response` (page 221) instead. See Modelling Transfer Functions (page 119) for an explanation.
- Removed the `open_loop` option from `frequency_response` (page 221). See MR 450 for a discussion.
- A complete rewrite of the installation guide (page 19).

Fixed

- The `Rc` parameter of `mirror` (page 142) and `beamsplitter` (page 149) did not have proper validation for the length of the array used to set `Rcx` and `Rcy`.
- **#814** indexing bug in `pseudo_lock_cavity` (page 232)
- **#381**, **#824** parsing error with different locale
- The `ArraySolution` `x0` and `x1` attribute confusion. `sol.x0` and `sol.x1` used to both return the sweep array for 1-dimensional `ArraySolution` (created by `xaxis` (page 209)). However, for multidimensional `ArraySolutions``s (like from :kat:analysis:`x2axis`), ``sol.x0` would return the second sweep array and `sol.x1` would return the first solution due to an indexing bug. The `ArraySolution` indexing for the sweep arrays is now 1-based. (**#809**, **#811**)
- Fixed an indexing bug in `PseudoLockCavity` (page 232) (**#814**)
- Fix transposed plots for the various camera detectors (page 338)
- Some yx-indexing bugs in the utility functions `HGModeWorkspace.u_nm` and `Image.at` (MR 435)
- A bug where the trace dependencies are not reset after calling `Model.cavity_mismatch` (MR 443)
- A parsing error when using symbolics in the KatScript and a “non-standard” locale setting (one that uses commas for decimal separators). (**#824** and **#381**)
- Ignoring comments in the katscript for the Replacing KatScript components (page 133) functionality.
- Inconsistent behaviour for the `compact` options for the utility tables class (page 258)

Added

- KatScript quick overview page (KatScript quick reference (page 390))
- A warning is printed if the compiled cython package is out-of-sync with the source. The warning is only display at import with a developer installation of Finesse. (MR 436)
- Ports and node overview tables in the documentation for most components, see Ports and nodes overview (page 148) for an example
- Add experimental support for Alignment Modulation (page 119)
- Added a new `lock_parameters` argument to the `dof` (page 190) component (defaults to `True`). See Symbolic links & Locking (page 290) for more details
- Added a method to check for broken symbolic links for the `dof` (page 190) component. See Symbolic links & Locking (page 290) for more details
- A quick overview of parsed files for the `Model` object. See Reviewing the history of parsed files (page 132)
- A new utility function `finesse.gaussian.analytic_un` (equation 9.34 from the living review)
- Option to place tags in the documentation to improve search ranking for pages.

- New overview page page for the Frequency Response (page 219) actions.
- A new page explaining the The model graph (page 35)

9.2.6 3.0b3 (2026-03-04)

Changed

- Restructuring of the documentation. We are working on a better flow and structure for the FINESSE documentation in preparation for the 3.0 release. Most notably the 'Physics of Finesse' section has been merged into the User guide (page 112). This means older links to documentation pages may be broken because the page is in a different place. Please contact us if you are missing a page
- Expanded documentation for the xaxis actions (page 208), including examples.
- The API are added back in. The look of the pages is slightly different, because we made changes under the hood to optimize our documentation build process.

Added

- New method to visualize symbolic relationships between parameters, see Visualizing the symbolic graph (page 239) for an example.
- New method for replace components in a Model. See Replacing KatScript components (page 133) for details.
- New argument `warnings` added to `finesse.model.Model.unparse`, which controls if unparsing warnings are shown
- New plane-wave examples for triangular cavity (page 40) and bow tie cavity (page 42)
- Added FINESSE 3 translations of the Living Review (page 561) examples
- New introduction text added for the Higher-order Hermite-Gauss modes (page 298) page.
- New introduction text on the The plane-wave approximation (page 112)
- New page on Transfer Functions and Error Signals (page 119)
- New introduction on Radiation Pressure (page 368)

Fixed

- Fixed issue **#797**: unparsing of a model where a parametervalue had been set using the python API to minus a reference failed with an AssertionError.
- Fixed issue **#810**: `tol` argument for `Maximize` (page 235) and `Minimize` (page 236) handled incorrectly for some methods.
- Fixed issue **#770**: RTL validation not being ran when using symbolic references.
- *getting_help* page being unavailable.

9.2.7 3.0b2 (2026-01-20)

Removed

- Support for python 3.10 (following NEP 29)

9.2.8 3.0b1 (2026-01-09)

Changed

- Reorganisation and rewrite of the documentation (work-in-progress)
- The data generated by the cameras (page 338) are now ordered such that the first index corresponds to the y axis and the second index to the x axis -> data[y, x].

- `Connector` components no longer assume that an identity ABCD matrix is the default. This means that if you have a connector that does not specify its ABCD matrix for each optical connection, you may see a `NoABCDCoupling`` exception raised now, when you would not have before.
- `NotChangeableDuringSimulation` now raised when trying to change a parameter that is not changeable during a simulation. This also fixes a bug where these changes are circumvented by using symbolics. Parameters that are not changeable during a simulation are flagged because some internal logic has not been implement yet to handle these changes.
- `ArraySolution` now only returns elements in its data array that has been filled, rather than the full allocated array.
- ParameterLocked error is now more userfriendly and suggests how to fix it by setting the parameter *is_tunable=True*.
- Map OPD methods now must accept a simulation keyword argument. This may be a *None* value and the map function should handle this sensibly with a default output of the map, such as all zeros.
- `finesse.analysis.actions.operator.Eigenmodes` now accepts a method argument that changes how the eigenvalues and eigenvectors are calculated. The *numpy.linalg.eig* does not perform well with larger more complex matrices, so the default method is now set to do a Schur decomposition and then calculate the eigenvectors using a QR to compute the null space of (A - λI).

Added

- Added ipykernel as doc build dependency as that doesn't seem to get installed now through other dependencies and leads to the doc build complaining that python3 doesn't exist.
- Added new Relaxation action that performs an iterative method to find the steady-state solution of a model that relies on non-linear behaviours, such as thermal deformations. This new action relaxes specific elements that provide a convergence and update interface.
- Added `HelloVinetDistortions` which wrap the new relaxation features for finding steady state solutions for models with thermal deformations. The `HelloVinetDistortions` element provides parameters to control the thermal state of an optic and automatically applies the maps and sets the curvature and focal length of mirrors and lenses.
- Added new detectors `MapOPDDetector` and `MapAmplitudeDetector` for extracting out the state of the map and storing it in the `ArraySolution` outputs for easy access.
- New dependencies colorama packages. Colorama provides a cross platform interface for doing multi-line terminal writing.
- `track_eigenmodes()` has been added for tracking the evolution of complex-valued cavity eigenvalues generated from roundtrip operators. This allows easier analysis of modal evolutions in cavities. A more generic finesse.utilities.misc.track_particles has also been added that tracks N-dimensional points.
- `finesse.analysis.actions.operator.EigenmodesSolution.estimate_orders()` added to give an approximate estimate of the the order of each eigenmode
- New documentation page (page 183) and docstrings for the readout components.
- Better error messages for **#711**, modulator signal injection is only partly implemented
- Possibility to parse katscripts and file at `finesse.model.Model` initialisation
- `finesse.model.Model.parse()` (and associated methods) now return the `finesse.model.Model` object
- Symbolic version of the standard error function *erf*
- Model.parse* now return Self to allows command chaining.
- Release dates in the changelog

Fixed

- Fixed issue **#775**: several tests failed with numpy 2.4.0 since we relied on long deprecated behaviour.
- Fix trivial typo that broke the `modulator` (page 159) `positive_only` option.
- Fixed issue **#745**: "Vacuum" material had the same name as FusedSilica.
- Fixed issue **#654**: Preventing doubly connected readout components.
- Fixed `finesse.components.dof.DegreeOfFreedom.dc_enabled`
- Fixed issue **#691**: Degree of freedom changes not propagating to angular parameters
- Fixed issue with DetectorWorkspace block_size not being updated causing issues with ArraySolution filling if the detector shape changes after initialisation
- Fixed issue with *finesse_sphinx* creating memory leak when the doc was build multiple time in successions.

Removed

- Removed runtime compatibility checks related to numpy v2

9.2.9 3.0a33 (2025-05-15)

Changed

- The handling of the RTL relationship for `Mirror` (page 142) and `beamsplitter` (page 149) has been improved. Energy conservation is now checked at every parameter update and when supplying 2 out of 3 values, the third one is automatically set to a symbolic expression to make sure $R+T+L=1$. See RTL relationship (page 144) for more details and examples.
- `finesse.components.general.Connector` components no longer assume that an identity ABCD matrix is the default. This means that if you have a connector that does not specify its ABCD matrix for each optical connection, you may see a `finesse.exceptions.NoABCDCoupling` exception raised now, when you would not have before.
- Improving Python error messages when strings are given as node names.

Added

- Added new `Telescope` (page 164) component which represents an ideal telescope between to parts of an optical system. For example, it allows the use to perfectly mode match two cavities without needing to design or specify the exact telescope details.
- `_trace_through` has been added as an internal flag to stop the beam tracer from trying to trace through a connector. This can be used for components like `Telescope` (page 164), which only connect optical fields and shouldn't allow beam tracing through it as it isn't well defined. Or the user could use this to fix a place that a mismatch should occur.
- Added an option to specify the transmission and reflection conventions for `Mirror` (page 142) and `beamsplitter` (page 149). The transmission and reflection phase relationship can be changed by setting the `imaginary_transmission` argument to either real or imaginary. By default this is imaginary (True), so a i phase on transmission. If set to be the real convention then the phase is 0 on transmission and 180 degrees on reflection on the port 1 side.

Removed

- `zero_tem00_gouy` optional argument was removed from `finesse.cymath.homs.HGModes` constructor as it was not being used internally or elsewhere.

Fixed

- Fixed issue **#699** by giving `ReadoutRF` (page 187) a default frequency value of `f=0`.
- Fixed passing arguments in the wrong order to `link()` not raising an exception.
- Operator action was returning operators that were not updated for the current parameter values. This was only an issue if parameters were varied without running any simulation. Extracting operators without running something like a Lock, Xaxis, Noxaxis, etc could have returned incorrect operators.
- `expand()` wasn't working correctly as the flat iterators that the detector workspaces use were not being updated correctly for the newly expanded data array.
- Improved exception messages from the beam tracer to removed "bug encountered!" which wasn't true in many cases. BeamTraceExceptions are now thrown with more useful messages to the user.

Changed

- `finesse.components.general.Connector` components no longer assume that an identity ABCD matrix is the default. This means that if you have a connector that does not specify its ABCD matrix for each optical connection, you may see a `finesse.exceptions.NoABCDCoupling` exception raised now, when you would not have before.

Added

- Added an option to specify the transmission and reflection conventions for `finesse.components.mirror.Mirror` and `finesse.components.beamsplitter.Beamsplitter`. The transmission and reflection phase relationship can be changed by setting the `imaginary_transmission` to either real or imaginary. By default this is imaginary (True), so a i phase on transmission. If set to be the real convention then the phase is 0 on transmission and 180 degrees on reflection on the port 1 side.

9.2.10 3.0a32

Fixed

- `modulator` (page 159) and `Nothing` (page 171) were not correctly handling mismatches changing during a simulation across their nodes which is now fixed.
- Nested `finesse.solutions.array.ArraySolution`s did not work correctly as :class:`finesse.solutions.array.ArraySolution` memoryviews were being held on a workspace basis. So running a `Noxaxis` (page 208) in a `Xaxis` (page 209) would conflict which memoryview was being used.
- Fixed not having the `IPython` library installed preventing you from importing Finesse
- Fixed incompatibility with new major `IPython` release (9.0)
- Fixed signal modelling example (page 268) to use the *pwr* node instead of the *amp* node on the laser
- Fixed bug report functionality for python 3.13 (*reporting_issues*)
- Fixed `finesse.model.Model.plot_graph()` not working in a notebook environment
- Fixed carrier simulation not being ran when running an optimizer on a `BeamPropertyDetector` (page 173)

Added

- MR !288 adds ring heater thermal deformation calculations to the thermal module. See `finesse.thermal.ring_heater` for more information.
- MR !289 allows model to be pickled for storing. Should only be used for short term storage of a model, as it is not guaranteed to be compatible with future versions or across different platforms or systems. See Saving and loading models (page 253)
- Missing katscript documentation for AstigmaticLens (page 158) component versions or across different platforms or systems.

Changes

- Added `ipykernel` as doc build dependency as that doesn't seem to get installed now through other dependencies and leads to the doc build complaining that python3 doesn't exist.

9.2.11 3.0a31 (2025-01-23)

Added

- Wheels for linux/macos 3.13

9.2.12 3.0a30 (2025-01-23)

Added

- `AstigmaticLens` (page 158) element added with x and y focal length parameters. Works the same as a `Lens` (page 156) and also has an OPD map attribute.
- Added simple sinusoidal and Helical LG mode calculation functions to `finesse.cymath.laguerre` module, see example (page 324)
- Added developer documentation page (page 484) on the internal matrix solver.
- Added `reverse_gouy` and `flip_lr` options to `finesse.knm.maps.Map.scatter_matrix()` to allow for more flexibility in how the scatter matrix is computed.

Fixed

- Fixed #685: Figure dictionary return by `solution.plot()` method has correct strings keys when single detector is present.
- Fixed #669: `finesse.knm.maps.Map.remove_piston()` accepts spot_size only with type `float` and not `np.float`
- Fixed #666: `finesse.analysis.actions.dc.DCFieldsSolutions` and `finesse.analysis.actions.lti.FrequencyResponseSolution` cannot select themselves by name
- Fixed #573: Symbolic changing division in lens focal length makes cavity unstable
- Fixed a collection of warnings in various tests

Changed

- Drop support for Python 3.9 due to various packages no longer supporting it
- Allows a superclass of `finesse.knm.maps.Map` to be used as a base class for custom maps. This allows for more flexibility in defining custom maps that define their own aperture and surface functions. The following example demonstrates:

```
class CustomMap(Map):
   def __init__(self):
      x = ...
      y = ...
      super().__init__(
         self,
         x,
         y,
         opd=self.surface,
         amplitude=self.aperture,
      )

   def aperture(self, model=None):
      return ...
```

(continues on next page)

(continued from previous page)

```
def surface(self, model=None):
    return ...
```

- `finesse.exceptions.NotChangeableDuringSimulation` now raised when trying to change a parameter that is not changeable during a simulation. This also fixes a bug where these changes are circumvented by using symbolics. Parameters that are not changeable during a simulation are flagged because some internal logic has not been implement yet to handle these changes.

9.2.13 3.0a29 (2024-11-14)

Fixed

- Fixed a critical bug in `DCFields` (page 235) where the fields were not always recalculated after parameter changes.
- Fixed broken `Isolator` (page 155) component, which caused a13 segfault in included in a model that was being run.

Added

- Added a new utility function `finesse.utilities.bug_report.bug_report()` to make it easier to report issues by automatically collecting relevant information. See m,`reporting_issues` for an example.
- Added a `show` and `path` argument to `finesse.model.Model.plot_graph()` to control whether to show the plot and whether to save it to disk.

Changed

- Use a more numerically stable method for calculating the frequency response of zpk filters which is more robust with filters with a large number of roots.
- FieldDetectors now work with planewave models and no longer throw a warning.

9.2.14 3.0a28 (2024-09-19)

This release includes two changes to default way of handling phases in FINESSE, which are listed below. The effect of these changes is subtle, and described in detail in Phase configuration settings (page 299) and Mirror and beamsplitter phase relationships (page 113).

> **Warning**
>
> If your simulations produce different results after updating to this release, it is likely that your previous results were incorrect, since the old default can break power conservation in certain setups.

You can switch between the two settings using `finesse.model.Model.phase_config()`

```
# new default
model.phase_config(False, True)
# old default
model.phase_config(True, True)
```

You can use this to check if your results are impacted by this change.

The changes have been tested with both the finesse-ligo and finesse-virgo packages, so if you are using these you will probably not be affected.

If you have any questions, do not hestitate to contact us via the matrix channel.

Breaking Changes

- The default value for `zero_k00` `finesse.model.Model.phase_config()` has been changed to `False`. See Phase configuration settings (page 299) for more information and an example of how the old default can break power conservation in the simulation Any simulation running with higher order modes and a cavity could be affected by this change and users using `finesse.knm.maps.Map` should make sure they use optimizers for locking.
- The phase relationship on transmission has been changed to a new default. It will be identical to the previous relationship for most cases, but will prevent power conservation issues in more complicated setups. You can set the `_settings.phase_config.v2_transmission_phase` to `True` if you want to revert to the old (Finesse 2) behavior, but the new behavior is likely more physically correct. See Mirror and beamsplitter phase relationships (page 113) for more details.

Added

- Add missing docs for many katscript Actions, Analyses, and Solutions (page 198) and Model elements (page 142) (incl. new group Mechanical Elements), fix many broken links
- Extra documentation on Phase configuration settings (page 299)
- New method: `finesse.model.Model.get_open_ports()`
- New class `finesse.utilities.collections.OrderedSet`, used in multiple places where ordering of results/components is relevant
- Distance-based filtering for visualizing subgraphs of the model graph, see Visualizing the model (page 247)

Changed

- `finesse.model.Model.get_elements_of_type()` now returns a tuple instead of a generator, and accepts element names as strings as well as python classes as arguments:
- Added an option `full_output` to `finesse.gaussian.optimise_HG00_q_scipy()` to return the optimized array of HG modes alongside the fit result. This replaces the previous `return_fit_result` argument.
- Include parity flip on reflection in Mirror and Beamsplitter ABCD methods. See also the new `_s` and `_t` suffixes in methods in `finesse.tracing.abcd` and the **#123**. For backward compatible code, use something like:

```
try:
   from finesse.tracing.abcd import space, beamsplitter_refl

   beamsplitter_refl_t = lambda *args: beamsplitter_refl(*args, "x")
   beamsplitter_refl_s = lambda *args: beamsplitter_refl(*args, "y")
except ImportError:
   # Handle newer versions which separate the beamsplitter refl # into the tangential
   and sagittal planes from 3.0a28 from finesse.tracing.abcd import (
      space, beamsplitter_refl_s, beamsplitter_refl_t,
   )
```

- Changes the examples in the documentation to indicate that parsing an action in KatScript and calling `model.run()` without any arguments is not recommended.

Removed

- The `'sagittal'` and `'tangential'` synonyms for `'y'` and `'x'` for the `modes` (page 197) command have been removed, since they can cause confusion with the new `plane` option for `beamsplitter` (page 149).

Fixed

- Single solution outputs can be seleceted by their name now, so you do not need to add extra actions or put them in series unnecessarily.
- Clarified the docstrings for the `Cavity` (page 195) class its `finesse.components.cavity.Cavity.path()` method.

- The shape of DCFieldsSolution is reverted to `[nodes, frequencies, HOMs]`, as described in `finesse.analysis.actions.dc.DCFieldsSolutions`
- When removing a Readouts (page 183) component, remove the associated output detectors as well (note that removing components from a model is unreliable in general)
- Parameters of autogenerated spaces and wires not generating a correct `full_name` attribute.
- Fixed issue **#659** - missing `sol = model.run()` in documentation for Maximise action

9.2.15 3.0a27 (2024-07-23)

- Allow detectors and ports to be visualized with component_tree method. See an example in the [docs](https://finesse.ifosim.org/docs/develop/usage/python_api/models_and_components.html#visualizing-the-model)
- Fix *finesse.gaussian.HGMode* ignoring shape of the given y vector when n=m.
- Option to keep only a subset of symbols in symbolic *Model.ABCD* method
- Add options to specify the plane of incidence for a beamsplitter and to misalign a beamsplitter
- Add pytest-xdist and configure it for faster (parallel) test running
- Fix slow optimal q detector test slowdown
- Fix broken cli test overwriting user config

9.2.16 3.0a26 (2024-07-10)

- Fixed inadvertently adding cython as a runtime dependency in 3.0a25
- Added documentation on defining manual beam parameters
- Expanded docstring on mirror curvature
- Better error message for degree of freedom illegal self referencing
- Generate conda files automatically from pyproject.toml

9.2.17 3.0a24 (2024-06-16)

New features

- Add FrequencyResponse4 action: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/202
- add plot_field method to EigenmodesSolution: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/220

Documentation changes

- Documentation on degree of freedom command/component: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/224
- new example: inference on RoC to examples: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/222
- Adding a link to the finesse-ligo documentation: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/218

Other

- Fix/benchmark creation tests: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/223
- Fix/641 ignore hidden folders during pyx files compilation checks: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/221

9.2.18 3.0a23 (2024-04-29)

- Fix memory leak issue during model building
- Changing the 'disabled' argument of the Lock component to 'enabled' to avoid double negative if conditions in the code
- Minor performance fixes
- Fix for the Optimizer action that would sometimes leave the model in an incorrect state after optimization
- "Did you mean" suggestions for katscript keyword arguments on syntax errors
- Adds warning for unreasonable katscript line lengths and better message on parsing recursion errors
- Evaluate symbolic references in the component info tables
- allows overlap_contour to work with (qx,qy) input

9.2.19 3.0a22 (2024-04-03)

- phase_config now locked when the model is built
- Symbol.lambdify was added to change Finesse symbolic expressions into a Python callable function
- Added initial benchmarking tests for tracking performance changes over time
- KnmMatrix.plot can be set to amplitude or phase now with the mode option
- Locks now throw an explicit exception LostLock when it fails
- Added Matplotlib helper function that plots arrows along a line at several points

9.2.20 3.0a21 (2023-11-20)

Adds support for Python 3.12

9.2.21 3.0a20 (2023-11-15)

- Python 3.8 support dropped: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/172
- Matplotlib 3.8 now works:
- FrequencyResponse action fixed when using two element frequency vector: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/169
- Now compiles with Cython 3, provides better debugging. Performance seems similar but not yet confirmed.
- Can trace beams in reverse for propagating through isolating components: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/181
- Wavefront curvature added to the beam propagation data: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/171
- KatScript will now take the Python class name as an option for elements and actions: https://gitlab.com/ifosim/finesse/finesse3/-/merge_requests/160
- EigenmodeSolution for a cavity now has method to compute roundtrip loss: https://gitlab.com/ifosim/finesse/finesse3/-/commit/db847bff9bf5ef4ffb109c5e234def6860f62525
- Map now has a *remove_piston* term method: https://gitlab.com/ifosim/finesse/finesse3/-/commit/ef83443addbfa4c99d4b662c6f8058a1740775fe
- New *DCFields* action to return a solution containing the DC optical fields at every node and frequency: https://gitlab.com/ifosim/finesse/finesse3/-/commit/b2cf34acae38d53a6dbf51906875f89e4589fee0

9.2.22 3.0a19 (2023-08-19)

- Requiring Matploblib < 3.8 until fixes are made for plotting routines
- Parameters that are external controlled, such as those set by a DegreeOfFreedom will explicitly shown the symbolic reference now as opposed to hiding it. See #571

9.2.23 3.0a18 (2023-08-17)

- Fixed FieldDetector not conjugating lower sideband
- Fixed DegreeOfFreedom using custom AC_IN and AC_OUT not filling the matrix correctly
- Variable element removed, now calls model.add_parameter instead. this means some code using *var.value.value* will no longer work.
- Added extra factorisation step when refactor returns a singular matrix with KLU, a warning will show when this happens.
- Model.display_signal_blockdiagram now takes nodes name list and only show paths with those nodes in for cleaner diagrams
- Model.path can take a symbolic=True keyword argument now for calculating symbolic path lengths
- phase_level deprecated and replaced with model.phase_config method
- unparing unsupported KatScript values will result in a __FIX_ME__ token
- Parameters now support boolean checks
- Python 3.11 wheels now built
- SetLockGains now just optimises gains and not demodulation phases
- OptimiseRFReadoutDemodPhaseDC now accepts Readout detector outputs *REFL9_I* or *REFL9_Q* for example, to optimise for either quadrature. Readout names supported and will default to *_I* with a warning
- Fixed Hello-Vinet function *substrate_thermal_expansion_depth* #567

9.2.24 3.0a17 (2023-08-02)

- Fixing block diagram generation for signal paths
- DOF has simpler interface for specifying just a DC actuation instead of using LocalDegreesOfFreedom
- This also allows user to specify their own AC connections to the DOF.AC.i and DOF.AC.o as they see fit
- Reworking signal node and port connections and attached_to attributes so they work
- Wires now connect anything to anything, had some logic about input and output that isn't needed anymore. Wires also have a gain now for simply rescaling inputs before summing with multiple other signal nodes

9.2.25 3.0a15 (2023-07-24)

Adding additional features for degrees of freedom to allow for better/easier modelling of ASC and other more complex effects. LocalDegreesOfFreedom replaces DOFDefinition, which now has separate AC input and output nodes. Also tested against Sidles-Sigg theory and no internal code changes were needed.

9.2.26 3.0a14 (2023-07-20)

Same as a13 but redoing conda dist for source

9.2.27 3.0a12 (2023-07-18)

Pinning to less than Cython 3

9.2.28 3.0a11 (2023-07-17)

Packaging/CI for windows still problematic, switching to conda instead of mamba due to 404 package errors

9.2.29 3.0a10 (2023-07-17)

Attempt at fixing bad windows tag processing

9.2.30 3.0a9 (2023-07-17)

New alpha update

9.2.31 3.0a8 (2023-06-02)

Bad pypi source pushed for a7

9.2.32 3.0a7 (2023-06-02)

Some recent fixes that improves memory allocation errors/checking and some usability errors.

9.2.33 3.0a6 (2023-03-11)

Alpha 6 release

9.2.34 3.0a5 (2022-10-09)

alpha 5, testing pypi deploy pipeline

9.2.35 3.0a4 (2022-10-09)

alpha 4

9.2.36 3.0a3 (2022-03-08)

Primarily fixes for Windows

9.2.37 3.0.0a2 (2022-01-19)

9.2.38 3.0a1 (2021-04-30)

9.2.39 0.1.0 (2018-06-14)

9.3 Finesse 2

9.3.1 Finesse 2 support

Supported legacy syntax

FINESSE 3 supports most common FINESSE 2 syntax via its legacy parsing mode. Legacy parsing is available either via the Command line interface (page 384) (using `--legacy` (page 387)) or in Python using `Model.parse_legacy()` or `Model.parse_legacy_file()`.

The FINESSE 2 syntax supported by the legacy parser is listed below, along with links to the relevent FINESSE 3 KatScript syntax and Python API. Directives not listed below are not supported. The supported parameters for each directive can be found in the FINESSE 2 manual.

Warning

Support for FINESSE 2 script support is only provided as a convenience to assist in transitioning to FINESSE 3. New scripts should always be written in FINESSE 3 script.

Components

Legacy name(s)	New name(s)	Python API

`bs`
`bs1`
`bs2`
`beamsplitter`
	`bs` (page 149)	`Beamsplitter`

`cav` `cavity`	`cav` (page 195) `cavity` (page 195)	`Cavity`
`dbs`	`dbs` (page 153)	`DirectionalBeamsplitter`

`gauss`
`gauss*`
`gauss**`
	`gauss` (page 195)	`Gauss`

`gouy`	`gouy` (page 177)	`Gouy`
`isol` `diode`	`isol` (page 155) `isolator` (page 155)	`Isolator`

`l`
`laser`
`light`
	`l` (page 165)
	`laser` (page 165)
		`Laser`

`lens`
`lens*`
`lens**`
`lens***`
	`lens` (page 156)	`Lens`

`m`
`m1`
`m2`
`mirror`
	`m` (page 142)
	`mirror` (page 142)
		`Mirror`

`mod`	`mod` (page 159) `modulator` (page 159)	`Modulator`
`sq` `squeezer`	`sq` (page 168) `squeezer` (page 168)	`Squeezer`
`s`	`s` (page 170)	`Space`

Detectors

Legacy name(s)	New name(s)	Python API
`ad`	`ad` (page 172)	`AmplitudeDetector`

`beam`	`ccd` (page 174)
	`ccdline` (page 174)
	`ccdpx` (page 174)
		`CCD`
		`CCDScanLine`
		`CCDPixel`

`bp`	`bp` (page 173)	`BeamPropertyDetector`
`xd`	`xd` (page 179)	`MotionDetector`

`pd[n]`	`pd` (page 180)
	`pd1` (page 180)
	`pd2` (page 180)
		`PowerDetector`
		`PowerDetectorDemod1`
		`PowerDetectorDemod2`

`qnoised[S/N]`	`qnoised` (page 181)
	`qnoised1` (page 181)
	`qnoised2` (page 182)
		`QuantumNoiseDetector`
		`QuantumNoiseDetectorDemod1`
		`QuantumNoiseDetectorDemod2`

`qshot[S/N]`	`qshot` (page 182)
	`qshot1` (page 183)
	`qshot2` (page 183)
		`QuantumShotNoiseDetector`
		`QuantumShotNoiseDetectorDemod1`
		`QuantumShotNoiseDetectorDemod2`

Commands

Legacy name(s)	New name(s)	Python API
`attr`	Replaced with arguments to the relevent components.	See relevent component constructors.
`attr mass` `attr tf`	Replaced with dedicated mechanics components e.g. `pendulum` (page 193), `free_mass` (page 192).	See components under `components.mechanical`.
`fadd` (Implemented in FINESSE 3 as `freq name frequency`)	Not implemented.	`Model.add_frequency`
`fsig name f` (Setting the frequency)	`fsig` (page 196)	`Model.fsig`
`fsig name comp ...` (Applying a signal)	`sgen` (page 169) `signal_generator` (page 169)	`SignalGenerator`

`func`
`set`
`put`
`put*`
	Replaced with KatScript Expressions and references (page 129).	`Symbol`
		`Parameter.ref`

`lambda` `lambda0`	`lambda` (page 196)	`Model.lambda0`
`lock` `lock*`	`lock` (page 190)	`Lock`
`mask`	Replaced by `mask` property of the relevant detector.	`MaskedDetector.mask`
`maxtem`	`modes` (page 197)	`Model.modes`
`pdtype`	Use *pdtype* keyword argument for detectors	Use *pdtype* keyword argument for detectors
`phase`	Not implemented.	`Model.phase_level` `Model.phase_config`

Axes

Legacy name(s)	New name(s)	Python API
`noxaxis`	`noxaxis` (page 208) *N.B.* This is now the default when no analyses are specified.	`Noxaxis`

`xaxis`
`x2axis`
`x3axis`
	`xaxis` (page 209)
	`x2axis` (page 213)
	`x3axis` (page 215)
		`Xaxis`
		`X2axis`
		`X3axis`

`yaxis`	Not implemented.	`Model.yaxis`
`scale`	Not implemented.	Not implemented.

Internal differences from Finesse 2

Modulator sideband amplitudes

Finesse 2 uses a Bessel function routine optimised for speed of calculation. This causes modulator field amplitudes to be incorrect by a small amount only for extreme values of modulation depth when more than one modulation is performed. Finesse 3 uses a correct implementation, and so will differ from Finesse 2 slightly.

The following table lists the modulation indices for which Finesse 2's error is distinguishable from numerical noise (greater than 10^{-14}):

Order, n	Modulation index, m
1	Always Correct
2	$0.482 \leq m \leq 2.0$
3	$0.958 \leq m \leq 3.0$
4	$2.128 \leq m \leq 4.0$
5	$2.920 \leq m \leq 5.0$

9.3.2 Trace verbosity

The `trace` command in Finesse 2 provided options for printing various beam tracing and coupling coefficient information associated with a file. On this page we will detail the replacements (and improvements) for these options.

For a brief summary of the Finesse 2 `trace` options and how these translate into the new features, see the table below. The descriptions of each integer, n, in terms of the Finesse 2 behaviour can be found in the old syntax reference.

Old `trace` option, n	New FINESSE equivalent
1	The `Model.modes()` method.
2	Print the `ModelElement.info_parameter_table()` of `Cavity` (page 195) objects.
4	See `Model.mismatches_table()`.
8	Display the `BeamTraceSolution` via printing return of `Model.beam_trace()`.
16	See `Model.space_gouys_table()`.
32	Use `KnmDetector` (page 177) objects.
64	Use `ModeMismatchDetector` (page 178) objects.
128	Draw the `TraceForest` via printing `Model.trace_forest`.

We will now go into a bit more detail on each of these options, in terms of the recommended ways to use them. For all of the following sections we will employ the example of a Fabry-Perot Michelson interferometer with aLIGO-like geometrical parameters for the arm cavities:

```
import finesse
finesse.configure(plotting=True)

model = finesse.Model()
model.parse("""
# Radii of curvature of arm cavity mirrors
var XROC1 -1934
var XROC2 2245
var YROC1 -1934
var YROC2 2245

l L0 P=1
link(L0, BS)
bs BS

s sy BS.p2 ITMY.p1 L=10
m ITMY R=0.99 T=0.01 Rc=[YROC1, YROC1]
s LY ITMY.p2 ETMY.p1 L=4k
m ETMY R=0.99 T=0.01 Rc=[YROC2, YROC2]

s sx BS.p3 ITMX.p1 L=10
m ITMX R=0.99 T=0.01 Rc=[XROC1, XROC1]
s LX ITMX.p2 ETMX.p1 L=4k
m ETMX R=0.99 T=0.01 Rc=[XROC2, XROC2]

cav cavXARM ITMX.p2
cav cavYARM ITMY.p2
""")
```

```
<finesse.model.Model at 0x7f1c62146900>
```

Note that the use of variables here for the radii of curvature of the mirrors allows us to ensure that these mirrors remain non-astigmatic; as the references will update for both planes whenever these variables change. We also separate these into X-arm and Y-arm variables such that updating one set doesn't affect the other arm cavity. If it was useful to do so, we could simply have a single pair of RoC values instead — thereby ensuring both arm cavities remain equivalent.

Each section heading below describes the task with the old 'trace' command equivalent in parentheses.

Listing TEM modes used ('trace 1')

One can simply query the modes included in a model with the `Model.modes()` function. In the case of our example above, we have:

```
print(model.modes())
```

```
[[0 0]]
```

Note that the existence of the `Cavity` (page 195) objects here automatically switches on the modal basis with only the fundamental (pure Gaussian) mode modelled. The modal mode can be explicitly turned off also using `Model.modes()`:

```
print(model.modes("off"))
```

```
None
```

If we select additional modes via `Model.modes()` and print the attribute again then we will see this reflected in the output:

```
model.modes(maxtem=2)
print(model.modes())
```

```
[[0 0]
 [1 0]
 [0 1]
 [2 0]
 [1 1]
 [0 2]]
```

See also

Selecting the modes to model (page 327) for a comprehensive guide on mode selection.

Accessing cavity parameters ('trace 2')

There are several ways to obtain properties of `Cavity` (page 195) objects. To display a verbose table of all the parameters (in their current state) associated with a cavity:

```
print(model.cavXARM.info_parameter_table())
```

```
┌──────────────────────┬──────────────────────────────────┐
│ Description          │ Value                            │
╞══════════════════════╪══════════════════════════════════╡
│ FSR                  │ 37474.05725                      │
├──────────────────────┼──────────────────────────────────┤
│ Loss                 │ 0.01990000000000003              │
├──────────────────────┼──────────────────────────────────┤
│ Finesse              │ 312.5845222828291                │
├──────────────────────┼──────────────────────────────────┤
│ FWHM                 │ 119.88455786717795               │
├──────────────────────┼──────────────────────────────────┤
│ Storage time         │ 0.002655136673535981             │
├──────────────────────┼──────────────────────────────────┤
│ Pole                 │ 59.94227893358897                │
├──────────────────────┼──────────────────────────────────┤
│ Round trip length    │ 8000.0                           │
├──────────────────────┼──────────────────────────────────┤
│ Waist size           │ [0.01195054 0.01195054]          │
├──────────────────────┼──────────────────────────────────┤
│ Waist position       │ [-1837.21538864 -1837.21538864]  │
├──────────────────────┼──────────────────────────────────┤
│ Stability (m-factor) │ [0.67018515 0.67018515]          │
├──────────────────────┼──────────────────────────────────┤
│ Stability (g-factor) │ [0.83509258 0.83509258]          │
```

(continues on next page)

(continued from previous page)

```
├──────────────────────┼──────────────────────────────────────┤
│ Round trip gouy phase │ [312.08135656 312.08135656]          │
├──────────────────────┼──────────────────────────────────────┤
│ Mode separation       │ [4988.07218818 4988.07218818]        │
├──────────────────────┼──────────────────────────────────────┤
│ Resolution            │ [41.60729519 41.60729519]            │
├──────────────────────┼──────────────────────────────────────┤
│ Stable                │ True                                 │
├──────────────────────┼──────────────────────────────────────┤
│ Critically stable     │ False                                │
└──────────────────────┴──────────────────────────────────────┘
```

In this case this displays the parameters associated with our `cavXARM` cavity.

For those parameters which have two values, e.g. the "waist size", the first value represents the quantity in the *tangential* plane whilst the second value is for the *sagittal* plane.

Accessing the actual values, rather than displaying them, is as simple as "calling" the relevant property of the `Cavity` (page 195) class. For example, if we want to get the eigenmode of `cavYARM` in the tangential plane and do something with the value (e.g. access the wavefront RoC of this mode), we can do:

```
qx_yarm = model.cavYARM.qx
# Do something the the BeamParam qx_yarm, e.g. get the RoC of the wavefront
print(qx_yarm.Rc)
```

```
-1934.0
```

Note

To obtain any of the parameters of the cavity during a simulation, use the `CavityPropertyDetector` (page 175).

Current mode mismatch parameters ('trace 4')

Printing the current mode mismatch state (see `BeamParam.mismatch()` for a description of the figure-of-merit mismatch quantity used) of a model is simple:

```
print(model.mismatches_table())
```

```
┌──────────┬──────────────┬──────────────┐
│ Coupling ╟ Mismatch (x) │ Mismatch (y) │
└──────────┴──────────────┴──────────────┘
```

Of course, this gives an empty table for this example currently as both arm cavities are equivalent in terms of geometric parameters and the interferometer is symmetric. For illustration, we can alter one of these cavities temporarily to show the resulting mode mismatch that then occurs at the central beam splitter (where the trace trees intersect):

```
# Temporarily change the RoC of ITMY using a context manager
with model.temporary_parameters():
    model.YROC1.value = -2000
    print(model.mismatches_table())
```

```
┌──────────────────┬──────────────┬──────────────┐
│     Coupling     ╟ Mismatch (x) │ Mismatch (y) │
╞══════════════════╬══════════════╪══════════════╡
│ BS.p1.i -> BS.p2.o ╟       0.0075 │       0.0075 │
```

(continues on next page)

(continued from previous page)

```
╞═══════════════════╪═══════════════╪═══════════════╡
│ BS.p2.i -> BS.p4.o ╫       0.0075 │       0.0075 │
╞═══════════════════╪═══════════════╪═══════════════╡
│ BS.p2.i -> BS.p1.o ╫       0.0075 │       0.0075 │
╞═══════════════════╪═══════════════╪═══════════════╡
│ BS.p4.i -> BS.p2.o ╫       0.0075 │       0.0075 │
╘═══════════════════╪═══════════════╪═══════════════╛
```

If you want to access the actual mismatch values themselves, you can use `Model.detect_mismatches()`. For example, we can access just the mismatch for the coupling `BS.p1.i -> BS.p2.o` with:

```
with model.temporary_parameters():
    model.spaces.LX.L = 4100
    mismatches = model.detect_mismatches()
    print(mismatches[(model.BS.p1.i, model.BS.p2.o)])
```

```
{'x': np.float64(0.04390880807812923), 'y': np.float64(0.04390880807812923)}
```

As you can see, this returns a dictionary of the mismatch values for this node coupling in both the tangential (x) and sagittal (y) planes.

Beam parameters for every node ('trace 8')

Displaying the values for the beam parameters at each node can be done via printing the `BeamTraceSolution` object — which is returned when performing a `Model.beam_trace()` and is also stored in the `Model.last_trace` attribute.

Again, using our example:

```
# Execute a beam trace using current model state
trace = model.beam_trace()
# Draw the trace solution in a forest format
print(trace)
```

```
Internal trace of cavity: cavXARM (q = -1837.22 + 421.68j, g = 0.8350925761717988)

    o ITMX.p2.o: [q = -1837.22 + 421.68j]
    └─o ETMX.p1.i: [q = 2162.78 + 421.68j]
       └─o ETMX.p1.o: [q = -2162.78 + 421.68j]
          └─o ITMX.p2.i: [q = 1837.22 + 421.68j]

Internal trace of cavity: cavYARM (q = -1837.22 + 421.68j, g = 0.8350925761717988)

    o ITMY.p2.o: [q = -1837.22 + 421.68j]
    └─o ETMY.p1.i: [q = 2162.78 + 421.68j]
       └─o ETMY.p1.o: [q = -2162.78 + 421.68j]
          └─o ITMY.p2.i: [q = 1837.22 + 421.68j]

Dependency: cavXARM

    o ETMX.p1.i: [q = 2162.78 + 421.68j]
    └─o ETMX.p2.o: [q = 2162.78 + 421.68j]

Dependency: cavXARM

    o ITMX.p2.i: [q = 1837.22 + 421.68j]
    └─o ITMX.p1.o: [q = 1837.22 + 421.68j]
       └─o BS.p3.i: [q = 1847.22 + 421.68j]
```

(continues on next page)

(continued from previous page)

```
          ├──o BS.p4.o: [q = 1847.22 + 421.68j]
          └──o BS.p1.o: [q = 1847.22 + 421.68j]
             └──o L0.p1.i: [q = 1847.22 + 421.68j]

Dependency: cavYARM

    o ETMY.p1.i: [q = 2162.78 + 421.68j]
    └──o ETMY.p2.o: [q = 2162.78 + 421.68j]

Dependency: cavYARM

    o ITMY.p2.i: [q = 1837.22 + 421.68j]
    └──o ITMY.p1.o: [q = 1837.22 + 421.68j]
       └──o BS.p2.i: [q = 1847.22 + 421.68j]
```

The order in which these solution trees are displayed corresponds exactly to the order in which the beam traces were performed on the model. The `TraceDependency` object associated with each trace tree is printed above the tree.

The `BeamTraceSolution` provides a dict-like interface for accessing beam parameters, and so obtaining such values is simple. For example, if we want the beam parameter at the laser output from our above `trace`:

```
# Get the beam parameters in both planes
qx, qy = trace[model.L0.p1.o]

# Equivalent but alternative syntax:
qx = trace[model.L0.p1.o].qx
qy = trace[model.L0.p1.o].qy

# As both qx and qy are BeamParam instances we can then access any
# beam parameter property of these or just print them, e.g:
print(qx)
```

```
BeamParam(w0=11.951 mm, z=-1.8472 km, w=53.697 mm, Rc=-1.9435 km)
```

Access Gouy phases for all spaces ('trace 16')

One can print all the Gouy phases accumulated over each individual space, using the current state of the model, via:

```
print(model.space_gouys_table())
```

```
┌──────────────┬──────────────────┬──────────────────┐
│    Space     │ Gouy (x) [deg]   │ Gouy (y) [deg]   │
╞══════════════╪══════════════════╪══════════════════╡
│ L0_p1__BS_p1 │           0.0000 │           0.0000 │
├──────────────┼──────────────────┼──────────────────┤
│      sy      │           0.0676 │           0.0676 │
├──────────────┼──────────────────┼──────────────────┤
│      LY      │         156.0407 │         156.0407 │
├──────────────┼──────────────────┼──────────────────┤
│      sx      │           0.0676 │           0.0676 │
├──────────────┼──────────────────┼──────────────────┤
│      LX      │         156.0407 │         156.0407 │
└──────────────┴──────────────────┴──────────────────┘
```

This shows each space in the model with the corresponding Gouy phases in both the tangential (x) and sagittal (y) planes. The values are in degrees by default, but can be switched to radians as shown by the parameter list of `Model.space_gouys_table()`.

If you want to access the actual Gouy phase values themselves, you can use `Model.compute_space_gouys()`. For example, we can access just the Gouy phase accumulated over the space `sy` with:

```
gouys = model.compute_space_gouys()
print(gouys[model.spaces.sy])
```

```
{'x': 0.0676471163098868, 'y': 0.0676471163098868}
```

See also

For tracking the accumulated Gouy phase over an arbitrary length sequence of spaces during a simulation, use the `Gouy` (page 177) detector.

And for outside of a simulation, you can use `Model.acc_gouy()` to calculate the Gouy phase accumulated over a given path. Another recommended method for doing this (and much more at the same time) is `Model.propagate_beam()` which returns a `PropagationSolution` containing methods and attributes for obtaining Gouy phases accumulated over paths and sub-paths; along with many more useful features.

Coupling coefficients of components ('trace 32')

The simplest way to obtain the values of coupling coefficients, for couplings at any component, is to add corresponding `KnmDetector` (page 177) instances to a model.

For example, let's misalign ITMX in our interferometer by rotating it by 0.5% of the cavity divergence angle in the yaw degree of freedom. Then we will add a `KnmDetector` (page 177) for probing the scattering matrix on reflection from this mirror at the inner-surface:

```
# Add a KnmDetector for K22 matrix of ITMX
model.parse("knmd K22_itmx ITMX 22")
# Temporarily misalign ITM.xbeta by 0.5% of the cavXARM divergence angle
with model.temporary_parameters():
    model.ITMX.xbeta = model.cavXARM.qx.divergence * 0.5 / 100
    # Run the model with no parameter sweep
    out = model.run()
```

Our `out` object now contains an entry `"K22_itmx"` which consists of the mode scattering matrix data for the specified coupling. We can wrap this up neatly using a `KnmMatrix` and print it, for example:

```
from finesse.knm.matrix import KnmMatrix

# The from_buffer method needs to know what higher-order modes were used so pass this as␣
↪second arg
K22_itmx = KnmMatrix.from_buffer(out["K22_itmx"], model.modes())

print(K22_itmx)
```

```
K[0][0]: 0 0 -> 0 0 = (0.9990013731832287-2.2182286522803378e-16j)
K[1][0]: 0 0 -> 1 0 = (6.938893903907228e-18+0.04465716662614314j)
K[2][0]: 0 0 -> 0 1 = 0j
K[3][0]: 0 0 -> 2 0 = (-0.0014115661870374835-1.0842021724855044e-19j)
K[4][0]: 0 0 -> 1 1 = (-0+0j)
K[5][0]: 0 0 -> 0 2 = 0j
K[0][1]: 1 0 -> 0 0 = (-2.0816681711721685e-17-0.04465716662614314j)
K[1][1]: 1 0 -> 1 0 = (-0.997005117137333+4.427592146859873e-16j)
K[2][1]: 1 0 -> 0 1 = 0j
K[3][1]: 1 0 -> 2 0 = (3.469446951953614e-18-0.06309167114051635j)
K[4][1]: 1 0 -> 1 1 = (-0+0j)
K[5][1]: 1 0 -> 0 2 = 0j
K[0][2]: 0 1 -> 0 0 = 0j
K[1][2]: 0 1 -> 1 0 = (-0+0j)
```

(continues on next page)

(continued from previous page)

```
K[2][2]: 0 1 -> 0 1 = (0.9990013731832287-4.4364573045606756e-16j)
K[3][2]: 0 1 -> 2 0 = 0j
K[4][2]: 0 1 -> 1 1 = (3.642919299551295e-17+0.044657166626143135j)
K[5][2]: 0 1 -> 0 2 = 0j
K[0][3]: 2 0 -> 0 0 = (-0.0014115661870374838+7.589415207398531e-19j)
K[1][3]: 2 0 -> 1 0 = (3.122502256758253e-17+0.06309167114051635j)
K[2][3]: 2 0 -> 0 1 = 0j
K[3][3]: 2 0 -> 2 0 = (0.9950108556023098+0j)
K[4][3]: 2 0 -> 1 1 = (-0+0j)
K[5][3]: 2 0 -> 0 2 = 0j
K[0][4]: 1 1 -> 0 0 = 0j
K[1][4]: 1 1 -> 1 0 = (-0+0j)
K[2][4]: 1 1 -> 0 1 = (-2.0816681711721685e-17-0.04465716662614314j)
K[3][4]: 1 1 -> 2 0 = 0j
K[4][4]: 1 1 -> 1 1 = (-0.997005117137333+8.855184293719746e-16j)
K[5][4]: 1 1 -> 0 2 = 0j
K[0][5]: 0 2 -> 0 0 = 0j
K[1][5]: 0 2 -> 1 0 = (-0+0j)
K[2][5]: 0 2 -> 0 1 = 0j
K[3][5]: 0 2 -> 2 0 = 0j
K[4][5]: 0 2 -> 1 1 = (-0+0j)
K[5][5]: 0 2 -> 0 2 = (0.9990013731832289+0j)
```

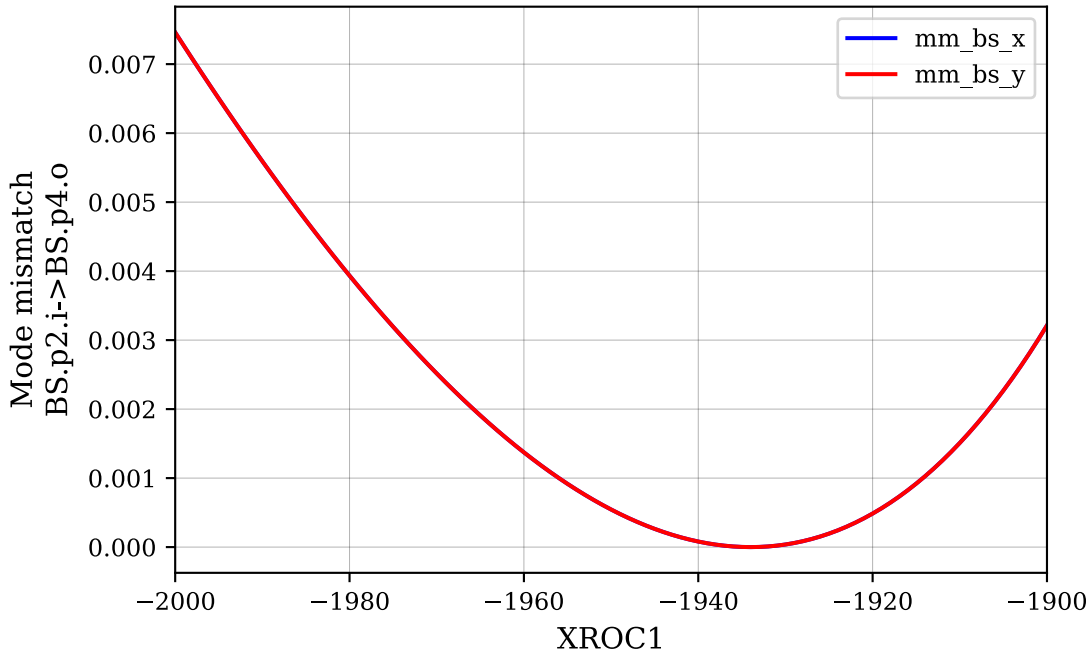
The displayed output here gives the coupling coefficients for the modes `n m -> n' m'` as shown.

Mode mismatches during a simulation ('trace 64')

Obtaining mode mismatch values during a simulation can be achieved via the use of `ModeMismatchDetector` (page 178) objects. Using our interferometer above, we can, for example, scan over the RoC of ITMX during a simulation and detect the corresponding mode mismatch which occurs from port 2 to port 4 of the beam splitter (due to the intersecting trace trees from `cavXARM` and `cavYARM` at this component):

```
model.parse("""
# This will detect mode mismatch from BS.p2.i -> BS.p4.o
mmd mm_bs_x BS.p2.i BS.p4.o direction=x
mmd mm_bs_y BS.p2.i BS.p4.o direction=y
""")

# Sweep the RoC of the ITM in the x-arm
out = model.run("xaxis(XROC1, lin, -1900, -2000, 100)")
out.plot("mm_bs_x", "mm_bs_y");
```



The resulting plot shows the requested mode mismatch quantity as a function of the swept parameter. One can specify the plane to detect in via the `direction` argument of this detector, as described in `ModeMismatchDetector` (page 178). The mismatch quantities are the same in both planes in this case as scanning `XROC1` changes both the tangential and sagittal planes of the RoC of the ITM in the X-arm.

Nodes found during the beam tracing ('trace 128')

This option was already essentially covered by the beam trace solution printing section earlier, however one can also access the underlying `TraceForest` structure of a model itself via the `Model.trace_forest` attribute. You can print this to get a representation of how the beam traces are structured in the current model state:

```
print(model.trace_forest)
```

```
Dependency: cavXARM [Cavity]

  o ITMX.p2.o
  └─o ETMX.p1.i
    └─o ETMX.p1.o
      └─o ITMX.p2.i

Dependency: cavYARM [Cavity]

  o ITMY.p2.o
  └─o ETMY.p1.i
    └─o ETMY.p1.o
      └─o ITMY.p2.i

Dependency: cavXARM [Cavity]

  o ETMX.p1.i
  └─o ETMX.p2.o

  o ITMX.p2.i
  └─o ITMX.p1.o
    └─o BS.p3.i
      ├─o BS.p4.o
      └─o BS.p1.o
```

(continues on next page)

(continued from previous page)

```
           └──o L0.p1.i

Dependency: cavYARM [Cavity]

  o ETMY.p1.i
  └──o ETMY.p2.o

  o ITMY.p2.i
  └──o ITMY.p1.o
     └──o BS.p2.i
```

This gives all of the trace trees (with the associated optical nodes) in the order in which beam traces will traverse them during the beam tracing algorithms.

One can easily change the priority of `TraceDependency` instances of the model and see how this is reflected in the new forest structure. For example, here we set `cavYARM` as the highest priority now such that this will be traced before `cavXARM`:

```
# Gives cavYARM the highest priority (one can also set
# priority manually via model.cavYARM.priority = value)
model.cavYARM.take_priority()

# We can check the tracing order via this method call
print("Trace order = ", model.trace_order_names)

# Perform a new beam trace so that trace_forest gets re-planted
model.beam_trace()

print(model.trace_forest)
```

```
Trace order =  ['cavYARM', 'cavXARM']

Dependency: cavYARM [Cavity]

  o ITMY.p2.o
  └──o ETMY.p1.i
     └──o ETMY.p1.o
        └──o ITMY.p2.i

Dependency: cavXARM [Cavity]

  o ITMX.p2.o
  └──o ETMX.p1.i
     └──o ETMX.p1.o
        └──o ITMX.p2.i

Dependency: cavYARM [Cavity]

  o ETMY.p1.i
  └──o ETMY.p2.o

  o ITMY.p2.i
  └──o ITMY.p1.o
     └──o BS.p2.i
        ├──o BS.p1.o
        │  └──o L0.p1.i
        └──o BS.p4.o

Dependency: cavXARM [Cavity]
```

(continues on next page)

(continued from previous page)

```
  o ETMX.p1.i
  └──o ETMX.p2.o

  o ITMX.p2.i
  └──o ITMX.p1.o
     └──o BS.p3.i
```

And now we can see that the ordering has changed as expected.

9.3.3 Migrating from Pykat + Finesse 2

Many of the features from Pykat have been integrated directly into FINESSE 3 and in some cases, expanded. This leads to some important difference in the usual recipes to produce an interferometer and start modelling, which are summarized below.

Creating a model

The `pykat.finesse.Kat` class has been replaced with the `finesse.Model`. Since FINESSE 2 style KatScript is not the primary way to build new models now, you also have to call `parse_legacy` to parse FINESSE 2 style scripts. For example:

```
from finesse import Model

# Create the empty model.
model = Model()

# Parse some components into the model.
model.parse_legacy(
    """
    l laser 2.5 0.0 0.0 input
    s s1 1 input n1
    m ITM 0.8 0.2 0.0 n1 n2
    s s2 70 n2 n3
    m ETM 0.999 1e-05 0.0 n3 n4
    xaxis ETM phi lin -95 95 1000
    """
)
```

```
<finesse.model.Model at 0x7eac84363e00>
```

There are two available functions for parsing FINESSE 2 style KatScript: `Model.parse_legacy()` and `Model.parse_legacy_file()`. There are also two similar functions for parsing the newer FINESSE 3 style script `Model.parse()` and `Model.parse_file()`.

> **Warning**
>
> Not all functions are supported by this legacy parser! See Finesse 2 support (page 542) for more details.

Major syntax changes

There have been a number of syntax changes and improvements in FINESSE 3. Instead of an object's optional attributes being defined via the `attr` command, it is now possible to define all attributes on the object definition line. This also means the numbered component types like `m1`, `bs2`, etc. are no longer needed. These changes mean the following code:

```
m1 ETM 5e-06 5e-05 0.0 nETM1 nETM2
attr EMAx Rcx 25
attr EMAx Rcy 20
```

can be replaced with:

```
m ETM T=1e-5 L=1e-6 Rc=10
```

Another new feature of Finesse 3 syntax is that it allows the use of arbitrary symbolic expressions with references to other parameters. This removes the need for the Finesse 2 `set`, `func` and `put` commands. An example of this syntax in the Finesse 2 (Section 3.5 of [41]) is shown in Finesse 3 form in this example (page 78).

See also

See KatScript (page 125) for more details.

The port and node system

In Finesse 2, each node needs to be uniquely and explicitly named. This requirement has been overcome in Finesse 3. Components now have one or more ports. For example, a mirror has two optical ports, corresponding to each surface. Each port then has multiple nodes. In the case of a mirror, each optical port has two nodes, corresponding to the input and output field at each side of the mirror.

```
model = Model()

# Create a laser and mirror.
model.parse(
    """
    l L0 P=1
    s s1 L0.p1 ETM.p1
    m ETM T=1e-5 L=1e-6 Rc=10
    """
)

# Grab the ETM nodes.
model.ETM.nodes
```

```
OrderedDict([('ETM.p1.i', <OpticalNode ETM.p1.i @ 0x7eac84153770>),
             ('ETM.p1.o', <OpticalNode ETM.p1.o @ 0x7eac84153850>),
             ('ETM.p2.i', <OpticalNode ETM.p2.i @ 0x7eac841a8120>),
             ('ETM.p2.o', <OpticalNode ETM.p2.o @ 0x7eac8415d610>),
             ('ETM.mech.z', <SignalNode ETM.mech.z @ 0x7eac8415d6d0>),
             ('ETM.mech.yaw', <SignalNode ETM.mech.yaw @ 0x7eac8415d790>),
             ('ETM.mech.pitch', <SignalNode ETM.mech.pitch @ 0x7eac8415d850>),
             ('ETM.mech.F_z', <SignalNode ETM.mech.F_z @ 0x7eac8415d910>),
             ('ETM.mech.F_yaw', <SignalNode ETM.mech.F_yaw @ 0x7eac8415d9d0>),
             ('ETM.mech.F_pitch',
              <SignalNode ETM.mech.F_pitch @ 0x7eac8415da90>)])
```

See also

See The port and node system (page 139) for more details.

Inline expressions

In FINESSE 3, expressions and functions can be specified inside parameter definitions. This removes the need for the `func` command:

```
m m2 R=m1.R*0.5 T=1-m1.R*0.5 phi=cos(l1.P/10)
```

Variables, constants and references

In FINESSE 2, in order to make a parameter of one component track another, a combination of *set* and *put* commands would have to be used:

```
m m1 0.99 0.01 0 n1 n2
m m2 0.5 0.5 0 n3 n4

set refl m1 R
set trans m1 T
put m2 R $refl
put m2 T $trans
```

In FINESSE 3, parameters can be referenced directly using their path (page 128):

```
m m1 0.99 0.01
m m2 R=m1.R T=m1.T
```

Running a model

Similar to Pykat, models can be run by calling the `run()` method on the model object. As before, this produces an output object, which can be plotted.

```
out = model.run()
```

Detector argument order

The order of arguments in the definition of detectors has changed. The nodes are now specified after the name of the detector, rather than at the end of the command.

Automatic conversion of Finesse 2 scripts

The **`kat3 convert`** command line program (page 384) bundled with FINESSE provides the ability to convert FINESSE 2 style KatScript to FINESSE 3 style. Note however that this is an **experimental** feature and does not support all forms of syntax yet. See its documentation (page 388) for more information.

Chapter 10

Index

- General
- KatScript commands
- KatScript elements
- KatScript analyses

Chapter 11

Living Review

The Living Review provides a comprehensive overview of interferometer techniques for gravitational-wave detection. It gives a textbook-style introduction to the optical science required for understanding modern gravitational-wave detectors, as well as other high-precision laser interferometers.

The article contains a number of FINESSE examples which are in a now outdated format (FINESSE version 2). The same examples are provided in FINESSE 3 syntax below.

11.1 2.7.1 Mirror reflectivity and transmittance

We use Finesse to plot the amplitudes of the light fields transmitted and reflected by a `mirror` (page 142) (given by a single surface). Initially, the mirror has a power reflectance and transmittance of $R = T = 0.5$ and is, thus, lossless. For the plot in Fig. 15 (generated here) we tune the transmittance from 0.5 to 0. Since we do not explicitly change the reflectivity, R remains at 0.5 and the mirror loss increases instead, which is shown by the trace labelled 'total' corresponding to the sum of the reflected and transmitted light power. The plot also shows the phase convention of a $90°$ phase shift for the transmitted light.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                # laser with P=1W at default frequency
    s s1 l1.p1 m1.p1 L=1    # space of 1m length
    m m1 R=0.5 T=0.5        # mirror with R=T=0.5 at zero tuning

    ad ad_trns m1.p2.o f=0  # an amplitude detector for transmitted light
    ad ad_refl m1.p1.o f=0  # an amplitude detector for reflected light

    # computing the sum of reflected and transmitted power
    mathd total abs(ad_refl)**2+abs(ad_trns**2) unit='W' label='Total power'
    """)

# changing the transmittance of the mirror 'm1'
out = base.run(Xaxis(base.m1.T, 'lin', 0, 0.5, 100))

fig, ax = plt.subplots(2, 1, figsize=(8, 7))
ax[0].plot(out.x1, np.abs(out['ad_trns']), label=r'ad_trns [$\sqrt{\mathrm{W}}$]')
ax[0].plot(out.x1, np.abs(out['ad_refl']), ls='--',
```

(continues on next page)

(continued from previous page)

```
           label=r'ad_refl [$\sqrt{\mathrm{W}}$]')
ax[0].plot(out.x1, out['total'], ls='--', label=r'total power [$\mathrm{W}$]')
ax[0].set_ylabel('Abs')

ax[1].plot(out.x1, np.degrees(np.angle(out['ad_trns'])), label='ad_trns')
ax[1].plot(out.x1, np.degrees(np.angle(out['ad_refl'])), ls='--', label='ad_refl')
ax[1].set_ylim(-10, 110)
ax[1].set_ylabel(r'Phase $[\degree]$')
ax[1].set_xlabel(r'T (m1)')

for a in ax:
    a.invert_xaxis()
    a.legend()
plt.show()
```

```
<Figure size 768x672 with 2 Axes>
```

11.2 2.7.2 Mirror tuning

This example demonstrates the conventions for lengths and microscopic positions introduced in Sect. 2.5. The top trace in Fig. 16 (generated here) depicts the phase change of a beam reflected by a `beamsplitter` (page 149) as the function of the beam splitter tuning. By changing the tuning from 0 to 180° the beam splitter is moved forward and shortens the path length by one wavelength, which by convention increases the light phase by 360°. On the other hand, if a length of a space is changed, the phase of the transmitted light is unchanged (for the default wavelength $\Delta k = 0$), as shown in the lower trace.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                    # laser with P=1W at default frequency
    s s1 l1.p1 bs1.p1 L=1       # space of 1m length
    bs bs1                      # beam splitter as 'turning mirror', normal incidence
    s s2 bs1.p2 nothing1.p1 L=1 # another space of 1m length
    nothing nothing1            # a 'nothing' element to connect the detector

    ad ad_r nothing1.p1.i f=0   # amplitude detector at default frequency
    """
)

out = base.run(Xaxis(base.bs1.phi, 'lin', 0, 180, 100))
out2 = base.run(Xaxis(base.s1.L, 'lin', 1, 2, 100))

fig, ax = plt.subplots(2, 1, figsize=(8, 7))
ax[0].plot(out.x1, np.degrees(np.angle(out['ad_r'])))
ax[0].set_xlabel(r'phi (bs1) [$\degree$]')
ax[0].set_ylabel(r'Phase [$\degree$]')

ax[1].plot(out2.x1, np.degrees(np.angle(out2['ad_r'])))
ax[1].set_xlabel(r'L (s1) [m]')
ax[1].set_ylabel(r'Phase [$\degree$]')
plt.show()
```

```
<Figure size 768x672 with 2 Axes>
```

11.3 3.8.1 Modulation index

This file demonstrates the use of a `modulator` (page 159). Phase modulation with up to five higher harmonics is applied to a laser beam and amplitude detectors are used to measure the field at the first three harmonics. Compare this to Fig. 18 as well (Fig. 22, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1            # laser with P=1W at default frequency
    s s1 l1.p1 eom1.p1  # space of zero length to connect laser and modulator

    # Phase modulator at 40kHz, with a modulation index of 0.05
    mod eom1 f=40k midx=.05 order=5 mod_type=pm

    # Three amplitude detectors, looking at the first three positive modulation
    # sidebands
    ad bessel1 eom1.p2.o f=40k
    ad bessel2 eom1.p2.o f=80k
    ad bessel3 eom1.p2.o f=120k
    """
)

out = base.run(Xaxis(base.eom1.midx, 'lin', 0, 10, 300))

plt.figure(figsize=(8, 4))
plt.plot(out.x1, np.abs(out['bessel1']), label='bessel1')
plt.plot(out.x1, np.abs(out['bessel2']), label='bessel2')
plt.plot(out.x1, np.abs(out['bessel3']), label='bessel3')
plt.xlabel('modulation index (eom1)')
plt.ylabel(r'amplitude [$\sqrt{\mathrm{W}}$]')
plt.legend()
plt.show()
```

```
<Figure size 768x384 with 1 Axes>
```

11.4 3.8.2 Mirror modulation

Finesse offers two different types of modulators: the `modulator` (page 159) component shown in the example above, and the `fsig` (page 196) command, which can be used to apply a signal modulation to existing optical components. The main difference is that `fsig` (page 196) is meant to be used for transfer function computations. Consequently Finesse discards all nonlinear terms, which means that the sideband amplitude is proportional to the signal amplitude and harmonics are not created (Fig. 23, generated here).

Note

In Finesse 2, the amplitude of a signal on a mechanical `z` node of a mirror or beamsplitter was scaled from meters to radians. The conversion factor is the reference wavenumber, which is equal to $k_0 = 5.91 \times 10^6 \mathrm{m}^{-1}$ for a reference wavelength $\lambda_0 = 1064\mathrm{nm}$.

Furthermore, the amplitude detector at a frequency offset of $f = 80$kHz is removed as amplitude detectors in Finesse 3 raise an exception when set to a frequency which is not present in the signal.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                    # laser with P=1W at default frequency
    s s1 l1.p1 bs1.p1 L=1   # space of 1m length
    bs bs1                  # beam splitter as 'turning mirror', normal incidence

    fsig(40k)               # set signal frequency to 40kHz
    sgen sig1 bs1.mech.z    # signal modulation applied to beam splitter

    # Amplitude detectors looking at lower and upper sidebands, and upper first harmonic
    ad upper bs1.p2.o f=40k
    ad lower bs1.p2.o f=-40k
    """
)

out = base.run(Xaxis(base.sig1.amplitude, 'lin', 1, 10, 100))

plt.figure(figsize=(8, 4))
plt.plot(out.x1, np.abs(out['upper']), label='upper')
plt.plot(out.x1, np.abs(out['lower']), ls='--', label='lower')
plt.xlabel('signal amplitude (bs1.mech.z)')
plt.ylabel(r'amplitude [$\sqrt{\mathrm{W}}$]')
plt.show()
```

```
<Figure size 768x384 with 1 Axes>
```

11.5 4.3.1 Optical beat

In this example two laser beams are superimposed at a $50:50$ beamsplitter (page 149). The beams have a slightly different frequency: the second beam has a 10 kHz offset with respect to the first (and to the default laser frequency). The plot illustrates the output of four different detectors in one of the beam splitter output ports, while the phase of the second beam is tuned from $0°$ to $180°$. The photodiode 'pd1' shows the total power remaining constant at a value of 1. The amplitude detectors 'ad1' and 'ad10k' detect the laser light at 0 Hz (default frequency) and 10 kHz respectively. Both show a constant absolute of $\sqrt{1/2}$ and the detector 'ad10k' tracks the tuning of the phase of the second laser beam. Finally, the detector 'pd10k' resembles a photodiode with demodulation at 10 kHz. In fact, this represents a photodiode and two mixers used to reconstruct a complex number as shown in Eq. (4.15). One can see that the phase of the resulting electronic signal also directly follows the phase difference between the two laser beams (Fig. 25).

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
```

(continues on next page)

(continued from previous page)

```
    var freq 10k                    # creating a parameter for the frequency offset
    l l1 P=1                        # laser with P=1W at the default frequency
    s s1 l1.p1 bs1.p1 L=1n          # space of 1nm length
    l l2 P=1 f=freq                 # a second laser with f=10kHz frequency offset
    s s2 l2.p1 bs1.p4 L=1           # another space of 1nm length
    bs bs1                          # 50:50 beam splitter

    ad ad1 bs1.p2.o f=0             # amplitude detector at default frequency
    ad ad10 bs1.p2.o f=freq         # amplitude detector at 10kHz frequency offset
    pd pd1 bs1.p2.o                 # simple photodetector
    pd1 pd10 bs1.p2.o f=freq        # photo detector with demodulation at 10kHz
    """
)

out = base.run(Xaxis(base.l2.phase, 'lin', 0, 180, 100))

fig, ax = plt.subplots(2, 1, figsize=(8, 7))
ax[0].plot(out.x1, np.abs(out['pd1']), label=r'pd1 [W]')
ax[0].plot(out.x1, np.abs(out['pd10']), label=r'pd10 [W]')
ax[0].plot(out.x1, np.abs(out['ad1']), label=r'ad1 [$\sqrt{\mathrm{W}}$]')
ax[0].plot(out.x1, np.abs(out['ad10']), ls='--', label=r'ad10k [$\sqrt{\mathrm{W}}$]')
ax[0].set_ylim(0, 1.1)
ax[0].set_xlabel(r'phi (l2) [$\degree$]')
ax[0].set_ylabel(r'Amplitude')
ax[0].legend()

ax[1].plot(out.x1, np.degrees(np.angle(out['pd1'])), label=r'pd1 [W]')
ax[1].plot(out.x1, np.degrees(np.angle(out['pd10'])), label=r'pd10 [W]')
ax[1].plot(out.x1, np.degrees(np.angle(out['ad1'])),
           ls='--', label=r'ad1 [$\sqrt{\mathrm{W}}$]')
ax[1].plot(out.x1, np.degrees(np.angle(out['ad10'])),
           ls='--', label=r'ad10k [$\sqrt{\mathrm{W}}$]')
ax[1].set_xlabel(r'L (s1) [m]')
ax[1].set_ylabel(r'Phase [$\degree$]')
ax[1].legend()
plt.show()
```

```
<Figure size 768x672 with 2 Axes>
```

11.6 5.6.1 Cavity power

This is a simple Finesse example showing the power enhancement in a two-mirror cavity as a function of the microscopic tuning of a mirror position (the position is given in degrees with 360° referring to a change of longitudinal position by one wavelength). Compare this plot to the one shown in Fig. 27, which instead shows the power enhancement as a function of the laser frequency detuning (Fig. 34, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                    # laser with P=1W at the default frequency
    s s1 l1.p1 m1.p1 L=1        # space of 1m length connecting laser to input mirror
    m m1 R=0.9 T=0.1            # cavity input mirror
    s scav m1.p2 m2.p1 L=1200   # cavity length of 1200m
    m m2 R=1 T=0                # cavity output mirror
```

(continues on next page)

(continued from previous page)

```
    pd pd1 m2.p1.i              # photodiode measuring the intra-cavity power
    """
)

out = base.run(Xaxis(base.m2.phi, 'lin', -50, 250, 300))
out.plot(logy=True);
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.7 5.6.2 Michelson power

The power in the South port of a Michelson detector varies as the cosine squared of the microscopic arm length difference. The maximum output can be equal to the input power, but only if the Michelson interferometer is symmetric and lossless. The tuning for which the South port power is zero is referred to as the dark fringe (Fig. 35, generated here).

```
import finesse
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                        # laser with P=1W at the default frequency
    s s1 l1.p1 bs1.p1 L=1           # space of 1m length
    bs bs1 R=0.5 T=0.5              # 50:50 beam splitter

    # The x-arm
    m ETMx R=1 T=0                  # east end mirror, lossless
    s Lx bs1.p3 ETMx.p1 L=3k        # east arm of 3km length

    # The y-arm
    m ETMy R=1 T=0                  # north end mirror, lossless
    s Ly bs1.p2 ETMy.p1 L=3k        # north arm of 3km length

    pd pd1 bs1.p4.o                 # photodetector in south port
    """
)

# using a series action to change the microscopic position of the north end mirror for
# three different reflectances and transmittances of the beam splitter. Note that the
# order of R and T inside the 'change' action is important as the parameters are changed
# from left to right and R + T + L may never exceed 1.
out = base.run("""
    series(
        xaxis(ETMy.phi, lin, 0, 300, 100),
        change(bs1.R=0.4, bs1.T=0.6),
        xaxis(ETMy.phi, lin, 0, 300, 100),
        change(bs1.T=0.45, bs1.R=0.45),
        xaxis(ETMy.phi, lin, 0, 300, 100)
    )
""")

plt.figure()
plt.plot(out[0].x1, out[0]['pd1'], label='symmetric, lossless')
plt.plot(out[1].x1, out[1]['pd1'], label='asymmetric, lossless')
plt.plot(out[2].x1, out[2]['pd1'], label='symmetric, lossy')
plt.xlabel('North mirror tuning [deg]')
plt.ylabel('South power [W]')
```

(continues on next page)

(continued from previous page)

```
plt.legend()
plt.show()
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.8 5.6.3 Michelson gravitational wave response

This is a simple Finesse example showing how the arm spaces can be modulated to produce the effect a gravitational wave would have on it. It outputs the amplitude and phase of the upper sideband that reaches the output port. The dips in amplitude occur when the travel time of the photons along the interferometer arms equals one gravitational wave period and hence the signal accumulated in the first and second half of the travel time cancel each other (the plot above does not have enough resolution to show that the dips indicate zero signal, the non-zero amplitudes are an artefact of the numerical plotting routine). In this example the frequencies of the dips are given as $f = N \cdot c/1200\text{m} = N \cdot 250\text{kHz}$, with N a positive integer (Fig. 36, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                        # laser with P=1W at the default frequency
    s s1 l1.p1 bs1.p1 L=1           # space of 1m length
    bs bs1 R=0.5 T=0.5              # 50:50 beam splitter

    # The x-arm
    m ETMx T=100e-6 L=0             # east end mirror, lossless
    s Lx bs1.p3 ETMx.p1 L=600       # east arm of 600m length

    # The y-arm
    m ETMy T=100e-6 L=0 phi=90      # north end mirror, lossless
    s Ly bs1.p2 ETMy.p1 L=600       # north arm of 600m length

    # apply signal to each arm, 180 degrees out of phase
    fsig(1)
    sgen sig1 Lx.h
    sgen sig2 Ly.h phase=180

    ad ad1 bs1.p4.o fsig.f          # amplitude detector in south port
    """
)

out = base.run(Xaxis(base.fsig.f, 'log', 1e3, 1e6, 4000))
out.plot(log=True);
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.9 6.7.1 Optical spring

A simple example for an optical spring in a two mirror cavity with suspended mirrors. The cavity is slightly detuned which is required for creating the spring. The output is the motion of a mirror while it is excited with a force of constant amplitude. The resonance feature shown in Fig. 46 (generated here) is the result of the optomechanical coupling of the mirror with the cavity field.

Note

The code below is a direct translation of the |Finesse| 2 code in the living review, using a `sgen` (page 169) and a `xd` (page 179) element. Transfer functions can be calculated more easily in |Finesse| 3 using the `freqresp` (page 221) action. See Optical springs (page 90) for a similar example using the frequency response action.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt

finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                        # laser with P=1W at the default frequency
    s s1 l1.p1 ITM.p1 L=1           # space connecting laser to input mirror
    m ITM R=0.99 L=0                # cavity input mirror
    s scav ITM.p2 ETM.p1 L=1        # cavity of 1m length
    m ETM R=1 L=0 phi=-0.048        # end mirror with roughly 40kHz detuning
    pendulum sus1 ETM.mech mass=1   # suspend end mirror on pendulum

    # Apply a force signal to the end mirror
    fsig(1)
    sgen force ETM.mech.F_z

    # a detector to measure the longitudinal motion signal of the end mirror
    xd ETM_z ETM.mech.z
    """
)

out = base.run(Xaxis(base.fsig.f, "log", 1, 100, 10000))
tf, phase = np.abs(out["ETM_z"]), np.degrees(np.angle(out["ETM_z"]))

fig, ax1 = plt.subplots()

ax1.loglog(out.x1, tf, label="Abs")
ax1.set_ylim(1e-6, 1e1)
ax1.set_xlabel("f [Hz]")
ax1.set_ylabel("TF [m/N]")
ax1.tick_params(axis="y", labelleft=True)

ax2 = ax1.twinx()
ax2.semilogx(out.x1, phase, ls="--", c="green", label="Phase")
ax2.set_ylim(-30, 210)
ax2.set_ylabel("Phase [deg]")
ax2.tick_params(axis="y", labelright=True)

lines1, labels1 = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()

ax1.legend(lines1 + lines2, labels1 + labels2, loc="upper left")
plt.show()
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.10 6.7.2 Homodyne detector and squeezed light

Note

The homodyne detector has not yet been implemented in Finesse 3. At the current time this example cannot be reproduced.

11.11 6.7.3 Quantum-noise limited interferometer sensitivity

This example shows the quantum-noise limited sensitivity of an advanced detectors. See Appendix B for the optical layout of the detector and Sect. 8.12 for more details about the interferometer operation. The model is loosely based on the Advanced LIGO design file and thus we expect to see the peak sensitivity around 100 Hz at a sensitivity of about $10^{-23}/\sqrt{\text{Hz}}$. We can see the both the `qnoised` (page 181) and `qshot` (page 182) detectors agree at high frequencies, where the sensitivity is purely limited by shot noise. At low frequencies the two traces differ because only `qnoised` (page 181) takes into account the radiation pressure effects (Fig. 48, generated here).

The Finesse input file for this example is more complex than for other examples because it contains a more complex interferometer setup and uses relatively advanced concepts such as setting mechanical transfer function. See Appendix A for more information on Finesse and where to find the documentation, such as the syntax reference, required to follow this example.

```
import finesse
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    ###############################################################################
    ###   Variables
    ###############################################################################
    var Larm 3995
    var Mtm  40
    var itmT 0.014
    var lmichx 4.5
    var lmichy 4.45

    ###############################################################################
    ###   Input optics
    ###############################################################################
    l L0 125
    s l_in L0.p1 prm.p1
    # Power recycling mirror
    m prm T=0.03 L=37.5u phi=90
    s prc prm.p2 bs.p1 L=53


    # Central beamsplitter
    bs bs T=0.5 L=0 alpha=45

    ###############################################################################
    ###   X arm
    ###############################################################################
    s lx bs.p3 itmx.p1 L=lmichx
    m itmx T=itmT L=37.5u phi=90
    s LX itmx.p2 etmx.p1 L=Larm
    m etmx T=5u L=37.5u phi=89.999875

    pendulum itmx_sus itmx.mech mass=Mtm fz=1 Qz=1M
    pendulum etmx_sus etmx.mech mass=Mtm fz=1 Qz=1M
```

(continues on next page)

(continued from previous page)

```
    ###############################################################################
    ###   Y arm
    ###############################################################################
    s ly bs.p2 itmy.p1 L=lmichy
    m itmy T=itmT L=37.5u phi=0
    s LY itmy.p2 etmy.p1 L=Larm
    m etmy T=5u L=37.5u phi=0.000125

    pendulum itmy_sus itmy.mech mass=Mtm fz=1 Qz=1M
    pendulum etmy_sus etmy.mech mass=Mtm fz=1 Qz=1M

    ###############################################################################
    ###   Output and squeezing
    ###############################################################################
    s src bs.p4 srm.p1 L=50.525
    m srm T=0.2 L=37.5u phi=-90

    # A squeezed source could be injected into the dark port
    sq sq1 db=0 angle=90
    s lsqz sq1.p1 srm.p2

    # Differentially modulate the arm lengths
    fsig(1)
    sgen darmx LX.h
    sgen darmy LY.h phase=180

    # Output the full quantum noise limited sensitivity
    qnoised NSR_with_RP srm.p2.o nsr=True
    # Output just the shot noise limited sensitivity
    qshot NSR_without_RP srm.p2.o nsr=True

    # We could also display the quantum noise and the signal
    # separately by uncommenting these two lines.
    # qnoised noise srm.p2.o
    # pd1 signal srm.p2.o f=fsig
    """
)

out = base.run("xaxis(darmx.f, log, 5, 5k, 100)")
out.plot(log=True, separate=False);
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.12 7.5.1 Michelson interferometer with arm cavities

This example shows how to setup a Michelson interferometer, tune it to the dark fringe and compute a transfer function from the differential length change to the output signal, using the sideband amplitude for simplicity (Fig. 57, generated here).

Note

In Finesse 2, the amplitude of a signal on a mechanical `z` node of a mirror or beamsplitter was scaled from meters to radians. The conversion factor is the reference wavenumber, which is equal to $k_0 = 5.91 \times 10^6 \mathrm{m}^{-1}$ for a reference wavelength $\lambda_0 = 1064$ nm.

After rescaling, the amplitude generated below is a factor of 2 larger than the result shown in Figure 57. The result produced here agrees with the result obtained using recent Finesse 2 code.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                            # laser with P=1W at the default frequency
    s s1 l1.p1 MPR.p1 L=1           # space of 1m length
    m MPR R=0 T=1                   # power recycling mirror
    s s2 MPR.p2 bs1.p1 L=50         # space of 50m length
    bs bs1                          # 50:50 beam splitter

    # north arm
    s s3 bs1.p2 ITMy.p1 L=7         # space of 7m length to north arm
    m ITMy R=0.99 T=0.01            # north input mirror, lossless
    s Ly ITMy.p2 ETMy.p1 L=4k       # north arm of 4km length
    m ETMy R=1 T=0                  # north end mirror, lossless

    # east arm
    s s4 bs1.p3 ITMx.p1 L=7         # space of 7m length to east arm
    m ITMx R=0.99 T=0.01 phi=90     # east end mirror, lossless
    s Lx ITMx.p2 ETMx.p1 L=4k       # east arm of 4km length
    m ETMx R=1 T=0 phi=90           # east end mirror, lossless

    # apply signal to north and east mirror positions, 180 degrees out of phase
    fsig(100)
    sgen sig1 ETMx.mech.z
    sgen sig2 ETMy.mech.z phase=180

    ad signal bs1.p4.o fsig.f       # amplitude detector in south port
    """
)

out = base.run(Xaxis(base.fsig.f, 'log', 1, 1e4, 300))

plt.figure(figsize=(8, 4))
plt.semilogx(out.x1, np.abs(out['signal']))
plt.xlabel(r'$f$ [Hz]')
plt.ylabel(r'Amplitude [$\sqrt{\mathrm{W}}$]')
plt.show()
```

```
<Figure size 768x384 with 1 Axes>
```

11.13 7.5.2 Michelson with signal recycling

This example recreates the plot shown in Fig. 54, the four traces show the transfer function for a Michelson interferometer with different signal recycling tunings (Fig. 58, generated here).

Note

In Finesse 2, the amplitude of a signal on a mechanical `z` node of a mirror or beamsplitter was scaled from meters to radians. The conversion factor is the reference wavenumber, which is equal to $k_0 = 5.91 \times 10^6 \mathrm{m}^{-1}$ for a reference wavelength $\lambda_0 = 1064$ nm.

```
import finesse
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                            # laser with P=1W at the default frequency
    s s1 l1.p1 MPR.p1 L=1           # space of 1m length
    m MPR R=0 T=1                   # power recycling mirror
    s s2 MPR.p2 bs1.p1 L=50         # space of 50m length
    bs bs1                          # 50:50 beam splitter

    # north arm
    s s3 bs1.p2 ITMy.p1 L=7         # space of 7m length to north arm
    m ITMy R=0 T=1                  # north input mirror, lossless
    s Ly ITMy.p2 ETMy.p1 L=4k       # north arm of 4km length
    m ETMy R=1 T=0                  # north end mirror, lossless

    # east arm
    s s4 bs1.p3 ITMx.p1 L=7         # space of 7m length to east arm
    m ITMx R=0 T=1 phi=90           # east end mirror, lossless
    s Lx ITMx.p2 ETMx.p1 L=4k       # east arm of 4km length
    m ETMx R=1 T=0 phi=90           # east end mirror, lossless

    s s5 bs1.p4 MSR.p1 L=50         # space of 50m length to signal recycling mirror
    m MSR R=0.9 T=0.1               # signal recycling mirror

    # apply signal to north and east mirror positions, 180 degrees out of phase
    fsig(100)
    sgen sig1 ETMx.mech.z
    sgen sig2 ETMy.mech.z phase=180

    ad signal MSR.p2.o fsig.f       # amplitude detector in south port
    """
)

out = base.run("""
    series(
        xaxis(fsig.f, log, 1, 10k, 300),
        change(MSR.phi=0.73026),
        xaxis(fsig.f, log, 1, 10k, 300),
        change(MSR.phi=2.4342),
        xaxis(fsig.f, log, 1, 10k, 300),
        change(MSR.phi=24.342),
        xaxis(fsig.f, log, 1, 10k, 300),
    )
""")

plt.figure()
plt.semilogx(out[0].x1, abs(out[0]['signal']), label='tuned')
plt.semilogx(out[1].x1, abs(out[1]['signal']), label='detuned 150 Hz')
plt.semilogx(out[2].x1, abs(out[2]['signal']), label='detuned 500 Hz')
plt.semilogx(out[3].x1, abs(out[3]['signal']), label='detuned 5 kHz')
plt.xlabel('North mirror tuning [deg]')
plt.ylabel(r'sideband amplitude [$\sqrt{\mathrm{W}}$]')
plt.legend(loc='lower left', framealpha=1)
plt.show()
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.14 7.5.3 Sagnac interferometer

This example demonstrates how to compute the frequency response of a simple Sagnac interferometer (Fig. 59, generated here). See section 7.4 for more information on the Sagnac interferometer.

Note

In Finesse 2, the amplitude of a signal on a mechanical `z` node of a mirror or beamsplitter was scaled from meters to radians. The conversion factor is the reference wavenumber, which is equal to $k_0 = 5.91 \times 10^6 \mathrm{m}^{-1}$ for a reference wavelength $\lambda_0 = 1064$ nm.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                            # laser with P=1W at the default frequency
    s s1 l1.p1 bs1.p1 L=1           # space of 1m length

    bs bs1                          # 50-50 beam splitter
    s s2 bs1.p2 my.p2 L=1k          # north arm of 1km length
    bs my R=1 T=0                   # north end mirror, lossless
    s s3 bs1.p3 mx.p1 L=1k          # east arm of 1km length
    bs mx R=1 T=0                   # east end mirror, lossless
    s s4 mx.p2 my.p1 L=1k           # third arm of 1km length

    # apply signal to north and east mirror positions, 180 degrees out of phase
    fsig(1)
    sgen sig1 mx.mech.z
    sgen sig2 my.mech.z phase=180

    ad signal bs1.p4.o fsig.f       # amplitude detector in south port
    """
)

out = base.run(Xaxis(base.fsig.f, 'log', 100, 1e6, 1000))

plt.figure(figsize=(8, 4))
plt.loglog(out.x1, np.abs(out['signal']))
plt.xlabel(r'$f$ [Hz]')
plt.ylabel(r'Amplitude [$\sqrt{\mathrm{W}}$]')
plt.show()
```

```
<Figure size 768x384 with 1 Axes>
```

11.15 8.17.1 Michelson modulation

This example demonstrates how a macroscopic arm length difference can cause different 'dark fringe' tuning for injected fields with different frequencies. In this case, some of the 10 MHz modulation sidebands are transmitted when the interferometer is tuned to a dark fringe for the carrier light. This effect can be used to separate light fields of different frequencies. It is also the cause for transmission of laser noise (especially frequency noise) into the Michelson output port when the interferometer is not perfectly symmetric (Fig. 73).

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                            # laser with P=1W at the default frequency
    s s1 l1.p1 eom1.p1 L=1          # space of 1m length
    mod eom1 f=10M midx=0.3         # phase modulation at 10 MHz
    s s2 eom1.p2 bs1.p1 L=1         # another space of 1m length

    bs bs1 R=0.5 T=0.5              # 50:50 beam splitter
    s Ly bs1.p2 ETMy.p1 L=100       # north arm of 110m length
    m ETMy R=1 T=0                  # north end mirror, lossless
    s Lx bs1.p3 ETMx.p1 L=110       # east arm of 100m length
    m ETMx R=1 T=0                  # east end mirror, lossless

    ad carrier bs1.p4.o f=0         # amplitude detector for carrier field
    ad sideband bs1.p4.o f=10M      # amplitude detector for +10MHz sideband
    """
)

out = base.run(Xaxis(base.ETMy.phi, 'lin', 0, 300, 100))

plt.figure(figsize=(8, 4))
plt.plot(out.x1, np.abs(out['carrier']), label='carrier')
plt.plot(out.x1, np.abs(out['sideband']), label='sideband')
plt.xlabel(r'phi (ETMy) [\degree]')
plt.ylabel(r'Amplitude [$\sqrt{\mathrm{W}}$]')
plt.show()
```

```
<Figure size 768x384 with 1 Axes>
```

11.16 8.17.2 Cavity power and slope

Figure 74 (generated here, same as Fig. 65) shows a plot of the analytical functions describing the power inside a cavity and its differentiation by the cavity tuning. This example recreates the plot using a numerical model in Finesse.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1 P=1                    # laser with P=1W at the default frequency
    s s1 l1.p1 m1.p1 L=1        # space of 1m length connecting laser to input mirror
    m m1 R=0.9 T=0.1            # cavity input mirror
    s scav m1.p2 m2.p1 L=1200   # cavity length of 1200m
    m m2 R=1 T=0                # cavity output mirror

    pd pd1 m2.p1.i              # photo diode measuring the intra-cavity power
    """
)
```

(continues on next page)

(continued from previous page)

```
out = base.run(Xaxis(base.m2.phi, 'lin', -50, 250, 1000))

fig, ax = plt.subplots(2, 1, sharex=True)
ax[0].semilogy(out.x1, out['pd1'])
ax[0].set_ylim(1e-2, 1e2)
ax[0].set_ylabel(r'$P$ [W]')
ax[1].plot(out.x1, np.gradient(out['pd1'], out.x1))
ax[1].set_ylim(-20, 20)
ax[1].set_xlabel('phi (m2) [deg]')
ax[1].set_ylabel(r'$\mathrm{d}P/\mathrm{d}\phi$ [W/deg]')
plt.show()
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.17 8.17.3 Michelson with Schnupp modulation

Figure 75 (generated here) shows the demodulated photodiode signal of a Michelson interferometer with Schnupp modulation, as well as its differentiation, the latter being the optical gain of the system. Comparing this figure to Fig. 68, it can be seen that with Schnupp modulation, the optical gain at the dark fringe operating points is maximised and a suitable error signal for these points is obtained.

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                                # laser with P=1W at the default frequency
    s s1 l1.p1 eom1.p1 L=1              # space of 1m length
    mod eom1 f=10M midx=0.3             # phase modulation at 10 MHz
    s s2 eom1.p2 bs1.p1 L=1             # another space of 1m length

    bs bs1 R=0.5 T=0.5                  # 50:50 beam splitter
    s Ly bs1.p2 ETMy.p1 L=100           # north arm of 110m length
    m ETMy R=1 T=0 phi=0                # north end mirror, lossless
    s Lx bs1.p3 ETMx.p1 L=110           # east arm of 100m length
    m ETMx R=1 T=0 phi=-ETMy.phi        # east end mirror, lossless

    pd1 south bs1.p4.o f=10M phase=-115 # demodulated output signal
    """
)

out = base.run(Xaxis(base.ETMy.phi, 'lin', 0, 180, 100))
arm_length_difference = out.x1/180

fig, ax = plt.subplots(2, 1, sharex=True, gridspec_kw={'hspace':0})
ax[0].plot(arm_length_difference, out['south'], label='demodulated signal')
ax[0].set_ylabel(r'$P$ [W]')
ax[0].legend(loc='upper right')
ax[1].plot(arm_length_difference, np.gradient(out['south'], arm_length_difference),
           label='optical gain')
ax[1].set_ylabel(r'$\mathrm{d}P/\mathrm{d}(\delta L/\lambda)$ [W]')
ax[1].set_xlabel(r'arm length difference $\delta L/\lambda$')
ax[1].legend(loc='upper right')
plt.show()
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.18 9.17.1 Beam parameter tracing

This example illustrates a possible use of the beam_property_detector (page 173) 'bp': the beam radius of the laser beam is plotted as a function of distance to the laser. For this simulation, the interferometer matrix does not need to be solved. 'bp' merely returns the results from the beam tracing algorithm of Finesse (Fig. 95, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    laser l1                        # laser with P=1W at the default frequency

    # Initializing the Gaussian beam with beam waist 1mm at 2m away from the laser.
    gauss g1 l1.p1.o w0=1m z=-2
    modes(maxtem=0)                 # we need only the u_00 mode

    s s1 l1.p1 nothing1.p1 L=1      # a space of 1m length
    nothing nothing1                # a nothing element to connect the detector
    bp width nothing1.p1.i w        # beam property detector to measure the beam width
    """
)

# tuning the length of s1
out = base.run(Xaxis(base.s1.L, 'lin', 0.1, 8, 200))

plt.figure()
plt.plot(out.x1, out['width']*1e3)
plt.xlabel('s1.L [m]')
plt.ylabel('Beam size [mm]')
plt.show()
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.19 9.17.2 Telescope and Gouy phase

This example shows the fine tuning of a telescope. The optical setup is similar to the optical layout on the Virgo North-end detection bench, resembling the telescope for the beam transmitted by the end mirror of one arm. The telescope consists of four sequential lenses with the purpose of reducing the beam size and provide a user defined Gouy phase for a split photo detector which is used for the alignment sensing system (Fig. 96, generate here).

```
import finesse
from finesse.analysis.actions import Xaxis
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                                # laser with P=1W at the default frequency
    gauss g1 l1.p1.o w0=0.014852735 z=-2.2462888k       # initializing gaussian beam
```

(continues on next page)

(continued from previous page)

```
    # multi-lens telescope, similar to the Virgo north end detection
    s sN2 l1.p1 lens1.p1 L=1.77                 # space between laser and first lens
    lens lens1 f=1.02                           # first lens, f=1.02
    s sN3 lens1.p2 lens2.p1 L=0.8996            # space between first and second lens
    lens lens2 f=-0.2                           # second lens, f=-0.2
    s sN4 lens2.p2 lens3.p1 L=1.1142-sN3.L      # space between second and third lens
    lens lens3 f=-0.1                           # third lens, f=-0.1
    s sN5 lens3.p2 lens4.p1 L=0.608             # space betwen third and fourth lens
    lens lens4 f=-0.1                           # fourth lens, f=-0.1
    s sN8 lens4.p2 nothing1.p1 L=0.759          # space between fourth lens and detector
    nothing nothing1                            # nothing element to connect detector

    gouy gouy1 from_node=l1.p1.o to_node=lens4.p1.i     # gouy phase through telescope
    bp w1 nothing1.p1.i w                               # beam size at end of telescope
    """
)

# tuning the position of lens2 by changing the lengths of the space in front and behind
# the lens.
out = base.run(Xaxis(base.sN3.L, 'lin', -1e-3, 1e-3, 400, relative=True))

fig, ax1 = plt.subplots()

ax1.plot(out.x1*1e3, out['w1']*1e3, c='green', label='Abs')
ax1.set_ylim(0, 12)
ax1.set_xlabel(r'$L$ (sN3) [mm]')
ax1.set_ylabel('beam size [mm]')
ax1.tick_params(axis='y', labelleft=True)

ax2 = ax1.twinx()
ax2.plot(out.x1*1e3, out['gouy1'], c='blue', label='Phase')
ax2.set_ylim(0, 180)
ax2.set_ylabel('Gouy phase [Deg]')
ax2.tick_params(axis='y', labelright=True)

lines1, labels1 = ax1.get_legend_handles_labels()
lines2, labels2 = ax2.get_legend_handles_labels()
ax1.legend(lines1 + lines2, labels1 + labels2, loc='upper left')
plt.show()
```

```
<Figure size 576x355.968 with 2 Axes>
```

11.20 9.17.3 LG33 mode

Finesse uses the Hermite–Gauss modes as a base system for describing the spatial properties of laser beams. However, Laguerre–Gauss modes can be created using the coefficients given in Eq. (9.43). This example demonstrates this and the use of a field camera `fcam` (page 175) to plot amplitude and phase of a beam cross section (Fig. 97, generated here).

```
import finesse
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                            # laser with P=1W at the default frequency
    gauss g1 l1.p1.o w0=1m z=0      # set up gaussian beam
    modes(maxtem=9)                 # include modes up to n+m=9
    tem(l1, 0, 0, 0, 0)             # HG coefficients to create LG33 mode
```

(continues on next page)

(continued from previous page)

```
    tem(l1, 9, 0, 0.164063, 0)
    tem(l1, 8, 1, 0.164063, -90)
    tem(l1, 7, 2, 0, 0)
    tem(l1, 6, 3, 0.125, -90)
    tem(l1, 5, 4, 0.046875, 180)
    tem(l1, 4, 5, 0.046875, -90)
    tem(l1, 3, 6, 0.125, 180)
    tem(l1, 2, 7, 0, 0)
    tem(l1, 1, 8, 0.164063, 180)
    tem(l1, 0, 9, 0.164063, 90)

    s s1 l1.p1 nothing1.p1 L=1      # a space of 1m length
    nothing nothing1                # a nothing element to connect the detector
    fcam ccd1 nothing1.p1.i xlim=5 ylim=5 npts=200  # ccd camera at end of space
    """
)

out = base.run()
out.plot(figsize_scale=2);

import numpy as np
import matplotlib.pyplot as plt
plt.imshow(np.degrees(np.angle(out['ccd1'])));
```

```
<Figure size 1152x711.936 with 4 Axes>
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.21 10.9.1 Higher-order mode resonances

This example illustrates the different resonance conditions of the higher-order modes in an optical cavity. In the simulation an input beam made up of 6 different order modes is generated using the `tem` (page 197) command. This beam is injected into an Advanced LIGO style cavity and the cavity length is tuned. Using an `amplitude_detector` (page 172) for each order mode, the different order modes are individually detected (Fig. 112, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
import numpy as np
import matplotlib.pyplot as plt
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                                # laser with P=1W at the default frequency
    modes(maxtem=5)                     # include modes up to n+m=5
    tem(l1, 1, 0, 0.5, 0)               # HG coefficients of input beam
    tem(l1, 2, 0, 0.3, 0)
    tem(l1, 3, 0, 0.2, 0)
    tem(l1, 4, 0, 0.1, 0)
    tem(l1, 5, 0, 0.15, 0)

    s s1 l1.p1 ITM.p1 L=1               # a space of 1m length
    m ITM R=0.986 T=0.014 Rc=-1934      # cavity input mirror
    s scav ITM.p2 ETM.p1 L=4000         # cavity length of 4km
    m ETM R=1 T=0 Rc=2245               # cavity end mirror
    cav arm ITM.p2                      # computing cavity parameters

    ad order0 ETM.p1.i f=0 n=0 m=0      # amplitude detectors for each order mode
```

(continues on next page)

(continued from previous page)

```
    ad order1 ETM.p1.i f=0 n=1 m=0
    ad order2 ETM.p1.i f=0 n=2 m=0
    ad order3 ETM.p1.i f=0 n=3 m=0
    ad order4 ETM.p1.i f=0 n=4 m=0
    ad order5 ETM.p1.i f=0 n=5 m=0
    """
)

out = base.run(Xaxis(base.ETM.phi, 'lin', -90, 90, 1000))

plt.figure()
for i in range(6):
    plt.plot(out.x1, np.abs(out[f'order{i}']), label=f'order {i}')
plt.legend()
plt.show()
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.22 10.9.2 Mode cleaner

This example uses the `tem` (page 197) command to create a laser beam which is a sum of equal parts in u_{00} and u_{10} modes. This beam is passed through a triangular cavity, which acts as a mode cleaner. Being resonant for the u_{00}, the cavity transmits this mode and reflects the u_{10} mode as can be seen in the resulting plots (Fig. 113, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                        # laser with P=1W at the default frequency
    modes(maxtem=1)             # Hermite-Gauss modes up to n+m = 1
    tem(l1, 0, 0, 1, 0)         # u00 mode
    tem(l1, 1, 0, 1, 0)         # u10 mode
    s s1 l1.p1 bs1.p1 L=1       # a space of 1m length

    # triangular mode cleaner cavity
    bs bs1 R=0.9 T=0.1          # input mirror
    s sc1 bs1.p3 bs2.p1 L=2     # 2m distance between bs1 and bs2
    bs bs2 R=0.9 T=0.1          # output mirror
    s sc2 bs1.p4 bs3.p2 L=49    # 49m distance between bs1 and bs3
    s sc3 bs2.p2 bs3.p1 L=49    # 49m distance between bs2 and bs3
    bs bs3 R=1 T=0 Rc=150       # curved end mirror
    cav cav1 bs1.p3             # computing cavity parameters

    fcam ccdrefl bs1.p2.o xlim=3 ylim=3 npts=200    # beam shape in reflection
    fcam ccdtrns bs2.p3.o xlim=3 ylim=3 npts=200    # beam shape in transmission
    """
)

out = base.run()
out.plot(figsize_scale=2);
```

```
<Figure size 1152x711.936 with 4 Axes>
```

```
<Figure size 1152x711.936 with 4 Axes>
```

11.23 10.9.3 Misaligned cavity

In this example a misaligned cavity is scanned and the circulating power is detected. Additional spikes in the cavity scan indicate the higher-order modes (order one and two are visible) created by the misalignment (Fig. 114, generated here).

```
import finesse
from finesse.analysis.actions import Xaxis
finesse.init_plotting()

base = finesse.Model()
base.parse(
    """
    l l1                                # laser with P=1W at the default frequency
    modes(maxtem=3)                     # Hermite-Gauss modes up to n+m = 3

    s s1 l1.p1 ITM.p1 L=1               # a space of 1m length
    m ITM T=0.014 L=35u Rc=-1934        # cavity input mirror
    s scav ITM.p2 ETM.p1 L=3994.5       # cavity length of 3994.5m
    m ETM T=5u L=35u Rc=2245 xbeta=0.3u # cavity end mirror
    cav cav1 ITM.p2                     # computing cavity parameters

    pd pd1 ETM.p1.i                     # photodiode measuring the circulating power
    """
)

out = base.run(Xaxis(base.ETM.phi, 'lin', -90, 90, 1000))
out.plot(logy=True);
```

```
<Figure size 576x355.968 with 1 Axes>
```

11.24 10.9.4 Impact of thermal aberrations

This example shows the power circulating in an Advanced LIGO style arm cavity versus input laser power when we consider the impact of thermal effects (lensing of the input mirror and change in curvature of the mirror surfaces). The mode mismatches these aberrations cause results in less power coupled into the cavity (Fig. 115, generated here).

We assume here that the thermal aberrations scale linearly with power (Vinet 2009). As we tune the incident laser power we also tune the thermal changes in curvature ($\mathrm{d}R_{c,\mathrm{ITM}}$ and $\mathrm{d}R_{c,\mathrm{ETM}}$). Here we use the change in R_c calculated for an Advanced LIGO cavity operating at high power (125 W) and then scale the R_c's accordingly. The curvatures are combined with the cold state curvatures to give the final state of the cavity mirrors at a given laser power. Similarly for the thermal lens in the ITM we scale the focal length, calculated for high power, with input power.

Finally we scale the power circulating in an individual arm cavity (P_c) by the gain afforded by the power recycling cavity (45 W/W) and the beam splitter (0.5) to represent the power in an arm of the full power recycled Michelson configuration.

We also plot the theoretical linear circulating power, when thermal effects are not considered. Here the 280.7 W is the circulating power in a cavity simulated with no thermal or higher order mode defects.

```
import matplotlib.pyplot as plt
import numpy as np

import finesse
from finesse.analysis.actions import Xaxis

finesse.init_plotting()

base = finesse.Model()
base.parse(
```

(continues on next page)

(continued from previous page)

```
    """
    l l1                                    # laser with P=1W at the default frequency
    gauss g1 l1.p1.o w0=12m z=-1.83522k     # set up gaussian beam
    modes(maxtem=6)                         # include modes up to n+m=6

    var Pin 0                               # variable for input power
    var dRcITM -1/(1/55000*Pin/125+1e-20)
    var RcITM -1/(1/1934+1/dRcITM)          # Rc of input mirror
    var dRcETM -1/(1/80000*Pin/125+1e-20)
    var RcETM 1/(1/2245+1/dRcETM)           # Rc of end mirror

    s s1 l1.p1 lens1.p1 L=1             # a space of 1m length
    lens lens1 f=6700*125/(Pin+1e-20)   # thermal lens
    s s2 lens1.p2 ITM.p1                # space of zero length connecting lens to mirror
    m ITM T=0.014 L=35u Rc=RcITM        # cavity input mirror
    s scav ITM.p2 ETM.p1 L=3994.5       # cavity length of 3994.5m
    m ETM T=5u L=35u Rc=RcETM           # cavity end mirror
    cav cav1 ITM.p2                     # computing cavity parameters

    pd pd1 ETM.p1.i                     # photodiode measuring circulating power
    """
)

out = base.run(Xaxis(base.Pin, "lin", 0, 125, 500))


def hot_power(x, P_c):
    return P_c * x * 45 / 2


def cold_power(x):
    return x * 280.7 * 45 / 2


plt.figure()
plt.plot(out.x1, hot_power(np.abs(out.x1), out["pd1"]), label="hot power")
plt.plot(out.x1, cold_power(np.abs(out.x1)), label="cold power")
plt.xlabel("P_in [W]")
plt.ylabel("P_c [W]")
plt.legend()
plt.show()
```

```
<Figure size 576x355.968 with 1 Axes>
```

Bibliography

[1] A Freise, G Heinzel, H Lück, R Schilling, B Willke, and K Danzmann. Frequency-domain interferometer simulation with higher-order spatial modes. *Classical and Quantum Gravity*, 21(5):S1067–S1074, 2004. URL: http://stacks.iop.org/0264-9381/21/S1067.

[2] D. D. Brown. *Interaction of light and mirrors: Advanced techniques for modelling future gravitational wave detectors*. PhD thesis, University of Birmingham, 2015. URL: http://etheses.bham.ac.uk/6500/9/BrownD16PhD_Final.pdf.

[3] J Aasi et al. (LSC). Advanced ligo. *Classical and Quantum Gravity*, 32(7):074001, 2015. URL: http://stacks.iop.org/0264-9381/32/i=7/a=074001.

[4] F. Acernese and others. Advanced Virgo: a second-generation interferometric gravitational wave detector. *Class. Quantum Grav.*, 32(2):024001, 2015. URL: http://stacks.iop.org/0264-9381/32/i=2/a=024001.

[5] T Akutsu et al. (KAGRA). First cryogenic test operation of underground km-scale gravitational-wave observatory KAGRA. *Classical and Quantum Gravity*, 36(16):165008, jul 2019. URL: https://doi.org/10.1088%2F1361-6382%2Fab28a9, doi:10.1088/1361-6382/ab28a9.

[6] M Punturo and others. The einstein telescope: a third-generation gravitational wave observatory. *Class. Quantum Grav.*, 27(19):194002, 2010. URL: http://stacks.iop.org/0264-9381/27/i=19/a=194002.

[7] Daniel Brown and Andreas Freise. Pykat: python interface and tools for finesse 2. http://www.gwoptics.org/pykat/. URL: http://www.gwoptics.org/pykat/.

[8] Daniel D. Brown, Philip Jones, Samuel Rowlinson, Sean Leavey, Anna C. Green, Daniel Töyrä, and Andreas Freise. Pykat: python package for modelling precision optical interferometers. *SoftwareX*, 12:100613, 2020. URL: http://www.sciencedirect.com/science/article/pii/S2352711020303265, doi:https://doi.org/10.1016/j.softx.2020.100613.

[9] B. P. et al. (LSC Abbott and Virgo). Observation of gravitational waves from a binary black hole merger. *Phys. Rev. Lett.*, 116:061102, Feb 2016. URL: http://link.aps.org/doi/10.1103/PhysRevLett.116.061102, doi:10.1103/PhysRevLett.116.061102.

[10] A. Freise, G. Heinzel, K. A. Strain, J. Mizuno, K. D. Skeldon, H. Lück, B. Willke, R. Schilling, A. Rüdiger, W. Winkler, and K. Danzmann. Demonstration of detuned dual recycling at the Garching 30 m laser interferometer. *Physics Letters A*, 277:135–142, November 2000. arXiv:arXiv:gr-qc/0006026.

[11] H. Lück, C. Affeldt, J. Degallaix, A. Freise, H. Grote, M. Hewitson, S. Hild, J. Leong, M. Prijatelj, K. A. Strain, B. Willke, H. Wittel, and K. Danzmann. The upgrade of geo 600. *Journal of Physics Conference Series*, 228(1):012012, May 2010. arXiv:1004.0339, doi:10.1088/1742-6596/228/1/012012.

[12] K. Danzmann, H. Lück, A. Rüdiger, R. Schilling, M. Schrempel, W. Winkler, J. Hough, G.P. Newton, N.A. Robertson, H. Ward, A.M. Campbell, J.E. Logan, D.I. Robertson, K.A. Strain, J.R.J. Bennett, V. Kose, M. Kühne, B.F. Schutz, D. Nicholson, J. Shuttleworth, H. Welling, P. Aufmuth, R. Rinkleff, A. Tünnermann, and B. Willke. *GEO 600 – Proposal for a 600m laser-interferometric gravitational wave antenna*. Volume 190 of MPQ. MPQ Garching, 1994.

[13] H. Lück, A. Freise, S. Goßler, S. Hild, K. Kawabe, and K. Danzmann. Thermal correction of the radii of curvature of mirrors for geo. In *Classical Quantum Gravravity 21*, 985–989. 2003.

[14] H. Grote, A. Freise, M. Malec, G. Heinzel, B. Willke, H. Lück, K. A. Strain, J. Hough, and K. Danzmann. Dual recycling for geo. In *Classical Quantum Gravravity 21*, 473–480. 2003.

[15] A. Freise. *The Next Generation of Interferometry: Multi-Frequency Optical Modelling, Control Concepts and Implementation*. PhD thesis, Universität Hannover, 2003. URL: https://www.repo.uni-hannover.de/handle/123456789/6218, doi:https://doi.org/10.15488/6166.

[16] Mauricio D.Ortiz jr, Aaron W. Jones, Marie-Sophie Hartig, Ali L. James, and Luis Ortega. Investigation into cavity axis shifts and tilts using the finesse modal model. Technical Report, LIGO Scientific Collaboration, 2021. URL: https://dcc.ligo.org/LIGO-T1900708/public.

[17] A.E. Siegman. *Lasers*. University Science Books, 1986. ISBN 9780935702118. URL: https://books.google.co.uk/books?id=1BZVwUZLTkAC.

[18] C. Bond, D. Brown, A. Freise, and K. Strain. Interferometer Techniques for Gravitational-Wave Detection. *Living Reviews in Relativity*, 2016. doi:10.1007/s41114-016-0002-8.

[19] Dana Z Anderson. Alignment of resonant optical cavities. *Appl. Opt.*, 23(17):2944–2949, sep 1984. URL: http://ao.osa.org/abstract.cfm?URI=ao-23-17-2944, doi:10.1364/AO.23.002944.

[20] H Wang. *Beware of Warped Surfaces: Near-Unstable Cavities for Future Gravitational Wave Detectors*. PhD thesis, University of Birmingham, 2017.

[21] N. Uehara, E. K. Gustafson, M. M. Fejer, and R. L. Byer. Modeling of efficient mode-matching and thermal-lensing effect on a laser-beam coupling into a mode-cleaner cavity. In U. O. Farrukh and S. Basu, editors, *Society of Photo-Optical Instrumentation Engineers (SPIE) Conference Series*, volume 2989 of Presented at the Society of Photo-Optical Instrumentation Engineers (SPIE) Conference, 57–68. May 1997.

[22] R.~W.~P. Drever, J.~L. Hall, F.~V. Kowalski, J Hough, G.~M. Ford, A.~J. Munley, and H Ward. Laser phase and frequency stabilization using an optical resonator. *Applied Physics B: Lasers and Optics*, 31:97–105, jun 1983. doi:10.1007/BF00702605.

[23] John A. Sidles and Daniel Sigg. Optical torques in suspended fabryâ??perot interferometers. *Physics Letters A*, 354(3):167–172, 2006. URL: https://www.sciencedirect.com/science/article/pii/S0375960106001381, doi:https://doi.org/10.1016/j.physleta.2006.01.051.

[24] Eiichi Hirose, Keita Kawabe, Daniel Sigg, Rana Adhikari, and Peter R. Saulson. Angular instability due to radiation pressure in the ligo gravitational-wave detector. *Appl. Opt.*, 49(18):3474–3484, Jun 2010. URL: https://opg.optica.org/ao/abstract.cfm?URI=ao-49-18-3474, doi:10.1364/AO.49.003474.

[25] Lisa Barsotti and Matthew Evans. Modeling of alignment sensing and control for advanced ligo. Technical Report, LIGO Scientific Collaboration, 2009. URL: https://dcc.ligo.org/LIGO-T0900511/public.

[26] Carlton M. Caves. Quantum-mechanical radiation-pressure fluctuations in an interferometer. *Phys. Rev. Lett.*, 45:75–79, Jul 1980. URL: https://link.aps.org/doi/10.1103/PhysRevLett.45.75, doi:10.1103/PhysRevLett.45.75.

[27] L. McCuller, C. Whittle, D. Ganapathy, K. Komori, M. Tse, A. Fernandez-Galiana, L. Barsotti, P. Fritschel, M. MacInnis, F. Matichard, K. Mason, N. Mavalvala, R. Mittleman, Haocun Yu, M. E. Zucker, and M. Evans. Frequency-dependent squeezing for advanced ligo. *Phys. Rev. Lett.*, 124:171102, Apr 2020. URL: https://link.aps.org/doi/10.1103/PhysRevLett.124.171102, doi:10.1103/PhysRevLett.124.171102.

[28] G. Heinzel. *Advanced optical techniques for laser-interferometric gravitational-wave detectors*. PhD thesis, Munich, Max Planck Inst. Quantenopt., 1999.

[29] Jun Mizuno. *Comparison of optical configurations for laser-interferometric gravitational-wave detectors*. PhD thesis, Hannover U., 1995.

[30] Joel S Cohen. *Computer algebra and symbolic computation: Mathematical methods*. AK Peters/CRC Press, 2003.

[31] Eric D. Black. An introduction to pound–drever–hall laser frequency stabilization. *American Journal of Physics*, 69(1):79–87, January 2001. doi:10.1119/1.1286663.

[32] Riccardo Maggiore. *Adventures in Gravitational Wave Detector Control: A Round-Trip Journey*. PhD-Thesis - Research and graduation internal, Vrije Universiteit Amsterdam, April 2024. doi:10.5463/thesis.566.

[33] Bayer-Helms F. Coupling coefficients of an incident wave and the modes of spherical optical resonator in the case of mismatching and misalignment. *Appl. Opt.*, 23:1369–1380, 1984. doi:10.1364/AO.23.001369.

[34] Andreas Freise Charlotte Bond, Paul Fulda. Analytical calculation of hermite-gauss and laguerre-gauss modes on a bullseye photodiode. Technical Report, LIGO Scientific Collaboration, 2016. URL: https://arxiv.org/abs/1606.01057.

[35] Samuel Rowlinson, Artemiy Dmitriev, Aaron W. Jones, Teng Zhang, and Andreas Freise. Feasibility study of beam-expanding telescopes in the interferometer arms for the einstein telescope. *Phys. Rev. D*, 103:023004, Jan 2021. URL: https://link.aps.org/doi/10.1103/PhysRevD.103.023004, doi:10.1103/PhysRevD.103.023004.

[36] D. D. Brown. *Interactions of light and mirrors: Advanced techniques for modelling future gravitational wave detectors*. PhD thesis, University of Birmingham, 2015.

[37] Jean-Yves Vinet. On special optical modes and thermal issues in advanced gravitational wave interferometric detectors. *Living Reviews in Relativity*, 12(1):5, 2009. URL: https://doi.org/10.12942/lrr-2009-5, doi:10.12942/lrr-2009-5.

[38] Joshua Ramette, Marie Kasprzack, Aidan Brooks, Carl Blair, Haoyu Wang, and Matthew Heintze. Analytical model for ring heater thermal compensation in the advanced laser interferometer gravitational-wave observatory. *Appl. Opt.*, 55(10):2619–2625, Apr 2016. URL: http://ao.osa.org/abstract.cfm?URI=ao-55-10-2619, doi:10.1364/AO.55.002619.

[39] Huy-Tuong Cao. Analytical solution to deformation of test mass due to ring heater. Technical Report, LIGO Scientific Collaboration, 2025. URL: https://dcc.ligo.org/LIGO-T2500014/public.

[40] A. Freise, G. Heinzel, H. Lück, R. Schilling, B. Willke, and K. Danzmann. Frequency-domain interferometer simulation with higher-order spatial modes. In *Classical Quantum Gravravity 21*, 1067–1074. 2003.

[41] A Freise and D D Brown. \sc Finesse 2.0 manual. Technical Report, University of Birmingham, 2014. http://www.gwoptics.org/finesse/download/manual.pdf.