theia

A 3D Gaussian Beam Tracer for Gravitational Interferometers

Version 0.1.1

User Guide

Raphaël Duque

July 11, 2017

theia is a command line program and Python library for 3D Gaussian beam tracing. It supports many different types of optical components, general 3D placing and orientation of these components and general astigmatic Gaussian beams, among other features. It allows the 3D visualization of the otical siulaiton by writing a CAD file read by the free software FreeCAD. theia was developed at the Optics Group of the Virgo gravitational observatory in Cascina, Italy. For more information on the theia Project, surf to http://theia.hopto.org:56000.

This document is a user's guide to the theia command line tool. It gives the information concerning the installation instructions, the usage and the input and output of theia necessary to operate the program from the command line. It also provides a short introduction to the theia library API. For more details on the theia Python library, please refer to the API Guide provided along with this User Guide, or go to http://theia.hopto.org:56000/docs/html/index.html for the online API guide.

Contents

1 `theia` Quick Start

For a quick start of `theia` once you are in the project repository, install `theia` locally with `make install`. The command `theia` will then be available to you everywhere and you can run the tutorial input files (with extension `.tia`) found in the `tutos/` folder with `theia FNAME.tia`, replacing `FNAME.tia` by one of the tutorial files. It is recommended when doing this to have the *Quick Reference* document not to far (provided with the project or found online at the project site) in order to learn the syntax for the input file.

The input `.tia` files provide (among other things) the optical setup information in text form. The output `.out` file reports the physical data (waist position relatively to the origin of the beam and waist size) of the beams generated by the propagation of the input beam. The `.fcstd` file provides the information for the 3D viewing by the `FreeCAD` software. The `alloptics.tia` tutorial file is particularly fit to quickly learn the input format because all the possible types of optics are used in the corresponding simulation.

When you have run the simulations, the text output file `.out` will contain the information of the traced beams, and the 3D viewing of the setup can be done by reading the `.fcstd` file with `FreeCAD`.

2 Physics behind of `theia`

In the section, we will briefly explain the ideas behind the development of `theia`, describe the way `theia` sees the physical objects it deals with and the tracing algorithm it implements.

2.1 The theia rationale

`theia` has been designed for flexible and practical operation. This why `theia` is not only a command line tool, but also a Python library aiming at scripting and written accordingly – please see the theia *API Guide* for more details on this library.

The `theia` command line tool has it its own right been designed with flexibility and pragmatism in mind. The `theia` input and output files were thought to allow high level features to insure ease of writing and reading by humans, to be printed out, brought to the optical bench and used as references to follow the evolution of the optical layout and its components, to be read as structured files containing figures one can readily compare to experimental data, etc.

Aiming for flexibility also implies liberty for the user when it comes to input. The user can specify as much information as she or he wishes. From specifying zero parameters and using default values for all the arguments to using built-in values such as handy units, users have a large radius of action for their input.

With liberty must also come caution. If the user specifies geometrically inconsistent parameters – leading to self-intersecting surfaces for instance –, then warnings may be issued to standard output (unless specific command line flags are used, see 3.2) but the simulation will carry on almost seamlessly, and may lead to unexpected behavior.

2.2 The operation of theia

`theia` is a command line 3D Gaussian beam tracing program. During its operation, input beams and an optical setup are read from an input text file and these beams are traced and interact with optical components. Following the rules of geometrical and Gaussian optics, and according to some selection rules designed to insure the termination of the program, this process produces new beams, by reflection and transmission of the former beams on the surfaces of the optical components. This creation and selection process is repeated recursively in order to calculate all the beams produced by the input beams and their geometrical and Gaussian characteristics.

This operation results in the writing of a text file containing the information on the traced beams and a CAD file for 3D visualization.

2.3 Beams and optics

The physical objects a representation of which `theia` deals with are general astigmatic Gaussian beams, and optical components in a 3D general position and orientation. This section describes how these objects are seen by `theia`.

Gaussian beams. Gaussian beams are described by two set of parameters: geometrical parameters and Gaussian parameters.

1. The geometrical parameters are the 3D position of the origin of the beam, the unitary 3D vector directing the beam and finally the length of the beam, from its origin to its end point. On input, the position of the origin is given by three coordinates and the direction by two angles forming spherical coordinates. See section 4.1 for details. The length of the beam is initialized to 0 and updated once we know if and where the beam ends by interaction.

2. The Gaussian parameters are threefold: one complex matrix and two unitary vectors. It is so because the general astigmatic beam, as it forms by repeated oblique incidences on optics, is described on one polarization direction by the following equation on the electrical field:

$$E(\vec{r},t) = \exp(i\eta(z) - i\frac{k}{2}t(x,y)Q(z)(x,y))e^{i(\omega t - kz)}$$

   in which z is the coordinate of $\vec{r}$, (x,y) its coordinates in the plane transversal to the beam's direction, and Q is a complex matrix, and η accounts for all other phase accumulation (Gouy, etc.). [1]

   Thus, to specify completely the state of the beam at a position z along the beam, one has to specify two vectors (the eigen-directions) which span the transversal plane, and the matrix $Q(z)$ in the basis of these vectors. From Q, one can derive the waists and positions of these waists, Rayleigh ranges, etc.

   In `theia`, to each beam are attached these two vectors and the value of Q *at the origin of the beam*.

Inputting Gaussian data. Inputting to the `.tia` file only allows for orthogonal beams. Thus it is asked to the user to specify the waists and positions of these waists, and the angle of rotation of the beam (in the clockwise sens looking down the beam) with respect to the beam which has as a first eigen direction the vector with the largest possible global Z coordinate (and which is of course orthogonal to the beam direction).

In other words, if the user specifies a direction $\vec{dir}$ and an angle of 0, then she or he is saying that the first semi-axis of the amplitude ellipsis (in which the first input waist and waist distance are found) is that unitary u vector orthogonal to $\vec{dir}$, and which has the largest possible Z coordinate. See figure 1 for an illustration. And the second semi-axis is the unitary v vector such that $(\vec{dir}, u, v)$ is a right-handed orthonormal basis.

What if $\vec{dir}$ is directed by $\pm e_Z$ and the "largest possible Z coordinate" condition makes no sens? Then u is $\pm e_X$ and v is $\pm e_Y$.

Now, if the angle `Alpha` is not 0, then the basis the user means is the rotation of the `Alpha=0.` case by an angle `Alpha` around $\vec{dir}$.

Optical components. Available optics are semi-reflective mirrors, thin lenses (defined by their focal length and diameter), thick lenses (thickness on axis, refractive index, curvatures and diameter), and beam-dumps (which stop light). Optical components all have cylindrical symmetry, except for mirrors which may have a wedged face. The labeled "HR" surface is the principal surface, where beam are meant to be reflected of transmitted. In the case of mirrors it is the non-wedged surface. See section 4.1 for inputting thin lenses, thick lenses and beam dumps.

[1] Note that in the case where the beam is orthogonal (but *not* in the general astigmatic case), there exists a orthogonal basis of the transverse plane in which the Q matrix is diagonal and the expression of the electrical field is reduced to a more familiar form $\exp(-i\frac{k}{2}(x^2/q_x(z) + y^2/q_y(z)))$. This description is amply detailed in [1, 2].

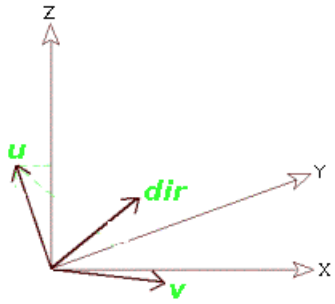

Figure 1: The `Alpha = 0` case. X, Y, Z is the global coordinate system, u is defined as being the vector orthogonal to the beam direction dir and having the largest global Z coordinate. v is chosen such that (dir, u, v) is a right-handed orthogonal basis. For a non-zero `Alpha` case, rotate u and v by `Alpha` around dir

Since mirrors can be wedged and do not have cylindrical symmetry, it is necessary to specify an angle to describe the position of the wedge in space. This is done in a similar fashion than in the beam case. If this angle (also called `Alpha` is 0, then the point of the AR (wedged face) which is not affected by the wedging (it is the point on the rim of the original cylinder the AR was wedged from) has the largest global Z component. Similarly as for beams, if the "maximum Z" condition makes no sense because the HR normal is $+e_Z$ (resp. $-e_Z$), then this reference point has largest (resp. smallest) X coordinate.

If `Alpha` is not 0, then this reference is situated at the image of the `Alpha=0` reference point by the rotation of `Alpha` around the normal (outpointing) vector of the HR surface. See figure 2 for an illustration.

On the contrary of lenses, the given center of the HR surface of mirrors on input is the center of the chord of the HR surface, and not the apex of the HR surface. Similarly, the thicknesses of mirrors as input are the thicknesses on the rim (rim of HR to closest point of wedged AR) and not on the optical axis.

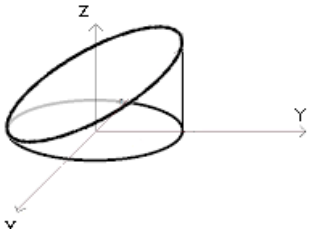

Figure 2: A mirror. Its center of HR is (0, 0, 0) and its normal to HR is $-e_Z$. Thus the reference point is (–1, 0, 0). Thus we can say that this is a thickness = 0, positive wedged mirror with `Alpha=` 225 ° (angle between the reference point and the point of the rim of the original cylinder which touches the AR face). Note that we could have also said a thickness = 1, *negatively* wedged mirror with `Alpha=` 45 °.

2.4 Algorithm and approximations

As previously described, `theia` traces beams by interaction of the input beams with the optics, recursively. Here are the rules to calculate the beams.

Physical rules. The laws of refraction and reflection are applied to calculate the directions of the new beams, as well as the origin Q matrix (and the base vectors to express it) of the new beams. The calculation of the Q matrix follows a phase matching method and is detailed in [1, 2]. Total reflection is also taken into account, and the powers of the new beams are calculated according to the reflectance and transmittance of the surface. This rule introduces the approximation that the radius of curvature of the beam must be much smaller than the radii of curvature of the optics at hand. Apart from this, there are no other approximations (nor for the geometrical optics calculations to determine the direction of the new beams, nor for the exact point of origin of the new beams on the curved surfaces[2]).

Computational rules. In order for the program to terminate, some computational rules have been introduced. Each simulation is led with an *order* and a *threshold*:

1. Each initial beam is created with a *strayness* order of 0. Any beam reflected by a mirror AR face, transmitted by a mirror HR face, or reflected by any face of a lens has the order of its parent beam, plus one. Beams with an order larger than the simulation order are excluded from the following calculation step and their children are not determined.

2. The power of children beams are determined with the powers of the parent beams and the reflectance and transmittance of the surface. Similarly, beams with powers smaller than the simulation threshold are excluded from the following calculation step and their children are not determined.

3 Installation and usage

3.1 Installation instructions

`theia` uses the Python standard library component `setuptools` to install the command line tool `theia` as well as the Python library.

Local installation. To install `theia` to your local environment, `cd` to the project repository root and issue the following commands:

- `make install` to install the `theia` command line program and library and compile the documentation in the `doc/` sub-directory of the project;
- `make build-theia` to only install the program and the library but not compile the documentation (useful if you do not have a latex environment running);
- `make build-doc` to only compile the documentation (useful if you have modified the library to your liking – please do).

This procedure will install the `theia` script to `$HOME/.local/bin`, and this directory **must** be in your `PATH` in order to have access to `theia` from anywhere in your file system.

System-wide installation. For a system-wide installation, you can issue `python setup.py install` with root privileges from the project root repository. The documentation must be compiled separately as indicated in the former paragraph and moved to some shared directory if you like.

[2] We are of course limited by the precision of floating pint numbers on the underlying machine.

Uninstalling. Uninstalling a local installation if fairly simple: issue `make clear` from the project root directory. This will wipe your `$HOME/.local` of anything that has to do with `theia`. All the documentation, tutorial files or `theia` input or output files elsewhere will of course stay in place after this procedure.

Uninstalling a system-wide installation is more tricky, and we do not provide an automated procedure for this and admins probably know better than us on this subject, though on most systems `setuptools` puts library files in `/usr/local/lib/python2.7/*-packages` and scripts in `/usr/local/bin`.

3.2 Usage on the command line

The general usage of `theia` is: `theia [options] FNAME`, where:

- `[options]` are command line options. See the next paragraph or the output of `theia -h` for more details;
- `FNAME` is the name of the configuration file to use for the simulation, with or without the `.tia` extension. See the next section for details on the format of the `.tia` file.

Command line options. As introduced in section 2. of this User Guide, `theia` takes one configuration file as an input and may write out to output files and to standard output (the terminal window). The command line options allow the user to control all these outputs. The command line options are summed up in table 1.

Command line option	Effect
`-h, --help`	show the usage and the command line options of `theia` and exit with a success exit code
`-l FCLIB, --FreeCAD-lib FCLIB`	specify[3] the location of the `FreeCAD` libraries for the 3D visualization as being `FCLIB` on your system
`-i, --info`	during the simulation, do not output tracing information to standard output (see section 4.3 for details on the information that is output)
`-w, --no-warn`	during the simulation, do not output warnings to standard output (see section 4.3 to find out what warnings may be written)
`-t, --no-text`	after the simulation, do not write the `.out` text output file (see section 4.2 for details)
`-c, --no-CAD`	after the simulation, do not write the CAD file

Table 1: Command lines options of `theia`.

Note on the `FreeCAD` libraries. `theia` writes the 3D visualization information for the simulation to a CAD file to be read by `FreeCAD`. If you do not wish to write the CAD file or or if you do not have the `FreeCAD` libraries installed, you may use the `-c` command line option.

On the contrary, if you have the `FreeCAD` libraries and you know where they are (usually `usr/lib/freecad/lib`), you can save `theia` the effort of finding them by aliasing `theia` with `theia -l [WHERE YOUR FREECAD LIB IS]`.

4 Input and output to theia

4.1 The `.tia` input format

`.tia` format input files are simple text files reporting the simulation parameters and the optical setup to the `theia` CLI tool. In such a file, each line is used to describe its own object[4]. These

[4]By the way, all spaces and tabs are ignored, so you can throw some in wherever you want.

objects can be simulation parameters, simulation meta-data or optical components. The object you are specifying with a given line depends on the format of the line, according to the following table 2.

Lines starting with ...	**... specify ...**	**... and the rest of the line is ...**
`#`	a comment	ignored
`order=`	the order of the simulation	a Python expression evaluating to an integer
`threshold=`	the threshold of the simulation	a Python expression evaluating to a floating point number
`bo`	the coordinates of an optical bench, i.e a new reference for coordinates (these coordinates will be added to the optics specified after this line and until the next `bo` line)	a comma-separated list of *entries* specifying the origin of the bench
`bm`	an input orthogonal Gaussian beam	a comma-separated list of *entries* (see following paragraph) specifying the characteristics of the beam
`mr`	a mirror object	a comma-separated list of *entries* specifying the characteristics of the mirror
`th`	a thinlens object	a comma-separated list of *entries* specifying the characteristics of the thin lens
`tk`	a thicklens object	a comma-separated list of *entries* specifying the characteristics of the thick lens
`bd`	a beamdump object (which stops all beams)	a comma-separated list of *entries* specifying the characteristics of the beamdump
`gh`	a ghost surface (which does not affect the beam but only make a new entry in the `.out` file)	a comma-separated list of *entries* specifying the characteristics of the ghost surface
anything else	the long name of the simulation (appears in the output file)	any text input (spaces and tabs will be stripped)

Table 2: Format of each type of line in a `.tia` input file

For beams, mirrors, beamdumps, ghost surfaces, thin and thick lenses, report to the *Quick Reference* document in order to know what are the order and the signification of the entries.

Entries and available Python expressions. An entry in a input file line is of the form (remember spaces and tabs are not taken into account) `VAR=EXPRESSION` or simply `EXPRESSION`. Use the second form when you have specified all the parameters since the beginning of the line and in the right order, and the first form when you want to input in the order you like, or you have omitted some parameters (think of this as a set of argument to a function, the arguments are considered in the right order and complete until the first `VAR=EXPRESSION`, and then its up to you to specify or not the following parameters).

`EXPRESSION` is any Python expression that is valid *in the scope where the input reading function is defined*. What this means is that you can use:

- Standard Python literal integers, floats and operators: `4.`, `.2e-3 * 3 / 2`, etc.
- The functions and constants that are available as built-ins in this scope (these are listed in the *Quick Reference* document): `arccos(3 * exp(-2.)) * mW`, `sqrt(4.*mm*mm)`, `arctan(pi/12)`, etc.

- Any user defined function or constant as long as it is declared as a global in the `theia.helpers.settings` module and initialized correctly (see section 5.1 on globals), declared in the `theia.helpers.units` module, or declared directly in `theia.running.parser`. After all of these operations, `theia` has to be rebuilt with `make install`.

  

  As we just mentioned, the parts of an entry line which are between two commas or between a comma and a `=` sign are evaluated as Python expressions, which allows for a great flexibility of the input. *Nonetheless*, you must beware of Python expressions evaluation which might lead to unexpected behavior, particularly on integers! For example, if one enters `(3/4)*pi`, the evaluation will lead to 0 because of integer division, and not $3\pi/4 = 135°$. To obtain 135°, one should enter `3*pi/4`, or `(3./4)*pi` for instance.

Empty constructors. In `theia`, all the constructors available to the user have a complete set of default values (see section 5.2 for details on this). Thus an input file can very well contain a line with a tag and *zero* arguments, such as:

```
#example usage of a empty constructor for a (thus default) input beam
bm
```

To find out what the default values of all the constructors are, report to the *Quick Reference* document.

Meta data-specifying lines overwrite each other. Lines used to specify the simulation order, threshold and long name overwrite each other. If you specify a "`threshold=`" line after another "`threshold=`" line, this latter one will overwrite the value specified in the former line. The same mechanics apply to "`order=`" and long-name lines.

4.2 The `.out` output file

The output file is also a simple text file. In contains formatted information on the simulation, the optics present in the setup and the beams generated by these. All this information is divided in sections:

- The `META DATA` section specifies the date at which the file was written, the input file used, the various simulation parameters.
- The `SIMULATION DATA` section gives information on the components used in the simulation (component type, reference and position of the HR surface chord center), and on the generated beam trees (a reference to their original beam and the number of beams the tree contains in total).
- The `BEAM LISTING` section reports, in a beam tree by beam tree fashion, the beams contained in the different beam trees. for each beam, you will find the optic and face the beam departs from, the length of the beam, the optic and face the beam finishes on, the reference to the beam, and then the waist positions (along the beam starting from the beam origin), waist sizes and finally beam direction. The waists are given in the proper basis attached to the beam, as described in section 2.3.

Beam references. The reference of each beam (as it appears at the end of each entry in the `BEAM LISTING` section of the output file) is the reference of its parents beam plus a tag which can be `r` or `t` according to whether the beam was produced by reflexion or transmission of the parent beam. The references of the input beams are read in the input file.

Here is a sample of output:

```
(M1, AR) 0.946496969941m (L1, HR) TBttt {
  Waist Pos:  (-1.8387309993210437, -1.8387309993210468)m
  Waist Size:  (0.00016310752802, 8.15537640098e-05)mm
  Direction:  (90.0, 4.68099236149e-13)deg
}
```

This sample describes a beam traveling 94.6 cm from the AR surface of the optic labeled M1 (surely a mirror) to the HR surface of the optic labeled L1 (surely a lens). This beam is the beam transmitted by the beam transmitted by the beam transmitted by the input beam labeled `TB`. It waists are about .1 μm and both located 1.83 m behind the origin of the beam (thus this beam is in its diverging phase all the way from M1 to L1). Its direction is along the positive X axis.

4.3 The `.fcstd` output file

`theia` writes the 3D visualization information to a `.fcstd` file to be read by the free software `FreeCAD`. This file is intended to ease navigation in the optical setup and also to review the characteristics of the traced beams.

Once the simulation is done, open the `.fcstd` file with `FreeCAD` and render all the objects visible by selecting all the objects on the left panel and hitting the space bar. Every simulation object (beam, optic, etc.) has two CAD objects associated to it: one which has the same reference as the optical object and which is the 3D viewer object (for example `M1`) and another which holds the optical information of the object. This one has the same name as the first one with `_doc` appended (for example `M1_doc`) and is listed right before the 3D viewer object on the object list on the left panel.

Thus, when selecting an object in the 3D view, the information of the object lies in the object listed right above the highlighted object in the left panel list. Of course, all these objects can be manipulated with the CAD software.

See figure 3 for an example of the 3D viewing.

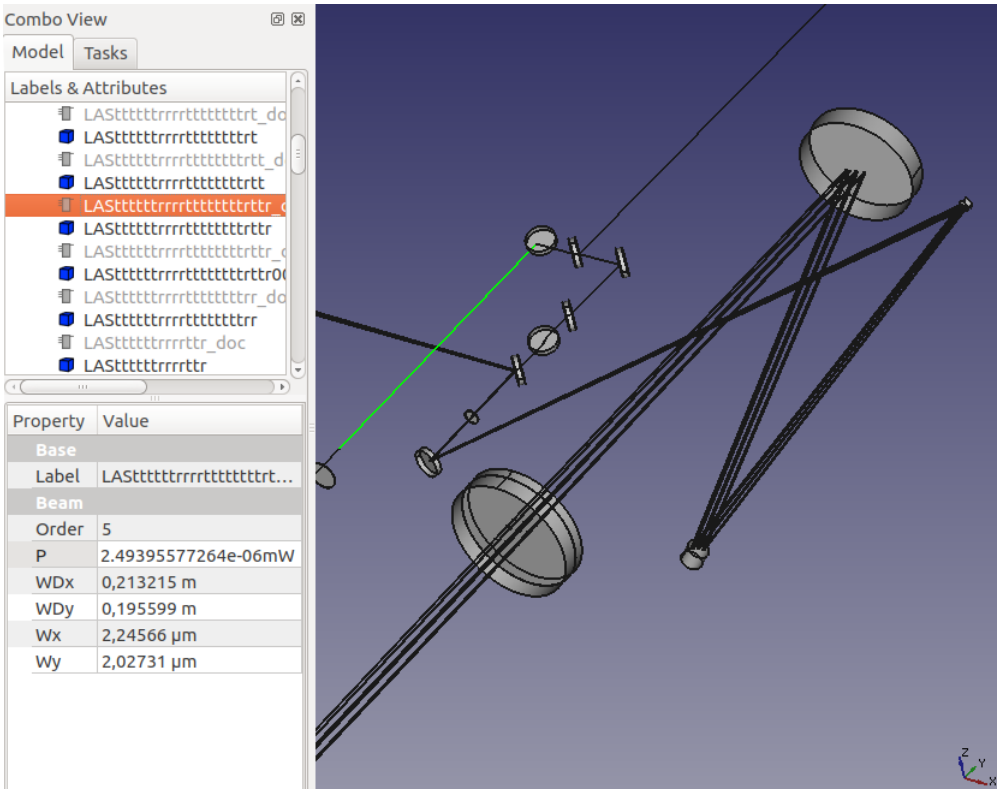

Figure 3: The 3D viewer reading the `.fcstd` file of `theia`. On the left panel, one sees a beam object (which was highlighted when the beam on the view was clicked on by the user) and the corresponding `_doc` object (highlighted now) showing the Gaussian and geometrical data of the beam.

5 An introduction to the theia API

This section is an introduction to the Application Programmers Interface to the `theia` library. It give somewhat more detail on the algorithm and data structures of theia and how they are implemented in `theia`. This guide may be useful to anyone who wants to use `theia` to develop their own optical simulation scripts, and to anyone who would like to contribute to `theia`. For a through and complete treatment of the `theia` API, see the *API Guide* document or the online documentation.

Throughout this section, Unix `paths/like/this` are understood as relative to the theia project root directory (*e.g.* `doc/img/flow.png`) and Python import statements `like.this.one` are understood as relative to the theia package (*e.g.* `running.simulation.Simulation.__init__`).

5.1 A note on global variables

The `theia` CLI tool uses a certain number of global variables in order to keep values which don't change along the execution. These global variables are consequently needed by a certain number of functions defined in the library in order for the CLI tool to be as functional as possible. When using `theia` as a library, one may not need all these globals and they may even get in the way of development.

How to take care of the globals once and for all. The global variables are *all* declared in `helpers.settings` and are initialized with `helpers.settings.init` at the very beginning of `main.main`, which takes in a dictionary and reads the globals from there. If you don't want to hear about the globals, you can place the following snippet (found in `tests/test_simulation.py`) at the beginning of your script and not worry about the globals.

```
# use this snippet and all globals worries are gone
from theia.helpers.settings import init

# initialize globals in a dictionary
dic = {'info': False, 'warning': False, 'text': False, 'cad': False,
                'fname': 'test_optics', 'fclib': '/usr/lib/freecad/lib'}

init(dic)

# you're all set
```

Who uses the globals? Here is a table (table 3) listing the global variables and which functions use them.

Global	Used by

`info`	`optics.beamdump.BeamDump.hit`
	`optics.lens.Lens.hitActive`
	`optics.mirror.Mirror.hitHR`
	`optics.mirror.Mirror.hitAR`
	`optics.optic.Optic.hitSide`
	`tree.beamtree.treeOfBeam`

`warning`	`optic.mirror.Mirror.__init__`
	`optic.thicklens.ThickLens.__init__`
	`optic.thinlens.ThinLens.__init__`
	`running.simulation.Simulation.run`

`text, cad, fname, fclib`	`main.main`

Table 3: The global variables of theia and the functions who use them

5.2 Classes and inheritance hierarchy

Figure 4 presents the inheritance hierarchy of the classes of `theia`. If you see a method twice, it just means the daughter class reimplements the method. You'll also find the signature of the methods and initializers.

A word on initializer default values. We try to avoid surprises and stay consistent throughout the code with the following policies concerning classes at the leaves of the inheritance graph:

1. For classes whose initializers will be called only with input form users (read from an input file or in a a script), *every* parameter of the constructor has a default value and the constructor can be called *without arguments*. What's more, the input of the user is processed through the class initializer and then fed to the initializer of the mother class. For example, the user may provide `X`, `Y` and `Z` to the constructor she or he calls, then these are processed and it is `[X, Y, Z]` as a list (with types checked etc.) which is fed to the mother initializer. This concerns the constructors for `ThinLens`, `ThickLens`, `Mirror`, `BeamDump`.

2. For classes whose initializers are called solely internally, there are *no default values*. These are the constructors of `SetupComponent`, `Optic`.

3. For classes that may be instantiated internally and by users, the module defining the class defines a global scope function, whose parameters *all have default values* and which is intended to be used with input from the user, as the constructors described in the previous point 1. This function is named `user$CLASSNAME` and processes the input of the user into input for the class's proper `__init__` initializer. On the other hand, this proper initializer is intended for internal use only and has *no default values*. This is for example the case of the `optics.beam.GaussianBeam` class, whose constructor is called internally to generate new beams and with user input read from the input file. In this last case it is `userGaussianBeam` which is called.

Abstract Base Classes. The highest class of the optical classes hierarchy is the `optics.component.SetupComponent` class. Its metaclass is set to `abc.ABCMeta`, making it an abstract base class[5]. This essentially means that no daughter class of this class can be instantiated unless all the methods decorated with `abc.abstractmethod` have been reimplemented by the daughter class. The methods concerned with this limitation are `optics.component.SetupComponent.lines` and `optics.component.SetupComponent.isHit`. Methods decorated with `abstractmethod` in an abstract base class can eventually be implemented in the mother class, but in theia they all `pass`, and could be called *pure virtual* for someone coming from C++.

5.3 Call graph

Here (figure 5) is the call graph of the `theia` CLI tool, from which one can easily deduce the call graph of any individual function. You can refer to the *API Guide* or the online API documentation to find out in which module the functions are defined.

Note on the call stack. According to this call graph (figure 5), the stack has a maximum height of $9 + 2(n-1)$ when there are n levels of recursion. Generally, the program crashes – if it crashes – by recursion depth limit exceeding (leading to a handled `RunTimeError` exception and an exit with an error code of 1) before causing a stack overflow.

5.4 Miscellaneous remarks

Coding style. In the development of `theia` we have tried to stick to a couple of coding style conventions, which may help to review the code and are important to know for anyone wishing to contribute.

[5] See `docs.python.org/2/library/abc.html` for details

- The code of `theia` is heavily commented and doc-stringed, and it should stay that way in order for `theia` to be an accessible library.
- Throughout the library, classes and attributes look `LikeThis` whereas objects and methods look `likeThis`.
- There is an approximate *one file* → *one class* correspondence and files are named accordingly with the objects they define. Generally, we have a tendency to distribute functions in different modules if they provide different functionalities, regardless of the total number of modules. Functions are together in a module if they belong together, consequently they are many modules in theia.
- We tend never to skip more than 1 line (Python is already very formatted).
- `# Provides` lines at the very beginning of modules allow to know at a glance what variables, functions and classes the module provides.
- Imports: import first from the Python standard library and third-party packages, then from `theia` sub-packages other than the current, then from the current `theia` sub-package. For `theia` sub-packages imports, always use the `from ... import` idiom, always use relative imports (`from ..helpers import interaction`) and for standard library and third-parties always `import` before you `from ... import`. We try to not import what we don't need.
- Class doc-string: present class attributes before instance attributes and mention if they are inherited.

Writing to `stdout` and files. Many classes reimplement the `__str__` method to have `print(object)` print a neatly formated description of the object. To this effect there are two important methods: `lines` (instance method) and `helpers.tools.formatter` (global scope function). `formatter` takes a list of strings (lines to output) and makes C-style indented output with curly braces in the right place in one large string. Basically, one has:

```
# inside class scope
def __str__(self):
        return formatter(self.lines())
```

References

[1] Kochkina, Wanner, Schmelzer, Tröbs, Heinzel: *Modeling of the General Astigmatic Gaussian Beam and its Propagation through 3D Optical Systems*, Applied Optics 24 (2013)

[2] Arnaud, Kogelnik: *Gaussian Light Beams with General Astigmatism*, Applied Optics 8 (1969)

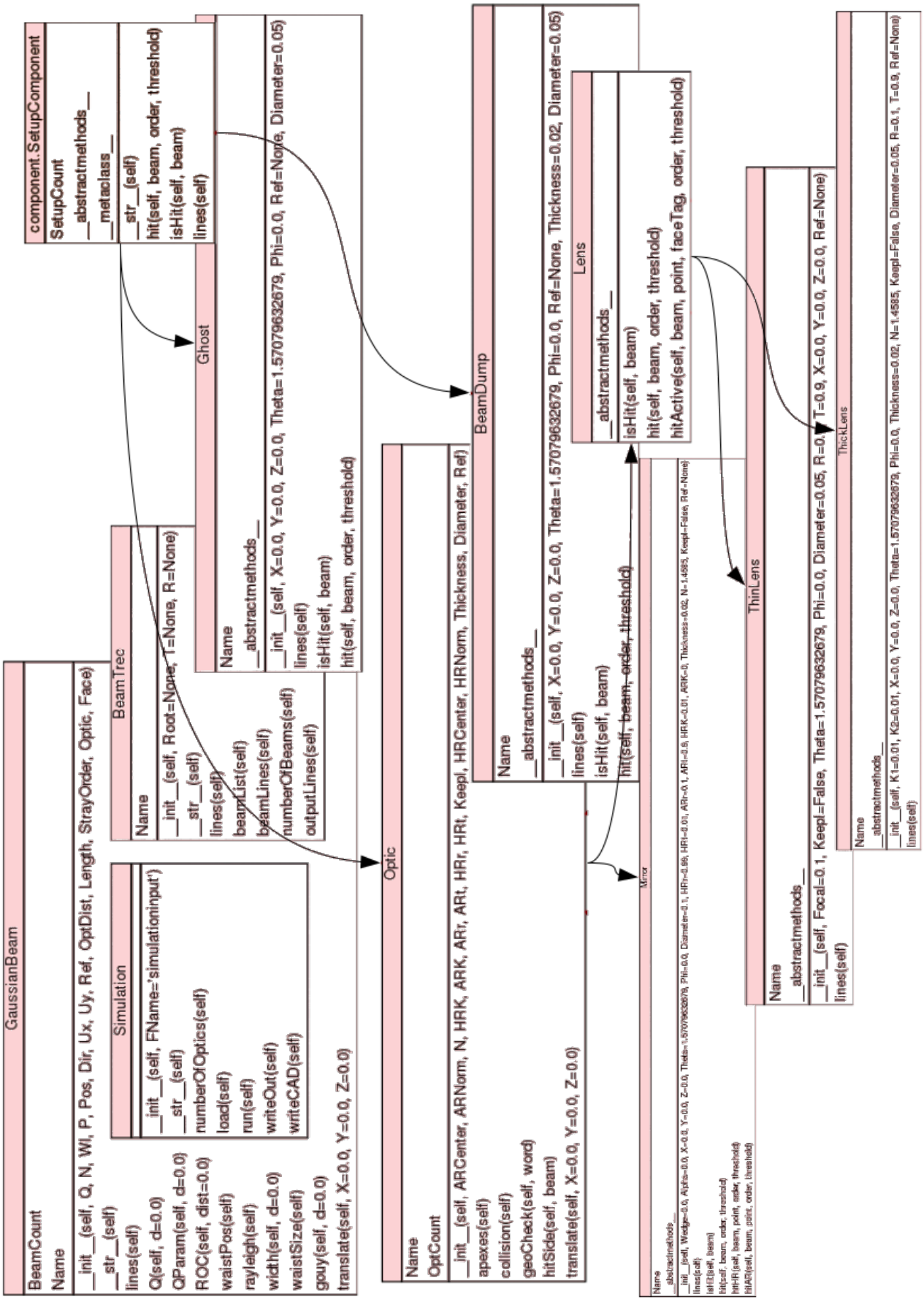

Figure 4: Inheritance hierarchy of theia

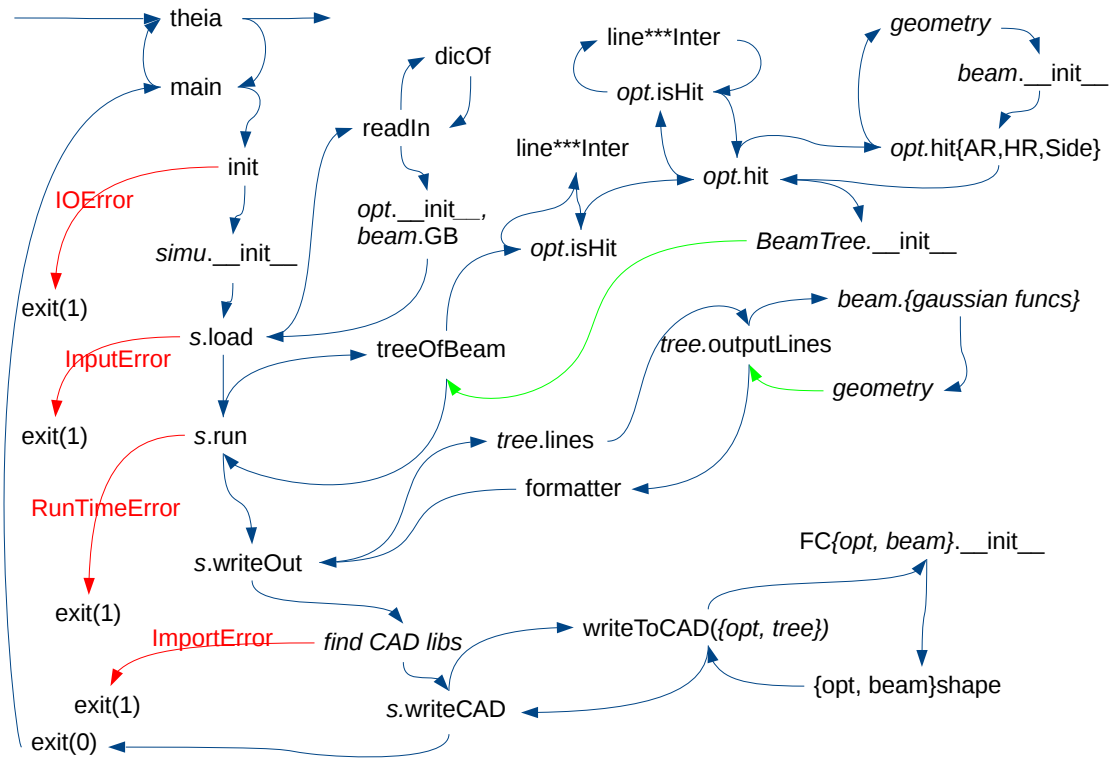

Figure 5: Call graph of the `theia` CLI tool. *italics*: generic object or group of methods, green: recursive call, red: handled exception.