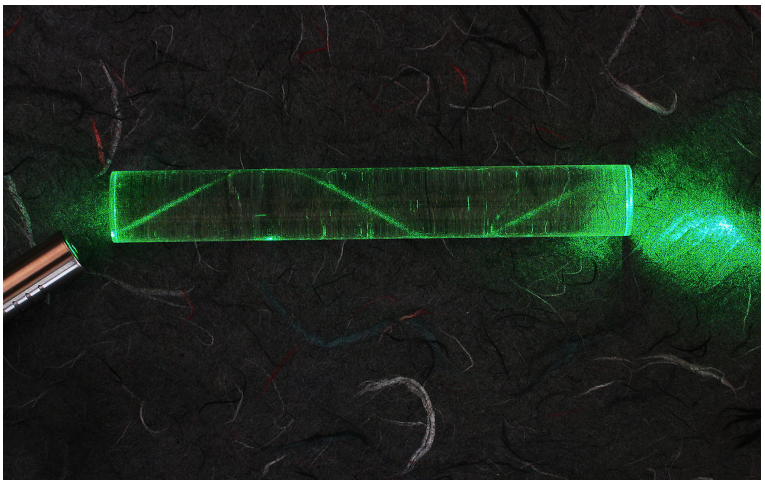
Numerical Ray Tracing

4.1 Introduction

The aim of ray tracing it to determine the geometric path of light through a system. Ray tracing arises in many contexts including optical instrument design, astronomy, and computer graphics, it is also important in areas like seismology where the rays represent the paths of seismic waves rather than light.

All but the most trivial ray tracing problems are intractable analytically and require numerical treatment. In this lab we will develop a program to perform ray tracing in two dimensions through a medium with a variable refractive index. We will first apply our code to a test case to verify its correctness. We will then use it to study total internal reflection, as shown below in the image of light from a laser travelling along a perspex rod.



To develop your numerical ray tracing program you will have to use all the programming skills you have learnt so far, putting them together to create a sophisticated program. We'll also learn a few more control structures to complete your understanding of basic MATLAB functionality: `while` loops and `break` statements.

Taking an abstract idea like an algorithm and realising it in code can be a formidable task. It is always simpler to develop a code in stages. That way we only need to think about a small part of the problem at any one time, and it also helps us debug the code.

4.2 Lab objectives

By the end of this lab sessions you should be able to:

1. Write code using `while` loops.
2. Solve problems using `for` loops or `while` loops.
3. Exit from loops using the `break` statement.
4. Combine different programming control structures to write a program.

If you still don't know how to do any of these things once you have completed the lab, please ask your tutor for help. You should also use these lists of objectives when you are revising for the end of semester exam.

4.3 Walkthrough: While loops and breaking

In Lab 2 we used `for` loops to iterate over a range of values, repeating a series of commands each time. `for` loops are good when you want to run for a fixed number of iterations. However, sometimes you want to calculate something for which you don't know the number of iterations in advance. A `while` loop lets you continue to iterate while some condition remains true.

For example, say you want to work out how many consecutive integers need to be summed before the total exceeds 100. If you start by setting a variable `total = 0` we can then iterate until `total` is equal to 100:

```
1 >> x = 0;
2 >> total = 0;
3 >> while total <= 100
4 >>   x = x + 1;
5 >>   total = total + x;
6 >> end
7 >> disp([x total]);
8     14   105
```

As the output shows, you need to sum all the integers from 1 to 14 before their sum exceeds 100.

A while loop repeats the following steps:

1. Evaluate the condition expression (in this case `total <= 100`)
2. If the condition is true, run the body (in this case, lines 4 and 5)
3. If the condition is false, jump to the statement after the body (`disp([x total])`)

4.3.1 While loops and for loops

Anything that can done with a `for` loop, can also be done with a `while` loop. For example, to print the numbers from 1 to 5 we could use either loop. A `for` loop would look like this:

```
1 >> for x = 1:5
2 >>   disp(x);
3 >> end
```

and a `while` loop to do the same thing would look like this:

```
1 >> x = 1;
2 >> while x < 6
3 >>   disp(x);
4 >>   x = x + 1;
5 >> end
```

In each iteration the *loop counter*, `x` has to be incremented. If you don't increment it you will end up with an **infinite loop**. Try removing line 4 (`x = x + 1`) from the code snippet above and rerunning it. This loop will continue running forever. Press CTRL-C while the cursor is in the command window to terminate the program and regain control of the command prompt. If this fails, then kill MATLAB, or kill the computer!

4.3.2 The break statement

In the example above, both code snippets result in exactly the same output, but the `for` loop is more convenient. Other examples lend themselves more naturally to a `while` loop. For example, say we want to print out the square of all integers where the square is less than 100. To do this with a `while` loop:

```
1 >> x = 1;
2 >> while x^2 < 100
3 >>   disp(x^2);
4 >>   x = x + 1;
5 >> end
6      1
7      4
8 ...
9     64
```

```
    81
```

To do this using a `for` loop we need to introduce a new concept: breaking out of a loop. Not surprisingly, this is done with the `break` command. The `break` command is usually used inside an `if` statement that checks whether some condition is true:

```
>> for x = 1:1000
>>   if x^2 >= 100
>>     break
>>   end
>>   disp(x^2);
>> end
     1
     4
...
    64
    81
```

In this example we start by looping between 1 and 1000. In each iteration, the value of x^2 is calculated and `if` it is greater than or equal to 100 we break out of the loop. If you remove lines 2 and 3, then the program will print out the squares of all the integers up to 1000 instead.

It is good practice to indent your code so that it is easier to read. Notice how in the example above the `if` statement block is indented inside the `for` loop block, which makes it easier to see the *flow of control* in the program. The MATLAB editor has a smart indenting facility that should automatically indent code inside loops.

4.3.3 Multiple conditions

Sometimes you need a `while` loop or `if` statement that depends on multiple conditions. For example, if you want to check whether a number is within a given range $-1 < x < 1$ you need to check two conditions. One way of doing this is to *nest* `if` statements inside each other:

```
>> x = -0.5;
>> if x > -1
>>   if x < 1
>>     disp('The number is in range');
>>   end
>> end
```

Try changing the value of `x` to something outside the range to check this code works.

A more convenient way is to combine the two conditions using the **and** operator `&&`:

```
>> x = -0.5;
>> if x > -1 && x < 1
>>   disp('The number is in range');
>> end
```

There is also an **or** operator, represented by the double pipe symbol `||`:

```
>> x = -0.5;
>> if x <= -1 || x >= 1
>>   disp('The number is out of range');
>> end
```

These operators work exactly the same way in a `while` loop. For example, to print out the value of the square numbers until either the value is greater than 100 or we have run 5 iterations of the loop:

```
>> x = 1;
>> count = 0;
>> while x^2 < 100 && count < 5
>>   disp(x^2);
>>   x = x + 1;
>>   count = count + 1;
>> end
```

4.4 Exercises

Question 1
Write a program that prints out all the multiples of 12 that are less than 200.

Hint: Be careful of the end case.

```
x = 1;
mult = x * 12;
while mult < 200
    disp(mult)
    x = x + 1;
    mult = x * 12;
end
```

Tutor note: check the first and last values are correct. Explain how the order of commands inside the loop affects the results.

Question 2
If you used a `for` loop in Question 1, rewrite your program with a `while` loop and check it gives the same results. If you used a `while` loop, rewrite it using a `for` loop.

```
for x = 1:100
    mult = x * 12;
    if mult >= 200
        break;
    end
    disp(mult);
end
```

Question 3
Consider the sum

$$s = 1^3 + 2^3 + 3^3 + ...$$

write a program to work out how many terms you need to add, so that the total exceeds 1000 ($s > 1000$). Write the number and the sum below.

```
sum1 = 0;
x = 0;
while sum1 < 1000
    disp([x sum1]);
    x = x + 1;
    sum1 = sum1 + x^3;
end
```

7 numbers gives a sum of 784
8 numbers gives a sum of 1296
Tutor note: check the end condition — a common error is incrementing x one extra time.

4.5 Walkthrough: Relational operators

When you want to compare two (or more) things you need to use a *relational operator*. These operators, such as less than (**<**) or equal to (**==**) return 1 if the comparison is true, and 0 if the comparison is false. For example:

```
>> 10 >= 5
ans =     1
>> 3 > 5
ans =     0
```

The operators we will use is given in the table below.

==	~=	<	>	>=	<=
Equal to	Not equal to	Less than	Greater than	Greater than or equal to	Less than or equal to

You can apply these operators to vectors to find elements that match particular criteria. Rather than a single 1 or 0 being returned, a vector of 1s and 0s is returned. For example:

```
>> a = [1 -3 2 -4 5 7];
>> a > 0
ans =     1     0     1     0     1     1
```

In this case, the first element of **a** is greater than 0, so the first element of the output vector is 1 (**true**). The second element of **a** is less than 0, so the second element of the output vector is 0 (**false**), and so on.

MATLAB has a nifty way of getting the values in an array that satisfy a certain condition: you can pass this output vector back in to the original vector like this:

```
>> a(a>0)
ans =     1     2     5     7
```

The new output vector contains only the positive elements of **a**. This trick uses the fact that you can pass one vector into another one to return the specified elements. For example, if we want to get the 2nd and 4th elements of **a** we could do this:

```
>> a = [1 -3 2 -4 5 7];
>> b = [2 4]
>> a(b)
ans =    -3    -4
```

Comparing vectors

In the above examples we have compared a vector to a scalar value. You can also compare two vectors in an element-by-element fashion. For example:

```
>> a = [2 4 5 1];
>> b = [1 6 2 7];
>> a > b
ans =     1     0     1     0
```

The output vector has a value of 1 (**true**) for each element of **a** that is greater than the corresponding element of **b**, and a value of 0 (**false**) otherwise. In other words 2 is greater than 1, so the first element of the output vector is 1. However, 4 is less than 6, so the second element of the output vector is 0.

As before, you can take this output vector and pass it in to one of the original vectors to get all the values that satisfy the condition. For example, to *find all the values of a that are greater than the corresponding value of b*:

```
>> a(a > b)
ans =     2     5
```

These operators are exactly the same as the ones you have used in *if statements*. For example:

```
x = 2
if x > 0
  disp(x)
end
```

4.6 Exercises

Question 4
Given the two vectors `a` and `b` below:

```
a = [1 -2 3 -4 5 -6 7 8];
b = [-40 -30 -20 -10 0 10 20 30];
```

Write MATLAB code to

1. Print out the elements of `a` that are less than 0.
2. Print out the elements of `b` that are greater than or equal to 3.
3. Print out the elements of `b` that are greater than the corresponding value of `a`.
4. Print out the number of elements of `a` that are positive.

Write your code in the box below.

```
a(a<0)
b(b>=3)
b(b>a)
length(a(a>0)) or sum(a>0)
```

Tutor note: emphasise the need for general solutions, not hard coding the answer for the particular case. i.e. if the content of vector `a` *changes their solution should still work.*

Question 5
Given the two vectors `a` and `b` defined in Question 4, write MATLAB code to:

1. Create a new vector `c` that consists of the negative values of `a`.
2. Create a new vector `c` that consists of the negative values of `a` and `b`.
3. Print out the first positive element of `a`.
4. Print out the last positive element of `a`.

Write your code in the box below.

```
c = a(a<0)
c = [a(a<0) b(b<0)]
c = a(a>0); c(1)
c = a(a>0); c(end)
```

Checkpoint 1:

4.7 Activity: Tracing a single straight ray

The goal of ray tracing is determine the path of a ray given an initial starting point and direction of the ray. We will write our ray tracing program in three stages, progressively implementing more advanced features. In this section we will implement the first stage: a program that draws the paths of straight rays.

The coordinates of the starting position of the ray can be denoted as $\mathbf{r}_0 = (x_0, y_0)$. In vector form, this is

$$\mathbf{r}_0 = x_0\hat{\mathbf{x}} + y_0\hat{\mathbf{y}}, \tag{4.1}$$

We can represent this real world vector as a MATLAB vector:

```
r0 = [x0, y0];
```

The first element is the initial x-position **`x0`**, and the second element is the initial y-position **`y0`**.

We will assume that the light is initially travelling in a direction that is inclined by θ_0 degrees from the horizontal. In vector notation, this is

$$\hat{\mathbf{s}}_0 = \cos(\theta_0)\hat{\mathbf{x}} + \sin(\theta_0)\hat{\mathbf{y}}, \tag{4.2}$$

Note that this vector has unit length.

Question 6
Implement Equation 4.2 as a MATLAB vector. Test it works using an angle of $\theta_0 = 45°$.

```
theta0 = 45;
s0 = [cosd(theta0), sind(theta0)];
```

Each component should be equal to $1/\sqrt{2}$.
Tutor note: It is critical that students understand the relationship between 'physics' vectors and MATLAB vectors.

4.7.1 An iterative model

In iterative models you compute the current parameters by taking small steps based on the parameters at the previous step. In this case we will update the position and direction of the ray by taking a small step in the direction $\hat{\mathbf{s}}_0$.

If our step-size if h, then the new position $\mathbf{r}_1$ is given by the vector addition

$$\mathbf{r}_1 = \mathbf{r}_0 + h\hat{\mathbf{s}}_0. \tag{4.3}$$

This vector sum is demonstrated in Figure 4.1.

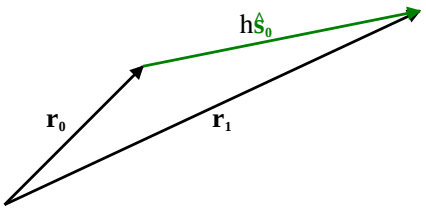


Figure 4.1: Illustration of one step of the position update.

Assuming the light continues to travel in the same direction, we can compute the next step $\mathbf{r}_2$ from $\mathbf{r}_1$ in the same way as we computed $\mathbf{r}_1$ from $\mathbf{r}_0$. The general expression for $\mathbf{r}_k$ is then

$$\mathbf{r}_{k+1} = \mathbf{r}_k + h\hat{\mathbf{s}}_0. \tag{4.4}$$

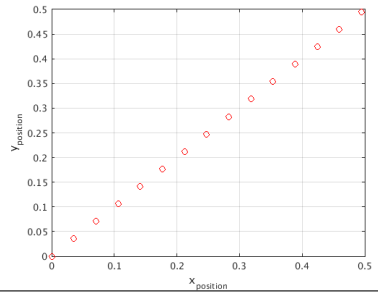
This is an example of an iterative method, since the new value is generated from previous values. After k applications of the method we expect to have travelled a distance of kh.

Question 7
Write a program that traces a ray according to Equation 4.4. The initial position and angle should be $\mathbf{r}_0 = (0, 0)$ and $\theta_0 = 45°$; s_0 is given by equation 4.4. Use a step size $h = 0.05$.

To do this, use a loop that iterates for 30 steps, plotting the position of your ray each time. You can use the same trick we used in Lab 2 Question 9 to plot in the loop with a `pause(0.2)` between each point so you can see it:

```
x=0;
y=0;
r=[x,y];%set the intial position.
for count = 1:30
  plot(r(1), r(2), 'ro');
  hold on
  pause(0.2);
  % Add your code to update the position here
end
```

In this example, `r` is a vector containing the x and y components of position. Sketch your final plot below.



Question 8
Our simulation would be better if, rather than running for a set number of iterations, it ran until the ray left some predefined region. Imagine we have a box that spans the range $0 \leq x \leq L_x$ and $0 \leq y \leq L_y$) where $L_x = L_y = 1$.

Make a copy of your program and modify it so that it uses a `while` loop instead of a `for` loop. Your while loop should have multiple conditions that means the ray keeps moving until it leaves the simulation box:

- $y > 0$: ray leaves bottom of box
- $y < Ly$: ray leaves top of box
- $x > 0$: ray leaves right side of box
- $x < Lx$: ray leaves left side of box

You should set the axes of your plot to be slightly bigger than the box so you can check if your loop is working correctly. For example, to have your axes extend 10% beyond the edge of the box you could use `axis([0, 1.1*Lx, 0, 1.1*Ly])`.

Run your program and make sure you understand the output. If you don't discuss it with your tutor before continuing. Add the changes in the box below.

Tutor note: Make sure Question 8 is working with the correct while loop conditions.

4.8 Activity: Ray tracing with refraction

In this section we will extend our program to model the refraction of light.

Refraction is the change in direction of light due variations in the refractive index. The simplest example is when the change occurs at a sharp boundary, such as the interface between air and glass. In this case a ray follows a piece-wise linear path. In general rays will follow curved paths in media where the refractive index varies continuously with position.

To model the refraction of light we have to update the direction vector $\hat{\mathbf{s}}$ after each step. We will derive a simple method for computing the new direction based on Snell's law for refraction. For simplicity our method assumed that the refractive index only varies with height: $n(\mathbf{r}) = n(y)$.

Assume we are at a point $\mathbf{r}_1$ and the direction of the ray at this point is $\mathbf{s}_1$. The new position is then

$$\mathbf{r}_2 = \mathbf{r}_1 + h\hat{\mathbf{s}}_1. \tag{4.5}$$

Our goal is to compute an update equation for $\hat{\mathbf{s}}_2$, the direction vector at the new position. To do this will treat n as if it were comprised of thin layers of constant n. Figure 4.2 shows two layers; the lower layer labelled 1 has refractive index $n_1 = n(\mathbf{r}_1)$ and the upper layer labelled 2 has refractive index $n_2 = n(\mathbf{r}_2)$. The ray starts at $\mathbf{r}_1$, travels a distance h in direction $\hat{\mathbf{s}}_1$ and encounters the boundary at $\mathbf{r}_2$. We apply Snell's law to calculate the direction of the ray $\hat{\mathbf{s}}_2$ in the upper layer.

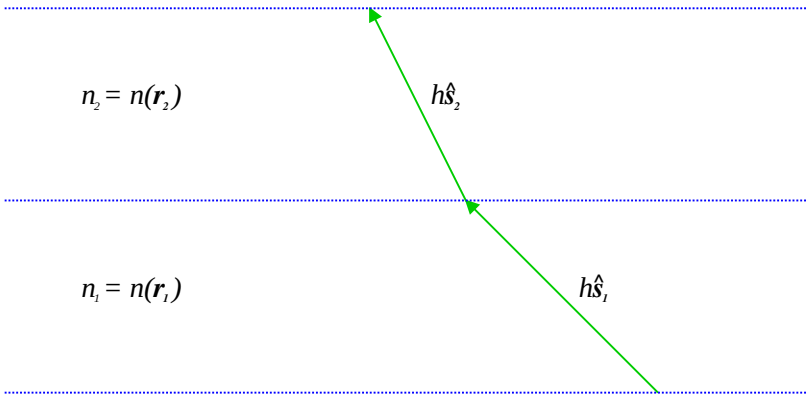


Figure 4.2: Diagram illustrating the position and direction update steps.

The left panel of Figure 4.3 shows the direction vectors of a ray travelling in the positive y direction. We can use basic trigonometry to relate the angle of incidence θ_1 to the x component of $\hat{\mathbf{s}}_1$. The vector $\hat{\mathbf{s}}_1$ has unit length and

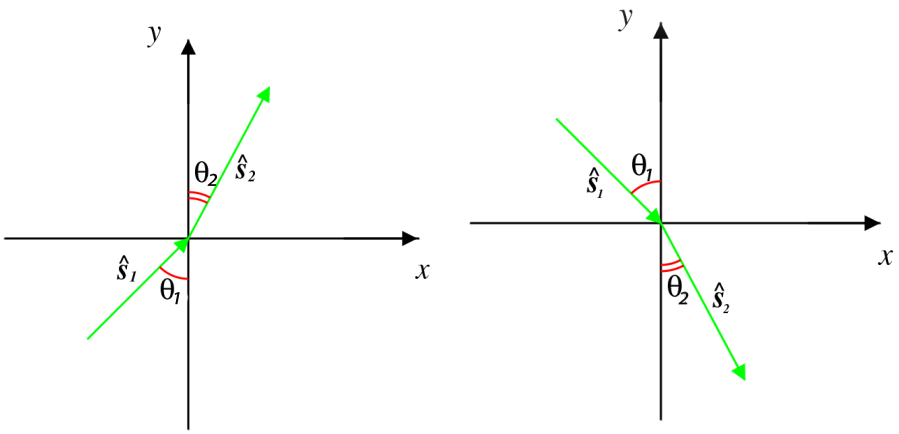


Figure 4.3: **Left:** Refraction of a ray travelling in the positive y direction. **Right:** Refraction of a ray travelling in the negative y direction.

it follows that

$$\theta_1 = \sin^{-1}(s_{1x}), \tag{4.6}$$

where s_{1x} is the x component of $\hat{\mathbf{s}}_1$. Using Snell's law,

$$n_2 \sin(\theta_2) = n_1 \sin(\theta_1), \tag{4.7}$$

we can determine the angle of refraction θ_2, since both n_2 and n_1 are known. Once θ_2 has been found, we can use trigonometry to determine the components of $\hat{\mathbf{s}}_2$. For the ray travelling in the positive direction we find that the new direction vector is

$$\hat{\mathbf{s}}_2 = \sin(\theta_2)\hat{x} + \cos(\theta_2)\hat{y}. \tag{4.8}$$

The right panel of Figure 4.3 shows the direction of a ray travelling in the negative y direction. Applying the same procedure to this case yields

$$\hat{\mathbf{s}}_2 = \sin(\theta_2)\hat{x} - \cos(\theta_2)\hat{y}. \tag{4.9}$$

We can combine the two cases into a single equation using the sgn(.) function:

$$\hat{\mathbf{s}}_2 = \sin(\theta_2)\hat{x} + \mathrm{sgn}(s_{1y})\cos(\theta_2)\hat{y}, \tag{4.10}$$

where s_{1y} is the y component of $\hat{\mathbf{s}}_1$ and

$$\mathrm{sgn}(x) = \begin{cases} +1 & x > 0 \\ 0 & x = 0 \\ -1 & x < 0. \end{cases} \tag{4.11}$$

The MATLAB version of the sgn function is called **`sign`**.

Question 9
Write a MATLAB function **`function s2 = refract_ray(n1, n2, s1)`** which computes the new direction vector using Equation (4.10).

The function should take the direction vector at the current position **`s1`**, the refractive index at the current position **`n1`**, and the refractive index at the new position **`n2`** as arguments. It should return the direction vector at the new position **`s2`**.

Test your function using an initial directon vector of $s1 = [\frac{1}{\sqrt{2}}, \frac{1}{\sqrt{2}}]$ and
a) $n1 = n2 = 1$ (air to air)
b) $n1 = 1, n2 = 1.5$ (air to glass)

Write your answers below.

```
function s2 = refract_ray(s1, n1, n2)
  s1x = s1(1);
  s1y = s1(2);

  theta_i = asind(abs(s1x));
  theta_r = asind(n1*sind(theta_i)/n2);

  s2(1) = sind(theta_r);
  s2(2) = sign(s1y)*cosd(theta_r);
end
```

(a) $s2 = [0.7071, 0.7071]$ ($[1/\sqrt{(2)}, 1/\sqrt{(2)}]$)
(b) $s2 = [0.4714, 0.8819]$
When the refractive indices are the same, the ray should continue in a straight line.

Question 10
Create a copy of your program from Question 8 and modify it so that at each iteration it calls **`refract_ray`** to take account of possible changes in the refractive index.

As a test, use a uniform refractive index of $n = 1$. Run your code and check that it produces straight lines.

Question 11
The final stage of our simulation is modelling the refraction of light as it passes from air downward into a slab of glass.

The refractive index for the air-glass system is

$$n(\mathbf{r}) = \begin{cases} 1.5 & y \leq 0.5 \\ 1.0 & y > 0.5. \end{cases} \tag{4.12}$$

Write a function called **`n = air_glass_interface(r)`** that takes a position vector **`r`** and returns the refractive index at that position, as given by Equation (4.12).

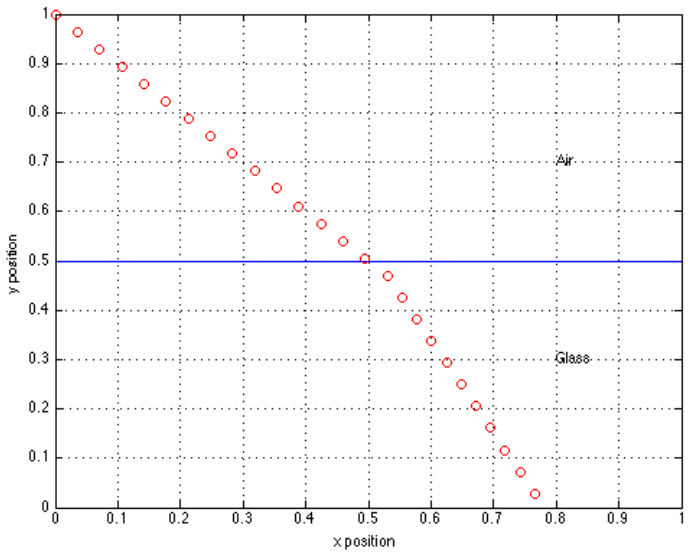
You should incorporate this function into the main loop of your simulation.

Add the line of code

```
plot([0 Lx], [Ly/2,Ly/2], 'b-');
```

to the plotting portion of your code to draw a horizontal line showing the air-glass interface.

Use your program to trace a ray from the top left corner ($r_0 = (0, 1)$) of the domain moving in a direction $\theta_0 = -45°$ to the horizontal. Draw the ray path in the box below.



Tutor note: check their initial position vector is [0, 1] (i.e. top left corner).
Also, you need to calculate n1 at the current position, and n2 at the next position (see solution code).
If path curves down, check that students have a line $n_1 = n_2$ after calculating the new s.

4.9 Total internal reflection (PHYS2921)

Optional for PHYS2011/2911

Total internal reflection is the phenomena where a ray is completely reflected from a transparent boundary. It occurs when a ray passes from a region of high refractive index to a region of low refractive index, but only when the angle of incident is large than the critical angle θ_c defined by

$$\sin\theta_c = \frac{n_2}{n_1}. \tag{4.13}$$

We will need to account for this phenomenon in our ray tracing code. In our model where n only depends on y, the direction of the reflected ray is

$$\hat{\mathbf{s}}_2 = s_{1x}\hat{\mathbf{x}} - s_{1y}\hat{\mathbf{y}}, \tag{4.14}$$

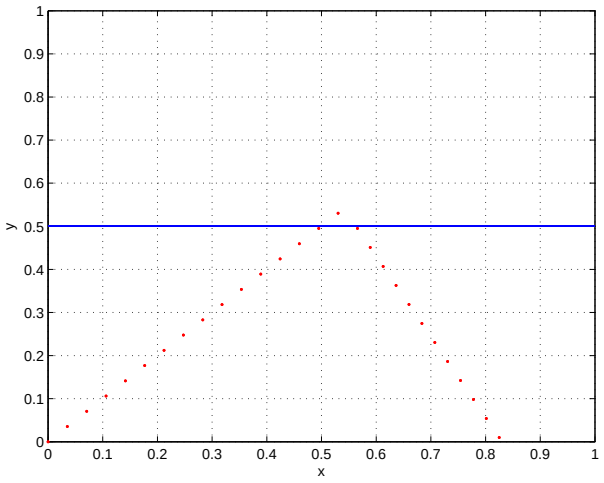
i.e. the y component changes sign, and the x component is unchanged.

Modify your solution to Question 11 to account for total internal reflection. To do this your need a way of detecting when the conditions are met for total internal reflection to occur. One way is to notice that when the incident angle exceeds θ_c, the quantity

$$\frac{n_1}{n_2}\sin\theta_1 \tag{4.15}$$

is greater than unity. When this occurs the angle of refraction $\theta_2 = \sin^{-1}(n_1/n_2 \sin\theta_1)$ becomes meaningless, because the inverse sine of a number greater than one is imaginary. Modify **`refract_ray.m`** so that the new direction is computed using Equation (4.10) if $(n_1/n_2)\sin\theta_1 \leq 1$, and is computed using Equation (4.14) when $(n_1/n_2)\sin\theta_1 > 1$.

Question 12 (PHYS2921)
Using **`air_to_glass_interface.m`** to compute n, draw a ray starting at $\mathbf{r}_0 = (0, 0)$ with initial angle $\theta_0 = 45°$. Use a step-size of $h = 0.05$. Draw the ray path in the box provided.



Checkpoint 2: