**SBC Bubble Image Simulation (Porting MATLAB to Python)**

**Lawrence Luo**

**Introduction:**

The point of this project was to port the SBC Image Simulation code, written in MATLAB, to Python. The original code was designed to produce an accurate simulation of the image captured by the cameras of the bubble chamber. This would be used to drive physical design revisions of the chamber with the goal of ensuring sufficient image quality. The original code attempted a few different methods for how to model the light, to some success, however the methods were generally too computation-heavy to produce high quality results. This project attempted to port the original MATLAB code to Python with the goal of improving program execution time—if the code could be ported to Python, then we could eventually figure out how to run it on GPUs.

Converting the code from MATLAB to Python was quite the challenge because of multiple reasons. Firstly, and most obviously, the code was long. Secondly, some parts of the structure of the MATLAB code had to be reorganized in order to achieve the same functionality in Python. Thirdly, many of the built-in MATLAB functions had to be brought in from other libraries in Python—the primary one was *numpy*. Because this was a heavy task, I spent the entire project simply converting the code over to Python without optimizing many of the low-level hand calculations that occur throughout the program. I simply ported code and restructured at the high-level where I could. I did not test or debug the code, I have only been able to convert all of the code to Python without syntax errors, but testing the code requires a more in-depth understanding of the underlying calculations in the model. In fact, most of the comments in file are the original comments left my previous code designers. The decisions I made are listed in this document. I describe my workflow in converting the code. I explain each file at a high level and describe how it can be used. I’ve included helpful tips for future users of parts of the code, as well as recommendations for future development. Finally, I list some potential issues that may be helpful when debugging certain parts of the code.

**Code Organization:**

The original code is broken up into 14 files which will each be explained below. Some of the files are scripts and some of them are classes to provide structure for data storage and corresponding functions. To understand each file as a black box, look at the beginning of the python file—there are comments that explain the inputs and outputs of each function. These comments are the original comments from the MATLAB code.

1. **BubbleImageCreator.py**

This file was ported from **CreateArBCGeometry\_WithLights\_Split.m.** This if the main file which runs and calls all other code. It defines geometric parameters necessary for defining geometry necessary in the model. It uses a new class called geospecswhich is defined in **geospecs.py** to store this information. An instance of geospecs is initialized with default values specified in its file, but can then be changed to hold new values that may be desirable for different models as shown in BubbleImageCreator. Once the geometric values have been prepared and stored in a geospecs instance, that instance is passed to createGeometry()which is a function defined in **createGeometry.py**. This returns 4 variables which define initial information used for modeling the light rays in the future: a list of the geometric surfaces which is stored in list of instances of a new class called surface (defined in **surface.py**), a list of rays, a list of starting points, and a list of pixels. Output variables can be changed, removed, and added to fit the proper modeling goal. In fact, in the template that is provided for this file, the pixels are nonexistent and the variable pixels is empty—they should be defined before use. They are already defined in the actual (non-template) file. Next, that information is passed to the core engine, RayTracer2(), which is defined **in RayTracer2.py**. Some other parameter values are passed to RayTracer2() as well. All of these input parameters are explained at the beginning of RayTracer2.py in the comments. Once the output of the RayTracer2() is obtained, it can be used in any way to display or plot the information. Further calculation can also be performed based on what model is desired.

* 1. **geospecs.py**

This file was added to support the necessary functionality in Python that was unnecessary in MATLAB. This file contains the geospecs class which holds general information about defining geometry in createGeometry(). The information stored in an instance of this class can overwrite the default values assigned at initialization, found in geospecs.py.

* 1. **finalRays.py**

This file was added to support the necessary functionality in Python that was unnecessary in MATLAB. This file contains the finalRays class which holds the information about the final rays after being obtained in BubbleImageCreator.py.

1. **createGeometry.py**

This file was ported from **CreateArBCGeometry\_WithLights\_Split.m**. Geometry is defined in this file and stored in an instance of surfacewhich is defined in **surface.py**. These surfaces are stored in a list called surface\_list. To define a new surface, instantiate an instance of surface and fill out its attributes which are initialized to empty by default. Once the geometry is defined, define the rays that you want to model. Finally, these values are returned.

* 1. **surface.py**

This file was added to support the necessary functionality in Python that was unnecessary in MATLAB. Surface.py defines a class called surface, which has attributes *description, n\_outside, n\_inside, surface\_type, absorption, shape, param\_list*. *Shape* is a string which has 4 valid values: “cylinder”, “sphere”, “torus”, and “plane”. These are the 4 ideal shapes that can be used to define the ideal shape of any surface in the model. These strings will be used later in the process, when rays are intersected with surfaces and the core engine needs to decide how to calculate the intersection; it will look at the string to determine what kind of shape the surface is and then it will call the corresponding intersection function. The parameters that are used in this calculation to determine the exact shape geometry are stored in the param\_list variable. For other information, check out the createGeometry.py and surface.py files which have comments at the top describing each parameter in more detail.

Right now, only surfaces with shape “cylinder” are supported because only the RayToCylinder function was ported to Python for testing purposes. To support the other shapes, the files RayToTorus, RayToPlane, and RayToSphere must be ported over to Python from their original MATLAB files.

* 1. **GenerateRaysFromCamera.py**

This file was ported over from GenerateRaysFromCamera.m. It is used to generate the rays originating from the camera, which are used in the model that this solution was ported over from. New models that do not require rays to originate from the camera may not require the use of this file. Currently, it is being called in createGeometry to define rays originating at the camera.

1. **CORE ENGINE: RayTracer2.py**

This file was ported over from RayTracer2.m. It is used to trace the rays generated in createGeometry. It is essentially a file that runs calculations by hand to model the light rays. It stores the return accounts for the geometric surfaces in instances of rayInterfaces whose structure is defined in **rayInterfaces.py.** For calculation, it then calls three other functions to perform calculation of the desired model: RayleighScattering(), UnifiedReflectorModel(), and RefractionReflectionAtInterface() which are each defined in its respective .py file. For information abot how RayTracer2 works and what kind of inputs and outputs it has, read the documentation at the top of the RayTracer2.py file.

RayTracer2 calculates Ray-Surface interesections differently than before in MATLAB. The new RayTracer2 calls IntersectFunction() defined in **IntersectFunction.py** which calls the corresponding RayToShape() function defined in **RayToShape.py** which calls the corresponding correct RayToXXXXX() function defined in the corresponding **RayToXXXXX.py** function. Right now, only RayToCylinder.py is written, for testing purposes.

* 1. **rayInterfaces.py**

This file was added to support the necessary functionality in Python that was unnecessary in MATLAB. It defines the class rayInterfaces which stores ray information calculated in RayTracer2.

* 1. **RayleighScattering.py**

This file was ported over from RayleighScattering.m. I wrapped the function RayleighScattering() in a class called RayleighScatteringClass in order to have a member attribute rayleigh\_azimuth that retained its value across multiple calls to the function.

* 1. **UnifiedReflectorModel.py**

This file was ported over from UnifiedReflectorModel.m.

* 1. **RefractionReflectionAtInterface.py**

This file was ported over from RefractionReflectionAtInterface.m.

* 1. **IntersectFunction.py**

This file was added to support the necessary functionality in Python that was unnecessary in MATLAB. Before in the MATLAB code, the proper RayToXXXXX function was stored as a lambda function attached to each surface. However, in Python, I found it easier to create a global Intersection Function that took in parameters stored in each surface to run the proper RayToXXXXX function. IntersectFunction() passes the information to the RayToShape() function. IntersectFunction() is essentially just a wrapper for the RayToShape() function. It gets the output from RayToShape() and then passes it back out to RayTracer2.

* 1. **RayToShape.py**

This file parses surface information and calls the proper RayToXXXXX() function, then returns the intersection information back to IntersectFunction.

* 1. **RayToXXXXX.py (RayToCylinder.py)**

This fil was ported over from RayToCylinder.m.

**Changes to Syntax**

The majority of the changes to small pieces of syntax were to convert array references from MATLAB to Python. This meant converting all indexing brackets () and {} into square brackets [] and subtracting 1 from all hardcoded indexes. I also had to change all matrix algebra to accommodate the Python operators and data storage. To achieve this, I used the *numpy* library which achieves the same functionality of MATLAB matrices by using numpy arrays. *Numpy* is based largely on MATLAB and contains many of the same functions, although some of the ordering of inputs and syntax are different.

Most of the built-in functions of MATLAB had to be converted to build in functions in Python such as np.matlib.repmat(), np.tile(), np.random.randn(), etc. Additionally, all built-in math functions had to be replaced with functions such as np.arccos(), np.sum(), etc. All semicolons from MATLAB had to be removed, and function definitinos had to be changed to fit Python’s syntax, including return statements. Variable instantiations had to be fixed and many arrays had to be converted to python lists or numpy arrays. All multiline statements in MATLAB were converted to single-line Python statements.

**Future Work:**

In the future, the amount of np.matlib.repmat() and np.tile() references can be decreased because numpy supports auto-repeat of arrays during calculation if inputs do not match in size, where MATLAB does not. However, I did not perform this during my work. Additionally, many of the long statements do not follow good code etiquette and should be into multiple lines of code with sub-calculations.

**Debugging and Testing:**

This code has not been tested or debugged. At the most simple level, the code does not have any syntax errors. All files contain the necessary imported files to link the entire project together, but there may be issues if the code is to be run with integration of the files. These are the main potential issues that may occur. NOTE: I thoroughly checked using pycharm, sublime, and xcode to see any easy to spot Python syntax errors and my workflow was geared towards catching any errors, but I expect that there will still be a few errors that fell through the cracks. These are a list of potential problems, although you should not expect that there are many occurrences. I am simply including this to bring awareness to what might be the most common problems, and to keep them in mind during future development:

* Array indexing off-by-one errors. I had to manually subtract 1 from all MATLAB indexes because MATLAB indexing starts at 1 and Python indexing starts at 0.
* Array indexing using () instead of []. I had to convert all () into [] for array indexing.
* Invalid built-in functions. Built-in MATLAB functions like sum() do not exist in the standard library of Python. They instead live in the numpy library. I had to manually prepend “np.” In front of many of the common functions and I may have missed a few here and there.
* Semicolons. MATLAB ends lines with semicolons but Python does not—I may have missed a couple.
* Numpy ndarrays. I designed the functions to take python lists as inputs and return python lists as outputs. Between those two steps, some of the data may have been converted to numpy arrays, which was necessary for calculation. This means that at the function interfaces, there may be issues where something expects a list and instead receives a numpy array, or vice versa.
* Bitwise & and OR. In many areas, Boolean calculation is performed. There is a difference between the python operator “and” and the python operator “np.logical\_and()”. The former evaluates arguments as Booleans and performs an AND operation. The latter performs an elementwise AND on the arguments. I may have missed a couple of the latter cases and instead used the first one.
* np.cross(). MATLAB assumes the third parameter in the built-in function is the “dim”, but numpy must assign this parameter explicitly. I chose to use np.cross, so it looks like “np.cross(input1, input2, axis=1)” rather than “cross(input1, input2, 1)”. I am pretty certain that axis is the correct parameter to use.
* importing files. The path for where the code is run must be figured out. All the files should include the correct “import” statements to be able to reference the other files, but I think they need to be put into a folder that correctly links them together to be able to find each other.

**How to Use this Code to Write an Image Simulation Model**

Use BubbleImageCreator.py to define overall geometric parameters. Then use createGeometry.py to define the actual geometry, the light ray locations, the pixel locations, and any other useful information that helps achieve the desired model. Once this is complete, make sure the outputs defined in createGeometry() match the outputs expected in BubbleImageCreator.py. Then call the core engine RayTracer2() with the proper inputs to model the light rays, along with some other parameters which are explained in the comments at the beginning of RayTracer2.py. Finally, in BubbleImageCreator, use the RayTracer2 output to perform any calculations and display any plots that are desired.

You can write new RayToXXXXX files to calculate intersections of rays with surfaces of new shapes other than Cylinder, Sphere, Plane, and Torus. Right now, only Cyinder is ported to Python. The other three exist in MATLAB and need to be ported over if desired. If another shape is required, write a new one.

Templates are provided for the main file (SimplifiedTemplates/BubbleImageCreator.py) and for geometry creation (createGeometry.py). These can be used to create a new model, in conjunction with the actual core engine files (See #3 in **Code Organization**) which do not need to be changed.

**How to Use Only the Core Engine for Other Simulations**

The core engine requires all the files listed in #3 in **Code Organization**. These files do not need to be changed and can be used as a black box. The main file is RayTracer2.py. As a black box, it takes in the inputs that describe the rays, physics model parameters, and geometry and it outputs the ray information. Detailed information about how these work are in the comments of RayTracer2.py. Additionally, see “**Recommendations for Future Development**” below.

**Recommendations for Future Development**

Firstly, the code must be tested and debugged. This requires the ability to validate the hand calculations that occur throughout the program. Since the calculations have many steps, it would be futile to just look at the end product (a simulate image) to diagnose code problems.

Secondly, in order to speed up the raytracing, I suggest looking into some open-source raytracer python libraries that exist in order to avoid much of the hand calculation. I am not sure what the capabilities of those existing raytracers are, but they may tradeoff decreased customization for increased power. There are many libraries that exist. I did not look into them because I was interested in porting over the exact customized code from MATLAB.

After the code is tested and debugged, then work can begin to get the code to run on GPUs and to execute with much quicker performance.