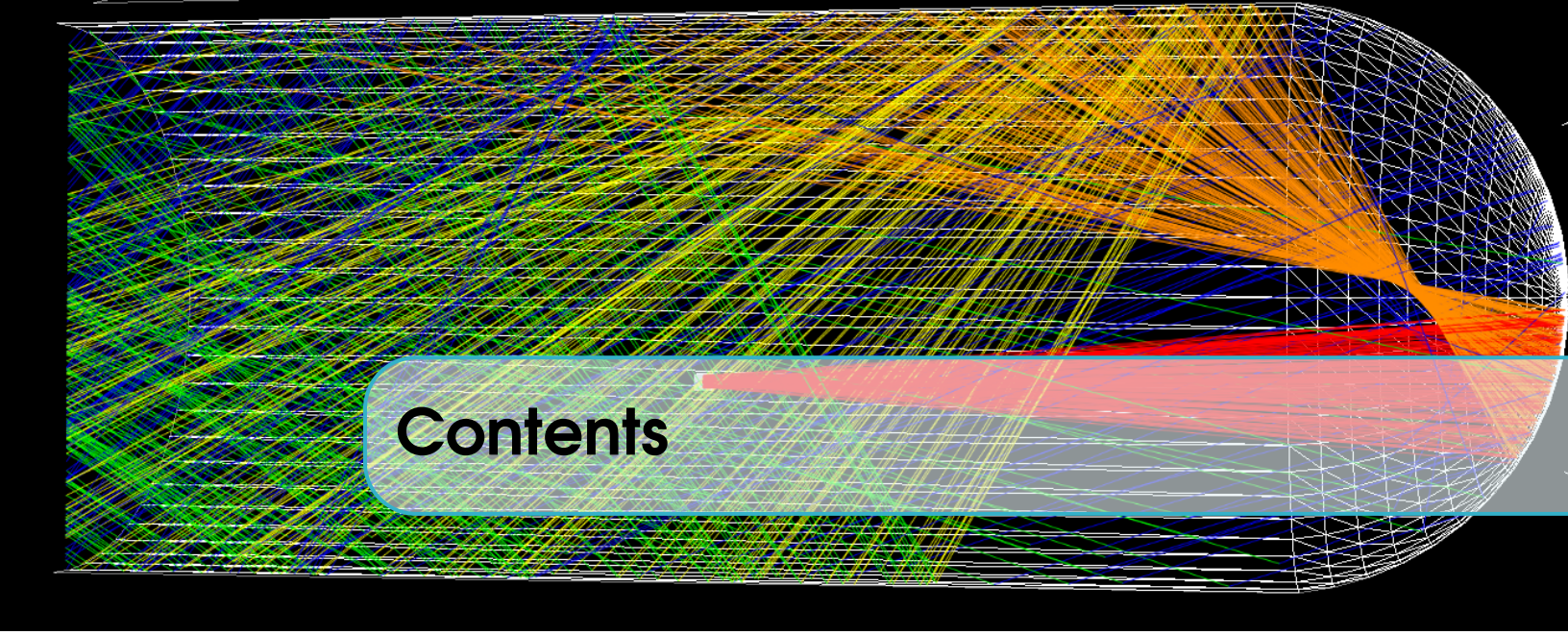
Modeling the reflective pattern of microwaves for use in ammonia target polarization

Matthew C Wild

Mentor: Dr. Robert Fersch



Contents

1. Introduction

1.1 Objective

For my capstone project, I wrote software to simulate the reflective pattern produced by the interaction of microwaves and surfaces with various geometric and absorptive properties. The goal was to model a given reflective surface and compare the illumination and absorption patterns to data gathered by SArah Clark and Dr. James Maxwell.

The complete process of this simulation from establishing parameters and initial conditions through comparison of data is outlined in appendix B as fig. B.3.

1.2 Motivation

Over the summer I had the opportunity of working with the polarized target group at Thomas Jefferson National Accelerator Facility. This group is in charge of creating two polarized targets for use in Hall B's detector. By directing the electron beam into these targets, physicists are able to study the intrinsic angular momentum, known as spin, of the proton. The free Hydrogen in the two target cells will be polarized via dynamic nuclear polarization [DNP] through interaction with microwaves with a frequency of about 150GHz. The two targets will be polarized opposite of each other, either with or against the polarization of the electron beam. I was made aware that the structural design of the experiment prohibited direct irradiation of the target cells (see fig. 1.1).

Therefore, they needed to implement a surface to reflect the microwaves onto polarized target cells. Up to that point the team had been testing various surface geometries and materials simply by 3D printing a shape and gathering rudimentary data produced by these reflectors.

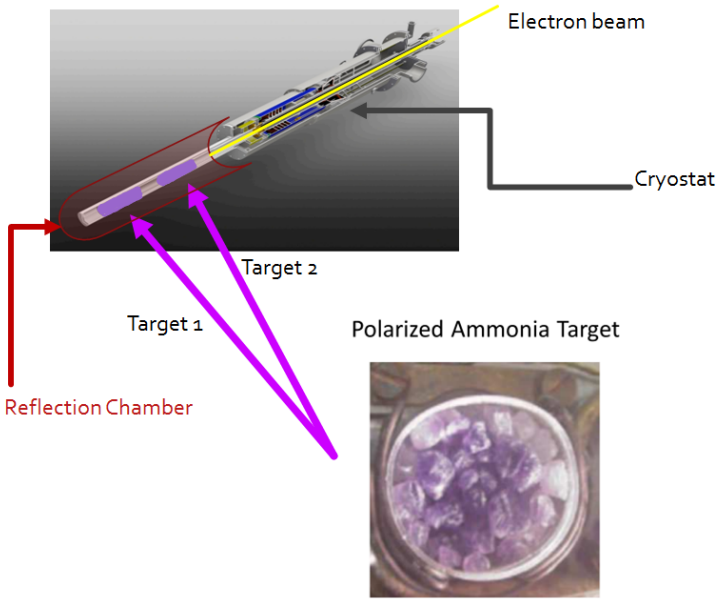


Figure 1.1: Physical Setup

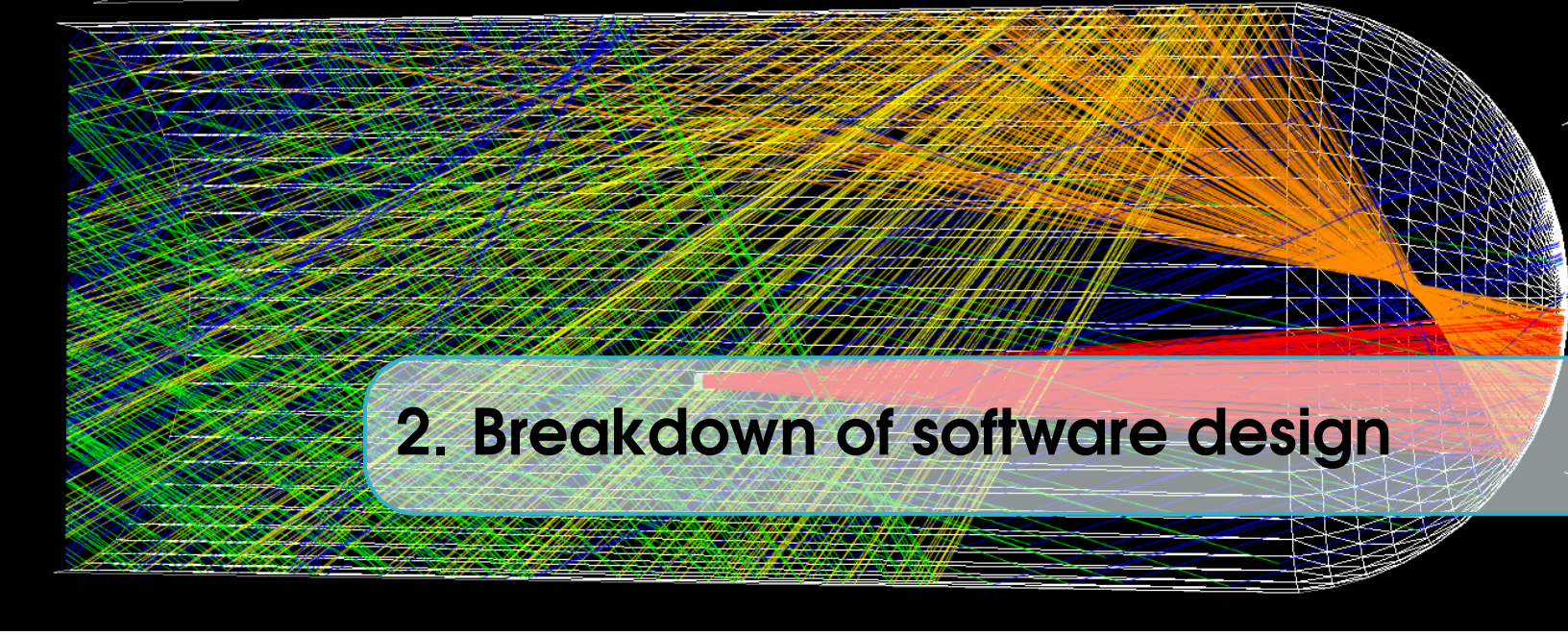
Dr. Fersch assigned me the task of writing code to predict the behavior of the microwaves as they reflected off of a surface inside the apparatus.

1.3 Rationale

The decision to write a 3D visualization of the simulated rays behavior rather than a simple mathematical model of the resulting reflective/absorptive pattern came from following desired functionality:

- The ability to share my results in an easily understood format
- The ability to effectively compare the visual results of my simulation to qualitative result

I chose to develop my simulation in Java, as I have about 6 years of experience with the language. I also was further influenced when I was made aware of the JavaFX library which was developed by Oracle for the purpose of simplified 3D rendering and object manipulation.



2. Breakdown of software design

2.1 Main class

The main class in my program is titled `Run_Experiment` and contains all methods necessary for detecting user input, initializing parameters, and writing the output file.

2.1.1 buildCamera

This method initializes the camera object necessary to render the simulation in 3D as well as manipulate the perspective (i.e. rotation/translation). The camera is then placed inside a group that is responsible for the rotation and this group is placed inside yet another group that is responsible for translating the image in 3D space. This method also initializes two light objects, one ambient and the other a point light. The ambient light ensures that every facet is illuminated while the point light allows for the visualization of depth. Finally, the parent camera group, the ambient light, and the point light are grouped together which is then returned when the method is called.

2.1.2 handleZoom

This method was specifically implemented to allow pinch zooming on touch screens. It achieves this by detecting the `zoomFactor` and translating the position of the root node along the z' axis by this factor, where the z' axis is perpendicular to the display.

2.1.3 handleMouse

This method captures mouse events such as mouse down (primary or secondary buttons), mouse movement, and if modifier keys (ctrl, Shift, etc.) are pressed. This starts with detecting the mouse down event. Pressing either button toggles the visibility of the rays. This function was implemented in order to massively reduce

strain on the graphics processor of a tablet, but it is not necessary for more powerful machines. Pressing a button down also begins tracking the difference between the start position and current position. If the primary button was pressed, then the change in the x position and y position are applied to a rotation transformation in the respective directions. If the secondary mouse button was pressed then it acts in a similar fashion to the `handleZoom` method by translating the image along the z' axis proportional to the change in the x position. In the event that both control and Shift are held while dragging either mouse button, the change in position is directly passed into translating the root node along the x' and y' axes, which are simply left/right and up/down on the display, respectively. This method concludes with setting the rays as visible once the mouse button is released.

2.1.4 handleKeyboard

This method captures keystrokes and consequently adjusts the layout of the rendered simulation. Table 2.1 shows the function bound to each expressed key.

Key	Function	Description
SPACE	default view	Sets camera position and angle to the initial view
Z	toggle absorption	Toggles the visibility of the absorption pattern
X	toggle surfaces	Toggles the visibility of all rendered surfaces
C	center	Places the simulation in the center of the window
V	toggle rays	Toggles the visibility of all rays
[Directional]	pan	Translates the simulation in given direction
Q	rotate cc	Rotates the simulation counter-clockwise
E	rotate c	Rotates the simulation clockwise

Table 2.1: Key bindings

2.1.5 ptsOnCircle

This method takes in the location of it center, the desired radius, and integer minimum/maximum separation (in degrees) between the points. Starting from the horizontal axis, a random integer is chosen between the two limits [inclusive] to be the distance to the next point. This continues over 360deg, with each point being added to the ArrayList which is ultimately the object returned by the method.

2.1.6 createBeam

This method takes in a central position, initial direction vector, and the radius of the apparatus from which the beam emerges. It randomizes a distance between 0 and a half of the provided radius which it then sets as the first radius and loops while adding a random discreet amount to the first radius until it reaches the radius of the apparatus. Within this loop, another loop is called to iterate over the points generated by the ptsOnCircle method at each radius. At every point that exists within the reflection chamber, a ray is produced. This ray's direction vector is the sum of the provided vector of the beam and a random divergence between 0 and a previously established constant.

2.1.7 bounce

The bounce method is passed a Ray object, otherwise known as the parent Ray, and the number of interactions it will calculate for said ray. It loops the given number of times and, as long as the ray was not absorbed by the last surface it interacted with, produces the resulting reflected ray's direction by calling the Ray.reflect() method. This is used in addition to the present ray's end position to create a new ray. This new ray is assigned a color based on how many reflections have occurred. Finally, this ray is added into the ArrayList of the parent Ray's reflections. This means that every ray in the simulation is a child linked to one of the original emitted rays.

2.2 Quadric Surfaces

The general equation for any quadric surface is:

$$0 = Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J \tag{2.1}$$

2.2.1 Parsing a given equation

By far one of the most complex tasks of this piece of software is parsing a given string into the standard quadric form (eq 2.1). The string that needs to be parsed is of the format: J=F[x,y,z]. This conversion is achieved through extensive use of Regex by the following process:

1. Split the string at the '=' and multiply the left side by -1 to attain the J constant for this surface

2. Throw away everything up through the '=' and focus on this right side of the equation
3. Replace all occurrences of any letter between [x-z] (let this letter be denoted as `r`) which is not preceded by a number or another letter with `1.0r`
4. Replace all occurrences of '-' with '+-'
5. Replace all occurrences of '-' followed by any letter between [x-z] (denote again as `r`) with '`-1.0r`'
6. Replace all occurrences of an integer preceding a letter between [x-z] with the double of that integer (i.e. $2x^2 \rightarrow 2.0x$).
7. Insert a space between every occurrence of a `double` and letter between [x-z]
8. Split the resulting string from the previous operations at each '+'
9. Iterate through this array and split each item at the ' '
10. If the resulting array is longer than 1, use a HashMap with entries formatted to have the variable expression as the key and the double as the value (i.e. 2.0 $x^2 \Rightarrow$ `.put("`x^2`",2.0)`).

Optional Pass these entries through another loop that assigns each mapped value to the proper constant ([A-I] in eq 2.1).
If this step is forgone, all future references to the constant must recall the value mapped to each string. For example, [map].getOrDeafault("x^2",0.0) to get the value for A in eq 2.1. The getOrDefault(key,default) is needed to handle the fairly common instance when not every constant is explicitly defined and should be implied to be 0.0.

2.2.2 Finding the intercept

Calculating the point where a ray intercepts a quadric surface begins with subtracting the offset position of the surface from the starting position of the ray. Then, we use the vector function:

$$\vec{e} = \vec{s} + t\vec{v} \tag{2.2}$$

where $\vec{e}$ is the end position of the ray, $\vec{s}$ is the starting position, and t$\vec{v}$ is the directional vector multiplied by the scalar value t. Splitting $\vec{s}$ into x,y,z and $\vec{v}$ into i,j,k results in

three useful equations:

$$\begin{aligned} e_{\hat{x}} &= s_{\hat{x}} + tv_{\hat{x}} \equiv x + ti \\ e_{\hat{y}} &= s_{\hat{y}} + tv_{\hat{y}} \equiv y + tj \\ e_{\hat{z}} &= s_{\hat{z}} + tv_{\hat{z}} \equiv z + tk \end{aligned} \tag{2.3}$$

Substitute these three equations (2.3) as the x,y,z parameters respectively for eq 2.1 and solve for t using the quadratic formula with terms a,b,c set to be:

$$\begin{aligned} a &= Ai^2 + Bj^2 + Ck^2 + Dij + Eik + Fjk \\ b &= 2(Axi + Byj + Czk) + D(xj + yi) + E(xk + zi) + F(yk + zj) + Gi + Hj + Ik \\ c &= Ax^2 + By^2 + Cz^2 + Dxy + Exz + Fyz + Gx + Hy + Iz + J \end{aligned} \tag{2.4}$$

If a comes out to be zero, simplify the equation and reevaluate the to end up with:

$$t = \frac{-c}{b} \tag{2.5}$$

In the event b is also equal to zero, there is no possible intersection. Finally, reevaluate eq 2.2 using the value calculated for t to determine the interception point. For later use in ray tracing, this value t can be passed into a map containing each surface as the key. Compare these keys to determine the order in which this ray intercepts surfaces.

2.2.3 Calculating the normal

The normal vector of a quadric surface is simply the gradient of eq 2.1:

$$\nabla f[x, y, z] = (2Ax+Dy+Ez+G)\hat{i}+(2By+Dx+Fz+H)\hat{j}+(2Cz+Ex+Fy+I)\hat{k} \tag{2.6}$$

Evaluate the expression at the calculated intercept to determine the numeric vector which is necessary for ray tracing calculations.

2.3 Ray Tracing

2.3.1 Snell's Law

The underlying principal of this ray tracing software uses Snell's Law, which states the angle between the incident and the normal to a surface at a given point is the

same as the angle between the reflected vector and the normal. In order to implement this into my code, it was necessary for to convert Snell's Law given as $|\theta|=|\theta'|$ into its vector form,

$$\vec{r} = \vec{i} - 2(\vec{i} \cdot \hat{n})\hat{n} \tag{2.7}$$

where $\vec{r}$ is the vector of the reflected ray, $\vec{i}$ is the vector of the incident ray, and $\hat{n}$ is the unit vector normal to the surface at the point of intercept. Finding this intercept point will be addressed in section 2.2.1 and solving for the normal vector will be discussed in section 2.2.2.

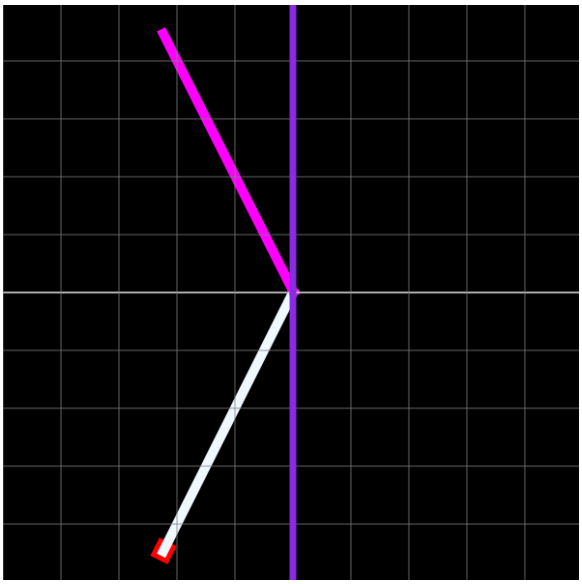


Figure 2.1: Simple 2D reflection off of a plane

Figure 2.1 is the result the first version of my code and exemplifies the basic principal of reflecting a single ray off of a plane in 2D.

For a complete map of the refection process, see figure B.2

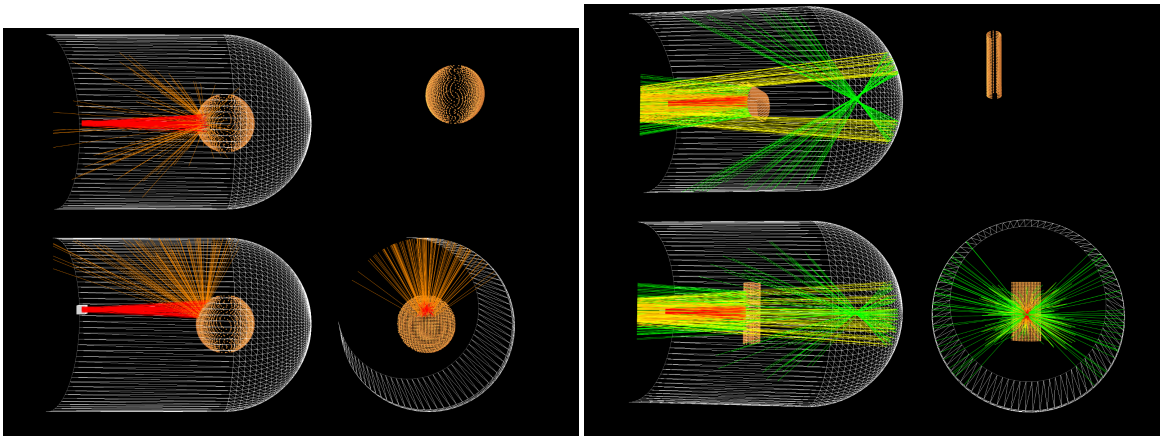
2.4 Xform class

The Xform class, which is available from Oracle, is an extension of the Group collection type. It utilizes Euler transforms to properly manipulate and orient nodes or shapes in 3D space. Almost every implementation of a group in the code is really a Xform object.

2.5 Rendering in 3D

The rendered simulation consists of the following three perspectives:

Perspective Name	Location on Screen	Orientation x-axis	Orientation y-axis	Orientation z-axis
Top	upper-left	right	out of screen	up
Side	lower-left	right	up	out of screen
Front	lower-right	out of screen	up	left
Reflector	upper-right	right	up	out of screen



(a) Reflected once with spherical reflector (b) Reflected twice with cylindrical reflector

2.5.1 Plotting the surface

The method of plotting a surface is called on the surface itself and essentially iterates over all 3D space by a specified resolution and attempts to solve eq 2.1 at each point. If a real solution exists, a small sphere is placed at the location and its color depends on the material of the surface. Figure 2.2 is an example of a surface with equation $0 = y^2 - z^2 - x$ that is bound by the volume (-2,-2,-2) and (2,2,2) being plotted using this method. While this methodology of rendering is obviously much more processor intensive compared to constructing a single mesh object, the performance impact has been determined not significant enough to putting the extensive time required to develop a mesh producing class.

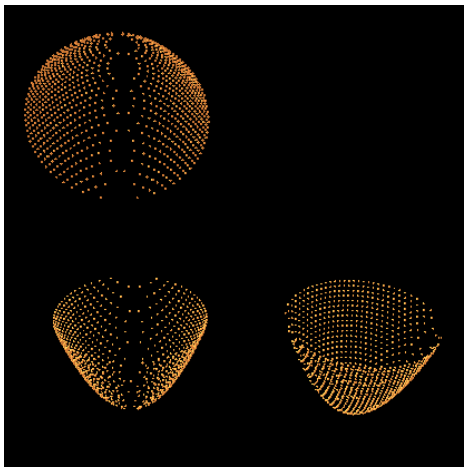


Figure 2.2: Surface rendered using small spheres

2.6 Constants interface

This interface is implemented by almost all of the classes an allows for global values to be passed between them without the need for additional parameters.

2.7 Material enum

An enumeration or, as it is known in Java, an enum is similar to a constant but it has set properties. In the case of this simulation, the Material enum contains various materials (Aluminum [Al], Copper [Cu], Ammonia [NH3], liquid Helium [He], and air [Air]). Each of these are created with the two parameters: absorption probability and render color. Determining the absorption probability of Ammonia in the lab set-up is well beyond the scope of this capstone but the polarized target group informed me to estimate it to be near zero.

2.8 Output file

The simulation saves a table of the number of rays absorbed per each thermocouple [column] at each incremental position [row] to a text file. Table 2.8 provides an example of this format.

Depth(cm)|#	1	2	3	4	5	6	7	8
0	10	6	333	7	56	0	28	8
..	..	..	..	..	..	..	..	..
13.5	107	68	303	7	569	0	238	68

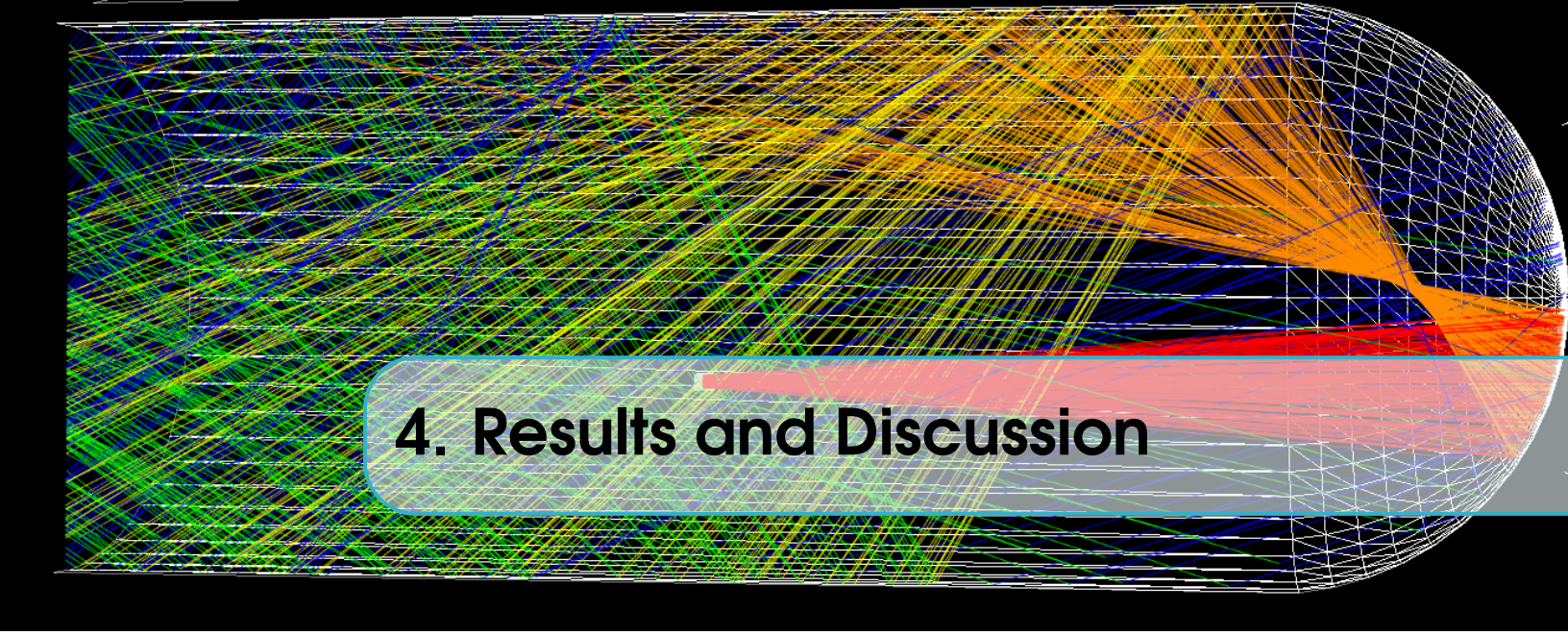
3. Dev Environment and Debugging

3.1 Dev Environment

This software is being developed using the Eclipse Neon IDE as it is the most up-to-date and comprehensive version of Eclipse available. Eclipse offers many of the same features as any other IDE such as compiling and running within the IDE itself. This IDE also contains an excellent step-by-step debugging features which displays the current values of variables in the current method while stepping through the executing code.

3.2 Debugging practices

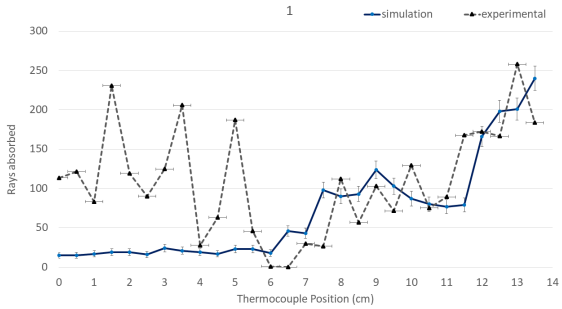
Due to the use of randomization in the ray production, broad test cases for final outcomes are impossible. Instead, utilizing print statements and breakpoints with the aforementioned step-by-step analysis of an unfolding process is extremely helpful when debugging. Given that this is a simulation, debugging certain visual bugs is a matter of evaluating the output by eye (such as inverted or improperly offset surfaces).



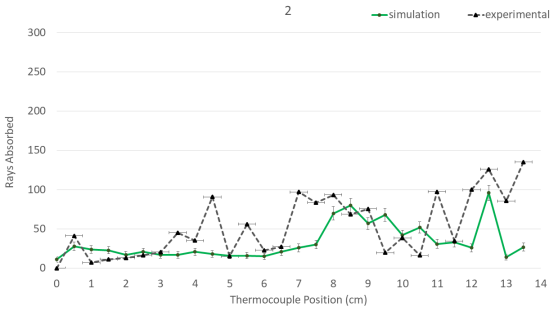
4. Results and Discussion

4.1 Results

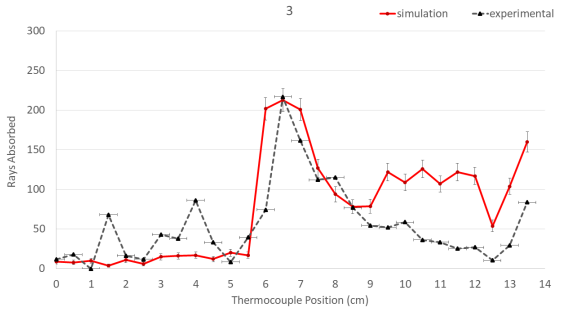
I used Microsoft Excel to plot the data stored in the output file. Each thermocouple was plotted separately as a graph of the number of absorbed rays vs. the distance into the cavity. In order to compare my simulation to the experimental data, which was temperature vs. distance, I had to modify the provided data's scale while maintaining its integrity. I started with filtering out the ambient temperature by subtracting every value in each data set by the lowest measurement of the set. This transformed the data set so that it started at zero. Next, each data set was multiplied by some factor so that it would be on the same scale as the simulation data. The transformed data set was superimposed over my simulation data. This allowed for the comparison of intensities and therefore reflective pattern generated by the surface. Fig. 4.1 displays the plot of each simulated thermocouple as a group. For larger images of these plots, see appendix A. The reason for transforming the experimental data rather than my simulation was due to the fact that my simulation is not capable of calculating an ambient background "temperature" as some number of rays.



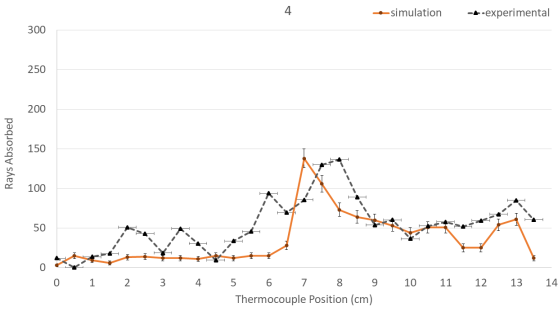
(a) Thermocouple 1



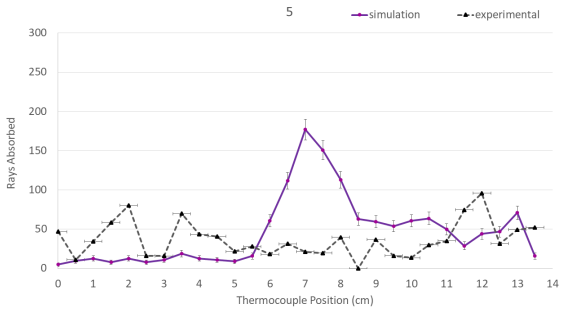
(b) Thermocouple 2



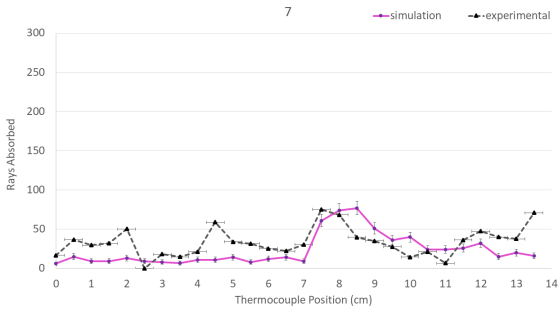
(c) Thermocouple 3



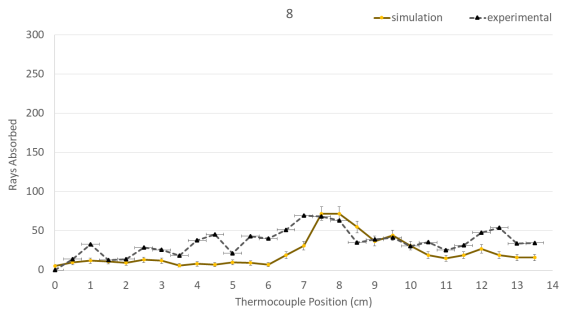
(d) Thermocouple 4



(e) Thermocouple 5



(f) Thermocouple 7



(g) Thermocouple 8

4.2 Discussion

The reason there is no plot for thermocouple 6 [fig:4.1] is because the physical thermocouple 6 is used to measure the ambient temperature throughout the cavity.

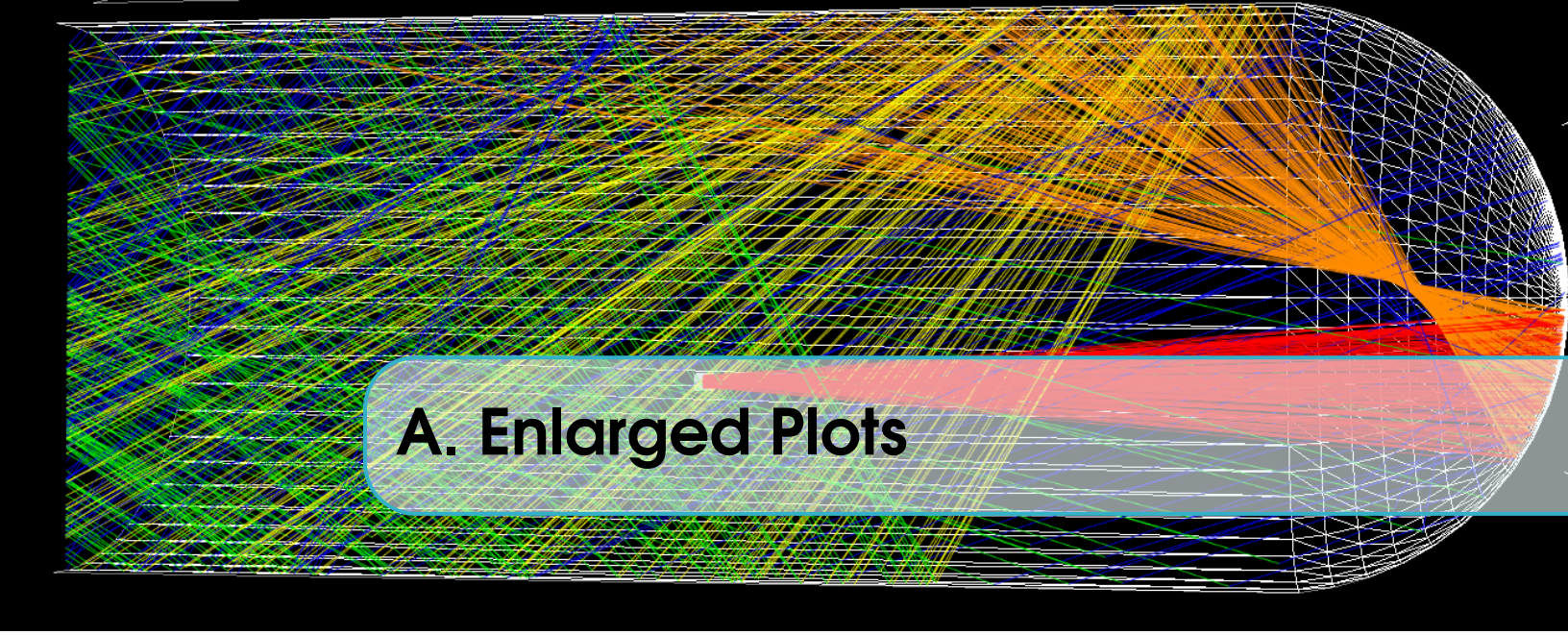
The relative location and intensity of the peaks in the charts indicates some degree of accuracy of the simulated results compared to experimental data.

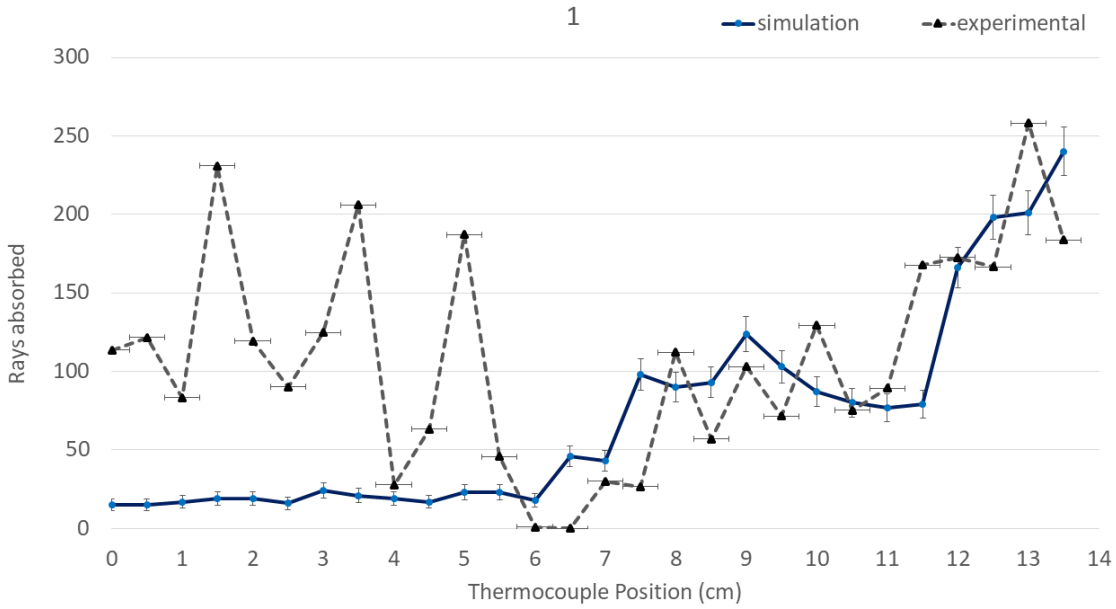
Because the number of rays absorbed by any thermocouple, compared to the millions of rays emitted, the vertical error bars on the simulation data can be estimated to be the square root of the number of rays absorbed $\delta N_{abs} = \pm\sqrt{N_{abs}}$. The horizontal error bars on the experimental data are simply one-half the smallest order of magnitude used to measure position in the physical set up (Δx=0.5cm and δx=$\pm$0.25cm).

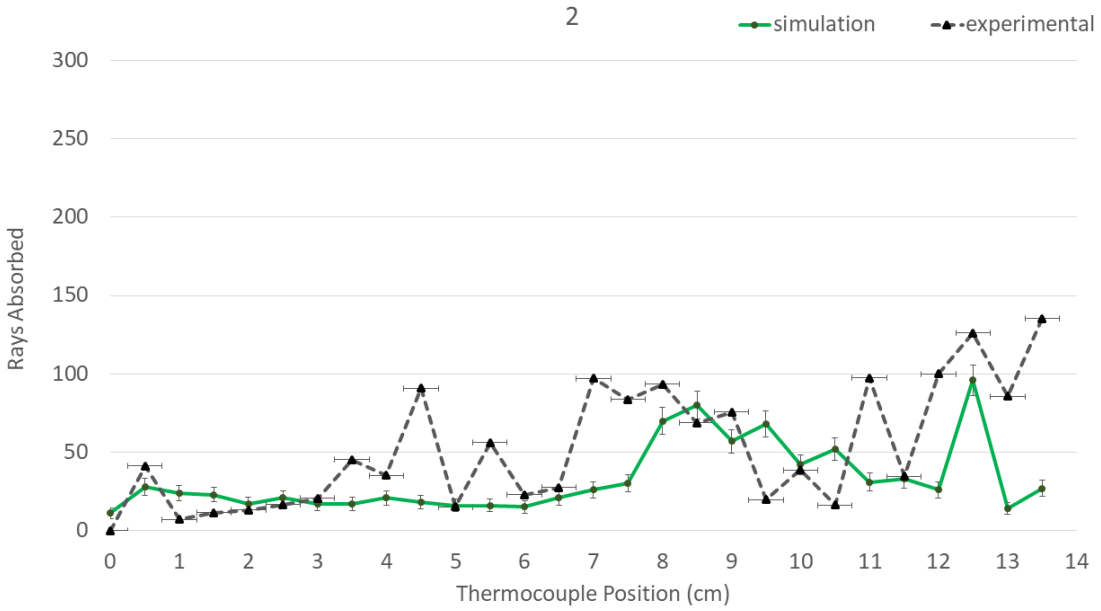
The major discrepancies lie with thermocouples 1 and 5. These thermocouples lie furthest from the reflective surface and away from the ramp's normal vector. Refer to fig B.1 for arrangement of thermocouples in reference to reflector. Another source of error could be that the ramp component of the reflective surface was potentially defined with an incorrect angle. This would lead to an offset pattern and potentially incorrect intensity in the number of rays absorbed by thermocouples on the outer edge of the array.

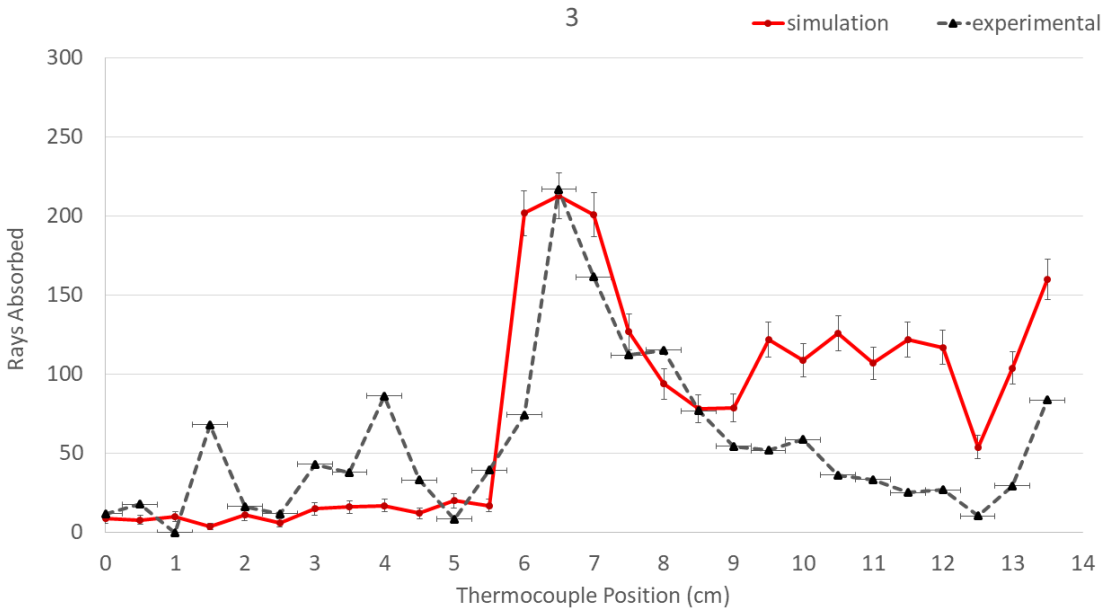
4.3 Conclusion

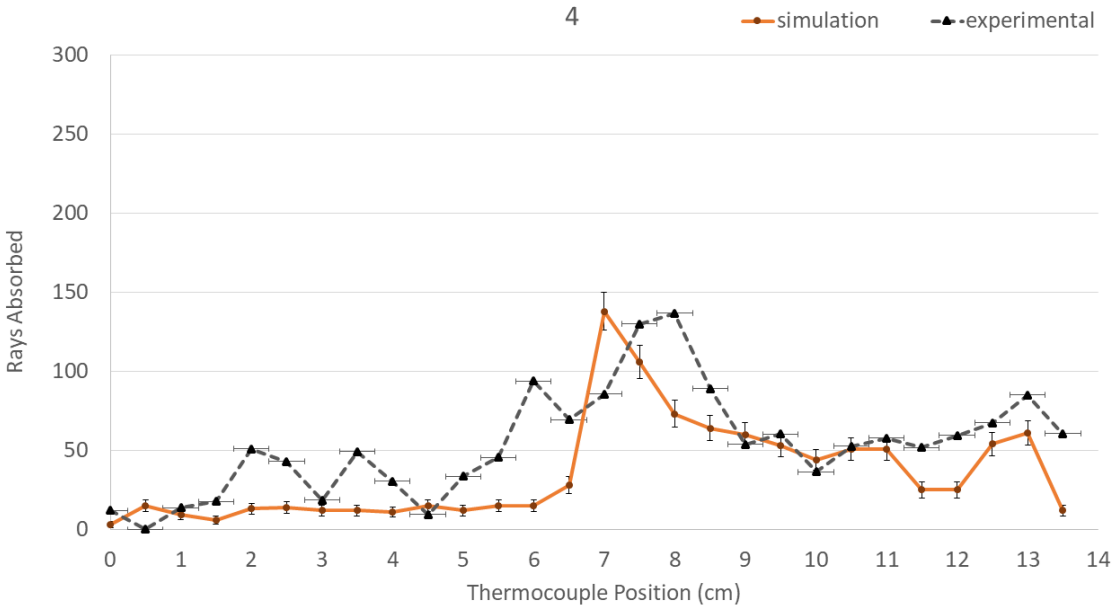
The minor discrepancies between simulation and reality stem from the inability to perfectly replicate any physical set-up in a simulation, which in this case lead to over/under-exposure of various thermocouples. Another source of error is the fact that the simulation does not take into account more advanced optical topics such as diffraction around objects with dimensions on the same order of magnitude as the microwave's wavelength. However, the results of the simulation developed for this project show that it is possible to emulate the behavior of microwaves being reflected off of a defined surface simply through the use of Snell's Law and quadric surface geometry.

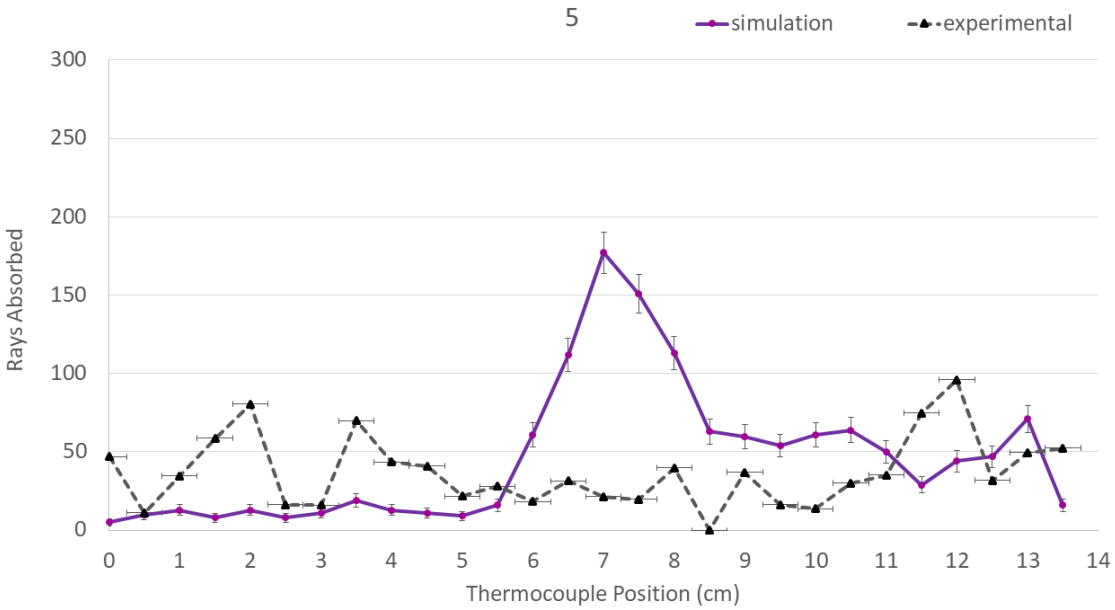

A. Enlarged Plots

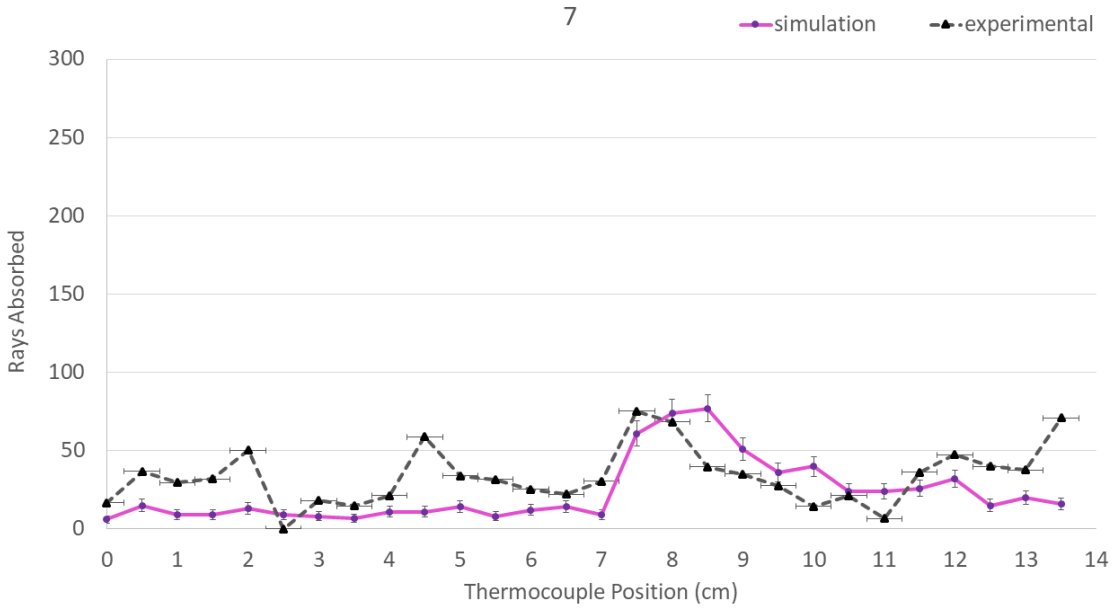

1
simulation
experimental
300
250
200
150
100
50
0
Rays absorbed
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
Thermocouple Position (cm)

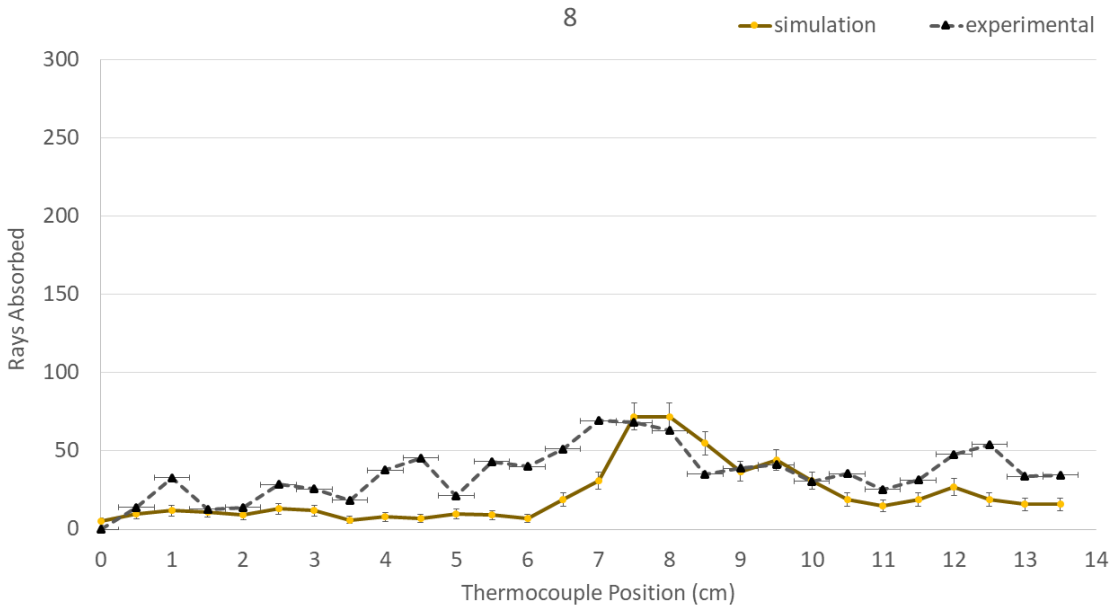

2
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)


3
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)

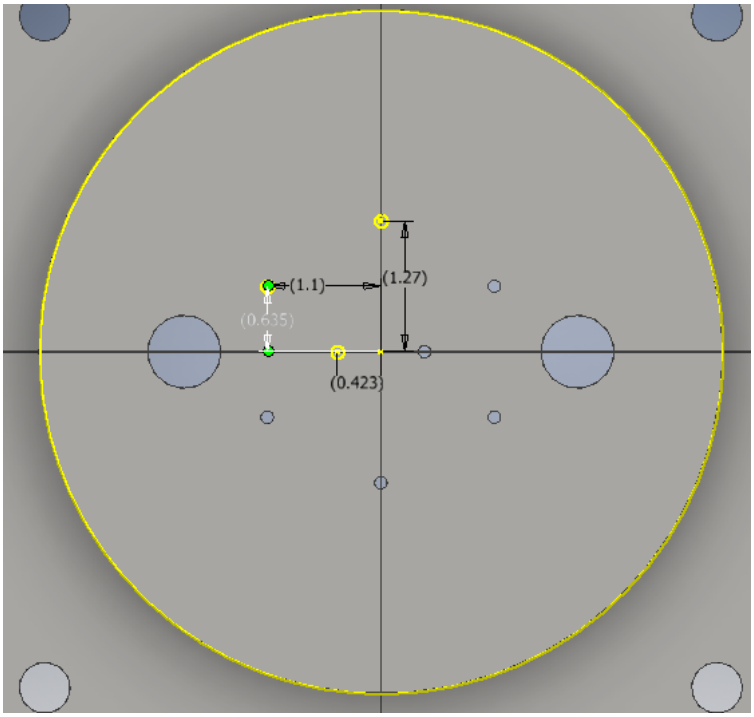

4
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)
0
50
100
150
200
250
300
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14


5
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)
0
50
100
150
200
250
300
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14

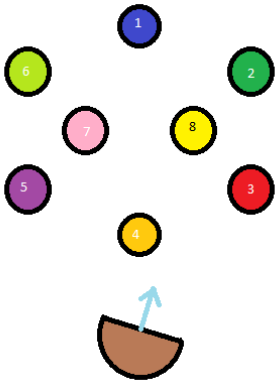

7
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)
0
50
100
150
200
250
300
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14


8
simulation
experimental
Rays Absorbed
Thermocouple Position (cm)
0
50
100
150
200
250
300
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14

B. Additional Figures



(a) Schematic of thermocouple array



(b) Location of thermocouples
Not to scale

Figure B.1: Arrangement and location of each thermocouple in experimental setup

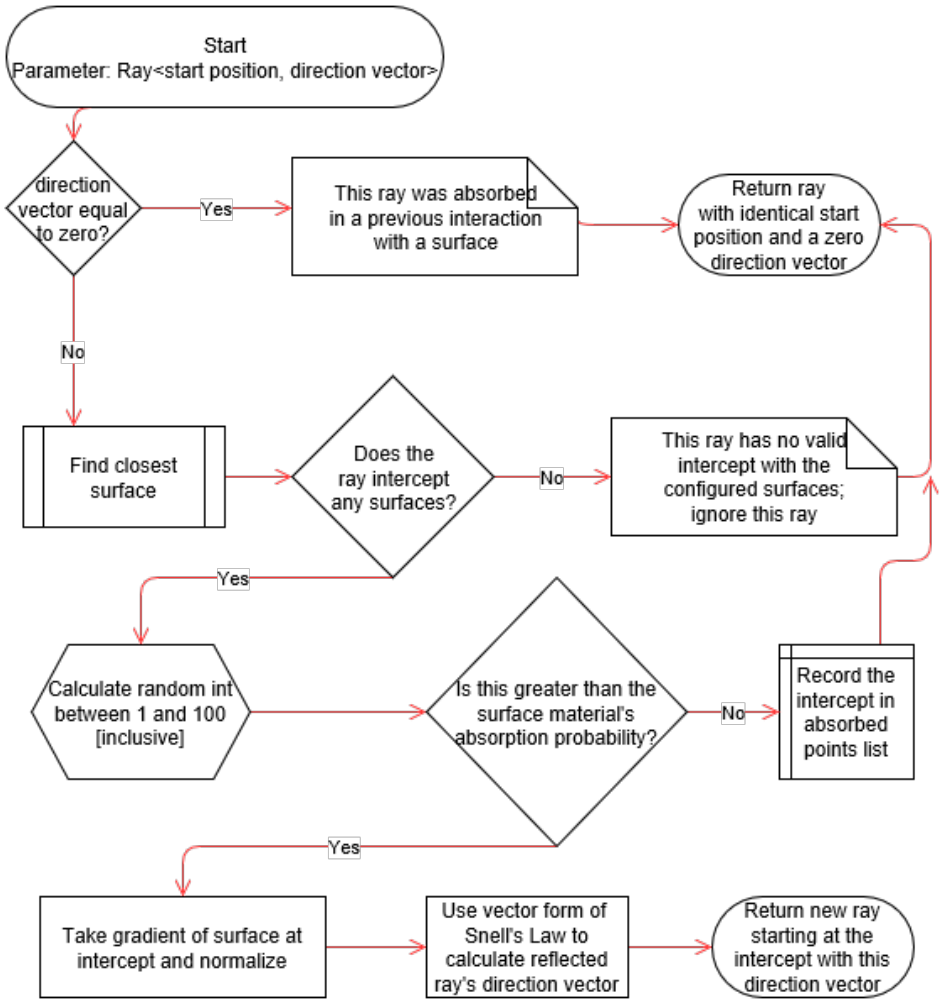


Figure B.2: Flowchart of reflecting a ray

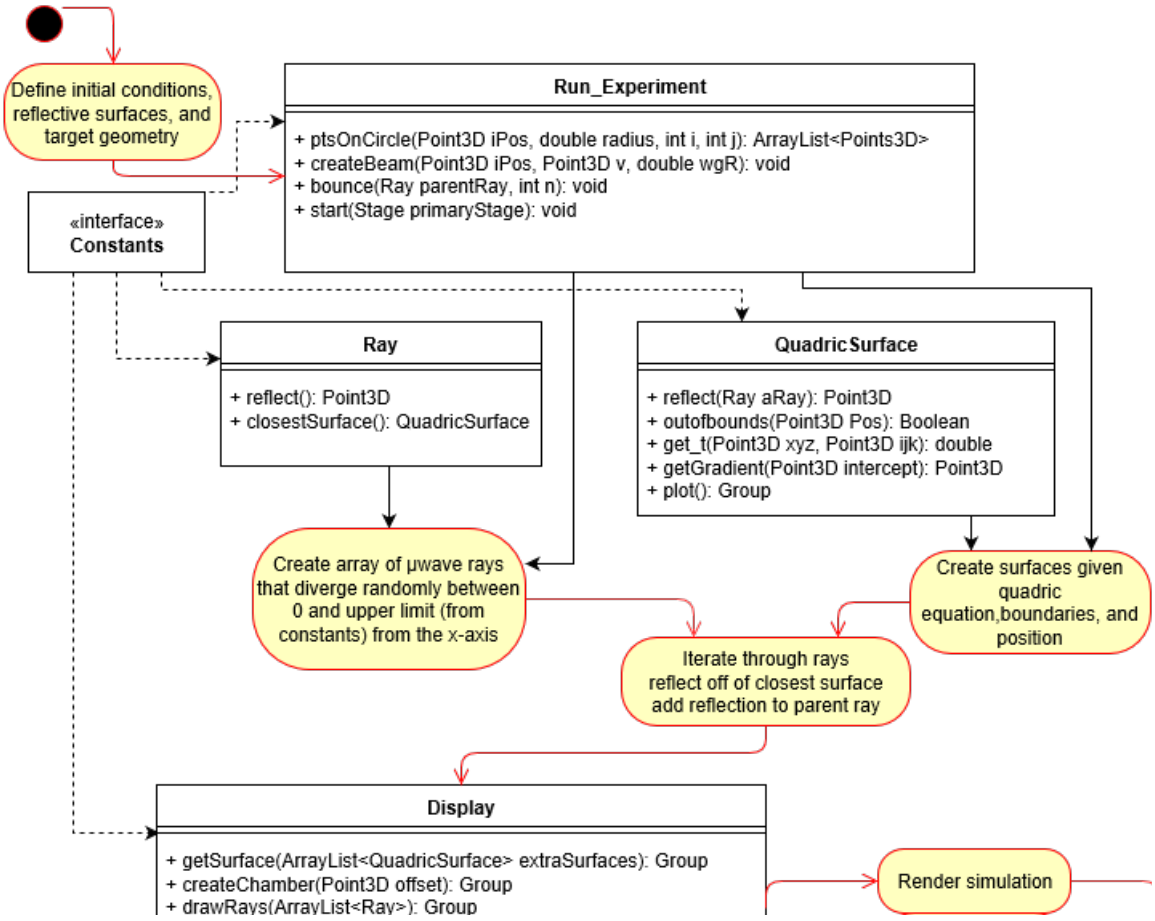


Figure B.3: Flowchart of overall simulation process