`theia`

A 3D Gaussian beam tracer

Version 0.1.0
API Guide

Raphaël Duque

June 6, 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. theia was developed at the Optics Group of the Virgo gravitational observatory in Cascina, Italy. Please see the `README.md` file of `theia` or surf to `http://37.117.61.221:56000` for more information.

This document is an Application Programming Interface Guide for 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.

Throughout this document, 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 the theia package root directory (*e.g.* `running.simulation.Simulation.__init__`).

Contents

0.1 A note on global variables 2
0.2 Classes and inheritance hierarchy 2
0.3 Call graph 3
0.4 Miscellaneous remarks 3

0.1 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'}

init(dic)

# you're all set
```

Who uses the globals? Here is a table 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	main.main

Table 1: The global variables of theia and the functions who use them

0.2 Classes and inheritance hierarchy

Figure 0.2 presents the inheritance hierarchy of the classes of theia. If you see a method twice, it just means the daughter class reimplements the method.

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 class has a `classmethod` decorated method, 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 method 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.SetupComponen` class. Its metaclass is set to `abc.ABCMeta`, making it an abstract base class[1]. 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++.

0.3 Call graph

Here is the call graph of the theia CLI tool, from which one can easily deduce the call graph of any individual function.

Figure 1: Call graph of the theia CLI tool

For a more concise view, here is the dependency tree of the functions of the theia library.

Figure 2: Dependencies of the functions of the theia library

Note on the stack. According to this call graph (figure 1), 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.

0.4 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.

[1] 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 (it isn't true only in the `helpers` sub-package) *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())
```