Python Bindings for the Open Source Electromagnetic Simulator Meep

Meep is a broadly used open source package for finite-difference time-domain electromagnetic simulations. Python bindings for Meep make it easier to use for researchers and open promising opportunities for integration with other packages in the Python ecosystem. As this project shows, implementing Python-Meep offers benefits for specific disciplines and for the wider research community.

In photonics and microwave design, it's essential to be able to accurately simulate electromagnetic wave propagation through subwavelength-scale structures. To achieve this, researchers often use the finite-difference time-domain (FDTD) method.[1] Because it models Maxwell's equations in a fully vectorial way, FDTD is one of the most powerful and general techniques, but it's also rather brute force. It's computationally intensive, but well suited for massive parallelism, making it scalable on large clusters or supercomputers. There are several commercial and open source FDTD packages available, but many researchers choose the open source package Meep, which was developed at MIT[2] and has a broad user community.

Meep's standard version defines a simulation 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] Mostly popular for educational purposes, Scheme can present newcomers with challenges in getting started. Although not inherently more difficult, Scheme has a somewhat different syntax, coding convention, and execution strategy than more mainstream, or *imperative*, languages. Many researchers interested in Meep aren't familiar with this programming paradigm.

In contrast, Python follows a more traditional approach. Like Scheme, it's a dynamically typed language and is thus well suited for scripting and rapid prototyping. It has also become widely adopted over the past decade, both in the industry (as in the Google Apps Engine platform) and in many open source projects. Python is especially popular in scientific and academic communities, and, as we discuss later, many Python libraries—most of them open source—are available and cover a wide spectrum of functionalities.

Scripting Meep using Python would make Meep easier for researchers to use, as well as permit seamless integration with other existing Python software.

1521-9615/11/$26.00 © 2011 IEEE
Copublished by the IEEE CS and the AIP

Emmanuel Lambert and Martin Fiers
Ghent University, Belgium
Shavkat Nizamov
Samarkand State University, Uzbekistan
Martijn Tassaert
Ghent University, Belgium
Steven G. Johnson
Massachusetts Institute of Technology
Peter Bienstman and Wim Bogaerts
Ghent University, Belgium

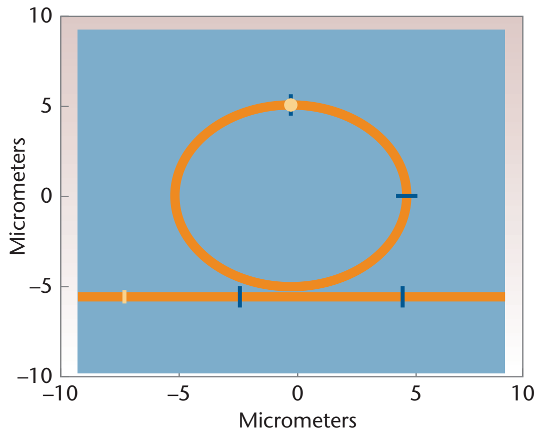


Figure 1. The automatic visualization of a 2D simulation landscape based on Python-Meep and Matplotlib. This visualization shows a ring resonator with access waveguide in silicon (orange), the position of the source (tan line), two fluxplanes (blue lines), and a probing point (tan circle).

Here, we describe how Python bindings for Meep leverage the tool in several ways, and how the research community benefits from this extension.

Leveraging Meep with Python

We've developed with Python for many uses over the years in our research on silicon photonics and plasmonics. At Ghent University (UGent)/IMEC, we've developed a litho mask design toolkit for silicon photonics in pure Python. We've also developed add-on tools and libraries for electromagnetic modeling, design optimization,[5] and process simulation.[6] Our long-term goal is to further automate closed-loop optimization of photonic circuits.[7] To this end, a powerful tool like Meep enriches our modeling framework. It also broadens our research capabilities in design optimization because it lets us leverage fully vectorial 3D FDTD simulations from inside a Python-driven design optimization process.

Benefits of Python Bindings

Python bindings offer several generic benefits to the wider community of Meep users. First, they enable the integration of Meep with existing Python open source libraries—such as the popular Numpy and SciPy (www.scipy.org)—for scientific computing. Numpy is an extension to the Python language that adds support for large, multidimensional matrix operations and related mathematical functions.[8] 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—that is, we want to use Meep to simulate the waveguide's electromagnetic behavior for various parameter values. It's now possible to use optimization algorithms, such as simulated annealing (provided by SciPy) or genetic algorithms (provided by PyGene), to explore this parameter space on a supercomputer and optimize against a particular target function. Numerical algorithms offered by Numpy can be used for processing simulation results. Combining these libraries with Meep is a promising option for the many researchers already familiar with them.

Visualizing Simulation Results with Python

In Meep's currently deployed versions, visualizing electromagnetic fields relies on external tools (with files for data interchange) and it's largely a manual process. With Python-aware Meep, we can develop visualization functionality using popular Python libraries such as Matplotlib for 2D (see http://matplotlib.sourceforge.net) and Mayavi2 for 3D (see http://code.enthought.com/projects/mayavi) and tightly integrate them with the simulation script. We can automatically generate the waveguide's visualization, 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 built this functionality on top of the standard Python-Meep, which we integrated with a more general simulation framework used by our research group (for this reason, it's currently a proprietary extension and isn't included in the public release of Python-Meep). Figure 1 shows 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 electromagnetic fields as the simulation progresses. Generally speaking, such automated and advanced visualization functionalities save time and can save reiterations of failed or ill-conditioned simulations.

Parallelizing Meep Simulations

Meep's standard version can be enabled for the message passing interface run (MPI-run), which means that the computation is distributed over multiple computing cores (on one or more nodes). MPI is an industry standard that defines message passing between software components executing in parallel.[9] Using MPI, we can easily parallelize an FDTD algorithm. We can split up the simulation problem in cells: in a given time step, the calculation for one cell is dependent only on the cell's previous states and the surrounding cells' boundaries. Each computing core processes one cell and exchanges boundary information with its neighbors.

The Python-Meep bindings are fully compatible with Meep's MPI-capabilities. However, such an MPI-distribution doesn't 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 often can't efficiently exploit the full capacity with one MPI-run alone.

Integration with the IPython Framework

At UGent, we're developing a generic photonic simulation framework based on IPython,[10] a Python environment enhanced for parallel computing. IPython largely abstracts the technical aspects of parallel computing from the user and allows robust error handling. It lets users submit scripts to a controller, which in turn scatters the code to engines on several nodes for execution. Results and exceptions are then gathered and presented to the client shell in a user-friendly manner.

The Python bindings for Meep let us integrate Meep with this IPython framework. Such integration shows a clear benefit, letting us combine MPI-runs of Python-Meep with IPython's scatter-gather capabilities. As Figure 2a shows, in this architecture, we basically have a 2D space over which we can spread many simulations (such as in a parametric scan). The first dimension is the number of computing cores to which we can scale one simulation in an MPI-run. The second 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. Finally, a user interface lets us launch simulations for a certain set of parameters and view a specific simulation's progress.

Suppose, for example, that we have a computer cluster with 1,600 cores and we want to scan a parameter space with 150 parameter combinations. 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, 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. Python-Meep's scaling behavior over the first dimension (the number of cores for MPI-run) is similar to standard Meep: the Python layer doesn't interfere with the MPI-specific commands in the Meep core.

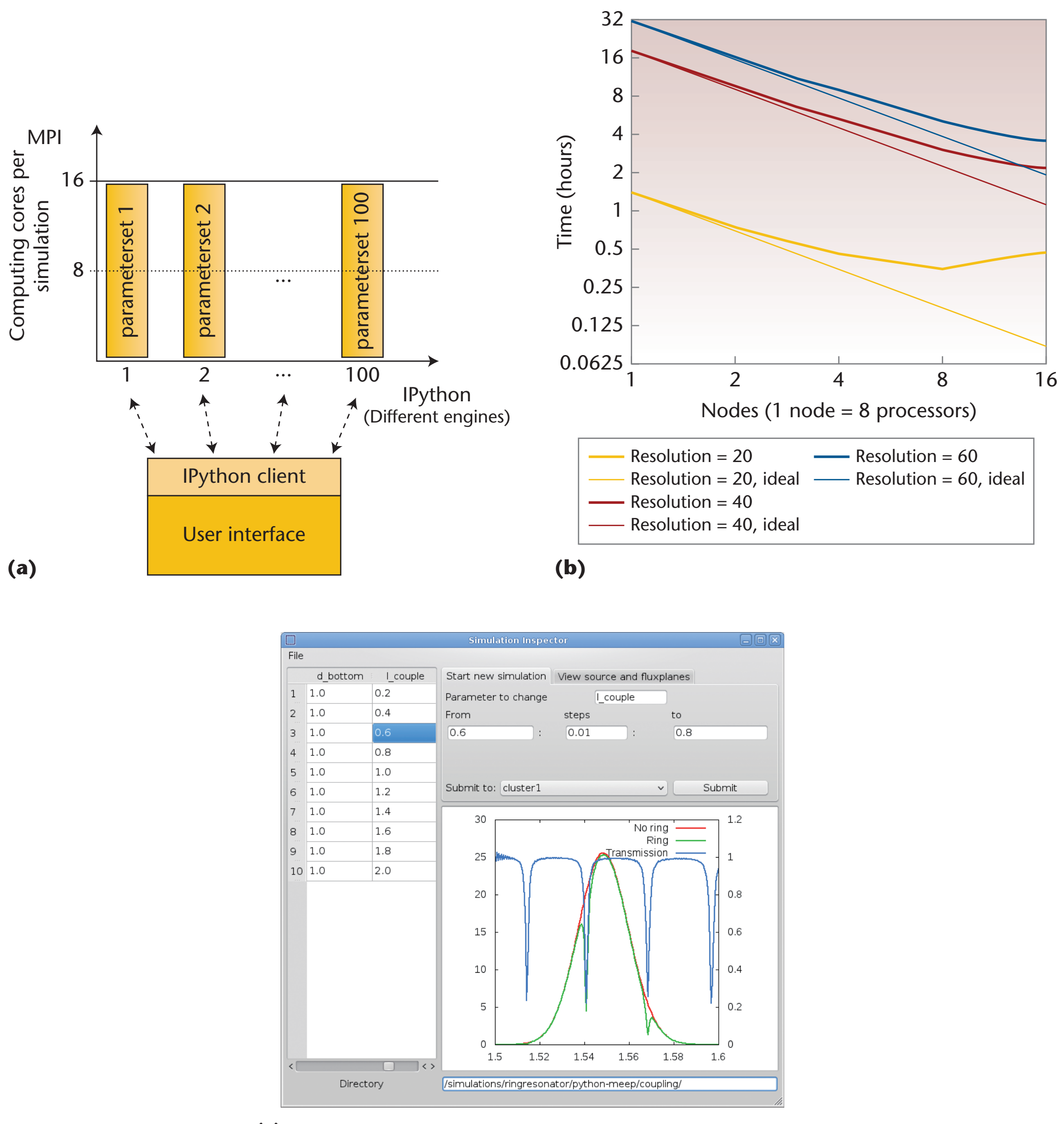
Figure 2b shows the scaling of a benchmark 3D simulation with MPI. The total calculation time is shown for different resolutions (sizes of computational volume). This is compared with the scaling we ideally expect—that is, when we double the number of nodes, we expect the calculation time to halve. For a given resolution, there's an upper limit to the number of cores over which we can scale efficiently. For a 3D 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 created: in such a case, the total running times increase. As Figure 2b shows, scaling performance is better for more complex, high-resolution problems.

IPython largely abstracts the technical aspects of parallel computing from the user and allows robust error handling. It lets users submit scripts to a controller, which in turn scatters the code to engines on several nodes for execution.

For the second dimension (the IPython engines), there's no inherent scaling limit as the different IPython engines are essentially separated programs running in parallel, with no intercommunication. Figure 2c shows a graphical user interface that we built with PyQt (www.riverbankcomputing.co.uk/software/pyqt/intro) on top of this IPython-based framework. Using it, we can conveniently launch new Python-Meep simulations and inspect results of terminated simulations.

A Taste of Python-Meep

Figure 3 shows a short example of a Python-Meep script, which offers a glimpse of the coding conventions. In this example, we calculate the 2D electromagnetic field profile in response to a line source located at the left of a straight waveguide.



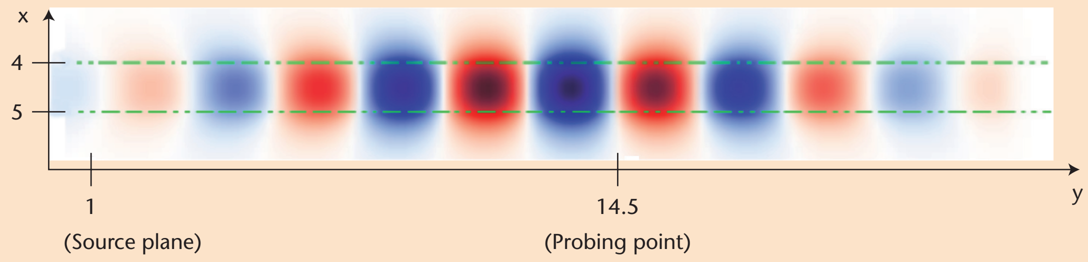
	d_bottom	l_couple
1	1.0	0.2
2	1.0	0.4
3	1.0	0.6
4	1.0	0.8
5	1.0	1.0
6	1.0	1.2
7	1.0	1.4
8	1.0	1.6
9	1.0	1.8
10	1.0	2.0

Parameter to change: l_couple
From: 0.6 steps: 0.01 to: 0.8
Submit to: cluster1
Directory: /simulations/ringresonator/python-meep/coupling/

Figure 2. Integrating Meep with the IPython framework. (a) A schematic representation of 100 simulations—each with different parameters—on a supercomputer. Each simulation executes in an IPython engine and is scaled with MPI over 16 computing cores. (b) Scaling a 3D Python-Meep simulation with MPI. The actual calculation times are shown for different resolutions and compared with the calculation times that we ideally expect. (c) The graphical user interface of UGent's photonic simulation framework, along with the parameters used in a range of Python-Meep simulations and the results for each simulation (that is, the transmission calculated from the fluxes). The GUI lets users 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.

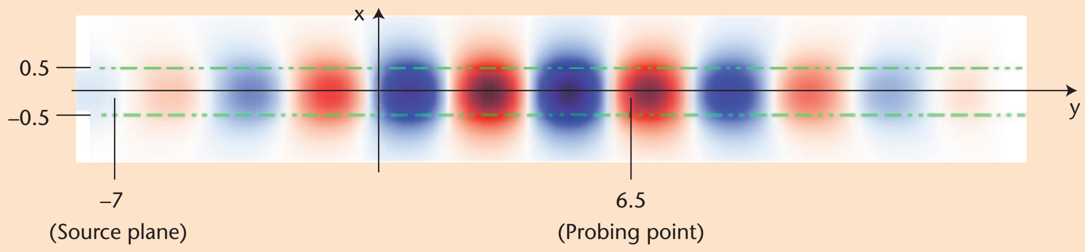
The field's Ez component is periodically written to a HDF5 file, which the user can then further process (HDF5 is a standard file format for scientific datasets; see www.hdfgroup.org).

Figure 4 shows an equivalent script implemented with Scheme. As these code samples show, the Scheme version defines the problem in terms of higher-level expressions. Functional languages such as Scheme are inherently highly expressive,[11,12] and the authors of Meep fully exploited this feature when they created the Scheme interface. They thus overcame the fairly low-level style of the Meep C++ core. Additionally, the Scheme interface was complemented with user-friendly functionality that isn't available in the underlying Meep C++ core (and thus, by default, isn't available yet in Python-Meep).



```
from meep_mpi import *

#define the waveguide material as a function of a vector(X,Y) :
#we create a straight waveguide of widdth 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
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(14.5,4.5)

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

Figure 3. Example of a basic Python-Meep simulation script, which uses its own coordination system.



```
;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 6.5 0 0)
             1e-3)
  (at-beginning output-epsilon)
  (at-every 0.6 output-efield-z))
```

Figure 4. Example of a basic Scheme simulations script. As the code sample shows, Scheme uses a different coordinate system than Python-Meep.

The Python-bindings directly expose the low-level Meep C++ core, which is reflected in the Python script's coding style. In Python-Meep, we're now adding similar high-level helper functions to facilitate simulation script writing, and we'll increase this effort in future versions. Although such functions are useful, they're not necessary to take advantage of Meep's functionalities. Scheme interface users are limited to the functionality it offers, while users of Python-Meep have more flexibility: they can use both the Meep C++ core's low-level functionality and the Python interface's higher-level helper functions.

Implementing the Python Bindings

In addition to outlining actual technical implementation of the Python bindings, we now explain why we choose the Simplified Wrapper and Interface Generator (SWIG) as the basic integration technology and weigh alternative implementations against each other.

Integrating the Meep Callback Mechanism

The Meep core library (written in C++) provides a callback mechanism that integrates with the simulation script: whenever the runtime engine needs information about a simulation's specific properties, it calls a user-defined function.

Choosing SWIG

As alternative approaches for implementing our Python wrapper, we initially compared both SWIG[1] and Boost.Python (www.boost.org/doc/libs/1_43_0/libs/python/doc/index.html).

Boost is a well-established and recognized set of open source C++ libraries that runs on almost any operating system. Its Boost.Python subset supports seamless interoperability between Python and C++. We had very good experiences with "Boost.Python; it offers a tutorial, the semantics of the API are clear, and it required only limited code writing. However, there was one important drawback: during the technical build process, we had to link our code to Boost-specific dynamic libraries. Although such libraries can be compiled from source, they have a large footprint. This is a major dependency that poses an additional threshold for deployment on third-party systems such as supercomputers. We prefer to keep Python-Meep lightweight, with as few dependencies as possible. Therefore, we decided to use SWIG.

SWIG is a dedicated framework for connecting C/C++ programs with many different programming languages. We must write an interface file, from which SWIG's engine generates two additional files: one with C code and the other 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 users don't need to install SWIG on their host systems.

SWIG's documentation is quite detailed, but the semantics of various constructs aren't always easy to understand. The technical implementation was rather complicated and required much trial and error before we obtained the required behavior. The typemap definition was especially error prone and hard to debug. These were serious drawbacks. However, once up and running, the Python/C++ interface works without a flaw.

Reference

1. D.M. Beazley, "Using SWIG to Control, Prototype, and Debug C Programs with Python," *Proc. 4th Int'l Python Conf.*, IOS Press, 1996; www.swig.org/papers/Py96/python96.html.

This mechanism is used intensively, such as in defining the simulation volume's material properties or defining a custom electromagnetic source.

We developed the Python-Meep bindings using SWIG, an open source tool that connects programs written in C/C++ with a variety of high-level programming languages.[13] As the sidebar, "Choosing SWIG" describes, SWIG's flexibility allows for an elegant integration with this callback mechanism. As Figure 5 shows, based on our experiences with performance and ease of use for the end user, the actual implementation technique evolved in three phases.

In a first straightforard 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 defining the material properties (see Figure 3). However, for many complex simulations—such as those with high resolution—the performance of this pure Python callback was insufficient because the callback function for defining 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, we addressed this drawback by letting users 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 user's overhead for mixing Python with C/C++. Nevertheless, combining two languages remains a drawback for some end users, particularly those who aren't familiar with C/C++.

In the original Scheme interface, the performance issue with this repeated callback occurs less often because Meep's authors largely bypass the standard callback mechanism. This results in a tighter integration of the C++ core and the Scheme definitions. We subsequently worked toward 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.

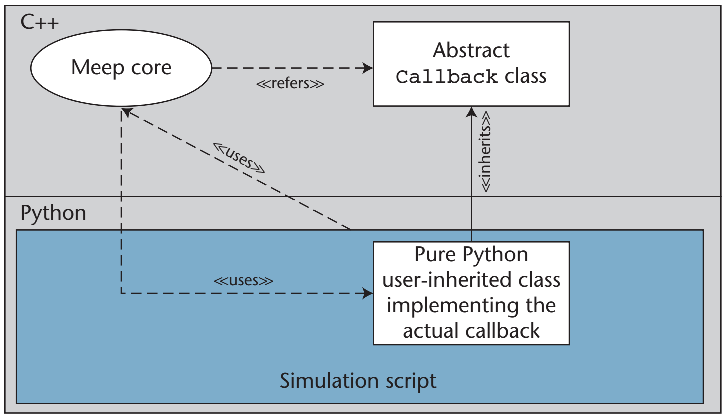
Numpy is known for its great performance because it stores and processes its data in C and exposes only a thin interface to Python. Therefore, if we define a Numpy matrix in Python with our simulation volume's material properties, the matrix is directly accessible from Meep using C coding conventions (basically, a pointer). The integration then comes down to writing a wrapper around the Meep callback

Peter Bienstman *is an associate professor at the Photonics Research Group of Ghent University/IMEC. His research interests include applications of nanophotonics in biosensors and photonic information processing, as well as nanophotonics modeling. Bienstman has a PhD from Ghent University. Contact him at Peter.Bienstman@intec.ugent.be.*

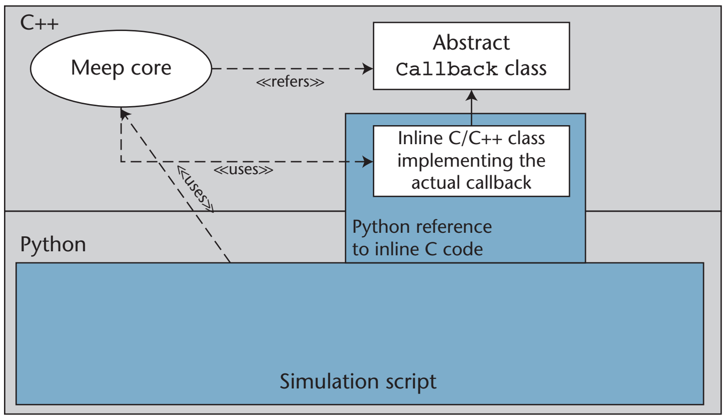
Wim Bogaerts *is a professor at the Photonics Research Group of Ghent University/IMEC, where he coordinates silicon photonics activities in process development, all-silicon integration, and photonic design tools. Bogaerts has a PhD in applied physics engineering from Ghent University. Contact him at Wim.Bogaerts@intec.ugent.be.*

References

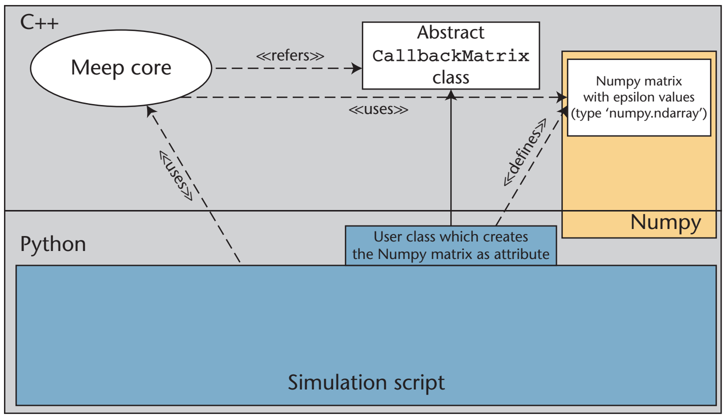
1. A. Taflove and S.C. Hagness, *Computational Electrodynamics: The Finite-Difference Time-Domain Method*, 3rd ed., Artech House Publishers, 2005; www.artech-house.com/Detail.aspx?strBookId=1123.
2. A.F. Oskooi et al., "MEEP: A Flexible Free-Software Package for Electromagnetic Simulations by the FDTD Method," *Computer Physics Comm.*, vol. 181, no. 3, 2010, pp. 687–702.
3. G.J. Sussman and G.L. Steele, Jr., "Scheme: An Interpreter for Extended Lambda Calculus," AI *Memos*, no. 349, MIT AI Lab, Dec. 1975.
4. *IEEE Std. 1178-1990, Scheme Programming Language*, IEEE CS, 1991.
5. D. Vermeulen et al., "Silicon-on-Insulator Nanophotonic Waveguide Circuit for Fiber-to-the Home Transceivers," *Proc. 34th European Conf. Optical Comm.*, 2008; doi: 10.1109/ECOC.2008.4729214.
6. P. Bienstman et al., "Python in Nanophotonics Research," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 46–47.
7. W. Bogaerts et al., "Closed-Loop Modeling of Silicon Nanophotonics from Design to Fabrication and Back Again," *Optical and Quantum Electronics*, vol. 40, no. 11, 2009, pp. 801–811.
8. T.E. Oliphant, "Python for Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 10, 2007, pp. 10–20.
9. W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.
10. F. Perez and B.E. Granger, "IPython: A System for Interactive Scientific Computing," *Computing in Science & Eng.*, vol. 9, no. 3, 2007, pp. 21-29.
11. J. Hughes, *Why Functional Programming Matters*, Addison Wesley, 1990.
12. M.P. Atkinson, P. Buneman, and R. Morrison, *Data Types and Persistence*, Springer Verlag, 1988.
13. D.M. Beazley, "Using SWIG to Control, Prototype, and Debug C Programs with Python," *Proc. 4th Int'l Python Conf.*, IOS Press, 1996; www.swig.org/papers/Py96/python96.html.
14. B. Spotz, "numpy.i: a SWIG Interface File for NumPy," SciPy, Dec. 2007; http://docs.scipy.org/doc/numpy/reference/swig.interface-file.html.



(a)


C++
Meep core
≪refers≫
Abstract Callback class
Inline C/C++ class implementing the actual callback
≪uses≫
≪uses≫
Python reference to inline C code
Python
Simulation script

(b)


C++
Meep core
≪refers≫
Abstract CallbackMatrix class
Numpy matrix with epsilon values (type 'numpy.ndarray')
≪uses≫
≪uses≫
≪defines≫
Numpy
Python
User class which creates the Numpy matrix as attribute
Simulation script

(c)

Figure 5. Alternative architectures implemented for definition of the material properties in the simulation volume. (a) The first architecture uses a pure Python class for callback. In this case, the C++/Python boundary is crossed whenever callback occurs (potentially millions of times for material definition). (b) The second architecture uses inline C/C++ for large simulation volumes with many grid points. The callback occurs completely in the C/C++ domain, offering great performance. (c) With the third architecture, users work in Python alone, creating a Numpy matrix with the material definition. Meep can directly access this matrix using a pointer. This also offers great performance, but with increased memory consumption.

functionality. This wrapper retrieves the actual values from the Numpy matrix and returns them to Meep.

Figure 5 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, users create a Numpy matrix, with its size corresponding to the discretized simulation volume (or a multiple for better accuracy). This architecture offers great performance and lets users work in pure Python. However, it increases memory consumption because we have to store the Numpy matrix before it's interfaced to Meep. Figure 6 illustrates the technique for the straight waveguide example in Figure 3.

Let's take a more detailed look at the technical implementation. As the last line of code in Figure 6 shows, 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`.

Our goal is to seamlessly pass the Numpy matrix as a parameter to the functions `set_matrix_2D` and `set_matrix_3D`. We therefore have 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*. Typically, defining typemaps is a complicated and tedious task. Luckily, a range of Numpy typemaps are already available in the open source community (numpy.i[14]). These typemaps are called `IN_ARRAY2` and `IN_ARRAY3` for 2D and 3D Numpy arrays, respectively.

In our SWIG definition file, we must link the signature of the `set_matrix_2D` function with the typemap. We do this using the code below. When we pass a Numpy array to the function in Python, it's automatically expanded in the C++ function's three or four corresponding parameters.

```
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)
```

Figure 6. Combining SWIG with Numpy matrices to describe the straight waveguide in Figure 3. The user inherits from `CallbackMatrix2D` and assigns the Numpy matrix to an attribute.

```
//Include the Numpy 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, we needed typemaps for interfacing parameters that represent complex numbers. Both Python and C++ have separate definitions of a `complex` type and thus we need a mapping or translation for seamless integration. The definition of these typemaps is quite complicated; for details, consult the file `py_complex.i` in the public Python-Meep distribution.

All three of these techniques for defining material geometries are available to Python-Meep users. The Numpy matrix approach is preferred for moderately sized simulations with relatively simple geometry. For very large simulation volumes, using a C/C++ callback function might be more appropriate, as it has lower memory requirements. It's also important to consider the simulation of bended waveguides: the approach with the Numpy matrix discretizes the geometry and thus creates a staircase approximation of the waveguide edges. In some cases, this might impact the simulation's accuracy. In such case, using a C/C++ callback function is more appropriate, as the simulator will then always dispose of a perfect representation of the geometry.

A fourth, more advanced technique was recently added to Python-Meep that allows the definition of the material geometry based on polygons. In this approach, the Python script defines a set of polygons, whereby each polygon outlines an area with unique material properties. The polygon coordinates are interfaced by the callback class with the Meep core engine without consuming large amounts of memory or processing time. Meep then disposes of an analytically correct representation of the materials and can resolve a full material geometry without recurring callback to Python. This results in excellent performance and great accuracy.

Interfacing External Data with a Python-Meep Script

Posters on FDTD mailing lists frequently express concerns about specifying external sources—that is, electromagnetic sources that are defined by some other software and exported as data files. Python has extensive features for interchanging

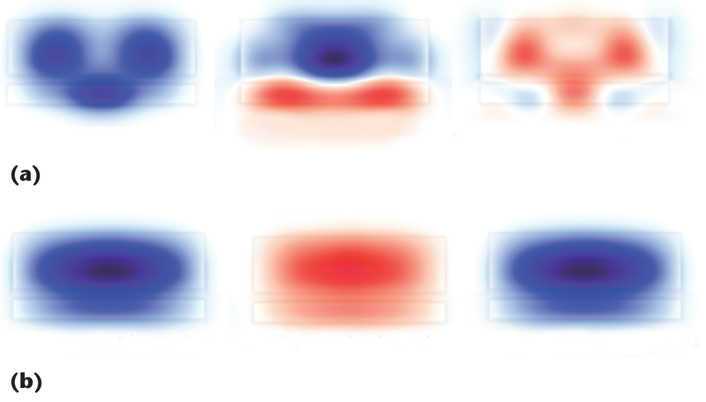


Figure 7. The shaping of an electromagnetic source in Python-Meep. (a) The field profile without spatial shaping of the source compared to (b) a field profile when the source is shaped according to an amplitude matrix calculated by Fimmwave and imported by Python-Meep. A field profile that is useful for a realistic design should have a constant spatial distribution of the power intensity over time for a given cross-section. In (a), there are major changes over time in the power intensity's spatial distribution for the chosen cross-section. In contrast, (b) shows a constant spatial distribution of the power intensity across the waveguide.

data that 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, or *modes*).

Realistic simulations often let just one specific mode be 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. Python-Meep conveniently addresses this problem. The commercial package Fimmwave (www.photond.com/products/fimmwave.htm) is well known for calculating such modes. We can use Fimmwave to calculate a target model's spatial amplitude and export the resulting matrix to a text file. In Python-Meep, we create a callback function that uses this matrix to calculate the source's exact amplitude profile. We then run the Python-Meep simulation with a custom source that matches accurately with the waveguide's physical properties. At UGent, we implemented such an integration scheme between Fimmwave and Python-Meep in several simulations (see Figure 7). During these efforts, the availability of Python's Numpy library proved useful because the resolution of the matrix that Fimmwave exports might not be the same as the resolution we want to use in the Meep FDTD simulations. Using Numpy, we can conveniently interpolate values to get the field profile value at each target position in the FDTD grid.

We distribute the Python-Meep bindings under the terms of the GNU General Public License, version 2. The source code is publicly available on Launchpad (https://launchpad.net/python-meep), and we welcome further contributions to the project's development.

Acknowledgments

The European Union, under its FP7-integrated project Helios, partially funded this work. Also, Martin Fiers and Martijn Tassaert received funding from the UGent Special Research Fund, Shavkat Nizamov's work is supported by a Russian Foundation for Basic Research grant 09-02-90205, and Wim Bogaerts received a postdoctoral grant from the Flemish Fund for Scientific Research (FWO).

***Emmanuel Lambert** is a research and development engineer with the Photonics Research Group of Ghent University-IMEC, where he's working on an integrated software framework for designing photonic components and circuits. His research interests include modeling nanophotonic circuits, large-scale computing, and integrating different software tools. Lambert has an MS in engineering from the University of Leuven. Contact him at Emmanuel.lambert@intec.ugent.be.*

***Martin Fiers** is a PhD student at Ghent University, where his research topic is photonic reservoir computing. His research interests include modeling of photonic components and designing software for phenomenological modeling. Fiers has an MS in electronic engineering from Ghent University. Contact him at Martin.Fiers@intec.ugent.be.*

***Shavkat Nizamov** is a postdoctoral researcher at Lausitz University of Applied Sciences in Germany. His research interests include investigating and improving a new biosensing methods based on surface Plasmon resonance. Nizamov has a PhD from the Heat Physics Institute in Tashkent (Uzbekistan). Contact him at nizamov.shawkat@gmail.com.*

***Martijn Tassaert** is a PhD student at Ghent University. His research interests include heterogeneous integration of SOI waveguides and III-V active devices. Tassaert has an MS in engineering from Ghent University. Contact him at Martijn.Tassaert@intec.ugent.be.*

***Steven G. Johnson** is one of the original authors of the Meep software and an associate professor at the Massachusetts Institute of Technology. His research interests include photonic crystals and electromagnetism in structured media and high-performance computation (from fast Fourier transforms to large-scale eigensolvers for numerical electromagnetism). Johnson has a PhD in physics from MIT. Contact him at stevenj@math.mit.edu.*