Zubin Mishra
UID: 604644805
CEE/MAE M20
June 16, 2017

Final Project

1 Ray Casting

1.1 Introduction

The goal in this problem is to develop a program that performs ray casting on a user generated set of walls in a fully-customizable 2D environment. This is accomplished by calculating 'slider' values for straight walls and performing Newton's method for curved walls to find intersections. In pursuit of this, MATLAB's struct functionality is used to package data for the walls and for the rays. Additionally, MATLAB's line function is used to draw the walls and rays and the norm function is used in the intersection finding procedure. It will be discussed how the program handles intersections at corners, how imperfect reflections affect the results, and how the time required to run the program changes with the number of walls.

1.2 Model and Methods

The script begins with a workaround to create empty arrays of structs. This is needed mainly because the code works for straight and curved walls, and if there are walls of one type but not the other, there still needs to be an array present, or an error will be thrown.

```
w(1) = makeWall(-10, -10, -10, 10, .9, 1);
w(1) = [];
cw(1) = makeCurvedWall(-4, 4, @(x) (16-x.^2), .9, 5);
cw(1) = [];
```

The walls are then constructed and added into the arrays. They are then drawn using the MATLAB line function:

```
for i = 1:length(w)
    line([w(i).end1(1) w(i).end2(1)], [w(i).end1(2) w(i).end2(2)]);
end

for i = 1:length(cw)
    curved_x = linspace(cw(i).end1, cw(i).end2);
    line(curved_x, cw(i).f(curved_x));
end
```

Some constants are then established. A for loop is then entered to allow for more than one user-input source. Within this for loop, the user is asked to input the source coordinates on the graph using MATLAB's ginput function:

```
[sx sy] = ginput(1);
source = [sx sy];
```

An array of starting ray angles is then created, the parameters of which can be changed to allow for focused or directional sources. The script then enters a for loop for each ray. Within this loop, a ray struct is constructed and placed in an array. Initial values for the bouncing calculations are then established, including P3 and P4, where P3 is the source and P4 is a long distance point the ray is pointing to (P4 is also stored initially as the next point when considering how the ray bounces). P4 is found using the following formula, where c is a constant and theta is the angle of the ray:

$$P4 = P3 + c\begin{bmatrix}\cos(\theta) & \sin(\theta)\end{bmatrix}$$

A while loop to carry out the bouncing is then entered. Within the while loop, the script will first check each straight wall for intersections. This is done by first establishing P1 and P2, the wall endpoints, and then generating 'slider' values using the following formulas:

$$s = \frac{(x4 - x3)(y1 - y3) - (y4 - y3)(x1 - x3)}{d}$$

$$t = \frac{(x2 - x1)(y1 - y3) - (y2 - y1)(x1 - x3)}{d}$$

Where,

$$d = (x2 - x1)(y4 - y3) - (y2 - y1)(x4 - x3)$$

s is the slider value for the wall and t is the slider value for the ray. If both slider values range between 0 and 1 inclusive, there is an intersection, given by:

$$T = P1 + s(P2 - P1)$$

In code:

```
if (((s >= 0 && s <= 1) || abs(s) < 10^-10 || abs(s-1) < 10^-10) && ((t >= 0
      && t <= 1) || abs(t) < 10^-10) || abs(t-1) < 10^-10)
            intersection = P1 + s*(P2-P1);
```

If this intersection is the closest one and does not lie on the same wall that the ray previously hit, then it is stored as the next point, and the wall's ID, phi, and lambda values are stored for later. It is also stored that an intersection was found:

```
if norm(intersection - P3) < norm(nextPoint - P3) && w(j).ID ~= hitwall
      nextPoint = intersection;
      wall = w(j).ID;
      phi = w(j).phi;
      lambda = w(j).lambda;
      intersectionFound = true;
```

The script then checks each curved wall for intersections. This is done by first establishing a function handle for the ray line function and a domain minimum and maximum using the x-values of P3 and P4:

```
g = @(x) tan(r(i).theta)*(x - P3(1)) + P3(2);
domain_max = max([P4(1) P3(1)]);
domain_min = min([P4(1) P3(1)]);
```

Then for each curved wall, an array of initial values is created and Newton's method is performed for each. If a vertical line is detected (using mod(theta, pi)) then Newton's method is overridden:

```
if (abs(mod(r(i).theta - pi/2, pi)) < 10^-3 || abs(mod(r(i).theta - pi/2,
      pi)-pi) < 10^-3)
            xr = P3(1);
            inter = 1;
```

If Newton's method detects convergence, the x-value of the resultant intersection is within the domain range of both the ray and the wall, and the y-value is appropriately above or below P3 (based on sin(theta)), then the potential intersection is stored:

```
if (inter == true && xr <= cw(j).end2 && xr >=cw(j).end1 && xr <= domain_max
      && xr >= domain_min && ((sin(r(i).theta) >= 0 && cw(j).f(xr) >=
      P3(2))||(sin(r(i).theta) <= 0 && cw(j).f(xr) <=
      P3(2))||abs(sin(r(i).theta)) < 10^-8))
            intersection = [xr cw(j).f(xr)];
```

If this intersection is the closest one and it is not at the same point as P3 and also does not lie on the same wall that the ray just previously hit (allowing for multiple reflections on the same curved wall), then it is stored as the next point, and the wall's ID, phi, and lambda values are stored for later:

```
if norm(intersection - P3) < norm(nextPoint - P3) && ~(cw(j).ID == hitwall &&
norm(intersection - P3) < 10^-8)
```

It is also stored that an intersection was found. Phi is retrieved by taking the arctangent of the derivative at the intersection (central difference approximation):

```
phi = atan((cw(j).f(xr+10^-5)-cw(j).f(xr-10^-5))/(2*10^-5));
```

After finding the appropriate intersection, the script stores which wall was hit and adds the intersection to the ray's x and y arrays:

```
hitwall = wall;
r(i).x(length(r(i).x)+1) = nextPoint(1);
r(i).y(length(r(i).y)+1) = nextPoint(2);
```

If no intersection was found, the script hits a break statement and continues to the next ray. Otherwise P3, P4, intensity, the next point, number of bouces, and intersection tracking are then updated/reset for the next run of the while loop:

```
P3 = nextPoint;
r(i).theta = -r(i).theta+2*phi;
r(i).intensity = r(i).intensity*lambda;
P4 = P3 + c*[cos(r(i).theta) sin(r(i).theta)];
nextPoint = P4;
bounces = bounces + 1;
intersectionFound = false;
```

After all the rays for a source have been considered, the rays are drawn:

```
for i = 1:numRays
    for j = 1:length(r(i).x)-1
      line([r(i).x(j) r(i).x(j+1)], [r(i).y(j) r(i).y(j+1)], 'Color', 'red');
      drawnow;
    end
end
```

The process repeats for each source the user inputs.

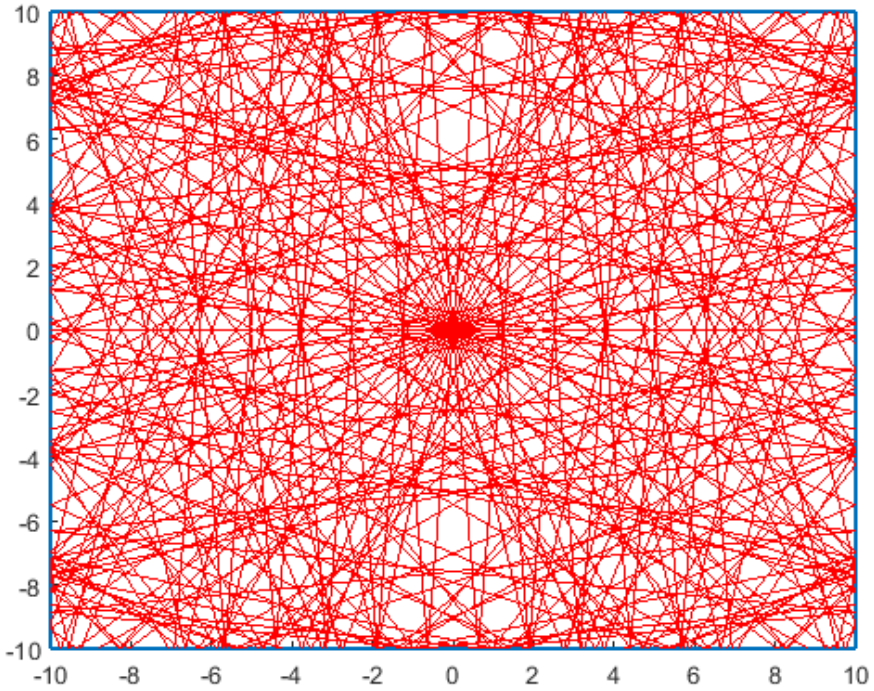
Four functions are used in this code: makeWall, makeCurvedWall, makeRay, and NewtonsI. makeWall, makeCurvedWall, and makeRay all work very similarly. makeWall takes in two x-coordinates, two y-coordinates, a lambda value, and an ID value. It creates a struct that contains two endpoints, the wall angle, the wall lambda value, and the wall ID. makeCurvedWall takes in two x-coordinates, a function handle, a lambda value, and an ID value and creates a struct that holds all of them. makeRay takes in an x-coordinate, a y-coordinate, an angle, and an intensity and creates a struct that holds all of them. The body of makeWall is shown below to illustrate the basic structure of these functions:

```
function [ W ] = makeWall( x1, y1, x2, y2, lambda, ID )
      W = struct('end1', [x1 y1], 'end2', [x2 y2], 'phi', atan((y2-y1)/(x2-
            x1)), 'lambda', lambda, 'ID', ID);
end
```

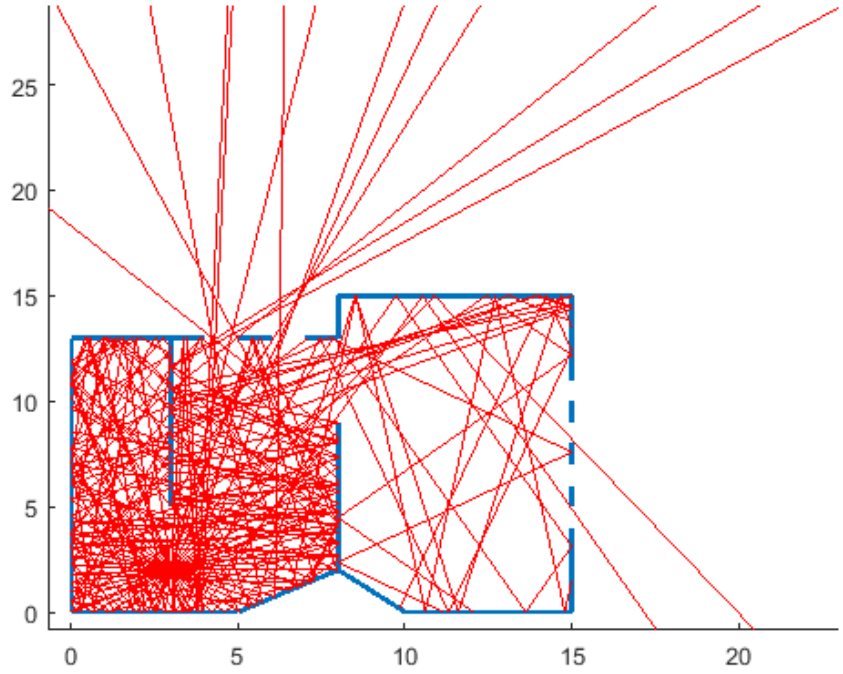
NewtonsI functions very similarly to the Newton's method function from homework 8, but takes in 2 function handles and performs Newton's method on their difference, using the central difference approximation.

1.3 Results and Calculations

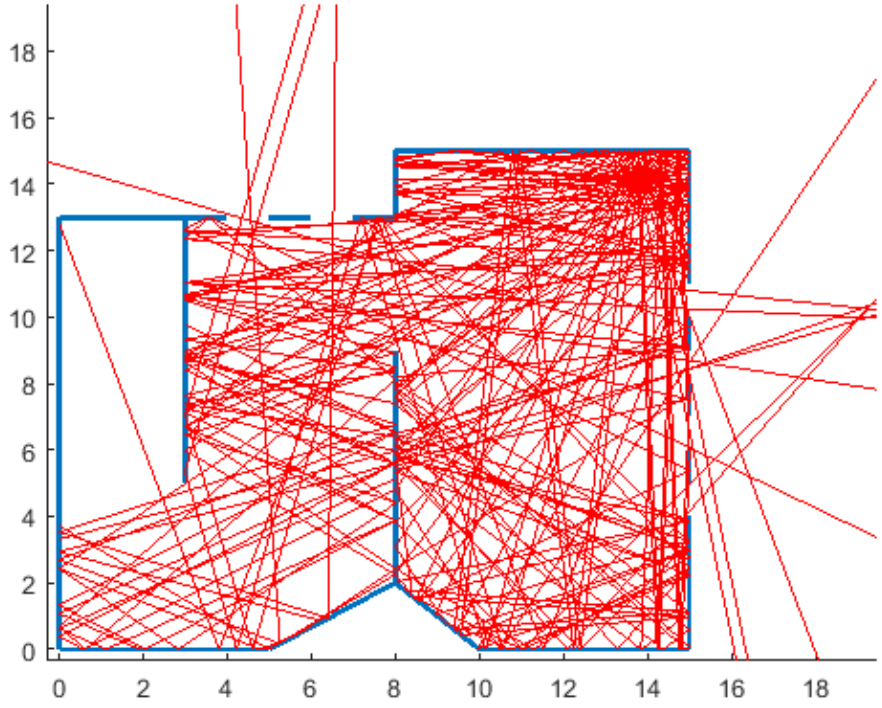
Simple case, 360° point source: runtime = 7.38 seconds



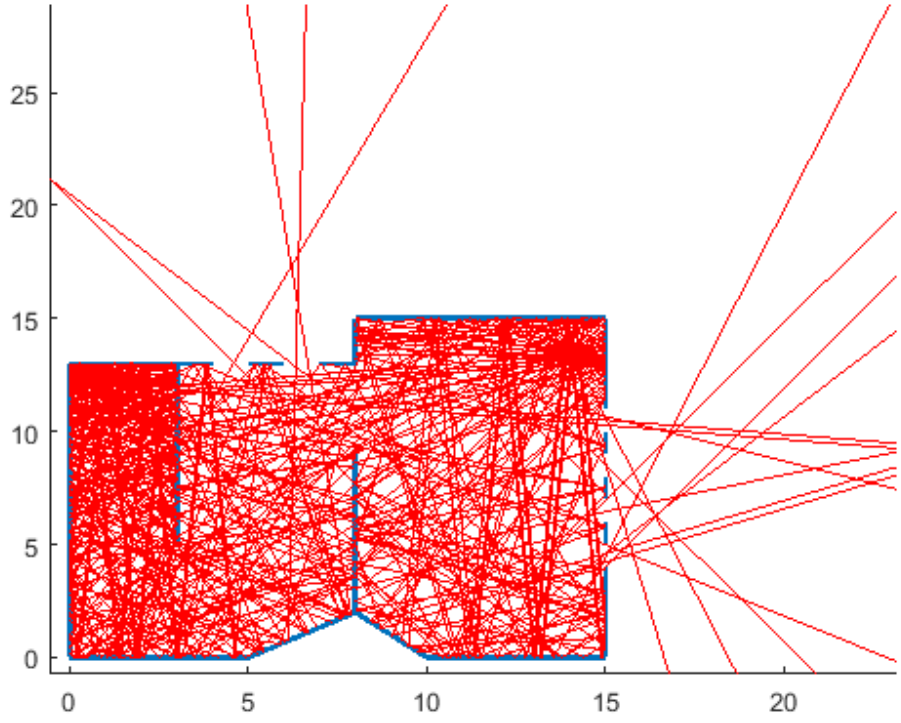
Complex case 1, 360° point source: runtime = 5.42 seconds



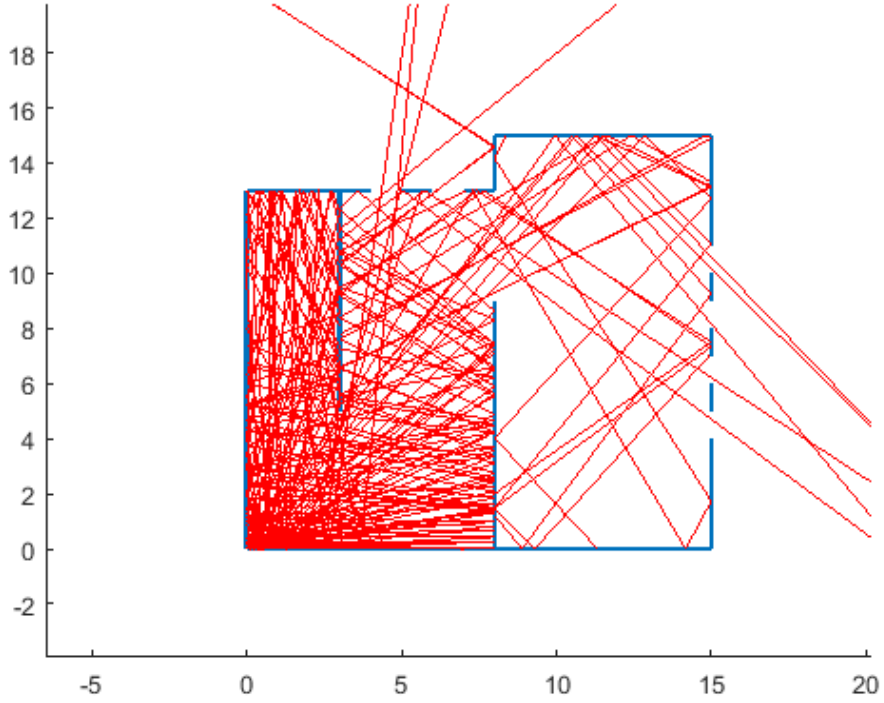
Complex case 2, 360° point source:



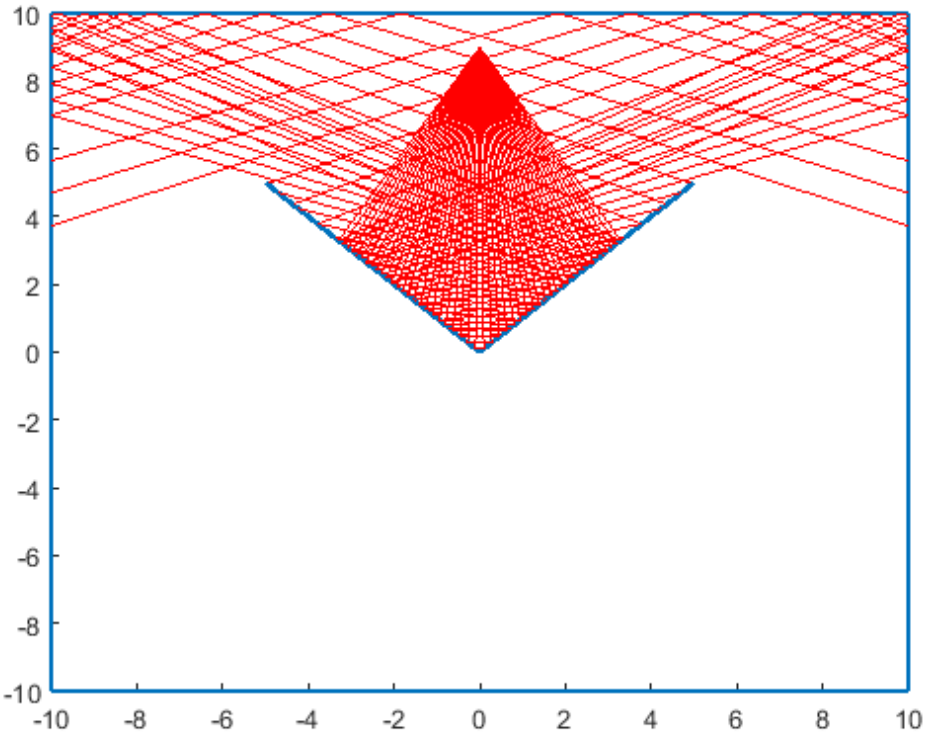
Complex case 3, two 360° point sources:



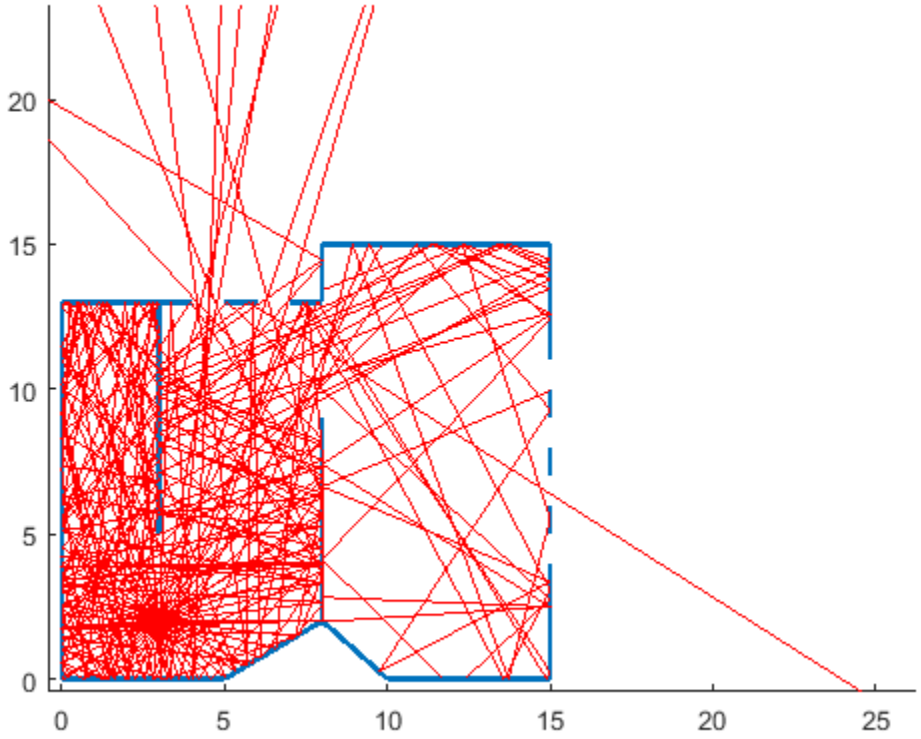
Complex case with focused source pointing into a wall:



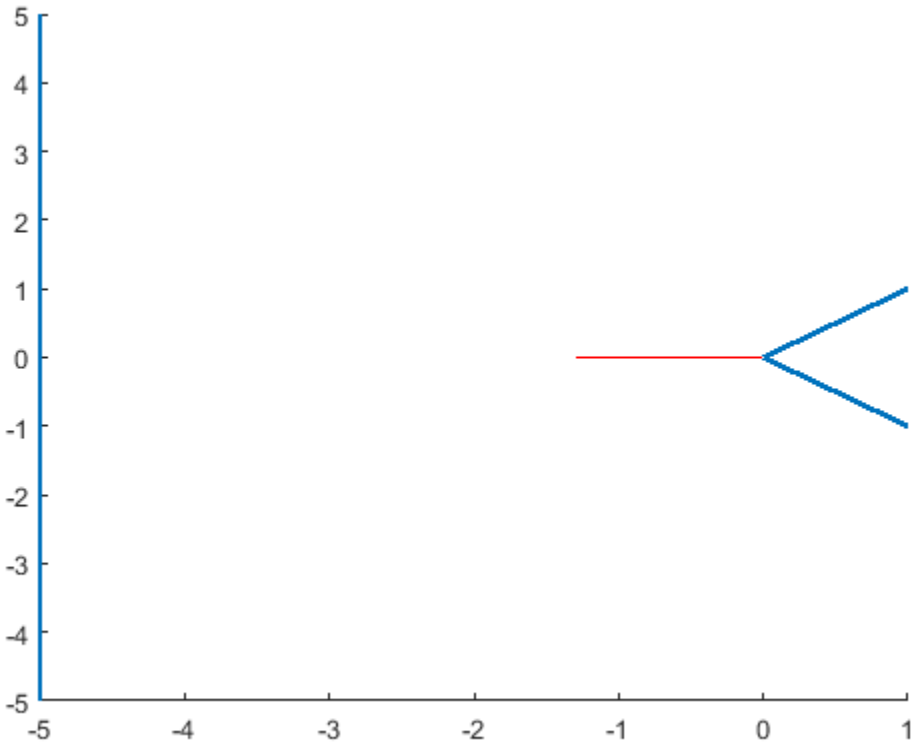
Focused source pointing at absorbing walls (lambda = .125):

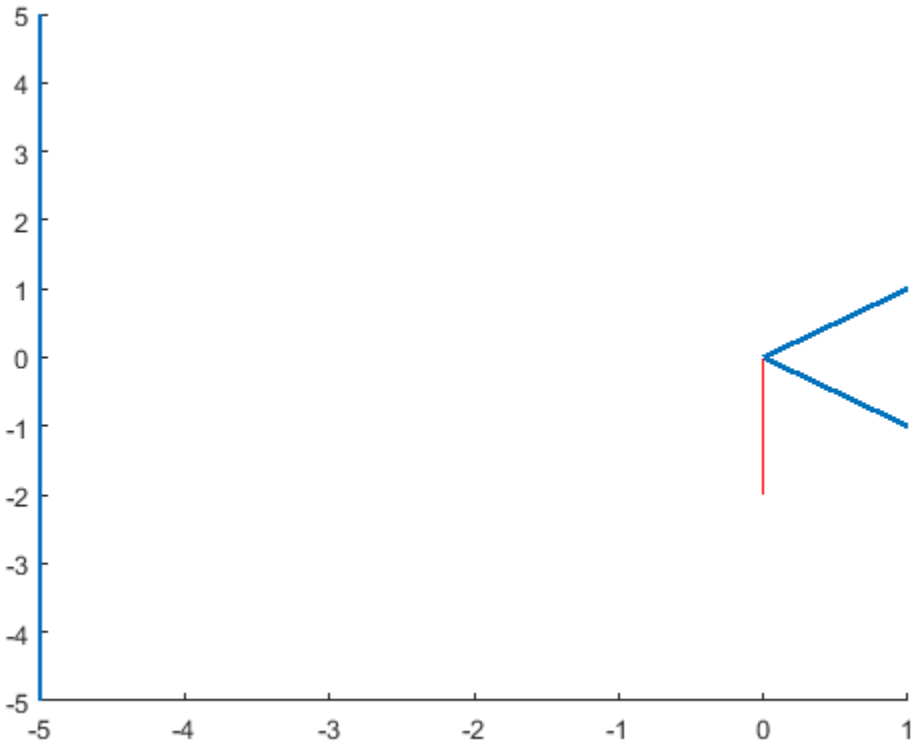


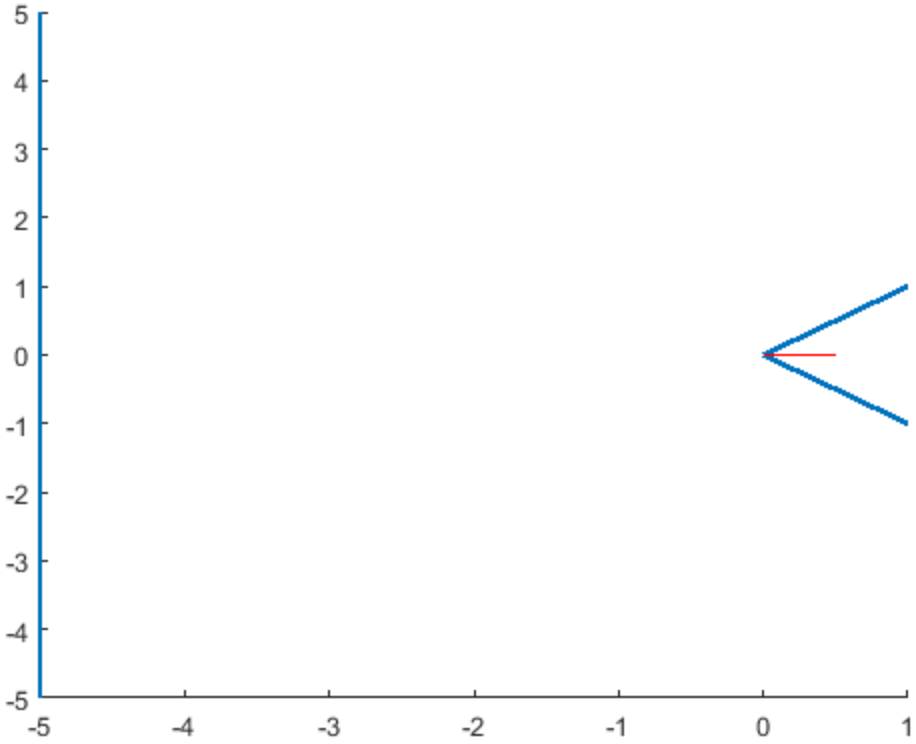
Small random perturbation added (0.1*rand) to model imperfect reflections for complex case 1:

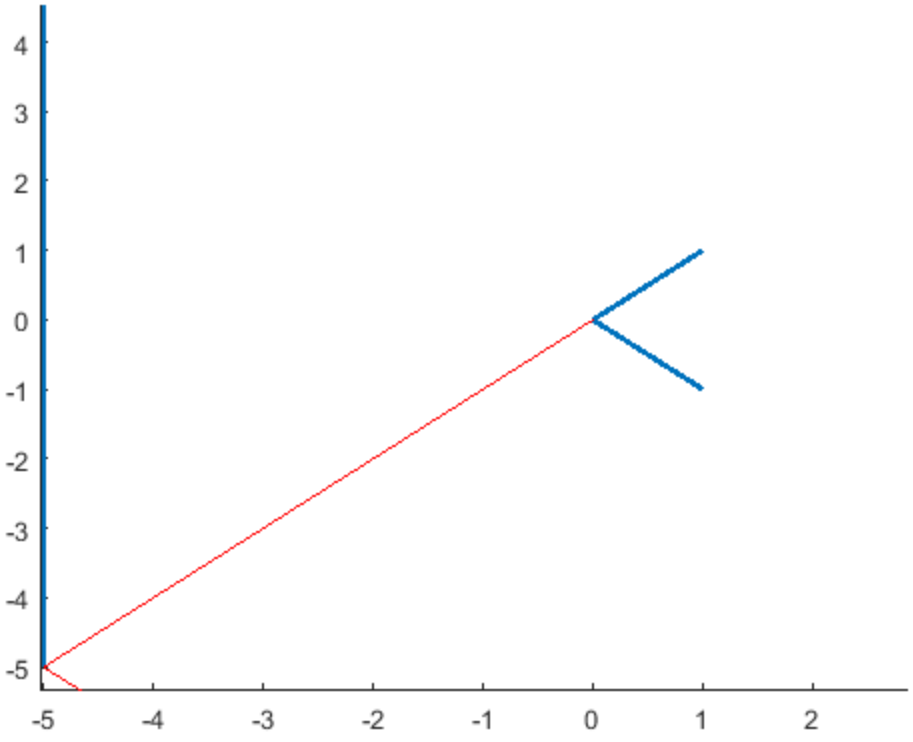


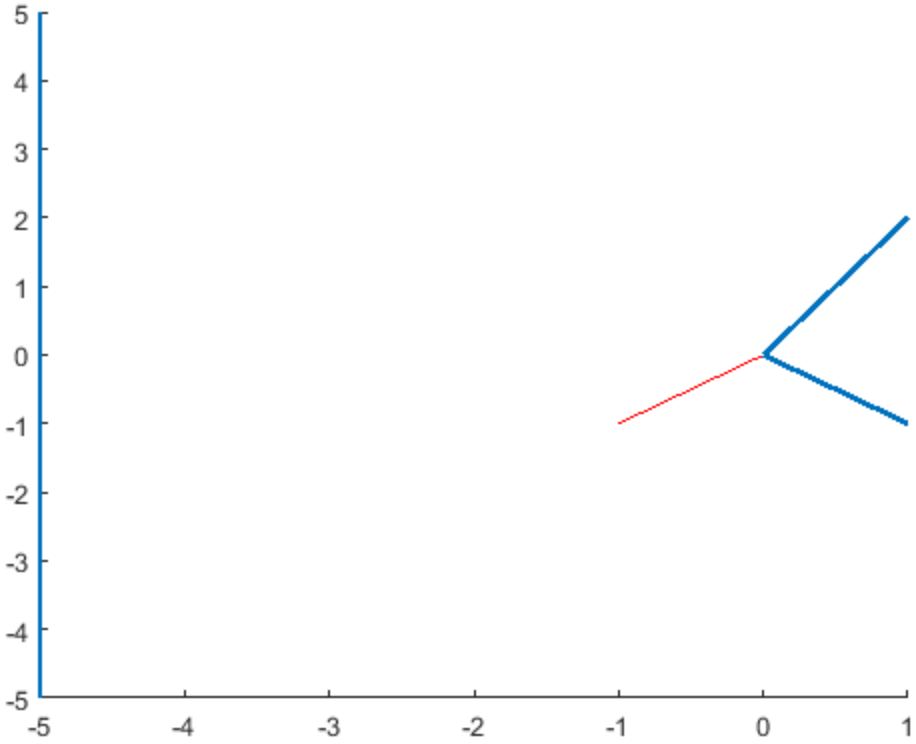
Corner tests:

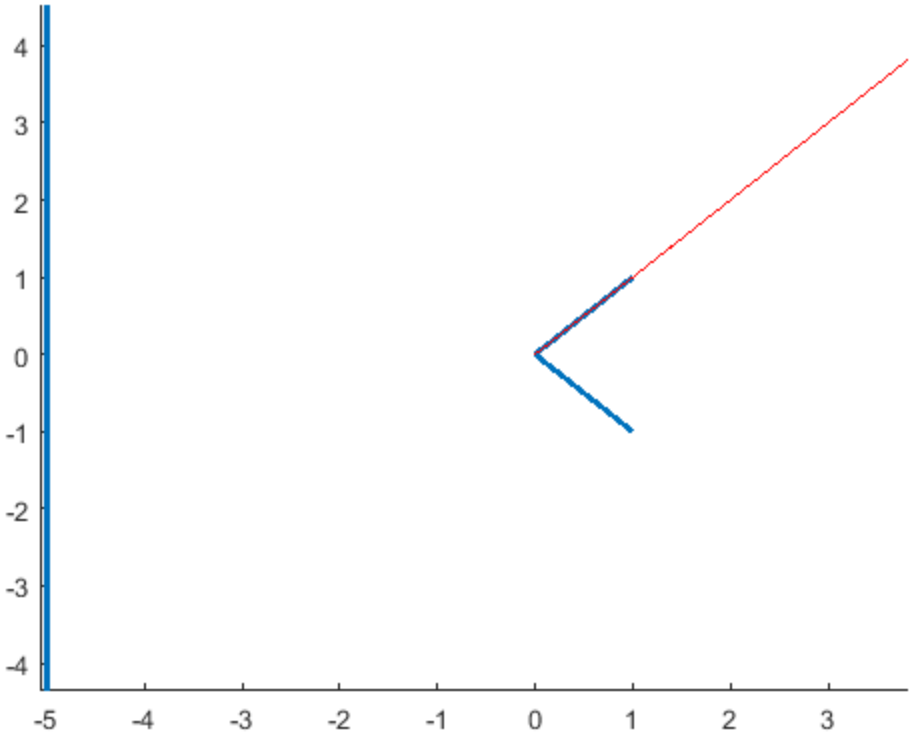




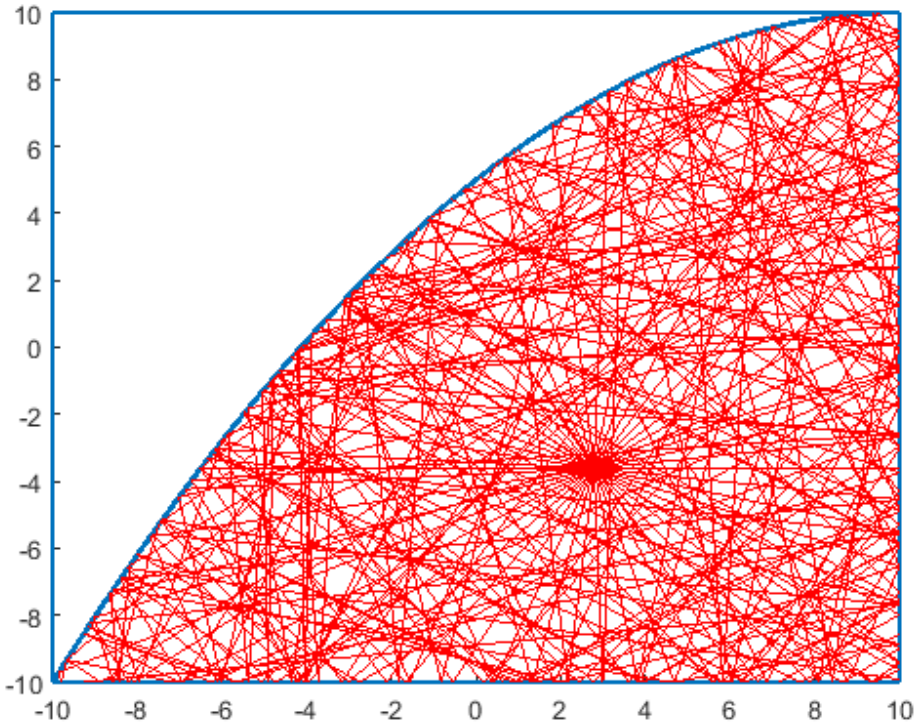

5
4
3
2
1
0
-1
-2
-3
-4
-5
-5
-4
-3
-2
-1
0
1


4
3
2
1
0
-1
-2
-3
-4
-5
-5
-4
-3
-2
-1
0
1
2

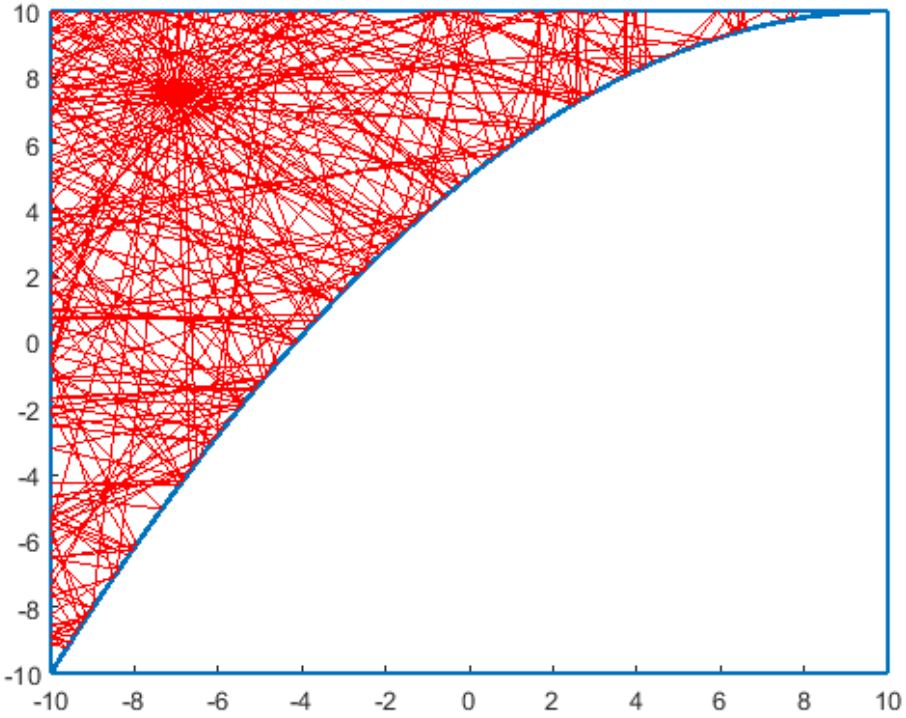

5
4
3
2
1
0
-1
-2
-3
-4
-5
-5
-4
-3
-2
-1
0
1


4
3
2
1
0
-1
-2
-3
-4
-5
-4
-3
-2
-1
0
1
2
3

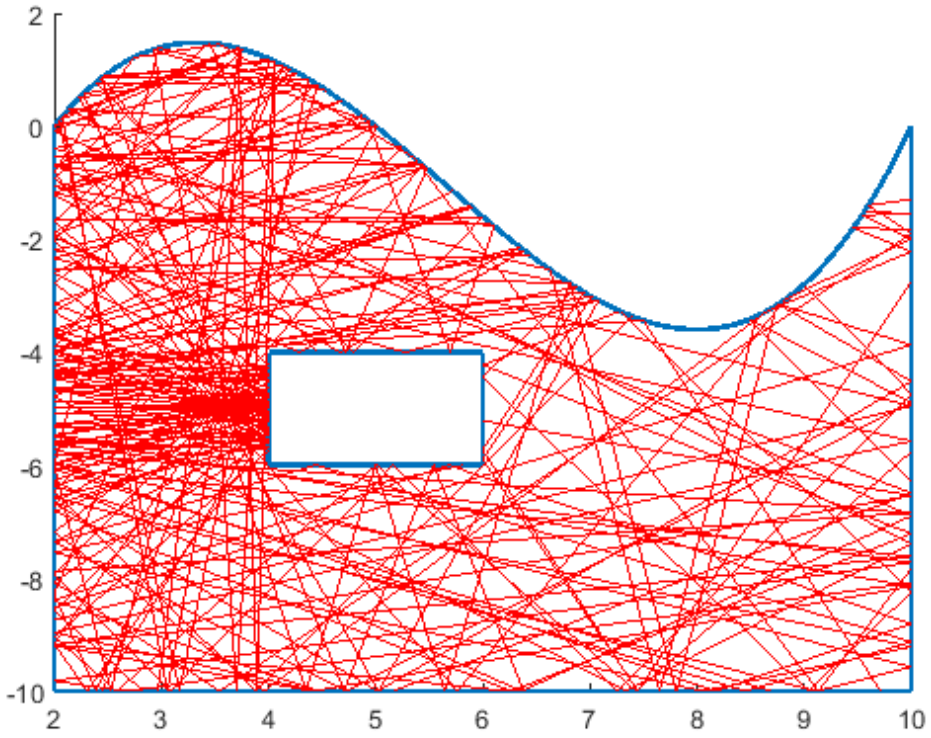
Quadratic curved wall case 1 (concave wall):



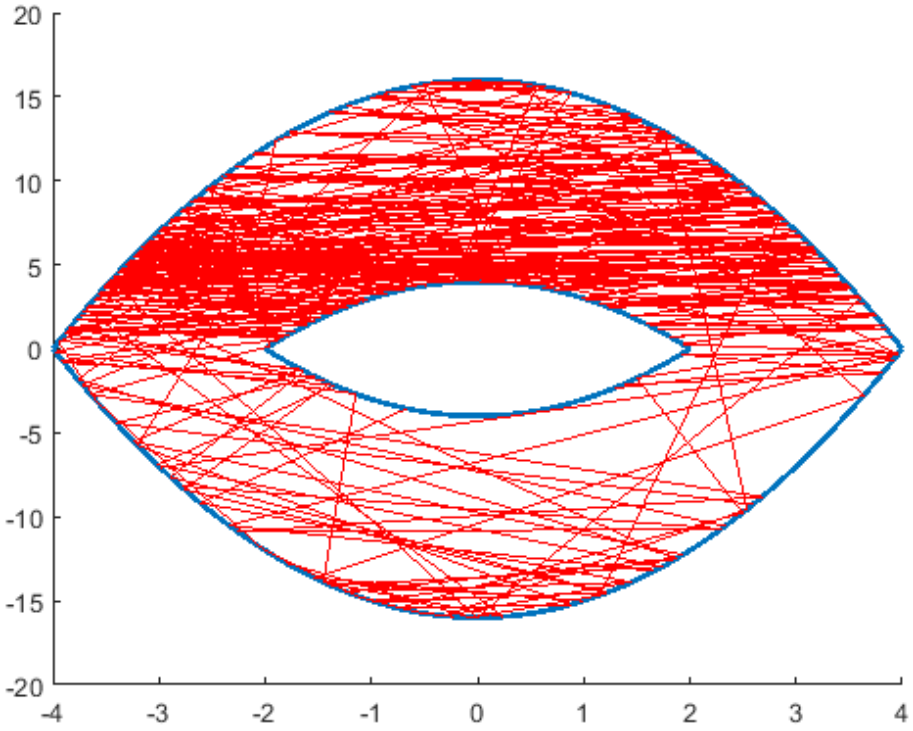
Quadratic curved wall case 2 (convex wall):



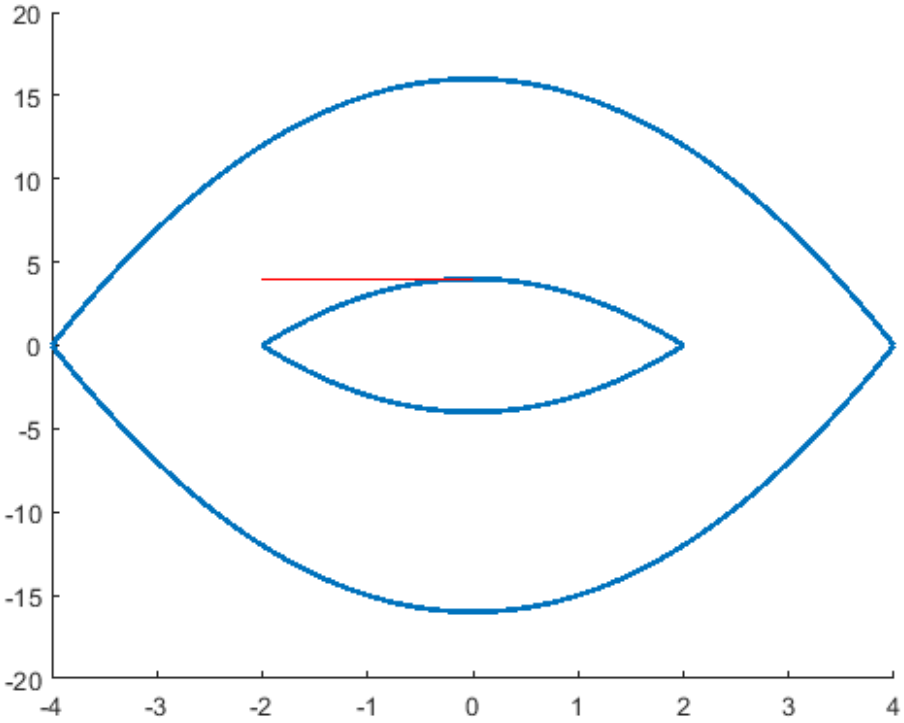
Cubic curved wall:



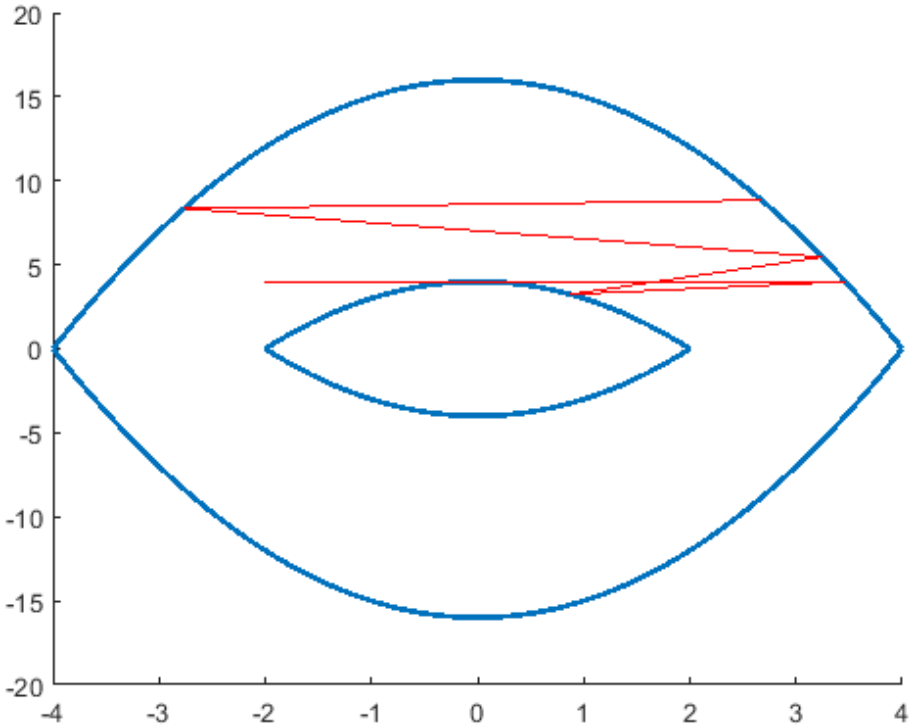
Only curved walls: runtime = 16.72 seconds



Curved wall tangent ray with low same-point tolerance:



Curved wall tangent ray with high same-point tolerance:



1.4 Discussion

First, program behavior when encountering a corner will be discussed. Typically, when a ray exactly hits a corner, the ray will stop there. Every intersection found after hitting the corner will be at the corner. The ray will first consider itself to have bounced off of whichever wall came earlier in the wall array. The ray will then no longer consider that wall, but there is another wall at the exact same point, so it will find that other wall in the corner instead. The ray will continue to find intersections like this at the exact same point (the corner) until the program continues to the next ray. This is what is happening in the 1st, 2nd, 3rd, and 5th cases shown in the results section above. In the 4th and 6th cases, the ray is parallel to one of the walls that makes the corner. This is a special case because the way intersections are found, parallel walls and rays are considered to have no intersections. As such, the ray simply bounces off of the wall that it is not parallel to. This can result in some odd behavior, such as the ray travelling directly through a wall, as is shown in the 6th case.

Second, overall coverage of rays when adding imperfect reflections will be discussed. When looking at the two graphs in the results section above, it can be seen that the coverages are not exactly the same, but are still very similar. This falls in line with the idea that, while the real-world physics is very complicated, this ray-casting model can provide a fairly accurate approximation.

Third, the program runtime will be discussed. Surprisingly, the addition of more walls did not result in a slower runtime than the simple source-in-a-box case (5.42 seconds vs 7.38 seconds). This is likely because the bulk of the program runtime is spent on drawing the rays, and in the simple case, every ray bounces 5 times, while in the complex case, some rays end up going out the windows, terminating their bouncing, and resulting in fewer lines to draw. One way that checking a ray against every wall could be avoided is by considering the angle that the ray is pointing. Using sine and cosine, it can quickly be found out if the ray is pointing up, down, left, right (I use this idea to some extent for finding curved wall intersections). Using this information, some walls can instantly be removed from consideration. For example, if it is known that a ray is pointing right and up, then if a wall has both endpoints' x-coordinates to the left of P3, it can be ignored. Similarly, if a wall has both endpoints' y-coordinates below P3, it can be ignored.

Fourth, a general observation about the way the program runs: the scaling value, c, that determines P4 in the code needs to be changed to suit the size of the environment being considered. Sometimes a value as low as c=5 is appropriate, but other times, a value as large as c=200 might be needed. This should be considered before running the script.

Finally, a brief discussion on handling curved walls. In order for the program to properly run, tolerances needed to be introduced in several areas, such as when testing for vertical or horizontal lines and when testing whether an intersection was, in fact, a new point or not. This same-point tolerance mainly affected how rays bounced off of convex surfaces. It also affected how rays tangent to the curved wall acted. This is shown in the results section above. With a low tolerance, rays hit the tangent point and stopped there. With a high tolerance, rays continued past the tangent point. Similar issues were run into with other convex surfaces, such as in the second

quadratic curved wall case. With too low of a tolerance, reflections could not be calculated properly.