A Photonic Optimization Package

Scott Kenning

April 25, 2022

1 Introduction

1.1 Motivation

Photonic design often takes place by using analytic principles to guide simulations and subsequent manufacturing runs to test devices. This process works well, but there are several reoccurring themes during the design process.

1. Convergence testing must be often performed to ensure simulation results are accurate.
2. Optimization of structures (beyond the realm available with analytic math) is often carried out through brute force means (i.e., parameter sweeps).
3. It is cumbersome to draw structures in simulation software when models in well-developed layout tools already exist.
4. Simulation scripts are often "quick and dirty", meaning they lack readability and structure. Often times, they are copy-and-pastes of previous problems modified slightly.

We seek to address these issues through the design of a Python package that carries out tasks relevant to solving the above issues. The vision for the Python package was shaped through interaction with photonic designers and firsthand experience working on photonic design problems.

1.2 Goals

The goals of this project are separated off based on the classification of them being a "process" problem or a "design" problem. Process problems refer to 1, 3, and 4 in the above motivation list, while design problems relate to the generation of a flexible optimization framework related to item 2.

1.2.1 Process Problems

These goals relate to generating useful code that abstracts the idea of a "simulation". This relates to breaking the problem into a few key components that are often universal across photonic simulations.

- One-time initialization routines: These routines often initialize the underlying simulation engine. They often are called once in the beginning of a script running.
- Loading desired simulation parameters: Photonic simulations often involve designing devices with an assorted variety of parameters.
- Drawing: Structures in a photonic simulation must be drawn once per iteration. In a single script, the structures may be changed (e.g., during design parameter sweeping).
- Running: After structures are drawn, some sort of simulation must take place.
- Post-processing: This may involve exporting fields, generating images of results, performing calculations, etc.

1.2.2 Design Problems

Built on the back of the above process problems are design problems. Here, we are concerned with efficiently running simulations that yield optimized photonic devices. This is a difficult concept to address, and we will therefore outline a few goals.

- Allow users to run optimization algorithms on design parameters.
- Allow for specialized tools like adjoint methods to be used for gradient calculations in optimization problems.

2 Addressing Process Problems

Before "Design Problems" are addressed, it is important to solve the "Process Problems". By addressing them in this order, it is possible to fit an optimization framework cohesively inside.

There are two standards for scientific computing that we will allow usage of with our code: MPI and HDF5. Ultimately, usage of MPI is at the mercy of the backend simulation software (Lumerical and MEEP [1] both support it, for example). Since this project provides wrapper functionality around an arbitrary software suite, we do not want it to interfere with MPI and cause deadlocking. Roughly speaking, this translates into all processes call identical backend simulation calls. To do this, we will define methods in the following section that allow MPI-interfacing software to be called within.

HDF5 is a standard file type that allows storage of large amounts of data in typically an array format. It is widely used and accessible. It is attractive for several reasons.

- Metadata may be easily stored alongside simulation results. This is useful for record keeping and knowing exactly what simulation parameters were used for a given dataset.
- It is compressed and in a binary format (it is more compact then other data formats).
- It is easily compatible with Python, Matlab, MEEP, Lumerical, etc. Newer .mat files are actually based off of it.

This standardized file format will be widely used in the software.

2.1 The Simulation Class

To address process problems, we propose the `Simulation` class. Although the exposed user-interface is rather simple, it provides behind-the-scenes machinery to ensure ease of use and fault-tolerant behavior. The user interface is previewed below.

```python
class Simulation:
        def __init__(self, logname : str, working_dir : str,
        ↪   catch_errors=True):
            ...
```

```

    def draw(self, parameters : dict[str, typing.Any]) -> None:
        pass

    def run(self, parameters : dict[str, typing.Any]) ->
    ↪   Result:
        return Result(parameters)

    def process(self, result : Result, parameters : dict[str,
    ↪   typing.Any]) -> Result:
        return result

    def oneOff(self, iteration_parameters: dict[str,
    ↪   typing.Any], iteration=1, total_iterations=1) ->
    ↪   typing.Union[Result, None]:
            ...

    def basicSweep(self, parameters: dict[str,
    ↪   list[typing.Any]]):
            ...
```

All commands to an MPI-dependent backend should be placed within `draw`, `run`, and `process`.

2.1.1 Running Basic Parameter Sweeps

A task that often comes up is running parameter sweeps. This is done by using `Simulation.basicSweep`. The block diagram of this helper function is shown in Figure 1. The goal of this routine is to provide fault-tolerant running of many simulations. For example, a list of simulations (their parameters) can be loaded from a CSV file using the `CSVParameterLoader` class and passed directly to the `basicSweep` function. If at any point in the `draw`, `run`, or `process` routines an exception is raised, the machinery will salvage as much as possible and then continue running other simulations. Ample logging allows the resulting suppressed exceptions to be debugged by the user.

To actually write a simulation, the user inherits from the `Simulation` class and overrides atleast `draw` and `run`. Overriding `process` is optional. The `process` routine ultimately gives the user freedom to add additional

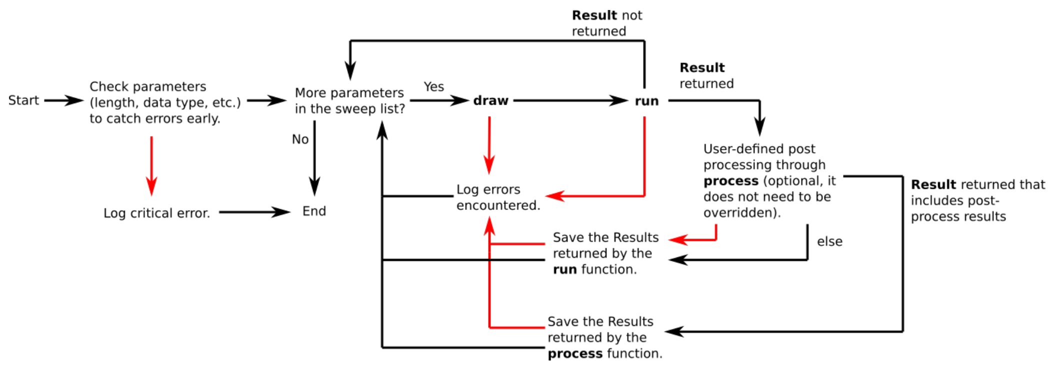

Result not returned
Start
Check parameters (length, data type, etc.) to catch errors early.
More parameters in the sweep list?
Yes
draw
run
Result returned
No
Log critical error.
End
Log errors encountered.
User-defined post processing through process (optional, it does not need to be overridden).
Result returned that includes post-process results
Save the Results returned by the run function.
else
Save the Results returned by the process function.

Figure 1: The block diagram of `Simulation.basicSweep`. Red lines indicate error handling routes and the black represent normal program flow in the absence of errors.

data to the results of the `run` routine.

2.1.2 Running a One-Off Simulation

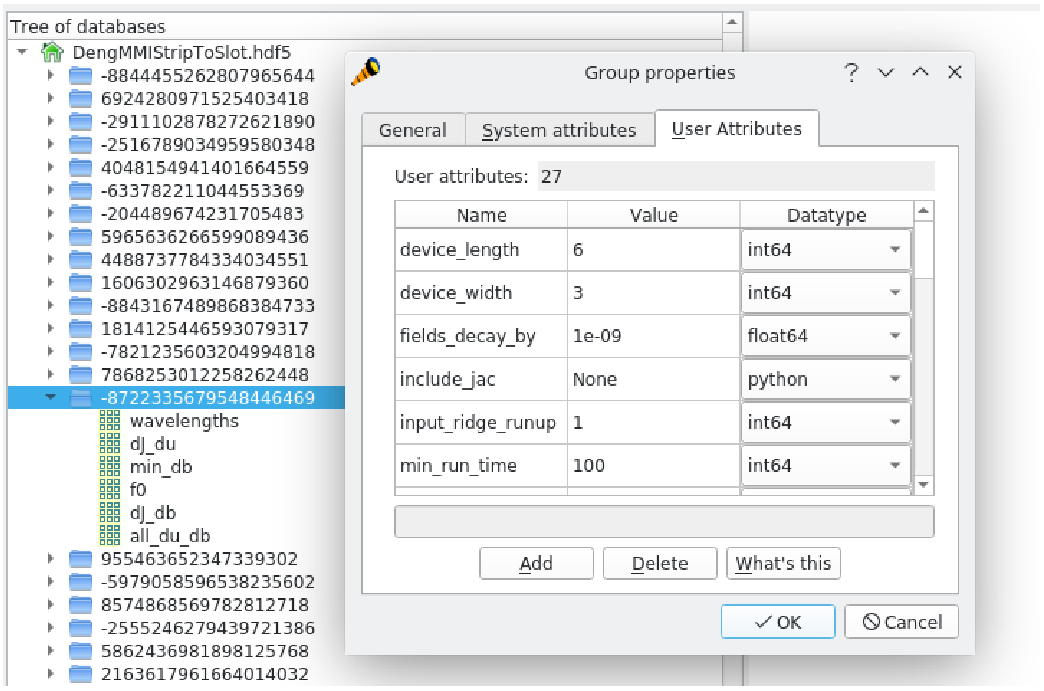
The `oneOff` function actually provides the same functionality as `basicSweep`, but for the case where there is only a single set of parameters. In fact `oneOff` is actually just called many times by `basicSweep` and implements the exception handling logic.

2.2 The Result Class

To continue addressing process problems, we propose the `Result` class. Arguably simpler than the `Simulation` class, it has one purpose: save desired results and their associated metadata.

```python
class Result:
        def __init__(self, parameters: dict[str, typing.Any],
         ↪  **values):
                ...
```

This is just achieved by returning a `Result` object from the `run` or `process` routines of the `Simulation` class. The constructor's signature is above, and is the only relevant function for the end user.


Tree of databases

- DengMMIStripToSlot.hdf5
  - -8844455262807965644
  - 6924280971525403418
  - -2911102878272621890
  - -2516789034959580348
  - 4048154941401664559
  - -633782211044553369
  - -2044896742317054830
  - 5965636266599089436
  - 4488737784334034551
  - 1606302963146879360
  - -8843167489868384733
  - 1814125446593079317
  - -7821235603204994818
  - 7868253012258262448
  - -8722335679548446469
    - wavelengths
    - dJ_du
    - min_db
    - f0
    - dJ_db
    - all_du_db
  - 955463652347339302
  - -5979058596538235602
  - 8574868569782812718
  - -2555246279439721386
  - 5862436981898125768
  - 2163617961664014032

Group properties

General | System attributes | User Attributes

User attributes: 27

Name	Value	Datatype
device_length	6	int64
device_width	3	int64
fields_decay_by	1e-09	float64
include_jac	None	python
input_ridge_runup	1	int64
min_run_time	100	int64

Figure 2: A screenshot of the resulting HDF5 file displayed in ViTables after many simulations were ran. The blue folder icons indicate HDF5 groups, which have the simulation parameters used saved in their metadata. Additional data (which is not metadata) was saved inside each group (e.g., `dJ_db`).

The `parameters` variable needs to be a Python `dict` that specifies metadata to be saved. For example, and as shown in Figure 2, design parameters for a particular device are stored as metadata. The actual folder name in the HDF5 file is generated randomly and uniquely, so the metadata is the differentiation feature of each folder of data in the simulation result files.

To save data that is not metadata (e.g., field quantities), they are simply passed in as `kwargs` to the constructor of `Result` as shown by example below.

```
        ...some simulation code...
        return Result(parameters, f0=f0, dJ_du=dJ_du,
         ↪  dJ_db=dJ_db, min_db=np.asarray(min_db),
         ↪  all_du_db=all_du_db, wavelengths=self.wavelengths)
```

For example, passing `f0=f0` means “save the variable in the above simulation code (right hand side of equal sign) as a variable named `f0` (left hand side of equal sign) in the HDF5 file”. Since there is often no need to change the name of the variable used in the script to something else in the HDF5 file, often times both sides of the equal sign will have the same variable name, leading to a notation that might look strange.

The `Result` class can handle data types that HDF5 supports. This includes most of the `numpy` array types.

2.3 The ConvergenceTest Class

The `ConvergenceTest` class performs exactly what it sounds like: convergence testing. Ultimately, the user guesses at what range of parameters needs to be swept. In addition, convergence criteria is supplied. It can optionally generate plots that would provide proof of convergence, and will return the suggested parameters required for appropriate convergence. See the in-code documentation for more information found in `pdt/core/Simulation.py`.

3 Addressing Design Problems

Addressing design problems, which are essentially in the form of optimization problems, is quite nuanced depending on what solver is used. In FDTD, it is possible to efficiently obtain gradient information of a design with two simulations. There are two strategies available that fall under “adjoint methods”, and usually one is implemented to a degree by the solver. In the case of MEEP, “density-based” methods are employed.

3.1 The DesignRegion and MaterialFunction Classes

Density-based methods are often used to design structures that are completely arbitrary. That is, as many degrees of freedom as possible are given to the optimization code. This often results in a grid with hundreds of “pixels” being changed around. This software has tools (such as `DesignRegion` and `MaterialFunction`) that are useful for designing “traditional” structures like tapers, mode converters, etc. It ultimately back-propagates the derivative given by the density methods (for each material pixel) to the design parameters. However, this is beyond the scope of this high-level document.

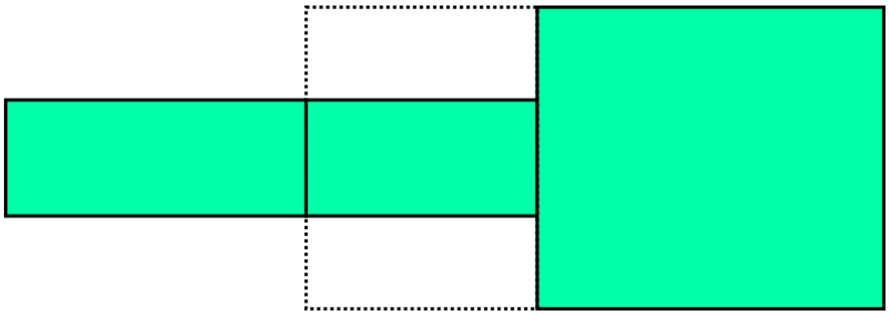


Figure 3: A simple taper example. We naively assume that we can simply connect two waveguides, and see if the computer can come up with something better in the dashed region.

These features are best explained by the examples in `pdt/designs/` or the Pydocs in the code.

3.1.1 Design Perturbation

Since any design that is optimized is likely projected onto a discrete grid, perturbing the design is not guaranteed to actually change the design once it is ultimately placed on the discrete simulation grid. Therefore, this software collects information about that grid and calculates the perturbation of each design parameter required to allow the simulator to see a unique design.

Consider the design of a simple taper, as shown in Figure 3. We assume no knowledge of taper design, and naively assume that the two waveguides can simply be connected together. We wish for the computer to optimize the waveguide width in the dashed box to maximize power transmission from the fundamental mode in one waveguide to the other. This is actually an example in `pdt/designs/`, so we will briefly examine the relevant sections of it.

To parameterize the design with respect to some degrees of freedom, we represent the width of the designed section along the horizontal direction with a weighted summation of Legendre polynomials up to a certain order. In effect, the computer can wiggle the boundaries around. If it modifies the weight of the linear Legendre polynomial, the yellow pixels in Figure 4 will then be shaded in, as shown.

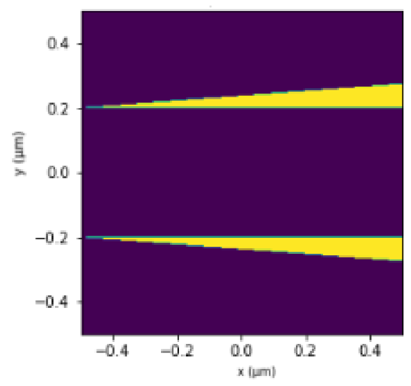


Figure 4: If the weight associated with the linear Legendre polynomial is marginally increased, the pixels of material in yellow will be changed. Note that this figure is what would occur if the design in Figure 3 is changed. In reality, the perturbation will be made as small as possible, so long as the underlying discrete grid of material is actually changed.

3.1.2 Sensitivity Information and Gradient Back-Propagation

Density-based adjoint solvers like MEEP return "sensitivity information" [2]. This information describes how, if pixel j of material changes, the figure of merit changes (e.g., the transmission of power through a structure). If we consider the design in Figure 3, a solver like MEEP would return data that is visualized by Figure 5. Combined with information from Figure 4, we can compute the gradient with respect to the design parameters (in this case, the coefficients to the polynomials generating the taper width).

Figure 4 gives us $\frac{\partial \epsilon_j}{\partial b_i}$, while Figure 5 gives us $\frac{\partial f}{\partial \epsilon_j}$. f represents our figure of merit, ϵ_j represents the dielectric constant at a pixel indexed by j, and b_i represents a design parameter indexed by i. Through the chain rule, we can compute $\frac{\partial f}{\partial b_i}$.

$$\frac{\partial f}{\partial b_i} = \sum_j \frac{\partial f}{\partial \epsilon_j} \frac{\partial \epsilon_j}{\partial b_i} \tag{1}$$

This computation of the gradient from sensitivity information and perturbation of design parameters is one of the key purposes of the classes described in this section.

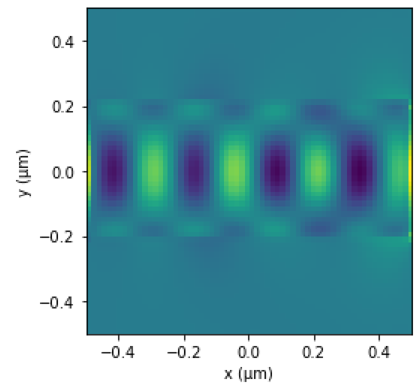


Figure 5: At yellow colors, an increase in the dielectric constant would increase transmission. At purple colors, a increase in the dielectric constant would decrease transmission.

3.2 The ScipyGradientOptimizer Class

This class wraps optimization function calls in a way that is compatible with the `Simulation` and `Result` class. Detailed documentation can be found in the Pydocs, but a brief outline of the important features that address design problems are presented here.

The key takeaway is that this class provides a single interface for performing optimization that seamlessly allows for computation of the gradient using first-difference or custom means.

3.2.1 First-Difference Gradient Calculation

This class is capable of using first-difference methods to compute the gradient, which will then be fed to a generic optimization algorithm. If there are N degrees of freedom in the design, $N + 1$ simulations are required. There are more efficient ways to obtain this gradient information, but the first-difference method is a nice way to debug.

Each design parameter is perturbed by an amount prescribed by the `DesignRegion` and `MaterialFunction`. Then the figure of merit is extracted from the `Result` class to perform the first-difference calculation.

3.2.2 Custom Gradient Calculation

The user may have their own solver-specific method of computing the gradient. In the examples provided in `pdt/designs/`, this is in the form of utilizing the adjoint solver in MEEP to provide sensitivity information. The `DesignRegion` and `MaterialFunction` classes can be used to compute the gradient as outlined in Section 3.1.2. This is then returned by the user in a `Result` object. `ScipyGradientOptimizer` will then utilize that information if the name of the field inside the `Result` containing the gradient is provided. Otherwise, `ScipyGradientOptimizer` will default to first-difference methods.

The importance of "custom gradient calculation" methods is that in electromanetics, adjoint methods can be used. Regardless of the number of design variables to optimize, only two simulations are required to obtain the information necessary.

3.3 Example Results: Waveguide Taper

A waveguide taper presents an interesting toy example: give the optimization script a poor initial state (depicted in Figure 3) and see if it comes up with a better design. The code for this example is found in `pdt/designs/` and is named `WaveguideTaperAdjvsFD.py`.

The device being designed transfers light from a 400 nm waveguide into a 1 µm waveguide. The allowed taper length is 1 µm, which makes it incredibly aggressive. The first ten Legendre polynomials are used in the construction of the taper width along the propagation direction.

The initial device has $|S_{11}|^2 \approx 0.82$, averaged across three wavelengths (1.50 µm, 1.55 µm, and 1.60 µm). The final optimized device has $|S_{11}|^2 \approx 0.98$. A cartoon of the final device is found in Figure 6. This software generates GIF renders of the optimization process for fun, and the one for this example can be found in `pdt/designs/renders/`.

3.4 Example Results: Strip-to-slot Mode Converter

We adapt a device from [3] to incorporate a stripload. The original device as outlined in [3] does not incorporate striploading, which is necessary if the resulting slot waveguide will be used in modulator designs. In effect, we add

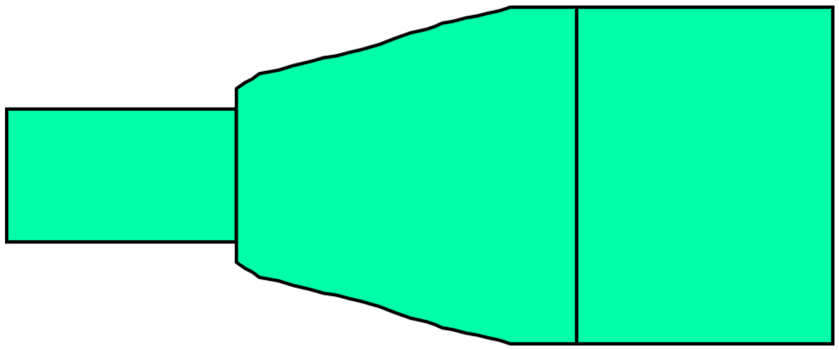


Figure 6: The final optimized state of the taper.

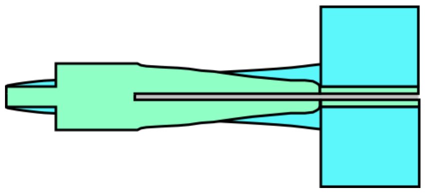


Figure 7: A modified design based off of the work of [3]. This is actually the optimized design.

the blue regions into the slot section of the device, depicted in Figures 8 and 7.

The optimizer was given the opportunity to wiggle around the green taper boundaries and the blue stripload boundaries. Figure 7 is a cartoon of what was designed by the program. It transmits roughly 97% of power in the fundamental mode of the strip waveguide to the slot waveguide. This is on-par with the claims made by [3]. Note that the simulation was 2D, and the index of the silicon ridge was calculated using rough averaging techniques.

4 Conclusion

This software provides an interface to structure simulation code around that addresses the "process problems". The `Simulation` class provides general logging facilities, error logging (and suppression, if desired), standardized functions to override for code clarity, and organization of results. Alongside the `Result` class, its functionality is extended to saving simulation metadata and data generated in a standardized manner.

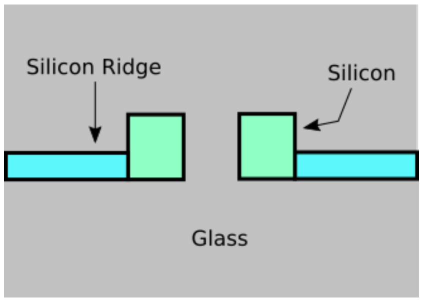


Figure 8: The cross section of a striploaded slot waveguide. This is a cross section of the far right side of Figure 7.

In addition, a simple interface is provided for optimization that wraps around the `Simulation` and `Result` classes. It offers generic first-difference gradient calculations and leaves room for using solver-specific adjoint information. Through this process, it abstracts away challenges including making sure a perturbational change in the design variables actually results in the discrete material grid used by the solver changing. It greatly facilitates optimization problems in FDTD simulations.

Examples have been provided in `pdt/designs/`. A taper example was discussed in this document, but a more sophisticated example such as the design of a strip-to-slot mode converter [3] is also available. These examples are 2D structures (they easily run on most computers), but there is nothing preventing the extension to 3D.

References

[1] A. Oskooi, D. Roundy, M. Ibanescu, P. Bermel, J. Joannopoulos, and S. Johnson, “Meep: A flexible free-software package for electromagnetic simulations by the fdtd method,” *Computer Physics Communications*, vol. 181, 2010.

[2] A. M. Hammond, A. Oskooi, M. Chen, Z. Lin, S. G. Johnson, and S. E. Ralph, “High-performance hybrid time/frequency-domain topology optimization for large-scale photonics inverse design,” *Optics Express*, vol. 30, no. 3, 2022.

[3] Q. Deng, L. Liu, X. Li, and Z. Zhou, “Strip-slot waveguide mode converter based on symmetric multimode interference,” *Opt. Lett.*, vol. 39, 2014.