UNIVERSITÉ DE LORRAINE

Report - Numerical Optics

Beampy - scientific Python software development, Guided User Interface and Finite Difference Beam Propagation Method feature

Marcel Reis Soubkovsky

Course lectured by:
Nicolas Fressengeas

In the scope of the Master's in Applied Physics and Physics Engineering
2nd year

Contents

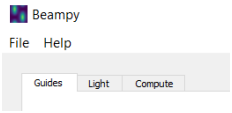

Figure 1: Beampy main tabs

1 Beampy

1.1 Description

Beampy is an open source software that allows the user to compute the propagation of light in a specific set of varying refractive index. The software is composed of three main tabs: the guides creation part, the light source creation and the propagation computational part (Figure 1). In the `Guides` tab it is possible to define the window settings such as number of calculation points in the x axis, the length of propagation (which increases the number of iterations). In `Guides` one can also create the waveguides. The space and waveguides are created simultaneously. The personalization options go from the guide's shape to their offset and their variation in refractive index. The second tab is `Light`, which enable the user to personalize the source of light. And at last the `Compute` tab computes the actual propagation with some extra options to perform kerr calculations.

The core of the software is based on Fast Fourier Transform Beam Propagation Method (FFTBPM), which is a tool for simulating light propagation. This method uses Fourier Transforms to generate the next slices of propagation. The software was developed by the students Jonathan Peltier and Marcel Soubkovsky. Jonathan was responsible for all the Physics behind and leading the development of the main functionalities of Beampy's version 0.0.1.4. Up to version 0.0.1.4 Marcel Soubkovsky participated in setting up the GUI and the interactive functionalities.

1.2 Guided User Interface (GUI) creation

There are several tools for Guided User Interface (GUI) creation. On Python, between some of the most popular libraries there is PyQt5, Tkinter and PySide2. The implementation is not always the easiest. Tkinter is a built-in library in Python that provides user interfaces directly from a python script. After several tests, I noticed Tkinter failed for our application in the following points: easiness of implementation of new features on already existing code, hard to manage since the code tends to get messy and also not very customizable.

Second option I turned to was Qt. Qt is a widely used GUI toolkit that is open-source and crossplatform. It has libraries on several programming languages and in Python the most popular two libraries are: PyQt5 and PySide2. The most important points for our application were: possibility of developing the GUI graphically, easiness of implementation, good scalability. The chosen library was PyQt5.

PyQt5 when installed on Python brings with it a graphical interface for designing GUIs called `designer` (Figure 2). The interfaces produced by this Graphical tool are save in an XML format under the extension `.ui`. PyQt5 has a conversion script with which it converts full `.ui` XML files into Python

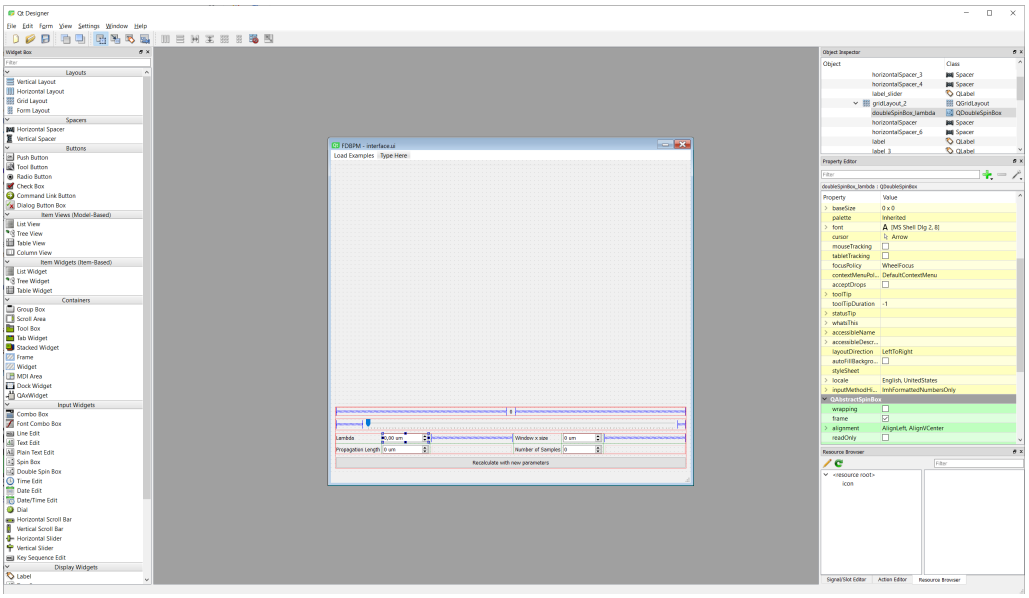

Figure 2: Designer Interface - Graphical tool for GUI creation that comes with PyQt5

GUI code. An example can be seen on Listing 1. The Python script that is then generated has to be imported into the main code. All the elements added in `designer` are present in the scope of the class present in the Python script generated during the conversion. The elements interactivity can then be programmed using PyQt5 methods and calling the elements (input boxes, sliders, radial inputs).

```
>> pyuic5 ./interface_designer.ui -o interface_python.py
```

Listing 1: Conversion from .ui file to .py

2 Fast Fourier Transform

A Fourier Transform allows us to manipulate a signal in the frequency space whether than in the original time space.

The Discrete Fourier Transform (DFT) is a method used to calculate the Fourier Transform of a discrete set of data. The core of the DFT resides in the following equation:

$$X(m) = \sum_{n=0}^{N-1} x(n) \left[\cos\left(2\pi nm/N\right) - i\sin\left(2\pi nm/N\right)\right] \tag{1}$$

where $X(m)$ is the output in the frequency domain. m is the index in the frequency domain and n in the time domain. The DFT is a powerful tool that comes out of Fourier Analysis that is today indispensable in digital signal processing. Nonetheless the parameter N which represents how many points are needed for the DFT at a specific point m is an important parameter. It defines the calculation time taken to process N points in a DFT. An algorithm that efficiently calculates the DFT of a signal is the Fast Fourier Transform (FFT), which was published by Cooley and Tukey in 1965[2]. In the last

decades the FFT proved to be the best method for calculating Fourier Transforms of discrete signals. In terms of speed of execution, a simple test can prove the efficiency of the algorithms in a short set of data.

The block of code in Listing 2 represents the calculation of a DFT in Python using the `scipy.linalg` library.

```
1 import numpy as np
2 from scipy.linalg import dft
3 x = np.array([1, 2, 3, 0, 3])
4 np.set_printoptions(precision=2, suppress=True)  # for compact output
5 m = dft(5)
6 result=m @ x  # Compute the DFT of x
```

Listing 2: DFT calculation in Python. (Source: Scipy Documentation)

The core of the DFT calculation (Listing 2) is in lines 5 and 6. On line 5 it creates the DFT matrix, which is a Transformation Matrix that when applied to the signal on line 6, gives its DFT calculation. On Python, some modules have FFT calculations within themselves, but the most popular one and also built-in Python is `numpy.fft`. The code needed to perform the FFT is seen on Listing 3.

```
1 from numpy.fft import fft
2 x = np.array([1, 2, 3, 0, 3])
3 result=fft(x)
```

Listing 3: FFT calculation in Python. (Source: Scipy Documentation)

Putting the core of the Python calculations in a loop and executing it one million times while timing the calculation gives us the comparison. `numpy.fft`'s FFT is over 5 times faster than `scipy.linalg`'s DFT calculation.

The FFT is undeniably of very simple application and a very powerful tool. Beam Propagation Method algorithms based on FFT have been widely used to study and analyze waveguides, coupled waveguides and the most varied photonic setups.

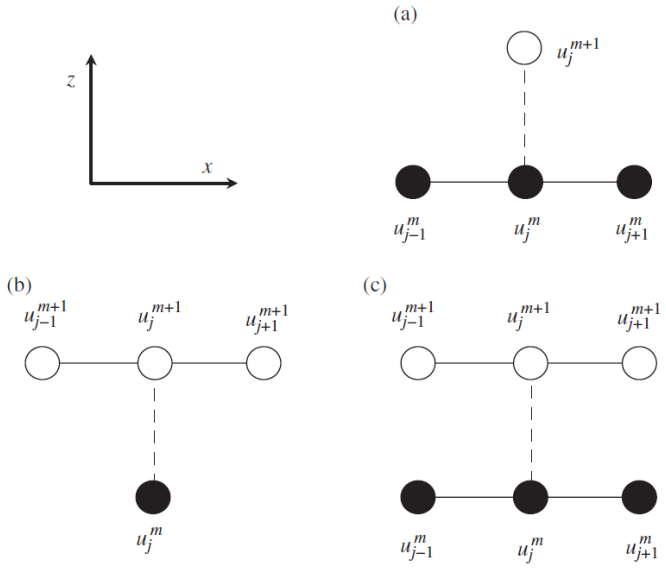

Figure 3: Three methods of computing propagation based on finite difference: (a) fully explicit, (b) fully implicit and (c) Crank–Nicolson. (Source: Pedrola)

3 Finite Difference Beam Propagation Method

3.1 Theory[3]

In the late 80s a technique based on a finite difference calculation of BPM emerged[1]. This technique proved to be up to 20 times faster than FFT-BPM[1]. This technique, named as Finite-Difference Beam Propagation Method (FD-BPM), can be applied in the most varied cases, such as: non-linear phenomena, mode analysis, optic fiber design, waveguide design. In order to simplify, let's consider a propagation only through one dimension on a 2D structure. The two dimensional wave equation can be expressed as follows:

$$2in_0k_0\frac{\partial u}{\partial z} = \frac{\partial^2 u}{\partial x^2} + k_0^2(n^2 - n_0^2)u \tag{2}$$

where z is the axis of propagation. When applied finite difference on $\frac{\partial u}{\partial z}$ becomes $\frac{u_j^{m+1} - u_j^m}{\Delta z}$, and on the second order partial derivative $\frac{\partial^2 u}{\partial x^2}$ becomes $\frac{u_{j-1}^m - 2u_j^m + u_{j+1}^m}{(\Delta x)^2}$. But three different methods can be used for calculating the finite difference equivalent. These are the fully explicit method, the fully implicit method and the Crank-Nicolson method (see Figure 3).

For this demonstration we will use the Crank-Nicolson method, which is the one used in this software application due to its greater stability compared to the other methods. After algebraically arrenging the terms, we end up with an expression structured as the following:

$$a_j u_{j-1}^{m+1} + b_j u_j^{m+1} + c_j u_{j+1}^{m+1} = r_j \tag{3}$$

where:

$$a_j = -\frac{\alpha}{(\Delta x)^2}$$

$$b_j = \frac{2\alpha}{(\Delta x)^2} - \alpha\left[(n_j^{m+1})^2 - n_o^2\right]k_o^2 + \frac{2ik_0n_0}{\Delta z}$$

$$c_j = a_j$$

$$r_j = \frac{(1-\alpha)}{(\Delta x)^2}\left[u_{j-1}^m + u_j + 1^m\right] + \left[(1-\alpha)[(n_j^m)^2 - n_0^2]k_0^2 - 2\frac{(1-\alpha)}{(\Delta x)^2} + \frac{2ik_0n_0}{\Delta z}\right]$$

This system can be solved by using a tridiagonal calculation as in Thomas method [3].

3.2 FD-BPM on Beampy

The initial code was based on a simple free space FD-BPM Matlab script written by PhD Edgar Guevara. The first task was translating the core of the code to Python. The task seemed simple, but surprisingly many functions in Matlab do not give the same result in Python. The main problem was the backslash operator ¨

ïn Matlab, which solves systems of linear equations directly. Python's numpy equivalents did not give the same results, thus not reproducing the free space propagation. The solution was accepting the difference in results and work from there on using `numpy.linalg.solve()`. The first result was observed reproducing the free space propagation correctly with a margin of up to 10-3 difference from the Matlab script. Different scripts could solve the system with the tridiagonal matrix, the two tested ones were `numpy.linalg.lstsq()`,`numpy.linalg.solve()`. In order to test the efficiency of these scripts, they were put in a loop to run more than 1 million times each and then the time was averaged and compared. The results show that `numpy.linalg.solve()` is more than 4 times faster.

np.linalg.lstsq(matrix,d)[0]	5.071 seconds
np.linalg.solve(matrix,d)	1.296 seconds

While testing the speed of the different functions, other parameters were tweaked for testing as well. And a big surprise to me was that the number of samples could be drastically reduced without affecting the result as much. On FFT-BPM the number of samples is a very important parameter for the calculation. If varied, the results can completely change. Which is unfortunate and makes it harder to reduce the computation time. On FD-BPM this variable is not as crucial as in FFT-BPM. By reducing the number of samples a time record of 0.0280 seconds was achieved. This opens several possibilities including real time rendering.

The original script was a first simple implementation with free-space propagation as the only feature. Next steps were: building a class structure, adapting core calculation to accept input shapes from Beampy's main class, developing waveguide implementation, creating GUI.

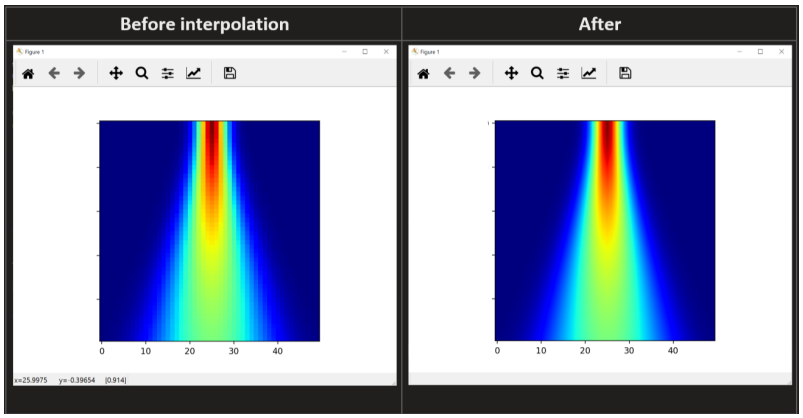

Figure 4: Propagation timed in 0.0280 seconds before and after interpolation.

First of all, the class structure was carefully designed. The code design has a parent-child pattern, the class with the main functionality would inherit the GUI functionalities from a GUI focused parent class. The main functions in the FdBpm class were split into the following methods:

- `create_space()` : sets all the needed variables for the next steps, selects the window size, number of samples, length of propagation.
- `create_source()` : enables the user to choose the light source and offset
- `create_guides()` : selects regions and applies difference in refractive index
- `calculate_propagation()` : core of the propagation calculation using FD-BPM
- + `make_tridiagonal_matrix()` : calculates tridiagonal matrix used to solve system

Next step was the implementation of waveguides creation. The objective was to achieve to have a guided wave at first, by introducing the low variation of refractive index to the calculation. The first result achieved is seen in Figure 5.

Next validation step was to achieve to couple two or three waveguides. An example from lumerical `https://support.lumerical.com/hc/en-us/articles/360042304694-Evanescent-waveguide-couplers` was taken and the focus was to reproduce something similar. The result was a three waveguide coupling.

3.3 EasyGui

The last step was to develop a GUI for later integration with the already existent GUI system I built for Beampy. Seen that the implementation of a GUI that contains matplotlib is not always easy, I decided to create a module for an easy implementation.

The idea was to still be able to develop the GUI using Designer, but to automatically compile it and import it in any code that it's being implemented on. I called the module EasyGUI.

In terms of implementation it is very straight forward. The user creates the .ui user interface on Designer, then they just need to import EasyGUI in the code as a parent class to their main class.

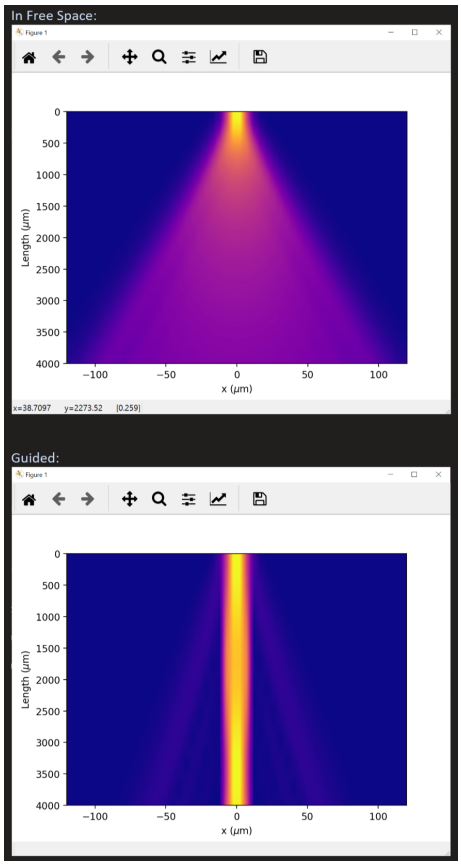

Figure 5: First result of refractive index variation implementation

Two methods enable the user to link the functionalities to the GUI elements and also to update the matplotlib.

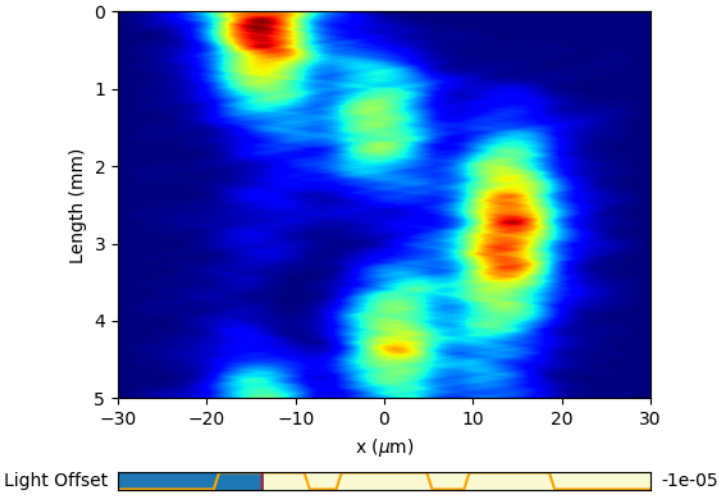

Figure 6: Reproduction of three waveguide coupling. Time taken: 0.3 seconds

4 Next steps for releasing FD-BPM in Beampy

- Perform more validation simulations
- Merge new GUI system based on PySide2
- Update docs

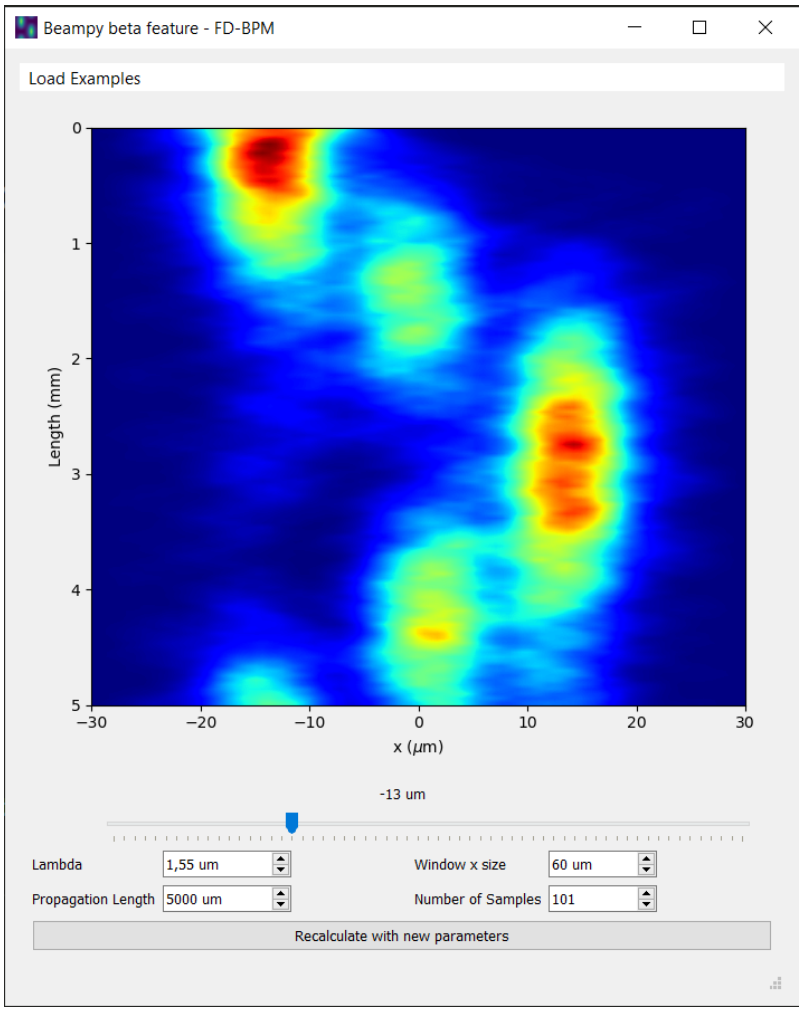

Figure 7: Guided User Interface. The slider moves the light source in real time.

References

[1] Y. Chung and N. Dagli. An Assessment of Finite Difference Beam Propagation Method. *IEEE Journal of Quantum Electronics*, 26(8):1335–1339, 1990. ISSN 15581713. doi: 10.1109/3.59679.

[2] J. W. Cooley and J. W. Tukey. An Algorithm for the Machine Calculation of Complex Fourier Series. *Mathematics of Computation*, 19(90):297, 4 1965. ISSN 00255718. doi: 10.2307/2003354. URL `https://www.jstor.org/stable/2003354`.

[3] G. L. Pedrola. *Beam Propagation Method for Design of Optical Waveguide Devices*. 2015. ISBN 9781119083375. doi: 10.1002/9781119083405.