Python bindings for the open source electromagnetic simulator MEEP.

Emmanuel Lambert[1], Martin Fiers[1], Shavkat Nizamov[2], Martijn Tassaert[1], Steven G. Johnson[3], Peter Bienstman[1], Wim Bogaerts[1]

1
Ghent University - IMEC, Department of Information Technology (INTEC), Photonics Research Group, Sint-Pietersnieuwstraat 41, B-9000 Gent, Belgium

2
Samarkand State University, Physics Faculty, 140104, University blvd. 15, Samarkand, Uzbekistan

3
Massachusetts Institute of Technology, Department of Mathematics, Center for Material Science & Engineering, Research Laboratory of Electronics, Cambridge MA 02139, USA

Part of this work was funded by the European Union in the framework of the FP7 integrated project HELIOS. Martin Fiers and Martijn Tassaert acknowledge the UGent Special Research Fund for a doctoral grant. Shavkat Nizamov acknowledges the RFBR grant 09-02-90205. Wim Bogaerts acknowledges the Flemish Fund for Scientific Research (FWO) for a postdoctoral grant.

Article abstract:

Meep is a broadly used and acknowledged open-source package for FDTD electromagnetic simulations. We describe how Python bindings for Meep leverage the tool. We outline new perspectives for integration of Meep with other libraries in the Python ecosystem. We show that Python bindings allow using Meep more effectively in a large scale parallel computing architecture. We describe the technical implementation of Python-Meep with SWIG and describe different architectures for interfacing data with the Meep core engine. Some applications of Python-Meep in our photonics and plasmonics research are briefly touched. We illustrate generic benefits to the wider research and open-source community.

Introduction

In photonics and microwave design, it is essential to be able to simulate the propagation of electromagnetic waves through sub-wavelength-scale structures with high accuracy. One of the common approaches for this purpose is the finite-difference-time-domain (FDTD) method [1]. Because it models Maxwell's equations in a fully vectorial way, it is one of the most powerful and general, but rather brute-force techniques. It is computationally intensive but well suited for massive parallelism, making it scalable on large clusters or supercomputers. While several commercial and open-source FDTD packages are commonly used, many researchers have appreciated the excellent open-source package Meep. Developed at MIT [2], it has a wide community of users. In this article we describe how Python bindings for Meep leverage the tool in several ways and how the research community benefits from this extension.

In the current standard version of Meep, a simulation is by default defined as a script written in the Scheme language. Scheme is a powerful and compact programming language, derived from LISP and belonging to the group of functional programming languages [3][4]. It is mostly popular for educational purposes. Newcomers can however experience a threshold in getting started with the language. Scheme is not inherently more difficult, but it has a somewhat different syntax, coding convention and execution strategy than more mainstream languages (the so-called imperative languages). Quite a lot of researchers interested in Meep are not familiar with this programming paradigm. On the other hand, Python follows a more traditional approach. Like Scheme, it offers the benefit of being a dynamically typed language and is thus well suited for scripting and rapid prototyping. It has become widely adopted during the past decade both in the industry (for example the Google Apps Engine platform) and in many open source projects. It is especially popular in scientific and academic communities: there are many Python libraries available (mostly open source), covering a wide spectrum of functionalities. Some of them will be discussed later in this article. Therefore, if Meep can be scripted through Python, it lowers the threshold for many researchers to use Meep and it allows for seamless integration with other existing Python software.

The use of Python in our research

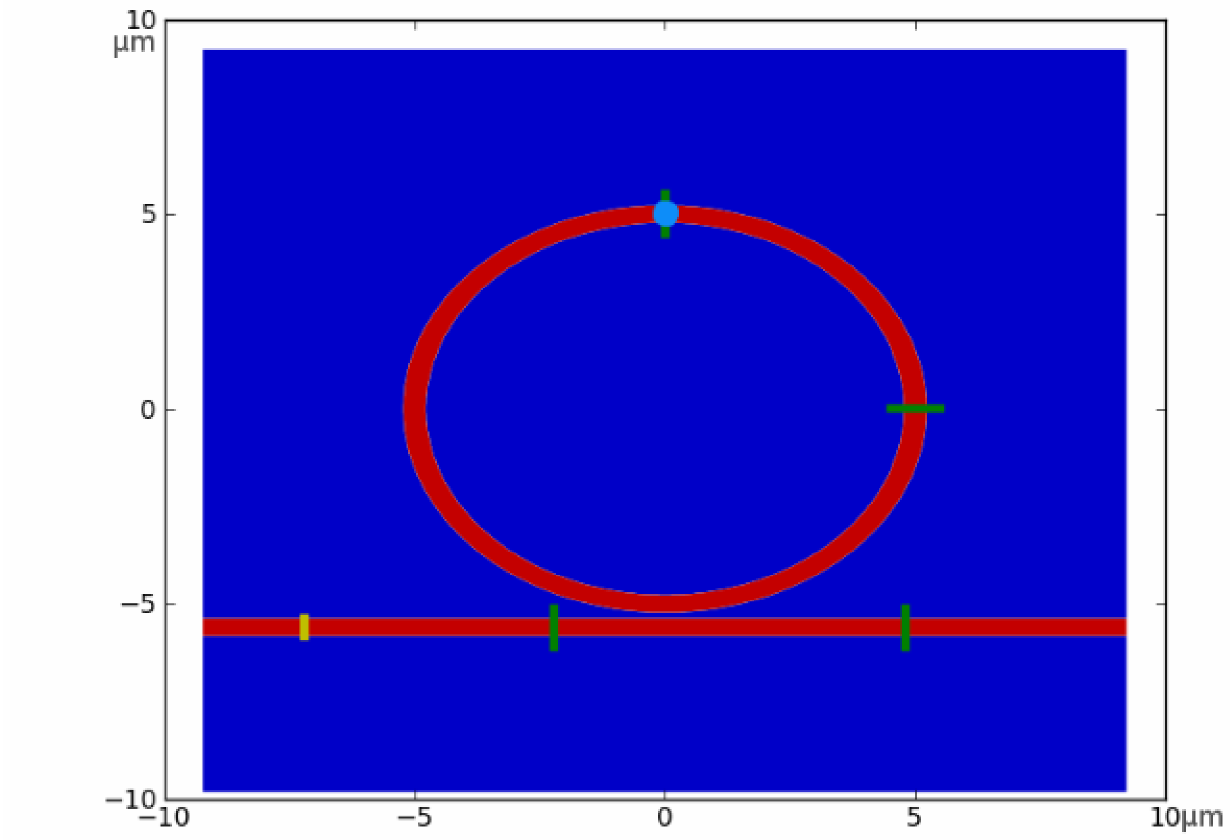
In our research on silicon photonics (UGent/IMEC) and plasmonics (SSU), we have deployed Python for many uses over the years. At UGent/IMEC, we have developed a litho mask design toolkit for silicon photonics in pure Python. Add-on tools and libraries have been developed for electromagnetic modeling, design optimization [5] and process simulation [6]. The long-term goal is to further automate closed-loop optimization of photonic circuits [7]. A powerful tool like Meep enriches our modeling framework. It broadens our research capabilities in design optimization because we can now leverage fully vectorial 3D FDTD simulations from inside a Python-driven design optimization process.

Leveraging Meep with Python

We see several generic benefits that Python bindings bring to the wider community of Meep users. Firstly, they enable the integration of Meep with existing Python open source libraries for scientific computing. The most acknowledged are Numpy and SciPy [8]. Numpy is an extension to the Python language which adds support for large, multi-dimensional matrix operations and related mathematical functions [9]. SciPy is a higher level library with mathematical tools and algorithms. Suppose for example that we want to explore a certain parameter space for the optimal configuration of a photonic waveguide (i.e. we want to simulate the electromagnetic behaviour of this waveguide with Meep for various parameter values). Optimization algorithms such as simulated annealing (provided by SciPy) or genetic algorithms (provided by PyGene), can now be used to explore this parameter space on a supercomputer and optimize against a certain target function. Numerical algorithms offered by Numpy can be used for processing of simulation results. Combining these libraries with Meep is a promising option for many researchers already familiar with them.

Visualization of the electromagnetic fields relies on external tools in the currently deployed versions of Meep (with files for interchange of data) and it is largely a manual process. With Meep now being Python-aware, we can develop visualization functionality using popular Python libraries such as Matplotlib (for 2D) [10] and Mayavi2 (for 3D) [11] and tightly integrate them with the simulation script. We can automatically generate the visualization of the waveguide, the position of the excitation source and the data-collecting flux planes. This allows for rapid, visual verification of the Meep script before running it. At UGent, we have built such functionality on top of the standard Python-Meep, which we integrated with a more general simulation framework used by our research group (for this latter reason, it is currently kept as a proprietary extension, not included in the public release of Python-Meep). The figure below illustrates a 2D-visualization made by this framework. Because the Python bindings provide direct access to core Meep functionality, we could even make a live visualization of the fluxes or the electromagnetic fields as the simulation progresses. The latter has however not yet been implemented. Generally speaking, such automated and advanced visualization functionalities save time and can save reiterations of failed or ill-conditioned simulations.

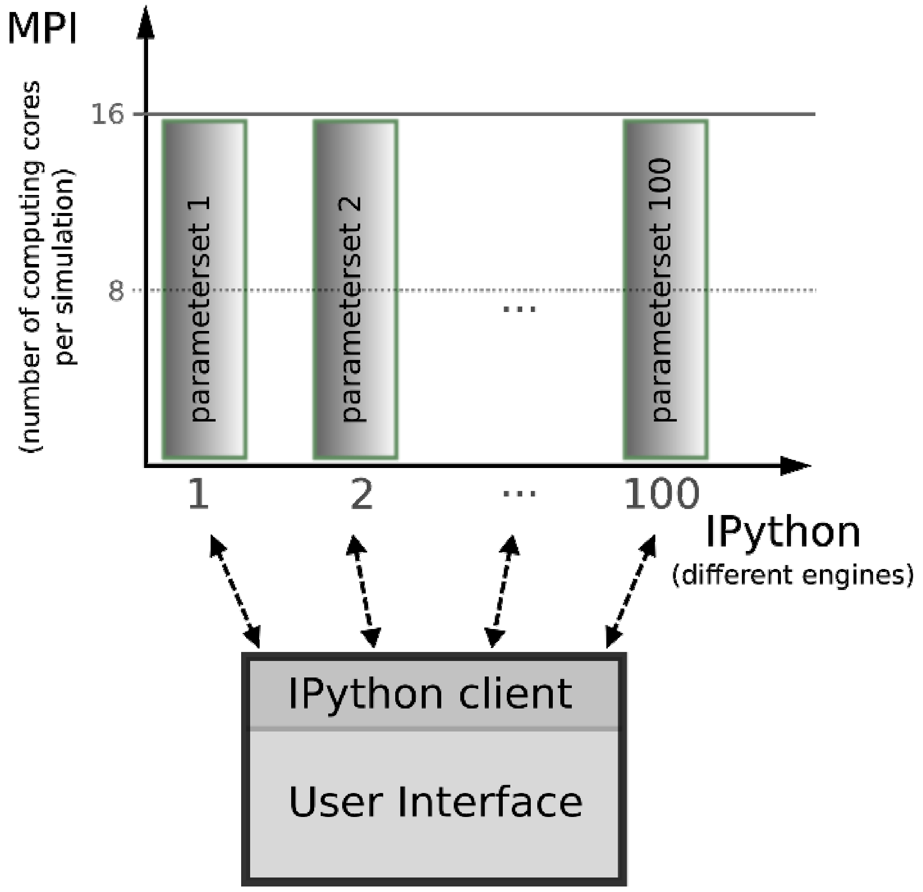
Figure 1 illustrates the automatic visualization of a 2D simulation landscape based on Python-Meep and Matplotlib : it shows a ring resonator with access waveguide in silicon (red), the position of the source (yellow line), two fluxplanes (green line) and a probing point (blue circle).



The standard version of Meep can be enabled for MPI-run, which means that the computation is distributed over multiple computing cores (on one or more nodes). MPI is an industry standard which defines message passing between software components executing in parallel [12]. An FDTD algorithm can easily be parallelized using MPI. We can split up the simulation problem in cells: in a given time step, the calculation for one cell is only dependent of the previous state(s) of the cell and the boundaries of the surrounding cells. Each computing core processes one cell and exchanges boundary information with its neighbors. The Python-Meep bindings are fully compatible with the MPI-capabilities of Meep. However, such an MPI-distribution does not scale infinitely: adding cores increases communication and synchronization overhead, which at some point limits further scaling. Even if we have a massive amount of cores at our disposal (such as on a supercomputer or cluster), we cannot efficiently exploit the full capacity with one MPI-run alone.
At UGent we are developing a generic photonic simulation framework based on IPython [13]. This is a Python environment which is enhanced for parallel computing. It largely abstracts the technical aspects of parallel computing from the user and allows robust error handling. It allows submitting scripts to a controller, which in turn scatters the code to engines on several nodes for execution. Results and exceptions are gathered back and presented to the

client shell in a user friendly manner.
The Python bindings for Meep enable the integration of Meep with this IPython framework. Such integration shows a clear benefit. We can now combine MPI-runs of Python-Meep with the scatter-gather capabilities of IPython. In this architecture, we basically have a 2-dimensional space over which we can spread a large number of simulations (e.g. in a parametric scan), as illustrated in figure 2a. The first dimension is the number of computing cores to which we can scale one simulation in an MPI-run. The other dimension is the number of different simulations that we want to run simultaneously (with each simulation assigned a set of MPI-enabled IPython engines). In this scheme, we can use the capacity of a cluster or supercomputer in an optimal way for a large set of simultaneous Python-Meep simulations. A user interface allows to launch simulations for a certain set of parameters and to view the progress of a specific simulation (figure 2c).

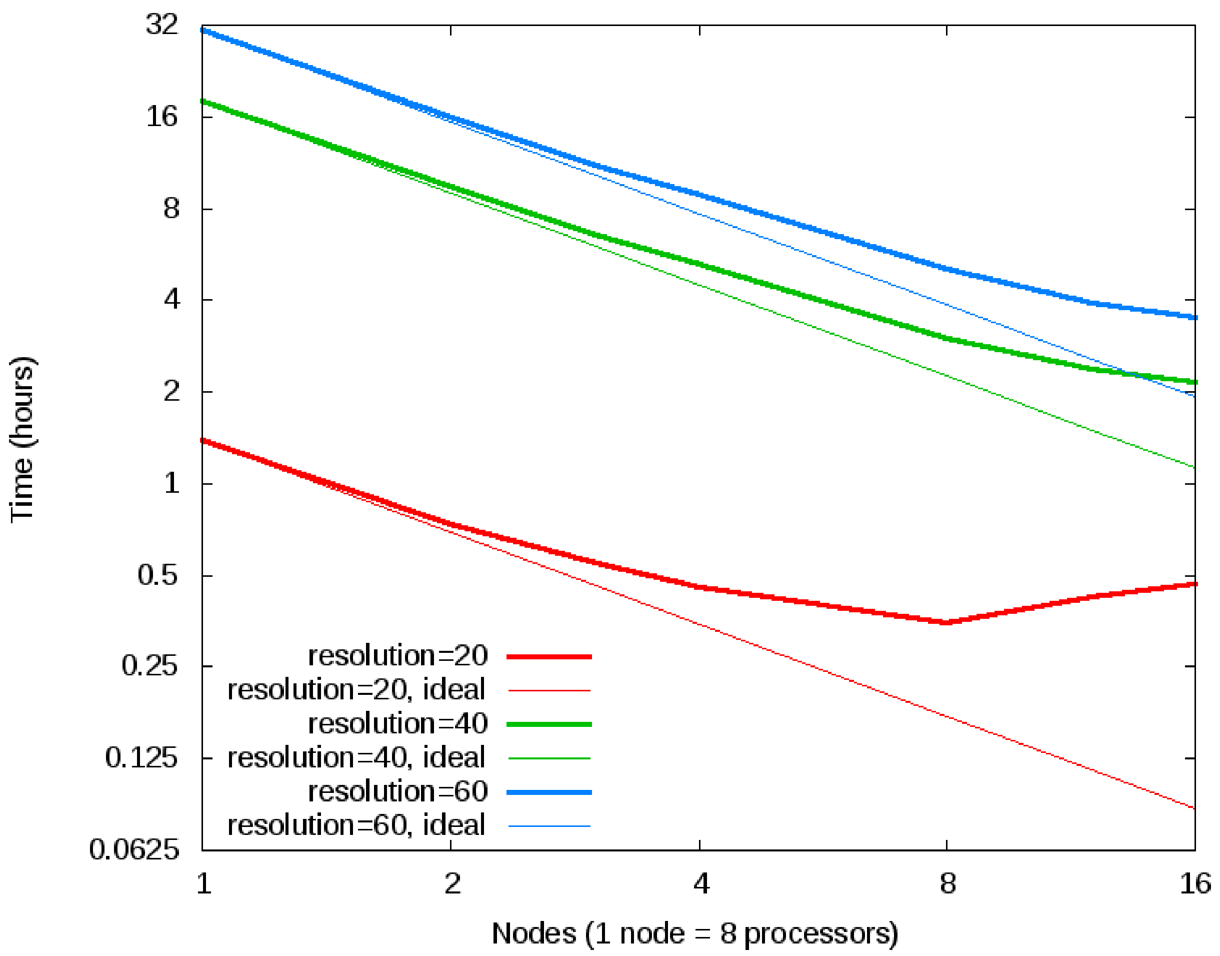
Figure 2a shows a schematical representation of 100 simulations (each with different parameter set) on a supercomputer. Each simulation executes in an IPython engine and is scaled with MPI over 16 computing cores.



Suppose for example that we have a computer cluster with 1600 cores at our disposal and that we want to scan a parameter space with 150 combinations of parameters. Let's assume that each simulation can be efficiently scaled over 16 cores with MPI. Combining MPI and IPython, we can run 100 Python-Meep

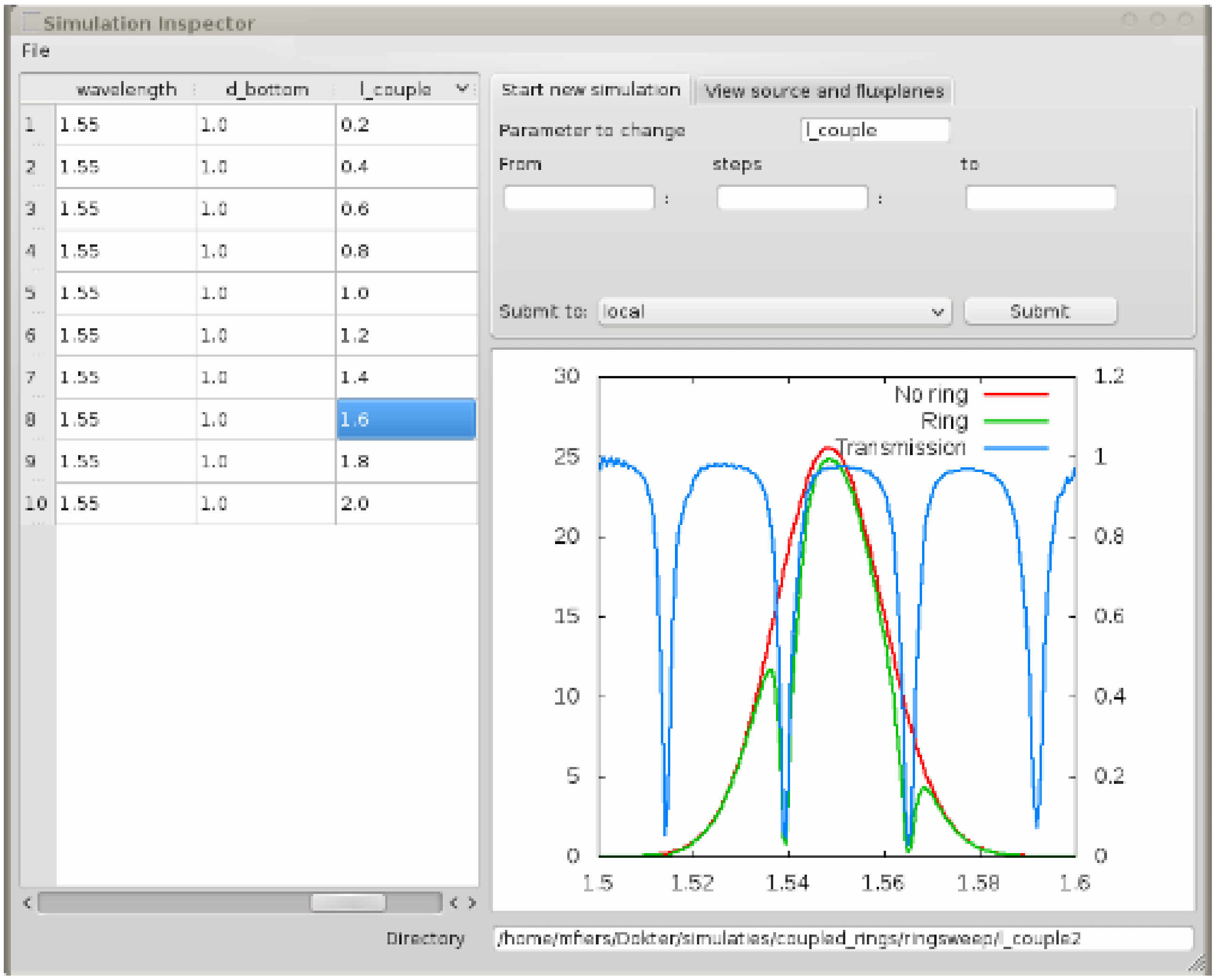
simulations simultaneously, with each simulation consuming 16 cores. If each simulation takes 30 minutes to complete, then we can execute the full parameter space in just one hour (30 minutes for 100 simultaneous simulations on 16 cores per simulation, followed by another 30 minutes for the subsequent 50 simultaneous simulations).
Both dimensions are independent of one another and have different scaling properties. The scaling behaviour of Python-Meep over the first dimension (the number of cores for MPI-run) is similar to the standard Meep: the Python layer does not interfere with the MPI-specific commands in the Meep core. Figure 2b shows the scaling of a benchmark 3-dimensional simulation with MPI. The total calculation time is shown for different resolutions (i.e. sizes of the computational volume). This is compared with the scaling that we ideally expect: i.e. when we double the number of nodes, we expect the calculation time to halve. For a given resolution, there is an upper limit to the number of cores over which we can scale efficiently. For a 3-dimensional simulation, the communication and synchronization overhead increases with the 4th power of the number of computing cores. At some point, the added benefit of extra calculation power is smaller than the additional overhead that is created: in such a case, the total calculation times even increases. In figure 2b, we can see that scaling performance is better for more complex, high resolution problems.

Figure 2b illustrates the scaling of a 3D Python-Meep simulation with MPI. The actual calculation times are show for different resolutions and compared with the calculation times that we ideally expect.



For the second dimensions (the IPython engines), there is no inherent scaling limit as the different IPython engines are essentially separated programs running in parallel, with no intercommunication.
Figure 2c below shows a graphical user interface that was built with PyQt [14] on top of this IPython based framework: we can conveniently launch new Python-Meep simulations and inspect results of simulations that have terminated.

Figure 2c illustrates the graphical user interface of the photonic simulation framework of UGent. It shows the parameters used in a range of Python-Meep simulations with the corresponding result for each simulation, i.e. the transmission calculated from the fluxes. It offers the possibility to inspect results and subsequently launch new simulations (with different parameters) to a computing cluster. This high level of automation aids in the rapid design of new components.



	wavelength	d_bottom	l_couple
1	1.55	1.0	0.2
2	1.55	1.0	0.4
3	1.55	1.0	0.6
4	1.55	1.0	0.8
5	1.55	1.0	1.0
6	1.55	1.0	1.2
7	1.55	1.0	1.4
8	1.55	1.0	1.6
9	1.55	1.0	1.8
10	1.55	1.0	2.0

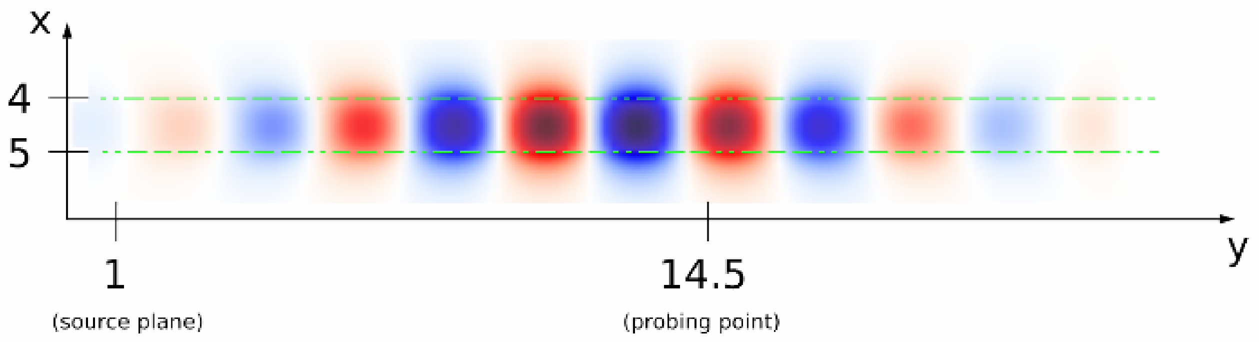
Parameter to change: l_couple
From : steps : to
Submit to: local
Directory: /home/mfiers/Dokter/simulaties/coupled_rings/ringsweep/l_couple2

A taste of Python-Meep

In figure 3a, we give a short example of a Python-Meep script, so that readers can get a flavor of the coding conventions. In this example, we calculate the 2D-electromagnatic field profile in response to a line source located at the left of a straight waveguide. The Ez component of the field is periodically written to a HDF5 file, which can then be further processed by the user (HDF5 is a standard file format for scientific datasets [15]). In figure 3b we show an equivalent script implemented with Scheme. From these code samples, it can be seen that the Scheme version defines the problem more in terms of higher level expressions. Functional languages like Scheme are inherently very expressive [16][17] and this feature was fully exploited by the authors of Meep when they created the Scheme interface. That way, they overcame the fairly low level style of the Meep C++ core. Additionally, the Scheme interface was complemented with user-friendly functionality which is not available in the underlying Meep C++ core (and thus not by default in Python-Meep).
The Python-bindings directly expose the low-level Meep C++ core and this is reflected in the coding style of the Python script. In Python-Meep, we are now also adding similar high level helper functions to facilitate the writing of

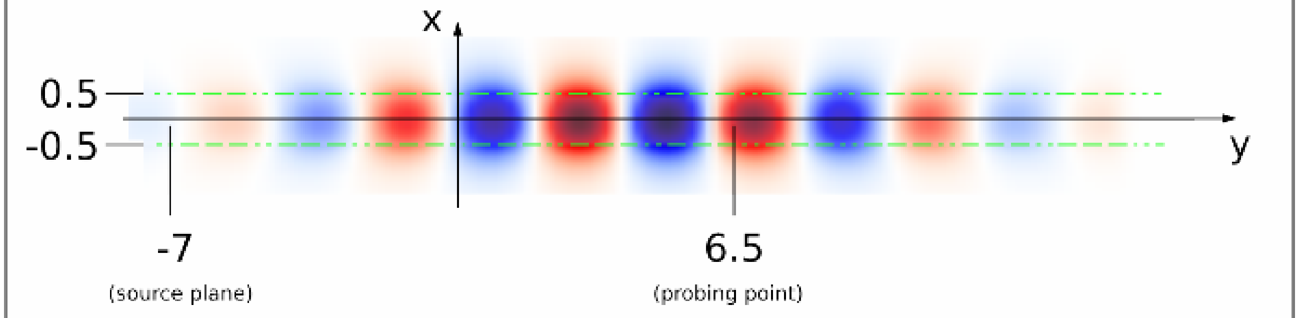
simulation scripts and we will increase this effort in future versions. While such functions are useful, they are however not necessary to use the functionalities that Meep offers.
Users of the Scheme interface are limited to using the functionality offered at that level while users of Python-Meep have more flexibility: they can use both the low-level functionality of the Meep C++ core and the higher-level helper functions that are being added to the Python interface.

Figures 3a/3b : a basic Python-Meep simulation script (a) and it's equivalent in Scheme (b). Note that the coordinate system is different in both versions.



```
from meep_mpi import *

#define the waveguide material as a function of a vector(X,Y) :
#we create a straight waveguide of width 1 over the full length
class epsilon(Callback):
    def double_vec(self,vec):
            if ((vec.y() >= 4) and (vec.y() <= 5)):
                return 12
            else:
                return 1

#create the computational grid of size 16 x 32 with resolution of 10
vol = voltwo(16,32,10)

#create a structure with PML of thickness = 1, using the class 'epsilon'
material = epsilon()
set_EPS_Callback(material.__disown__())
s = structure(vol, EPS, pml(1))

#define a gaussian line source of length 1 at X=1, Y=4
#with center frequency 0.15 and pulse width 0.1
srcGaussian = gaussian_src_time(0.15, 0.1)
srcGeo = volume(vec(1,4),vec(1,5))

#create the fields
f = fields(s)
f.add_volume_source(Ez, srcGaussian, srcGeo)

#export the dielectric structure
epsFile =  prepareHDF5File("./sample-eps.h5")
f.output_hdf5(Dielectric, vol.surroundings(), epsFile)

#define the file for output of the field components
ezFile = prepareHDF5File("./sample.h5")

#define a probing point at the end of the waveguide
#to check if source has decayed
probingPoint = vec(1.5,4.5)

#start the simulation, sending HDF5 output to the file 'ezFile'
runUntilFieldsDecayed(f, vol, Ez, probingPoint, pHDF5OutputFile = ezFile)
```

(a)



```
;define the simulation volume
(set! geometry-lattice (make lattice (size 16 8 no-size)))

;define the geometry of the straight waveguide and the PML layer
(set! geometry (list
                (make block (center 0 0) (size infinity 1)
                      (material (make dielectric (epsilon 12))))))
(set! pml-layers (list (make pml (thickness 1.0))))

;define the Gaussian source
(set! sources (list
               (make source
                  (src (make gaussian-src (frequency 0.15) (fwidth 0.10)))
                  (component Ez)
                  (center -7 0))))

;define the resolution
(set! resolution 10)

;start the simulation, sending HDF5 output to file
(run-sources+
  (stop-when-fields-decayed 50 Ez
               (vector3 7.5 0 0)
                1e-3)
  (at-beginning output-epsilon)
  (at-every 0.6 output-efield-z))
```

(b)

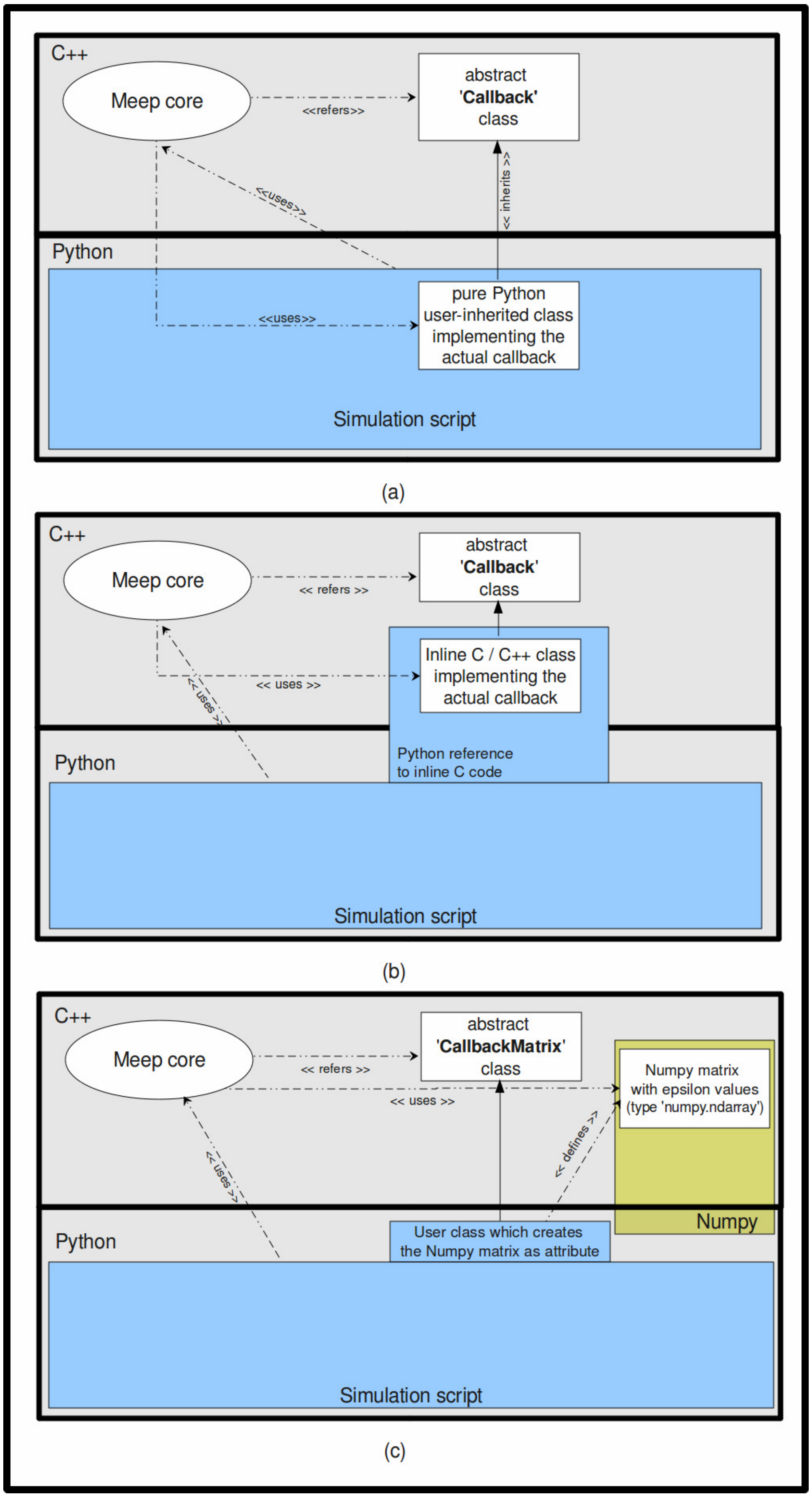
Technical implementation of the Python bindings

Integrating the Meep callback mechanism

The Meep core library (written in C++) provides a mechanism of callbacks for integration with the simulation script: whenever the runtime engine needs information about specific properties of the simulation, a function defined by the user is called. This mechanism is used intensively, for example in the definition of the material properties of the simulation volume or in the definition of a custom electromagnetic source.
The Python-Meep bindings were developed using SWIG, an open source tool

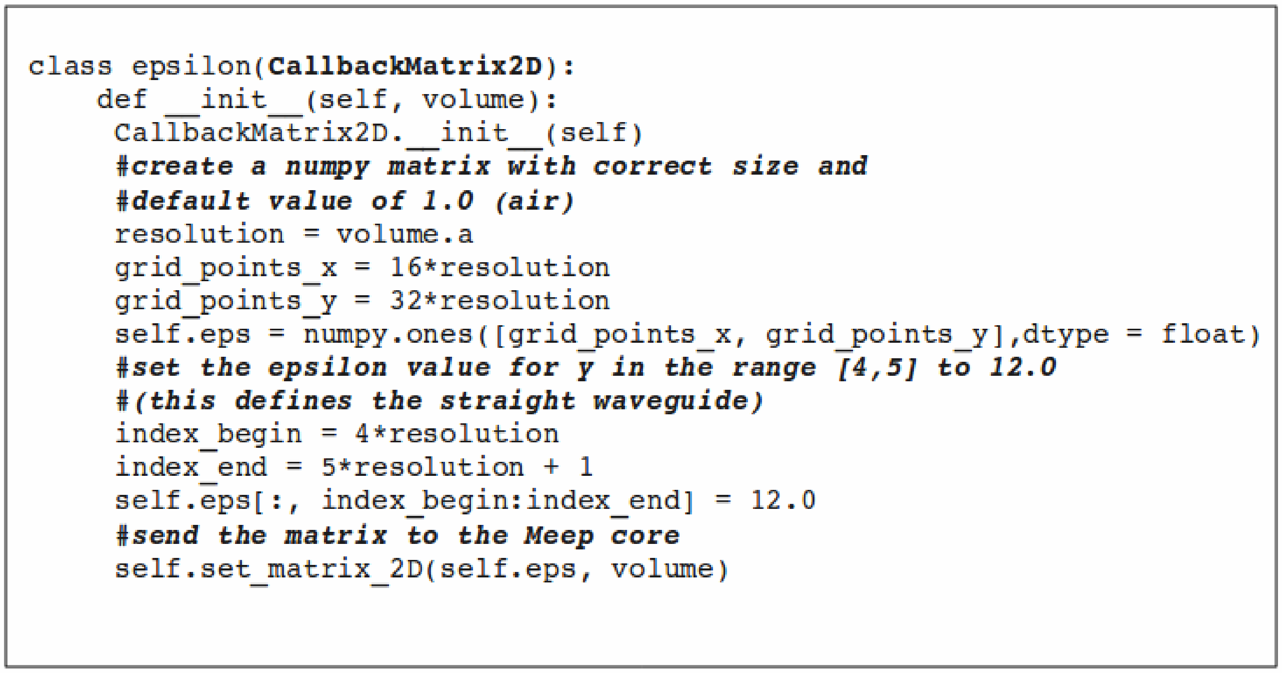
that allows connecting programs written in C/C++ with a variety of high-level programming languages [18]. The flexibility of SWIG allows for an elegant integration with this callback mechanism. Based on our experiences with performance and ease of use for the end user, the actual implementation technique evolved in three phases (described below and illustrated in figure 4).

Figure 4 illustrates the alternative architectures that were implemented for definition of the material properties in the simulation volume. First architecture: using a pure Python class for callback (a). In this case, the C++/Python boundary is crossed whenever callback occurs (potentially millions of times for material definition). Second architecture: using inline C/C++ for large simulation volumes with many grid points (b): the callback occurs completely in the C/C++ domain (great performance). Third architecture: the user works in Python only, creating a Numpy matrix with the material definition (c). Meep can directly access this matrix using a pointer, while the user works in pure Python (also with great performance but with increased memory consumption).


C++
Meep core
<<refers>>
abstract 'Callback' class
<<uses>>
<< inherits >>
Python
pure Python user-inherited class implementing the actual callback
<<uses>>
Simulation script
(a)
C++
Meep core
<< refers >>
abstract 'Callback' class
Inline C / C++ class implementing the actual callback
<< uses >>
<< uses >>
Python
Python reference to inline C code
Simulation script
(b)
C++
Meep core
abstract 'CallbackMatrix' class
<< refers >>
<< uses >>
Numpy matrix with epsilon values (type 'numpy.ndarray')
<< defines >>
<< uses >>
Numpy
Python
User class which creates the Numpy matrix as attribute
Simulation script
(c)

In a first straightforward implementation, Python-Meep provides an abstract `Callback` class, from which the user inherits in pure Python. In that class, the user implements the required functionality, such as definition of the material properties (see figure 3). For many complex simulations however (i.e. with high resolution), the performance of this pure Python callback was not sufficient : the callback function for definition of the materials is typically called a million times or more. The overhead of swapping from C++ to Python, subsequently running a piece of interpreted Python code and returning the results back to C++ is small, but it becomes problematic when the callback is executed hundreds of thousands or millions of times.
Initially, this drawback was solved by allowing users to define a callback function in C or C++, with the rest of the simulation script in Python. In this scheme, the user's C++ code is compiled at runtime and dynamically linked with the Python-Meep bindings: the callback is then done completely inside the C++ domain. This solution provides the required performance. The Python package "weave" allows for very elegant inclusion of inline C/C++. It largely abstracts the overhead for the user of mixing Python with C/C++. Nevertheless, combining 2 languages remains a drawback for certain end users, many of whom are not familiar with C/C++.
In the original Scheme interface, the performance issue with this repeated callback occurs less often: in this implementation, the standard callback mechanism is largely bypassed by the authors of Meep. A tighter integration of the C++ core and definitions in Scheme is realized.
We subseqently worked towards a similar solution that would allow a pure Python definition of even complex high-resolution simulations. The breakthrough came by combining SWIG with Numpy matrices. Numpy is known for its great performance, thanks to the fact that Numpy stores and processes its data in C and exposes only a thin interface to Python. Therefore, if we define a Numpy matrix in Python with the material properties of our simulation volume, that matrix is directly accessible from Meep using C coding conventions (basically a pointer). The integration then comes down to writing a wrapper around the Meep callback functionality. This wrapper retrieves the actual values from the Numpy matrix and returns them to Meep. Figure 4 further illustrates this architecture in contrast with the other two. Code-wise, we provide a user-friendly class `CallbackMatrix` from which the user inherits. In the class, he creates a Numpy matrix with size corresponding to the discretized simulation volume (or a multiple for better accuracy). This architecture offers excellent performance, while allowing the user to work in pure Python. A drawback is the increased memory consumption, as we have to store the Numpy matrix before it is interfaced to Meep. Figure 5 illustrates the technique for the straight waveguide example of figure 3.

Figure 5 : use of the technique with Numpy matrix for describing the straight waveguide of figure 3. The user inherits from CallbackMatrix2D and assigns the Numpy matrix to an attribute.



```
class epsilon(CallbackMatrix2D):
    def __init__(self, volume):
     CallbackMatrix2D.__init__(self)
     #create a numpy matrix with correct size and
     #default value of 1.0 (air)
     resolution = volume.a
     grid_points_x = 16*resolution
     grid_points_y = 32*resolution
     self.eps = numpy.ones([grid_points_x, grid_points_y],dtype = float)
     #set the epsilon value for y in the range [4,5] to 12.0
     #(this defines the straight waveguide)
     index_begin = 4*resolution
     index_end = 5*resolution + 1
     self.eps[:, index_begin:index_end] = 12.0
     #send the matrix to the Meep core
     self.set_matrix_2D(self.eps, volume)
```

As we see in the last line of the code snippet of figure 5, the Python-Meep function `set_matrix_2D` is used for interfacing the Numpy matrix with the underlying C++ code. In the C++ code of the Python-Meep wrapper, the function signature is :

```
void set_matrix_2D(double* matrix, int dimX, int dimY, ...)
```

Similarly, for a 3D-simulation we have :

```
void set_matrix_3D(double* matrix, int dimX, int dimY, int dimZ, ...)
```

The first parameter is of type `double*` and is a pointer to the actual values in the Numpy matrix. The following two or three `int` parameters indicate the matrix dimensions. In Python the matrix is of type `numpy.ndarray`.
We want to seamlessly pass the Numpy matrix as parameter to the functions `set_matrix_2D` and `set_matrix_3D`. It is therefore required to define some kind of translation between the Python type `numpy.ndarray` and an equivalent tuple of parameters `double*` and `int` in C++. In SWIG, the technique for such a translation is called a typemap. Normally, the definition of typemaps is a complicated and tedious task. Luckily, a range of typemaps for Numpy are already available in the open source community ("numpy.i" [19]). They are called `IN_ARRAY2` and `IN_ARRAY3` for respectively 2- and 3-dimensional Numpy arrays.
In our SWIG definition file, we have to link up the signature of the `set_matrix_2D` function with the typemap. This is done using the code below. When we pass a Numpy array to the function in Python, it is automatically expanded in the three or four corresponding parameters of the C++ function.

```
//Include the Numy header file, so that Numpy types are known
%{
#define SWIG_FILE_WITH_INIT
#include <numpy/npy_common.h>
%}

//Include the Numpy typemaps
%include "numpy.i"

%init %{
  import_array();
%}

%apply (double* IN_ARRAY2, int DIM1, int DIM2)
       {(double* matrix2, int dimX, int dimY)};

%apply (double* IN_ARRAY3, int DIM1, int DIM2, int DIM3)
       {(double* matrix3, int dimX, int dimY, int dimZ)};
```

Similarly, typemaps were needed for interfacing parameters that represent complex numbers. Both Python and C++ have seperate definitions of a `complex` type and thus a mapping or translation is required for seamless integration. The definition of these typemaps is quite complicated. Interested readers can consult the file `py_complex.i` in the public Python-Meep distribution.

All three of the above techniques for defining material matrices are available to users of Python-Meep. The approach with the Numpy matrix is the preferred one for simulations of moderate size. For very large simulation volumes, using a C/C++ callback function may currently be more appropriate, as it has lower memory requirements. In future versions, we are planning to explore PyTables [20] as an approach for processing very large matrices: PyTables combines HDF5 and Numpy and allows storing huge matrices on disk, thus limiting the memory consumption.

The choice for SWIG

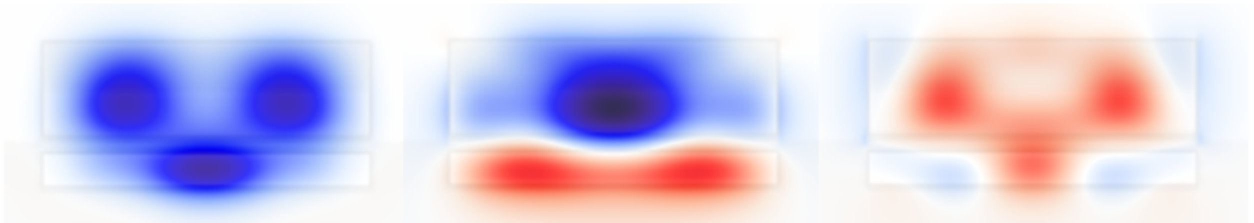
Initially we compared both “SWIG” [18] and “Boost.Python” [21] as alternative approaches for implementing our Python wrapper.
Boost is a well established and recognized set of open source C++ libraries which runs on almost any operating system. “Boost.Python” is a subset which supports seamless interopability between Python and C++. We had very good experiences with “Boost.Python” during our evaluation: a tutorial is available, the semantics of the API are clear and the amount of code that we had to write was limited. However, there was one important drawback: during the technical build process, our code needed to be linked to Boost-specific dynamic libraries (dll’s). While these libraries can be compiled from source, they have a large footprint. This is a major dependency which poses an additional threshold for deployment on third party systems like a supercomputer for example. We preferred to keep Python-Meep lightweight with as little dependencies as possible. Therefore, we decided to use SWIG.

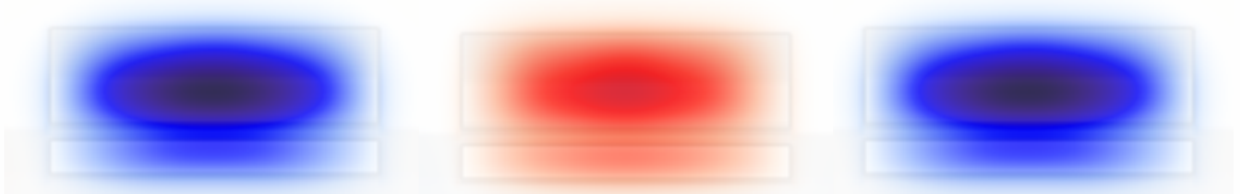
SWIG is a dedicated framework for connecting C/C++ programs with a large variety of programming languages. One must write an interface file from which the SWIG engine generates two additional files: one file with C code and one file with Python code. There are no other dependencies. Once this code is generated, it can be transferred to any operating system and compiled there. The footprint is thus limited and a SWIG installation is not needed on the host system. The SWIG documentation is very detailed but the semantics of various constructs are not always easy to understand. The technical implementation was rather complicated and we needed a lot of trial and error before the required behaviour was obtained. Especially the definition of typemaps was error prone and hard to debug. These were serious drawbacks, but once up and running, the Python/C++ interface works without a flaw.

Interfacing external data with a Python-Meep script

A frequently asked question in FDTD mailing lists concerns the problem of specifying "external" sources, i.e. electromagnetic sources that are defined by some other software and exported in the form of a datafile. Python has extensive features for interchanging data which come in handy in such a case. One example is the excitation of a specific mode of a photonic waveguide (a photonic waveguide can typically guide waves with specific profiles, called modes). In realistic simulations, it is often required that only one specific mode is excited at a time. The only solution then is to create a source with the exact spatial amplitude shape of the mode that we want to excite. This problem is conveniently addressed with Python-Meep. The commercial package Fimmwave is well known for calculation of such modes [22]. We can use Fimmwave to calculate the spatial amplitude profile of the mode that we want to excite and export the resulting matrix to a text file. In Python-Meep, we create a callback function that uses this matrix to calculate the exact amplitude profile of the source. We then run the Python-Meep simulation with a custom source that matches accurately with the physical properties of the waveguide. At UGent, we have implemented such an integration scheme between Fimmwave and Python-Meep in a couple of simulations. During these efforts, the availability of the Python library Numpy proved useful: the resolution of the matrix that is exported by Fimmwave may not necessarily be the same as the resolution that we want to use later on in the Meep FDTD simulations. Using Numpy, we could conveniently interpolate values to get the field profile value at each wanted position in the FDTD grid.

Figure 6 illustrates the field profile without spatial shaping of the source (a), versus a field profile when the source is shaped according to an amplitude matrix calculated by Fimmwave and imported by Python-Meep (b). A field profile that is useful for a realistic design should have a constant spatial distribution of the power intensity over time for a given cross-section: in (a), we see that there are major changes over time in the spatial distribution of the power intensity for the chosen cross-section. In constrast, the profile in (b) shows a constant spatial distribution of the power intensity over the full length of the waveguide.


(a) Cross section of waveguide, with unshaped source at different times


(b) Cross section of waveguide, with shaped source at different times

Open source

The Python-Meep bindings are distributed by its authors under the terms of the GNU General Public License (v2). The source code is publicly available on Launchpad [23] and the community is invited to further contribute to the project's development.

Conclusion

We conclude that the recently released Python bindings for Meep bring interesting benefits for the wider research and open source community. First of all, Python is a convenient alternative for those researchers who want to use Meep but are not familiar with the Scheme programming language. The Python bindings enable the integration of Meep with other software libraries in the Python ecosystem (such as libraries for visualization and libraries with numerical and scientific algorithms). We can also leverage the parallel computing capabilities of Meep by combining MPI with the IPython framework. We discussed the technical implementation of the Python-Meep bindings with SWIG and three different architectures for interfacing data with the Meep core engine. We have illustrated how we use Python-Meep in our silicon photonics and plasmonics research. Some options for improvement in future versions were discussed. We have released the Python-Meep bindings as open source: in this way, the community of users can contribute to its further development.

References :

[1] - Allen Taflove and Susan C. Hagness (2005). Computational Electrodynamics: The Finite-Difference Time-Domain Method, 3rd ed. Artech House Publishers. ISBN 1-58053-832-0. http://www.artechhouse.com/Detail.aspx?strBookId=1123.

[2] - Ardavan F. Oskooi, David Roundy, Mihai Ibanescu, Peter Bermel, J. D. Joannopoulos, and Steven G. Johnson, "MEEP: A flexible free-software package for electromagnetic simulations by the FDTD method," Computer Physics Communications, vol. 181, pp. 687-702 (2010).

[3] - Gerald Jay Sussman and Guy Lewis Steele, Jr. (December 1975), "Scheme: An Interpreter for Extended Lambda Calculus" (postscript or PDF), *AI Memos* (MIT AI Lab) AIM-349

[4] - IEEE Standard for the Scheme Programming Language, IEEE part number STDPD14209.

[5] - D. Vermeulen, G. Roelkens, J. Brouckaert, D. Van Thourhout, R. Baets, R. Duijn, E. Pluk, G. Van den Hoven, "Silicon-on-insulator nanophotonic waveguide circuit for fiber-to-the home transceivers", ECOC, Belgium, p.Tu.3.C.6 (2008)

[6] - P. Bienstman, L. Vanholme, W. Bogaerts, P. Dumon, P. Vandersteegen, "Python in Nanophotonics Research", Computing in Science & Engineering, 9(3), p.46-47 (2007)

[7]- W. Bogaerts, P. Bradt, L. Vanholme, P. Bienstman, R. Baets, "Closed-loop modeling of silicon nanophotonics from design to fabrication and back again", Optical and Quantum Electronics, 01/2009 - 40(11) p.801-811

[8] – Numpy and SciPy project page : http://www.scipy.org

[9] - Travis E. Oliphant, "Python for Scientific Computing", Comput. Sci. Eng. 9, 10 (2007). From the same author, the book "Guide to Numpy" (December 7, 2006) was released in the public domain. It can be downloaded at http://www.tramy.us/numpybook.pdf

[10] – Matplotlib is an open source Python library for 2D plotting. http://matplotlib.sourceforge.net/

[11] – Mayavi2 is a Python library for 3D Scientific Data Visualization and Plotting. http://code.enthought.com/projects/mayavi/

[12] - Gropp, William; Lusk, Ewing; Skjellum, Anthony (1994). "Using MPI: portable parallel programming with the message-passing interface". MIT Press In Scientific And Engineering Computation Series, Cambridge, MA, USA. 307 pp. ISBN 0-262-57104-8

[13] - Fernando Perez, Brian E. Granger, "IPython: A System for Interactive Scientific Computing," Computing in Science and Engineering, vol. 9, no. 3, pp. 21-29, May/June 2007, doi:10.1109/MCSE.2007.53.

[14] - PyQt are Python bindings for Nokia's Qt application framework. It runs Windows, MacOS/X, Linux. http://www.riverbankcomputing.co.uk/software/pyqt/intro

[15] – HDF5 is a set of file formats and libraries designed to store and organize large amounts of numerical data. Originally developed at the National Center for Supercomputing Applications, and currently supported by HDF Group. http://www.hdfgroup.org

[16] - John Hughes. "Why Functional Programming Matters", in D. Turner, editor, Research Topics in Functional Programming. Addison Wesley, 1990.

[17] - M. P. Atkinson,Peter Buneman,Ronald Morrison, “Data types and persistence”, par 4.2.1

[18] - David M. Beazley - "Using SWIG to Control, Prototype, and Debug C Programs with Python". 4th International Python Conference, Livermore, California, June, 1996.

[19] – Bill Spotz, Sandia National Laboratories, “numpy.i: a SWIG Interface File for NumPy”, Decmber 2007, document available in the Numpy distribution.

[20] – PyTables is a package for managing hierarchical datasets designed to efficiently cope with extremely large amounts of data. http://www.pytables.org

[21] - Boost.Python, a C++ library which enables seamless interoperability between C++ and Python. http://www.boost.org/doc/libs/1_43_0/libs/python/doc/index.html

[22] - Fimmwave by Photon Desgin : http://www.photond.com/products/fimmwave.htm

[23] - Python-meep project page: https://launchpad.net/python-meep