A Photonic Optimization Package

Scott Kenning

February 19, 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 finite difference optimization algorithms on simulation parameters.
- Generate models for designing particular structures using adjoint methods present in a particular simulation suite.

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 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.

```
1 class Simulation:
2         def __init__(self, logname, working_dir="working_dir"):
3             ...
4
```

```
    def draw(self, parameters):
        pass

    def run(self, parameters):
        return Result(parameters)

    def process(self, result, parameters):
        return result

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

All commands to an MPI-dependent backend should be placed within `draw` and `run`. The `process` function is assumed to contain no code dependent on the simulation backend, and it will only be called on the main MPI process. Hence, all post-processing should be doable on one process (e.g., making plots).

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 2. 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 data to the results of the `run` routine.

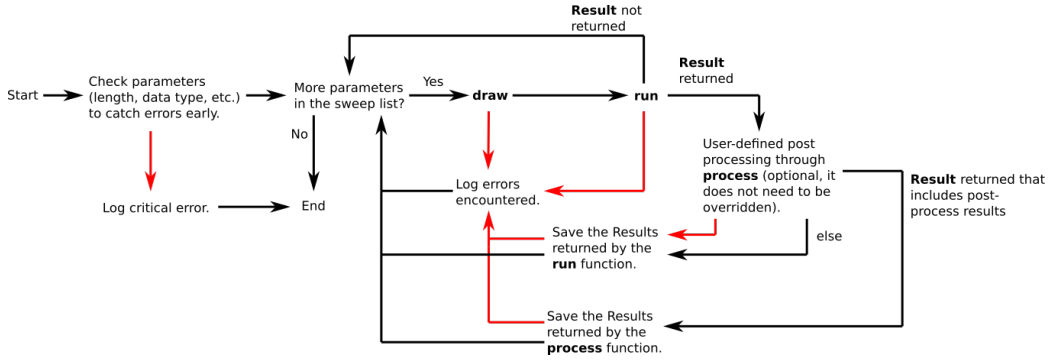

Figure 1: The block diagram of `Simulation.basicSweep`. Red lines indicate error handling routes and the black represent normal program flow in the absence of errors.