x

Computational Analysis of Optical Systems

Thales Swanson, Anango Prabhat

June 2025

Acknowledgements

Our thanks go to the British Physics Olympiad (BPhO) for their work in creating a wide range of engaging and thought-provoking physics challenges, including the optics tasks that formed the basis of this project.

We are also grateful to our physics teacher, Dr Harrison. His excellent teaching and passion for the subject equipped us with the foundational knowledge and enthusiasm needed to undertake this challenge.

Finally, we thank Ella Ulbrich for providing the test image that is used for our simulation.

Abstract

This paper presents a comprehensive investigation into various optical phenomena. We explore key concepts such as refractive indices, lens behaviour, and the formation of rainbows by systematically addressing tasks that apply fundamental principles of optics. Utilising mathematical modelling and computational tools, we derive essential equations and create interactive visualisations to facilitate understanding. The study employs Python, JavaScript, and HTML/CSS to enhance user engagement and illustrates the elegance of light's behaviour in diverse scenarios, including reflection, refraction, and virtual image formation. We create two Flask websites which we convert into android apps. The first contains our solutions to the 2025 Optics Tasks, display interactive simulations. The second is for explaining short and long sight using a vision simulator with ray diagrams and an eye testing game. Our creations can be used as interactive, engaging educational tools for optics, both in terms of the physics through simulations and for understanding refractive error and how it can be corrected.

Contents

1 Introduction **7**

2 Task 1 **7**

3 Task 2 **8**

4 Task 3 **10**

5 Task 4 **11**

6 Task 5 **12**

7 Tasks 6 and 7 **13**

8 Task 8 **15**

9 Task 9 **17**

10 Task 10 **19**

11 Task 11 **20**
11.1 Part a . . . 20
11.2 Part b . . . 21
11.3 Part c . . . 22
11.4 Part d . . . 23

12 Task 12 **24**
12.1 Part a . . . 24
12.2 Part b . . . 28

13 Vision Simulator Website and App Development **31**
13.1 Introduction . . . 31
13.2 Libraries and Technologies . . . 31
13.3 Architecture . . . 32
13.4 Functionality: Vision Simulator . . . 32
13.6.3 Tree 3: Eye Test Game (Client-Side) . . . 39

14 Optics Tasks Website **40**
14.1 Introduction . . . 40
14.2 Libraries . . . 40
14.3 Architecture . . . 40
14.4 Initialisation and Configuration . . . 41
14.5 Homepage and Navigation . . . 41
14.6 Task Display . . . 42
14.7 Plot Generation (Backend) . . . 42
14.8 Deployment . . . 42
14.9 Functionality Forest . . . 43
14.9.1 Tree 1: Core Application Setup & Main Navigation Routes . . . 43
14.9.3 Tree 3: Interactive Task Page & Client-Side Logic . . . 46
14.9.4 Tree 4: Backend Plot Generation (Interactive Tasks via `/plot/<task_id>`) . . . 47
14.9.5 Tree 5: Backend Plot Generation (Static Tasks via `/plot/<task_id>`) . . . 48

14.9.6 Tree 6: Key Helper Utilities 49
14.10Evaluation 49
14.11Website Links 49

15 References **50**

1 Introduction

Optics, a fundamental branch of physics, delves into the behaviour and properties of light, including its interactions with matter and the instruments used to detect it. This field encompasses phenomena such as reflection, and refraction, offering a window into the wave and particle nature of light. The study of optics not only enhances our comprehension of natural phenomena like rainbows and mirages but also underpins advancements in technologies such as lenses, microscopes, and lasers. Through a systematic exploration of optical principles, this investigation seeks to illuminate the intricate ways light shapes our perception of the world. Our work is fuelled by a shared fascination for the elegance of light.
This paper investigates key optical concepts, including Snell's Laws of Reflection and Refraction, the behaviour of light in lenses, and the physics behind rainbow formation. We derive essential equations and develop interactive visualisations to enhance comprehension, employing Python, JavaScript, and HTML/CSS to create two engaging and interactive websites and apps. The first of these displays our interactive solutions to the Optics Tasks, and the second is an interactive demonstration of short and long sight and how these can be corrected. In our work, we utilised several open-source Python libraries to facilitate our research and development, including Matplotlib [1] for data visualization, NumPy [2] for numerical operations, Flask [3] for web application deployment, ipywidgets [4] for creating interactive user interfaces, Pillow [5] for image manipulation, and scikit-image [6] for advanced image processing. We also used Render [7] and Railway [8] in order to host our service. We used capacitor [9] in order to create our app.

2 Task 1

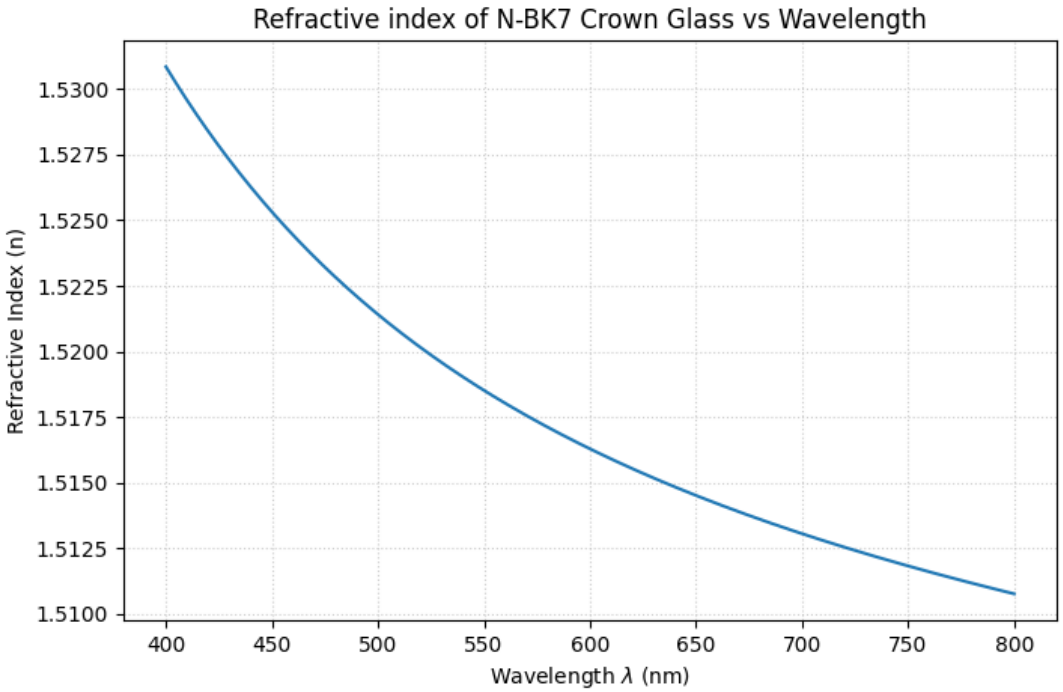
Both parts of this question were simple tasks which required the plotting of data, using the given equations

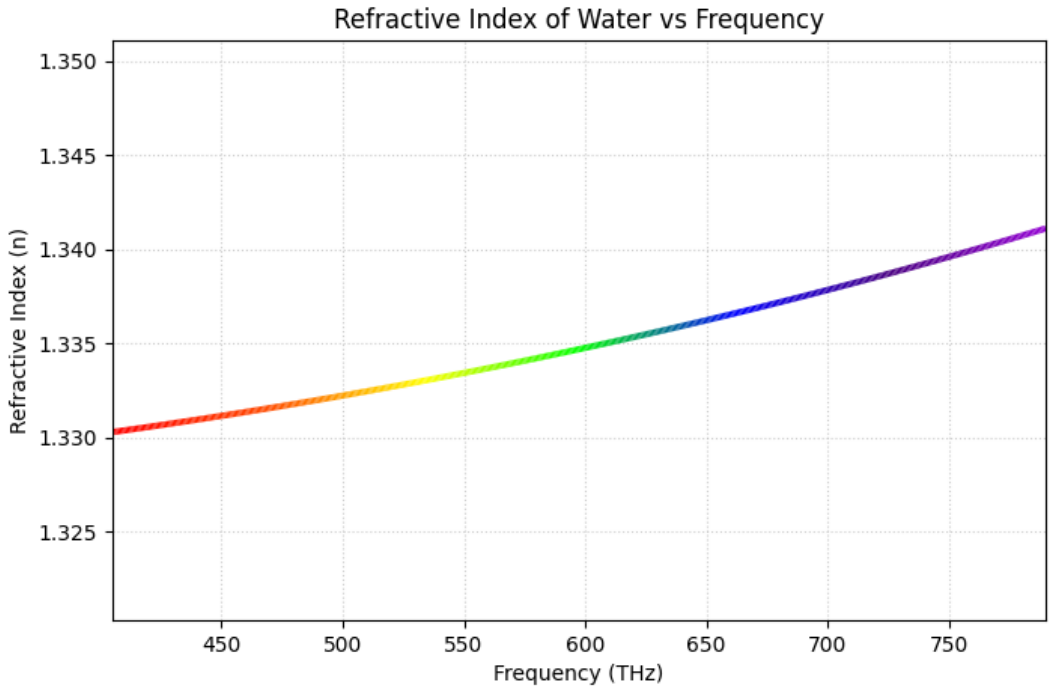
$$n = \sqrt{1 + \sum_k \frac{a_k \lambda^2}{\lambda^2 - b_k}}$$
$$a = [1.03961212, 0.231792344, 1.01146945]$$
$$b = [0.00600069867, 0.0200179144, 103.560653]$$
$$(n^2 - 1)^{-2} = 1.731 - 0.261(\frac{f}{10^{15}\text{Hz}})^2$$

we first rearranged the final equation to make n the subject in order to get

$$n = \sqrt{1 + \sqrt{\frac{1}{1.731 - 0.261(\frac{f}{10^{15}\text{Hz}})^2}}}$$

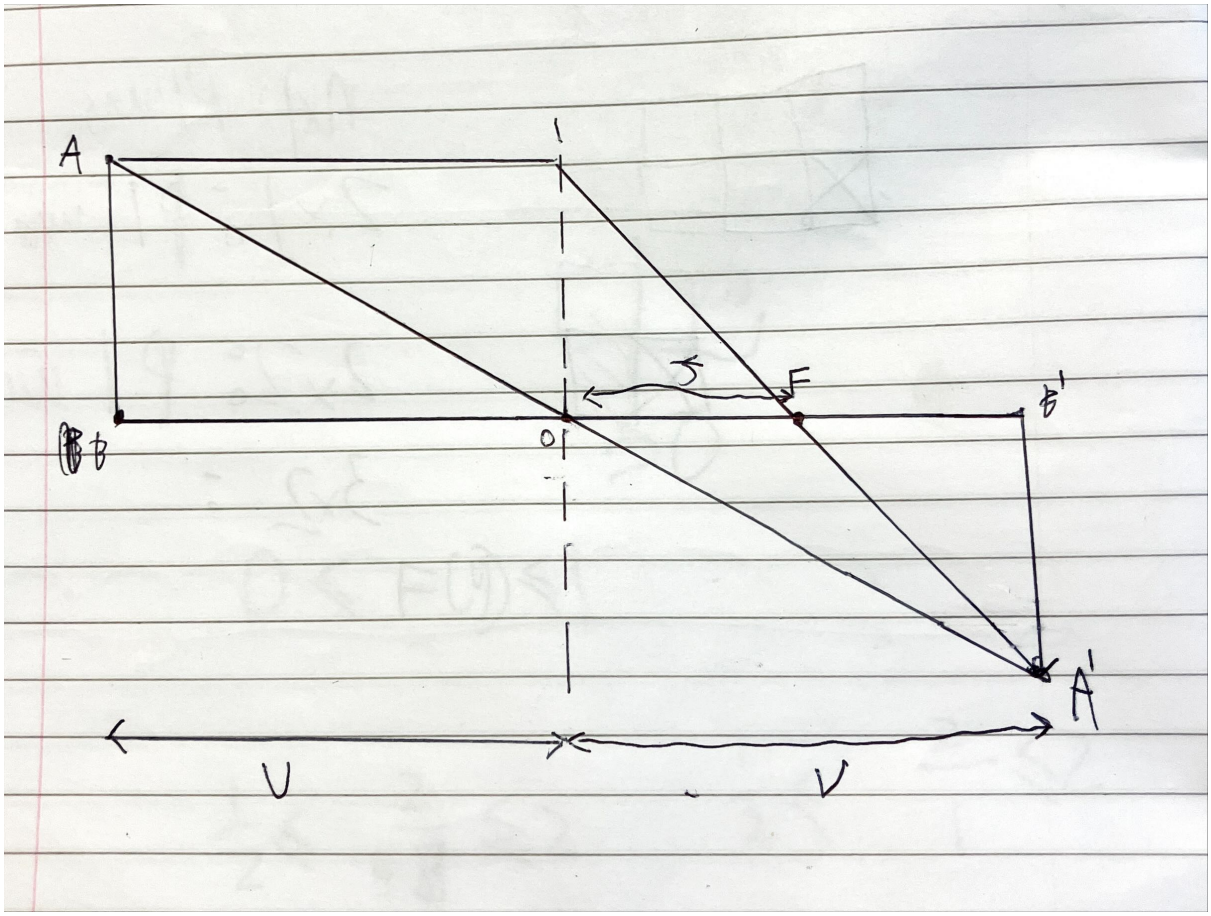
we then used the Matplotlib library in order to plot the necessary data.
In order to create a smooth colour map based on frequency for the second part of this question we used the LineCollection library to break our plot up into lines, and then the LinearSegmentedColorMap library in order to colour these lines, in the colours of the rainbow based on their frequency.
Shown below are the graphs of Refractive index of N-BK7 Crown Glass vs Wavelength, and Refractive index of water vs frequency.



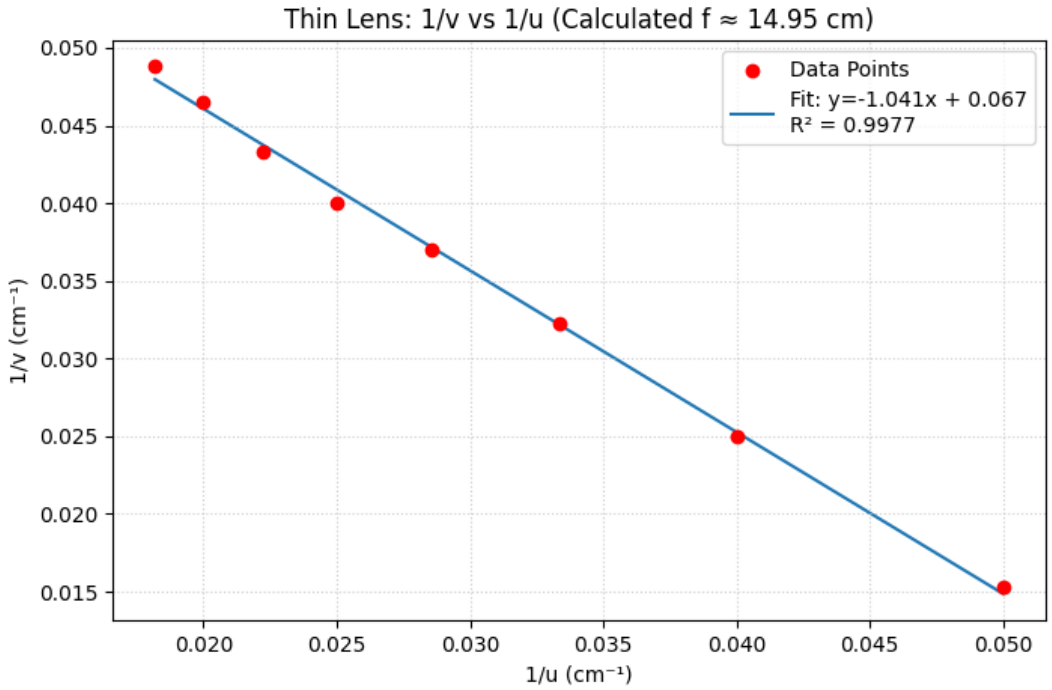


3 Task 2

This task relies on the thin lens equation $\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$. We will start by deriving this equation.



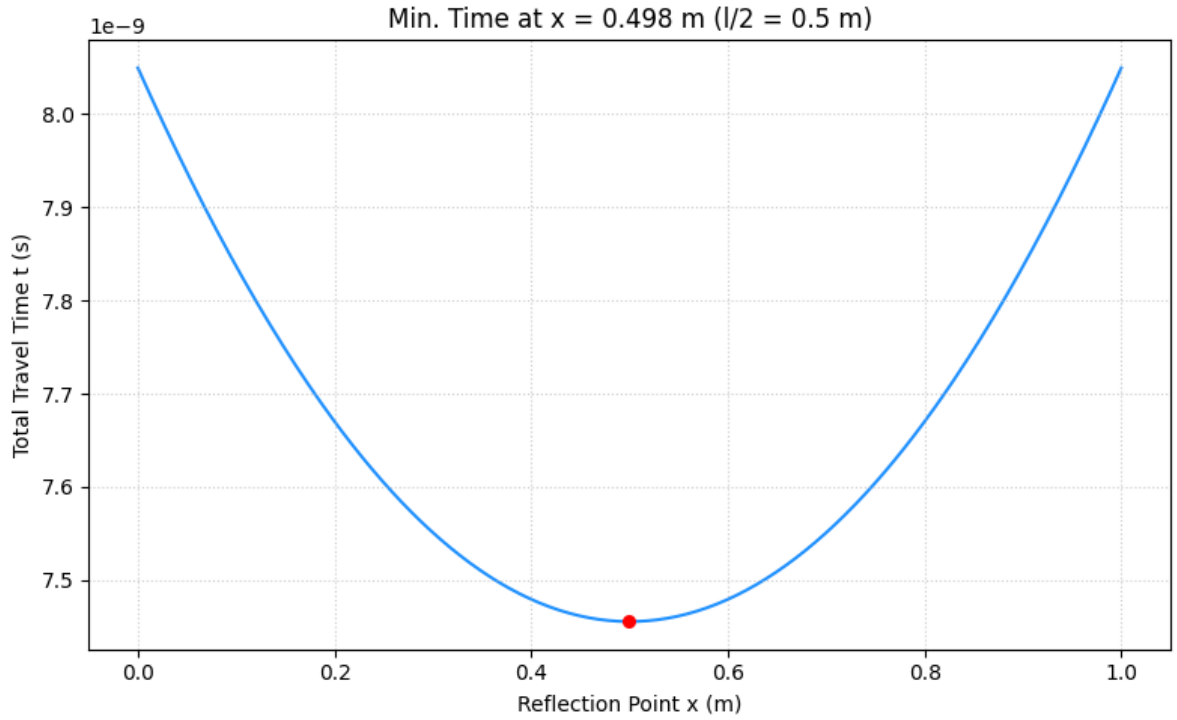
Note that since $B'OA' = BOA$ and $ABO = A'B'O = 90°$ we know that $\triangle ABO$ is similar to $\triangle A'B'O$. Hence $\frac{A'B'}{AB} = \frac{OB'}{OB}$, we will call this equation 1.
Additionally since we know that $CFO = A'FB'$ and $FB'A' = FOC = 90°$ we know that $\triangle A'FB'$ is similar to $\triangle CFO$. Hence $\frac{A'B'}{OC} = \frac{FB'}{OF}$. Now as the direction AC is horizontal we know that $AB = OC$. Substituting this in gives us $\frac{A'B'}{AB} = \frac{FB'}{OF}$ we will call this equation 2.
By equating equations 1 and 2 we get that $\frac{OB'}{OB} = \frac{FB'}{OF} = \frac{OB'-OF}{OF}$.
Now notice that $OB' = v$, $OB = u$, and $OF = f$. By substituting these all in we can see that $\frac{v}{u} = \frac{v-f}{f}$ multiplying through by uf gives us $vf = uv - uf$ and so $uv = vf + uf$. Now we divide by uvf in order to get that $\frac{1}{f} = \frac{1}{u} + \frac{1}{v}$, and so we have derived the thin lens equation.
Now, in order to complete this task we plotted the points shown using the Maplotlib library, and used np.polyfit in order to create the line of best fit. Using the equation $\frac{1}{u} + \frac{1}{v} = \frac{1}{f}$ we can see that when $\frac{1}{u} = 0$, we will have $\frac{1}{v} = \frac{1}{f}$, this means that the y-intercept should be $\frac{1}{f}$. This tells us that $f = \frac{1}{0.06688644140147935} = 14.950713164 \approx 15$, so the focal length is approximately 15cm.
By rearranging the equation into the equation of a line we can see that $\frac{1}{v} = -\frac{1}{u} + \frac{1}{f}$. Since $\frac{1}{f}$ is a constant, this means that we should expect the gradient of $\frac{1}{v}$ vs $\frac{1}{u}$ to be exactly -1. In our graph we can see that instead the gradient is -1.041. This means there is an error of 4.1% between the theoretical gradient and the experimental gradient, a small error like this is to be expected, and means our data is likely mostly accurate.
The R^2 value we got was 0.9977, indicating that $\frac{1}{v}$ and $\frac{1}{u}$ have a strong linear relationship. This is exactly equal to the R^2 value given in the sample graph (in the slides), implying that our plotting and calculation is highly likely to be fully correct. A further verification is that the equation of the line is the same as in the sample graph to three significant figures.
Shown below the given data is plotted.



4 Task 3

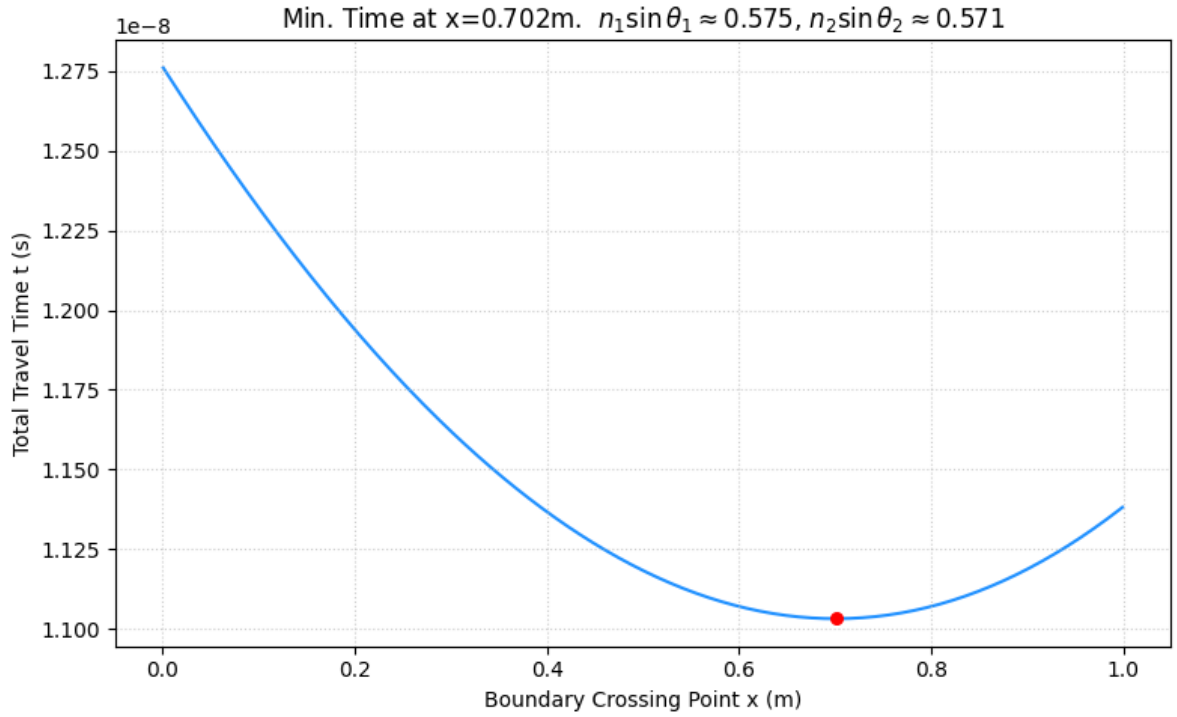
This was a simple task which involved plotting t vs x using the formula $t = \frac{\sqrt{x^2+y^2}}{c/n} + \frac{\sqrt{(L-x)^2+y^2}}{c/n}$, we used Matplotlib to accomplish this. Additionally we initially used the ipywidgets library in order to add sliders to allow the user to change y, L, n, and c (we used . The program we created shows that the minimum travel time occurs when $x = \frac{L}{2}$, which shows Snells Law of Reflection is true by utilising Fermat's principle.

Shown below is a graph of t vs x for reflection of light in a vacuum. Both the length and height have been set to 1m. It can be seen that the smallest time is reached when x is halfway through, which demonstrates Snells Law of Reflection.



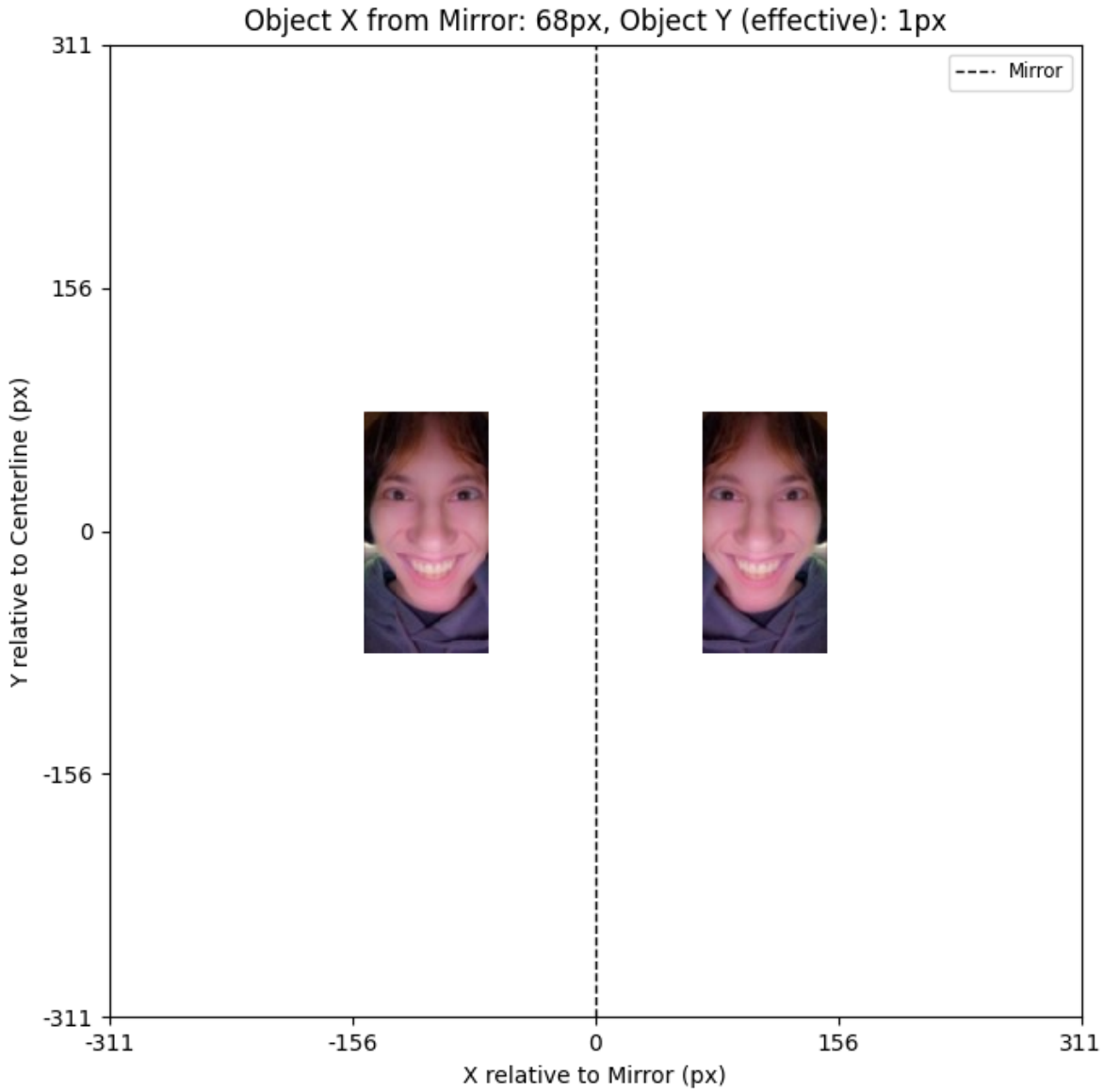
5 Task 4

This task involved plotting t vs x. To do this we used the formula $\frac{\sqrt{x^2+y^2}}{c/n_1} + \frac{\sqrt{(L-x)^2+y^2}}{c/n_2}$, we used Matplotlib to accomplish this. We also used the ipywidgets library in order to add sliders to allow the user to change y, L, n_1, n_2, and c. The program showed that the minimum travel time occurs when $n_1 \sin\theta = n_2 \sin\phi$ which shows Snells Law of Refraction is true by utilising Fermat's principle.
Shown below is a graph of t vs x for refraction of light moving from a material with refractive index 1, to a material with refractive index 2. Both length and height have been set to 1m. It can be seen that the smallest time is reached when $n_1 \sin\theta_1 \approx n_2 \sin\theta_2$, which demonstrates Snells Law of Refraction.



6 Task 5

For this task we had to plot an object (represented as an array of pixels) and its reflection in a plane mirror, in order to mimic the style we will have to use for later tasks we achieved this by transforming pixels. We did this by first creating a white canvas, and then drawing both the object and virtual image onto this canvas. An offset is added to both the coordinates of the pixels contained within the original image, as well as the pixels contained within the virtual image in order to create the canvas with (0,0) at the centre. We made this interactive with the use of sliders, in order to allow the user to move the image around on the canvas, as well as increase or decrease the canvas size. Shown below is an example output.



7 Tasks 6 and 7

These tasks involved plotting an object and the images of this object when it is placed near an ideal, thin convex lens. We created an interactive model using sliders which the user can modify to change the starting X and Y positions of the object, as well as the focal length. We used the same method for both Task 6 and 7, which is why we have combined them into one simulation. The code displays the object, transforms each pixel of the object according to some formulae, and then creates the output pixel grid. Linear interpolation along first rows then columns is then used to fill in the remaining pixels, to prevent gaps occurring in the image if the image is larger than the object. This is superior to bilinear interpolation for this specific task because it ensures that all pixels outside the image shape are white. Here is a detailed explanation of how the pixel transformations are calculated:

The simulation is based on two fundamental principles of geometric optics: the Thin Lens Equation and the Magnification Equation. The code applies these formulas to every single pixel of the source object to determine the position of the corresponding pixel in the final image.

To find the image distance of a pixel v, and thus the transformed x-coordinate, we use a rearranged form of the Thin Lens Equation, which we derived for Task 2:

$$v = \frac{uf}{u - f}$$

where u is the object distance from the lens and f is the focal length. We use the Cartesian Sign Convention here, since this is useful for implementation.

To find the transformed y-coordinate, we use the Magnification equation. Having found v already, and knowing u, we find $M = -\frac{v}{u}$. Then, we can compute the height of the image element relative to the optical axis as $h_i = M \cdot h_o$, and thus we can get the transformed y-coordinate.

We make this interactive with sliders, and use banned slider functionality in JavaScript to prevent the object from overlapping the focal point, since that would lead to an infinitely large image.

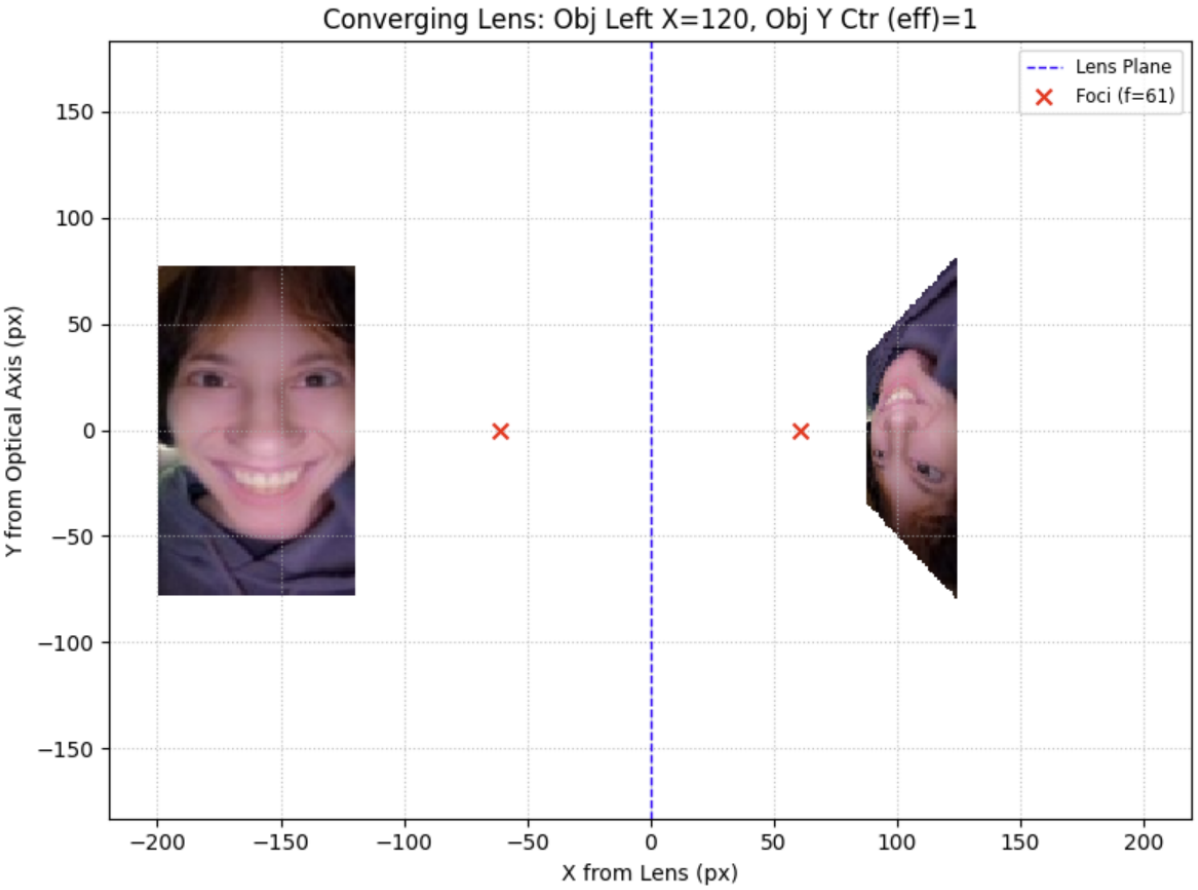


Figure 1: Task 6 Example

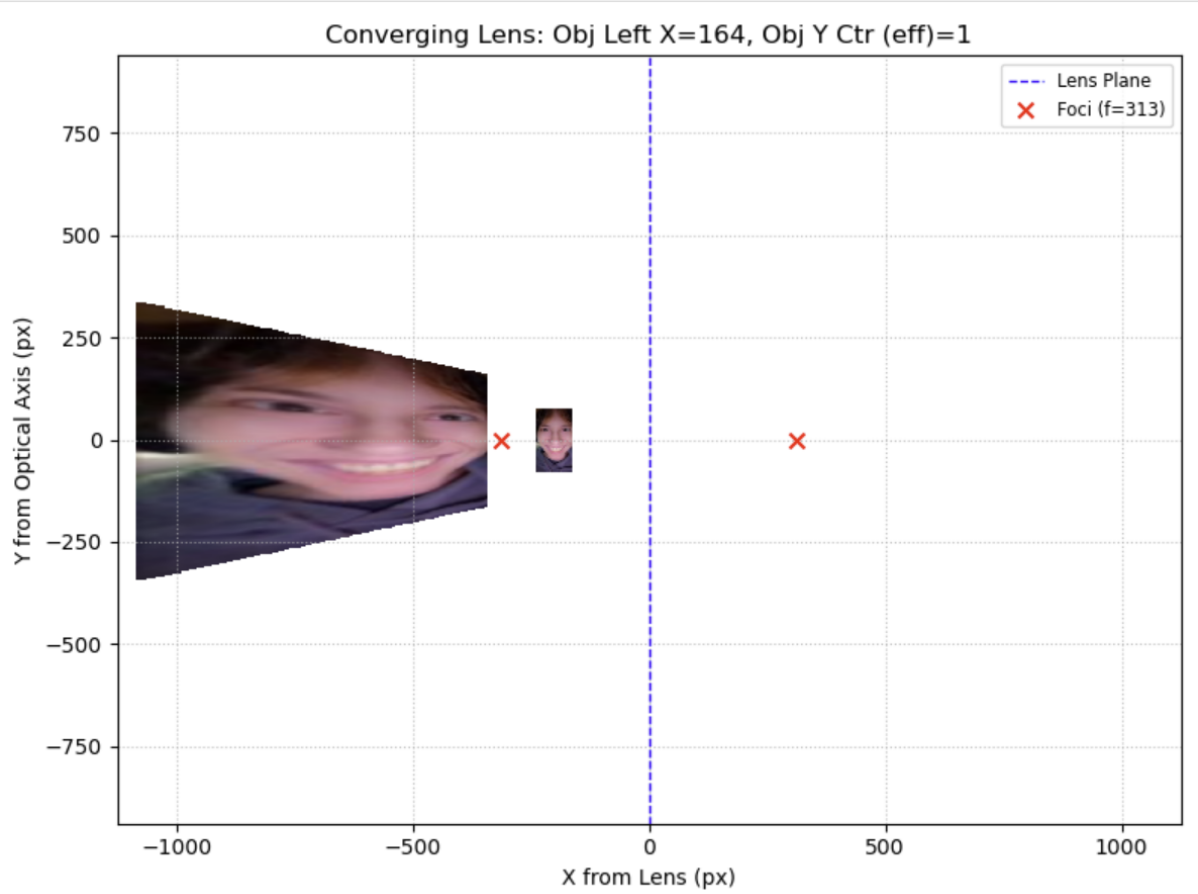


Figure 2: Task 7 Example

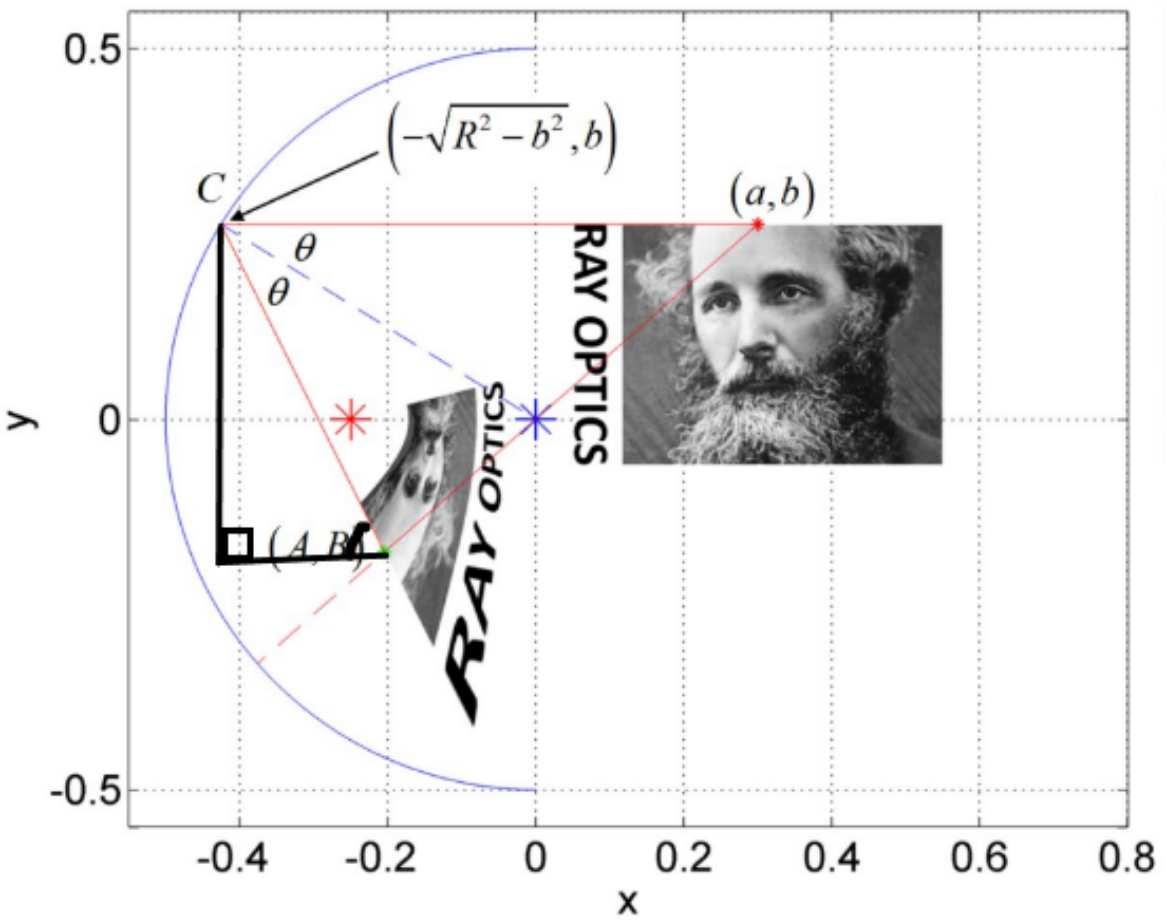
8 Task 8

To do this task we noticed that the given equations are inconsistent, and so we started by deriving the correct equations used to find the coordinates of a transformed pixel. These equations will be used to perform the pixel transformations in our code.

Firstly we explain why $\theta = \tan^{-1}(\frac{b}{\sqrt{R^2-b^2}})$.

The circle is centred at the origin so its equation is $x^2 + y^2 = R^2$. The reflection point is the point on the circumference of the circle when $y = b$ so $x^2 + b^2 = R^2$, and so $x^2 = R^2 - b^2$ and so $x = \pm\sqrt{R^2 - b^2}$. However as the mirror is solely to the left of the origin we must have $x = -\sqrt{R^2 - b^2}$. So the coordinates are $(-\sqrt{R^2 - b^2}, b)$

$\tan\theta = \frac{\text{opposite}}{\text{adjacent}} = \frac{b}{-\sqrt{R^2-b^2}}$ and so $\theta = \tan^{-1}(\frac{b}{-\sqrt{R^2-b^2}})$ now we are able to derive equations for m_i, X_i and Y_i



We consider drawing this triangle as shown above. Note that the angle at the bottom right of this triangle must equal 2θ as it and the angle at C are alternate.

Next we considered the line connecting C to (a', b'), this must have gradient of $\frac{\triangle y}{\triangle x} = \frac{-1\times\text{opposite}}{\text{adjacent}} = -\tan 2\theta = -m$. Additionally we know this line goes through the point $C = (-\sqrt{R^2-b^2}, b)$, so the equation of the line is given by $y - b = -m(x - -\sqrt{R^2-b^2})$ and so $y = -mx + b - m\sqrt{R^2-b^2}$

Now notice the line connecting (a, b) to (a', b'). This line has gradient $\frac{b}{a}$ and goes through the origin, so has equation $y = \frac{b}{a}x$.

The coordinates (a', b') is the point where these two lines intersect. We can find this point by setting $\frac{b}{a}x = -mx + b - m\sqrt{R^2-b^2}$ and simplifying to get $(m + \frac{b}{a})x = b - m\sqrt{R^2-b^2}$ and so $x = \frac{b-m\sqrt{R^2-b^2}}{m+\frac{b}{a}} = -\frac{m\sqrt{R^2-b^2}-b}{m+\frac{b}{a}}$. This gives us $a' = -\frac{m\sqrt{R^2-b^2}-b}{m+\frac{b}{a}}$. We can find the y-coordinate by substituting $x = -\frac{m\sqrt{R^2-b^2}-b}{m+\frac{b}{a}}$ into $y = \frac{b}{a}x$ to get that $y = -\frac{b}{a}\frac{m\sqrt{R^2-b^2}-b}{m+\frac{b}{a}}$. And so we have $b' = -\frac{b}{a}\frac{m\sqrt{R^2-b^2}-b}{m+\frac{b}{a}}$

Using these equations we plotted the pixel transformations using a similar method to that which we used for the previous 3 tasks.

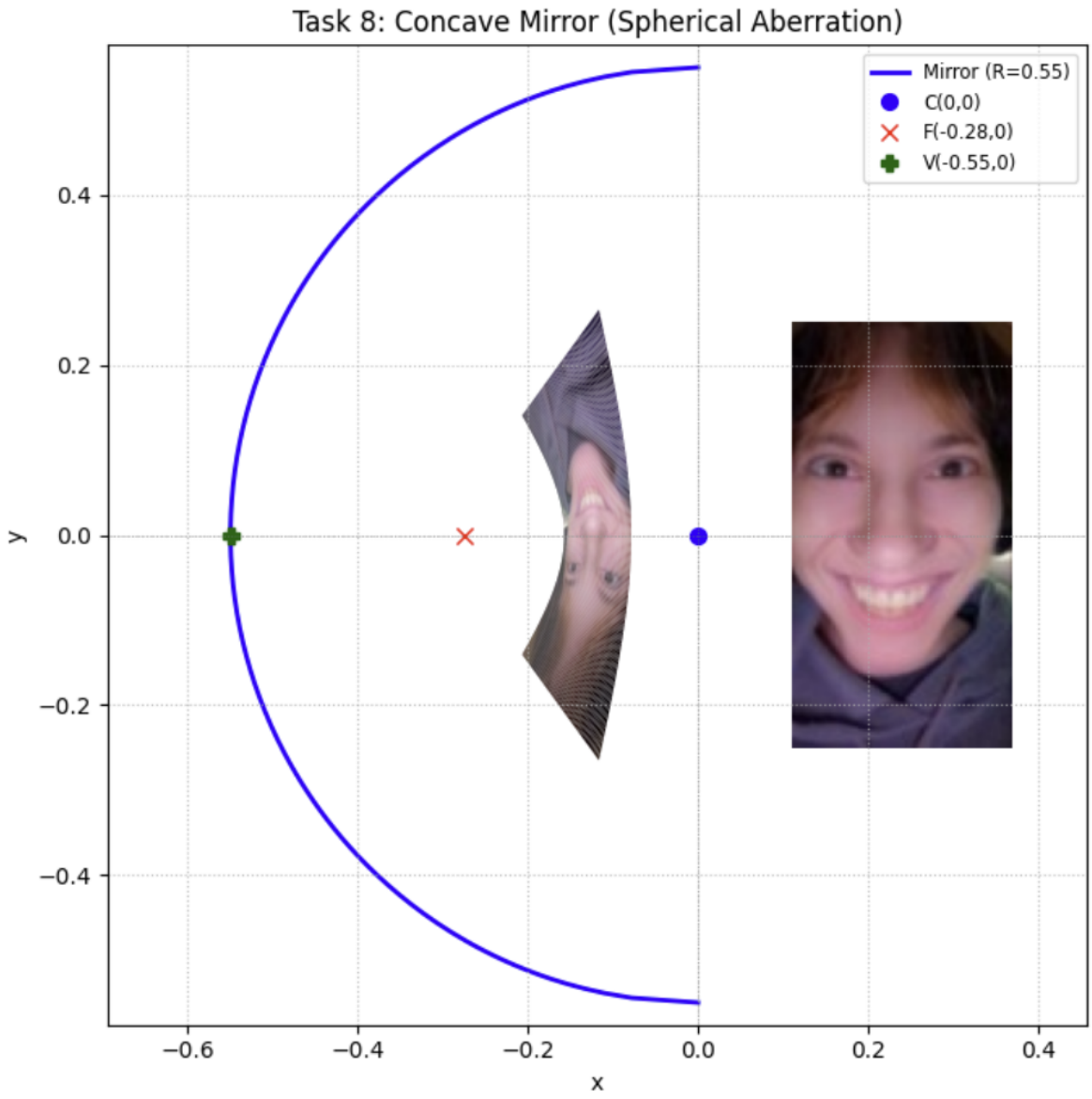
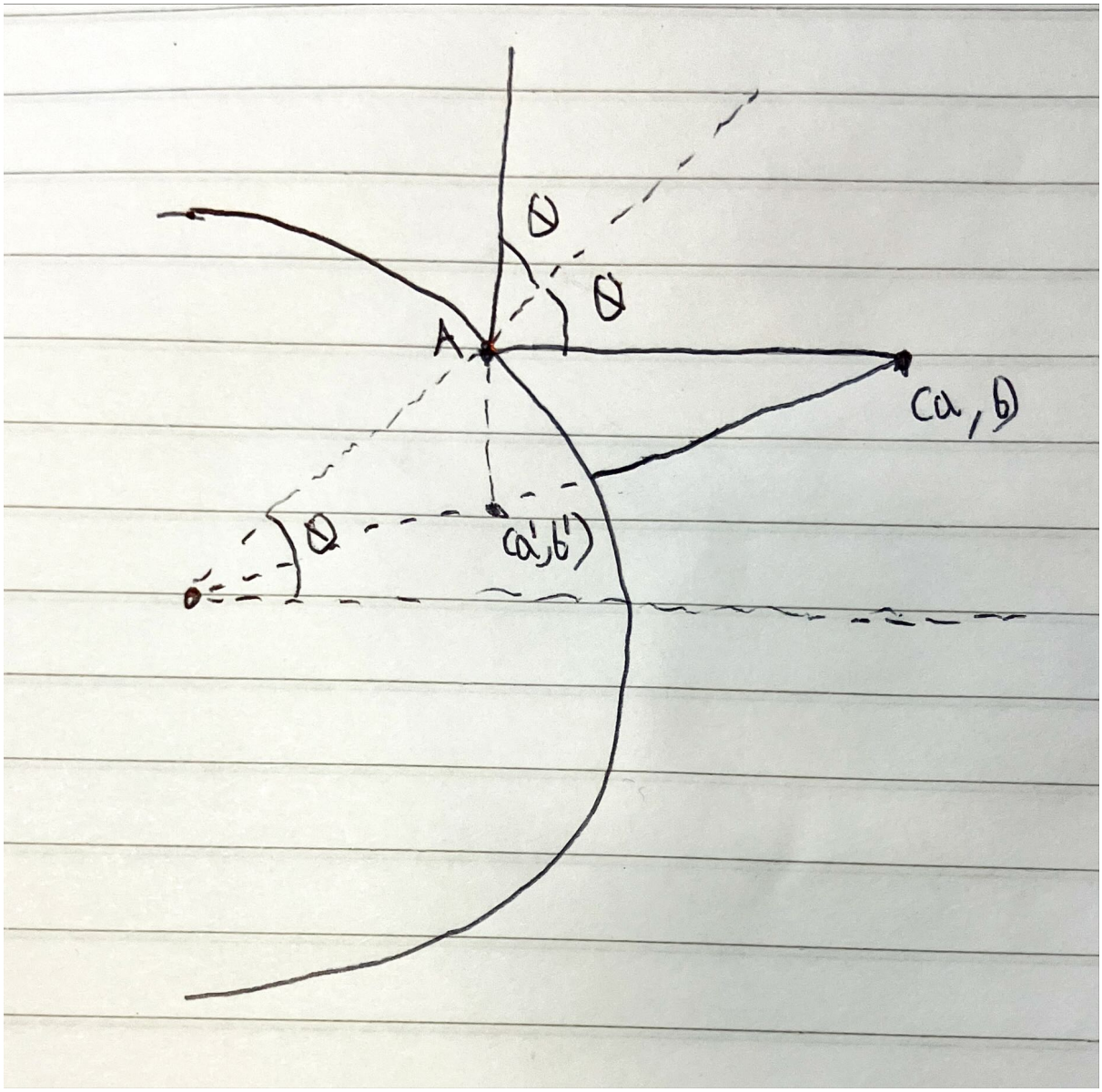


Figure 3: Task 8 Example

9 Task 9

Firstly, we noticed the given equations are inconsistent, and so we derived the equations for the coordinates of the virtual image of an object in a convex spherical mirror, based on the original coordinates of the objects. These equations will be used to perform the pixel transformations in our code



We consider the diagram shown above, where (a, b) are the initial coordinates of a point, and (a', b') are the transformed coordinates of the point. The lens is a circle with radius R centred at the origin, so its Cartesian equation is $x^2 + y^2 = R^2$. Now let's consider point A. We know that this point has y-coordinate b, and so because it lies on the circle we can find its x-coordinate using $x^2 + b^2 = R^2$, and so $x = \sqrt{R^2 - b^2}$. Hence $A = (\sqrt{R^2 - b^2}, b)$.

Now let's consider θ. We know that $\tan\theta = \frac{\text{opposite}}{\text{adjacent}} = \frac{b}{\sqrt{R^2-b^2}}$, and so $\theta = \arctan(\frac{b}{\sqrt{R^2-b^2}})$.

Now we consider the line from A which makes an angle 2θ with the x-axis. We can find the gradient of this line by gradient $= \frac{\triangle y}{\triangle x} = \frac{\text{opposite}}{\text{adjacent}} = \tan 2\theta$, we will define m such that $m = \tan 2\theta$. Additionally we know this line goes through the point $(\sqrt{R^2 - b^2}, b)$ and so its equation is $y - b = m(x - \sqrt{R^2 - b^2})$, and so $y = mx - m\sqrt{R^2 - b^2} + b$.

Now let's consider the line going from the origin to (a, b). This line goes through the origin, and has gradient $\frac{b}{a}$ and so its equation is $y = \frac{b}{a}x$.

Now notice that the point (a', b') is simply the point where these two lines intersect, and so we can find its coordinates as follows:

$mx - m\sqrt{R^2 - b^2} + b = \frac{b}{a}x$

$mx - \frac{b}{a}x = m\sqrt{R^2 - b^2} - b$

$(m - \frac{b}{a})x = m\sqrt{R^2 - b^2} - b$
$x = \frac{m\sqrt{R^2-b^2}-b}{m-\frac{b}{a}}$
By substituting this into $y = \frac{b}{a}x$ we can see that $y = \frac{b}{a}\frac{m\sqrt{R^2-b^2}-b}{m-\frac{b}{a}}$. Hence we have found the coordinates of (a', b').
To conclude we have found:
$a' = \frac{m\sqrt{R^2-b^2}-b}{m-\frac{b}{a}}$
$b' = \frac{b}{a}\frac{m\sqrt{R^2-b^2}-b}{m-\frac{b}{a}}$ where
$m = \tan 2\theta$
$\theta = \arctan(\frac{b}{\sqrt{R^2-b^2}})$
It is interesting that these equations are very similar to those in Task 8, although this is to be expected due to the symmetry of the two tasks. Using these equations we performed pixel transformations in order to draw the true image.

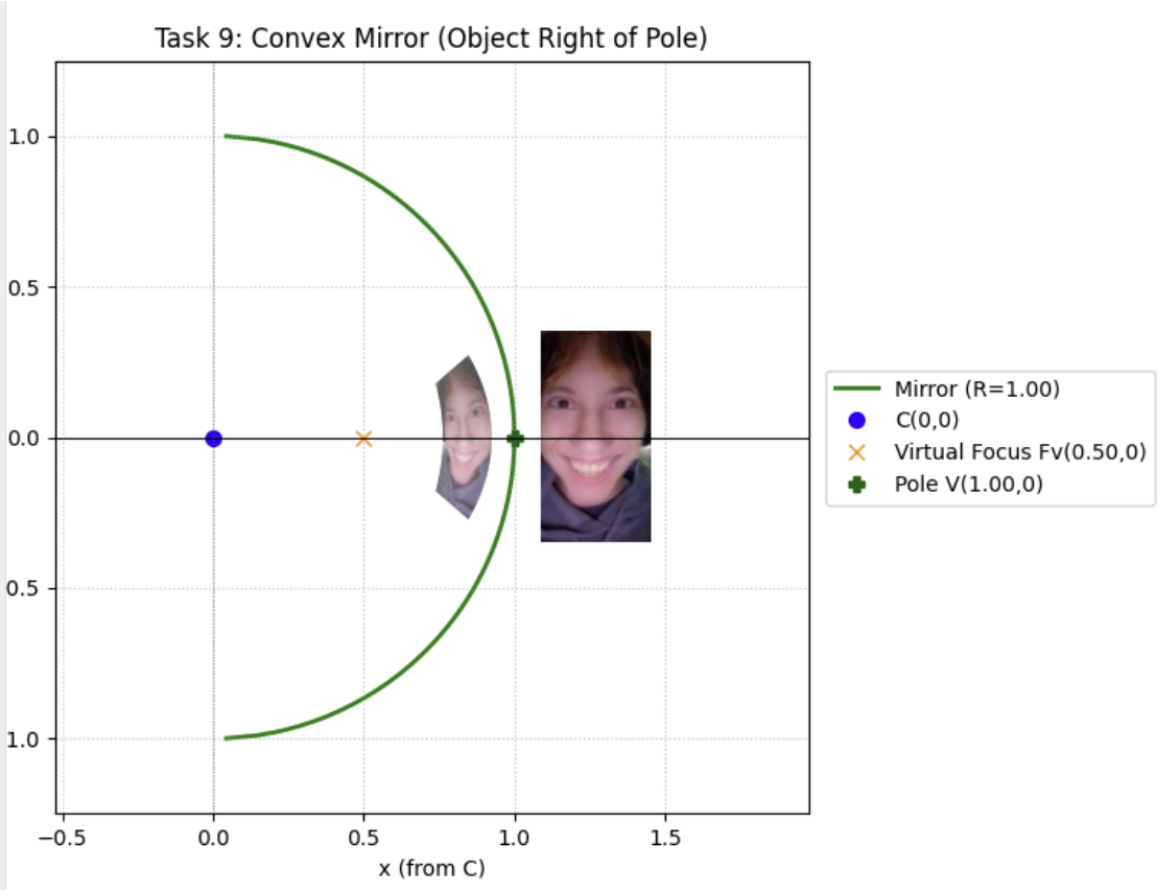


Figure 4: Task 9 Example

10 Task 10

For this task, we have to form a mapping of pixel coordinates to an arc centred at the base of the object. To get notation straight, let the height of the image in pixels be H, the width be W, the angle of the arc be θ in degrees, and R_f be the thickness of the object (as a multiple of the radius of the image). Let R denote the radius (in pixels) of the circle in which the image is inscribed. Instead of using the equations provided, we decided to perform the inverse transformation in the following way: for each row of the image r_i, we form an arc of radius r_{here} centred at the base of the image, evenly distributing the pixels of the row around the arc, and using bilinear interpolation to fill in areas of the object which are

not the direct mapping of a pair of image coordinates. We assumed that r_{here} scales linearly from R to $R(R_f + 1)$ – this assumption can easily be checked using the given formulae. This gives that for row i, the radius is $r_{here_i} = R(1 + \frac{(H-i)R_f}{H})$. This allows us to plot the points.

1

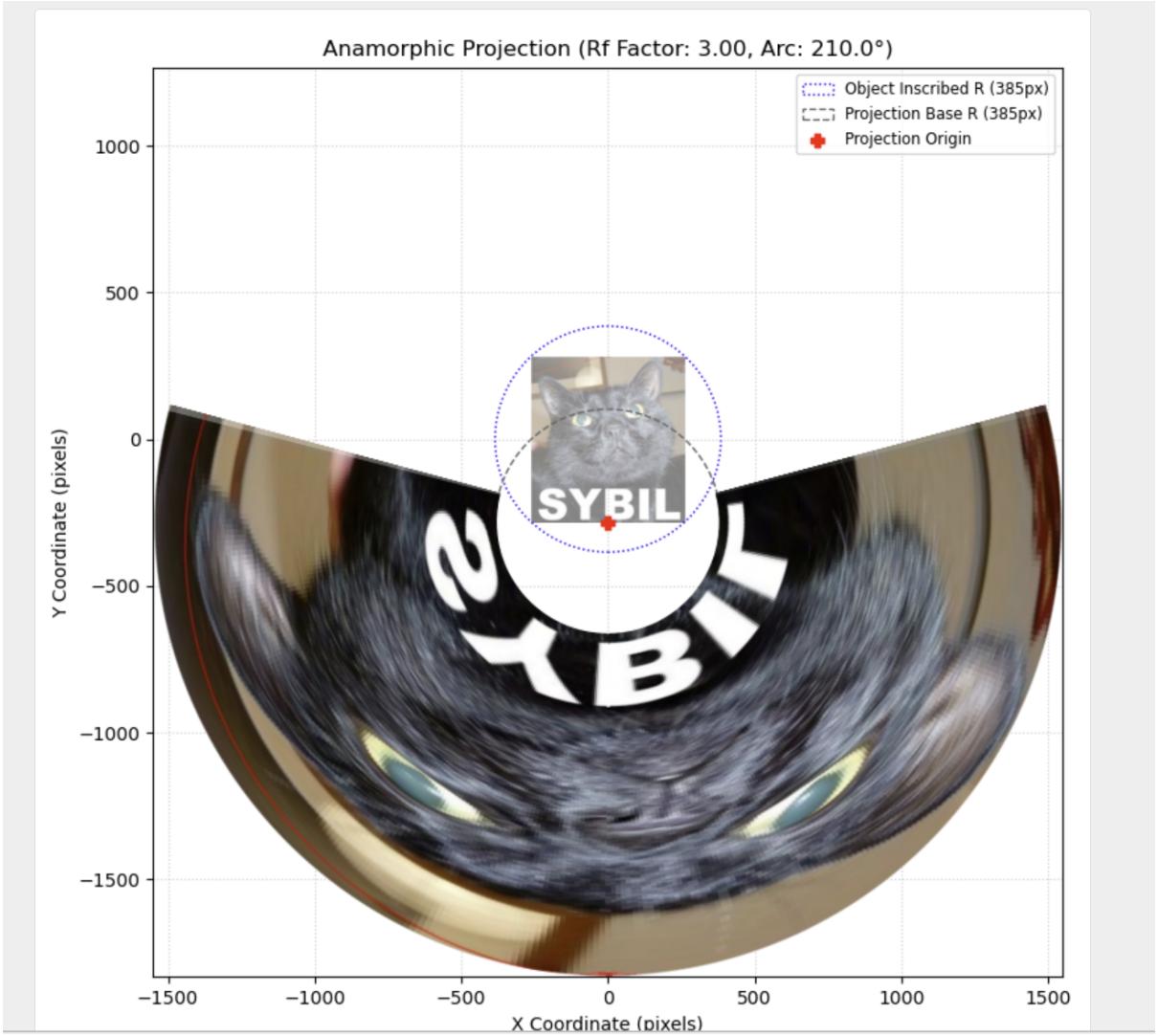


Figure 5: Sybil the Cat - Stretched to 210°

11 Task 11

11.1 Part a

For this subtask we were challenged to plot the Elevation angle ϵ of the rainbow against the angle of incidence θ_i for different frequency's of light for both primary and secondary bows. In order to do this we used the equations for ϵ based on n and θ.

$$\epsilon = 4\sin^{-1}(\frac{\sin\theta}{n}) - 2\theta$$
$$\epsilon = \pi - 6\sin^{-1}(\frac{\sin\theta}{n}) + 2\theta$$

We also used the equations for the θ which minimize ϵ as these will be where a rainbow is observed.

$$\theta = \sin^{-1}\sqrt{\frac{4-n^2}{3}}$$
$$\theta = \sin^{-1}\sqrt{\frac{9-n^2}{8}}$$

Where n depends on frequency based on the formula $(n^2-1)^{-2} = 1.731 - 0.261(\frac{f}{10^{15}\text{Hz}})^2$ from task 1b. Using this we were able to plot each colour of light in order to achieve the desired graph.

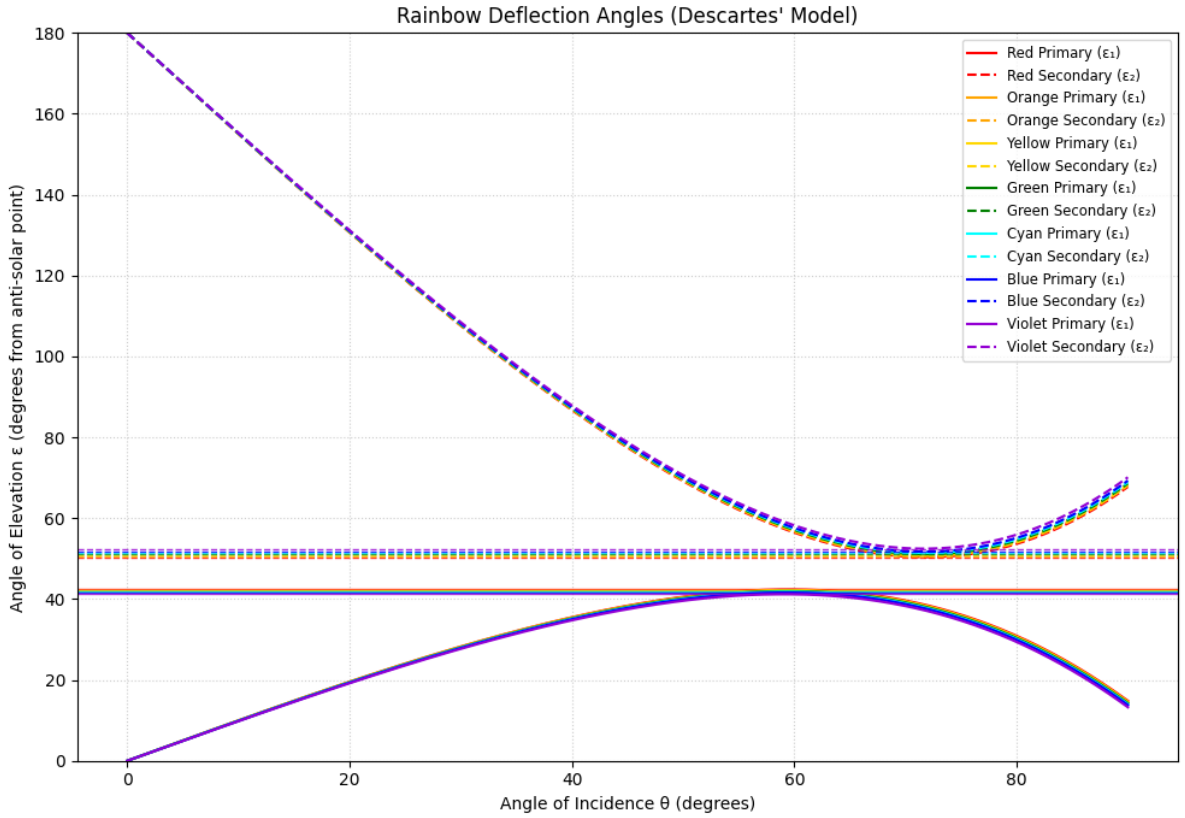


Figure 6: Task 11a Graph

11.2 Part b

For this subtask we were challenged to plot the elevation of primary and secondary rainbows against the frequency of light, using the same equations as in the previous part. We used a similar method to Task 1b to accomplish this, by taking advantage of the LineCollection and LinearSegmentedColorMap libraries.

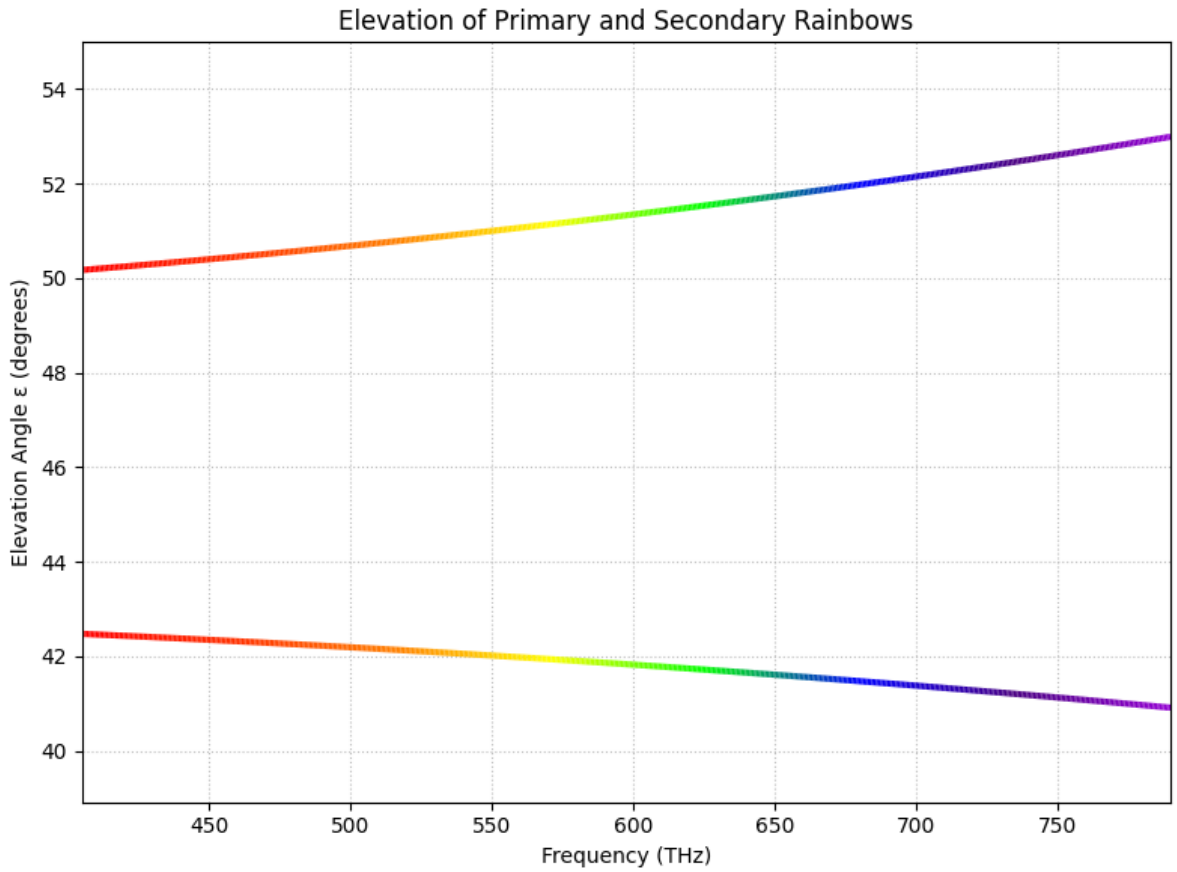


Figure 7: Task 11b Graph

11.3 Part c

For this subtask we were challenged to plot ϕ vs the frequency of light for primary bows, secondary bows, and the critical angle. In order to achieve this we utilised the equations

$$\theta = \sin^{-1}\sqrt{\frac{4-n^2}{3}}$$

$$\theta = \sin^{-1}\sqrt{\frac{9-n^2}{8}}$$

$$n = \sqrt{1+\sqrt{\frac{1}{1.731-0.261(\frac{f}{10^{15}\text{Hz}})^2}}}$$

$$\phi = \frac{\sin\theta}{n}$$

In order to plot ϕ vs frequency for both primary and secondary bows, we then used a similar method in order to plot the critical angle by noticing that this would result in $\phi = \arcsin(\frac{1}{n})$ and plotting accordingly. We also coloured the data based on frequency of light.

When plotting the graph we expect the secondary rainbow to occur at a larger refraction angle than the primary rainbows, and this is indeed the true output which is observed. However, in the graph shown in the task the legend is the other way around, this is likely due to an error by the organisers while creating the legend.

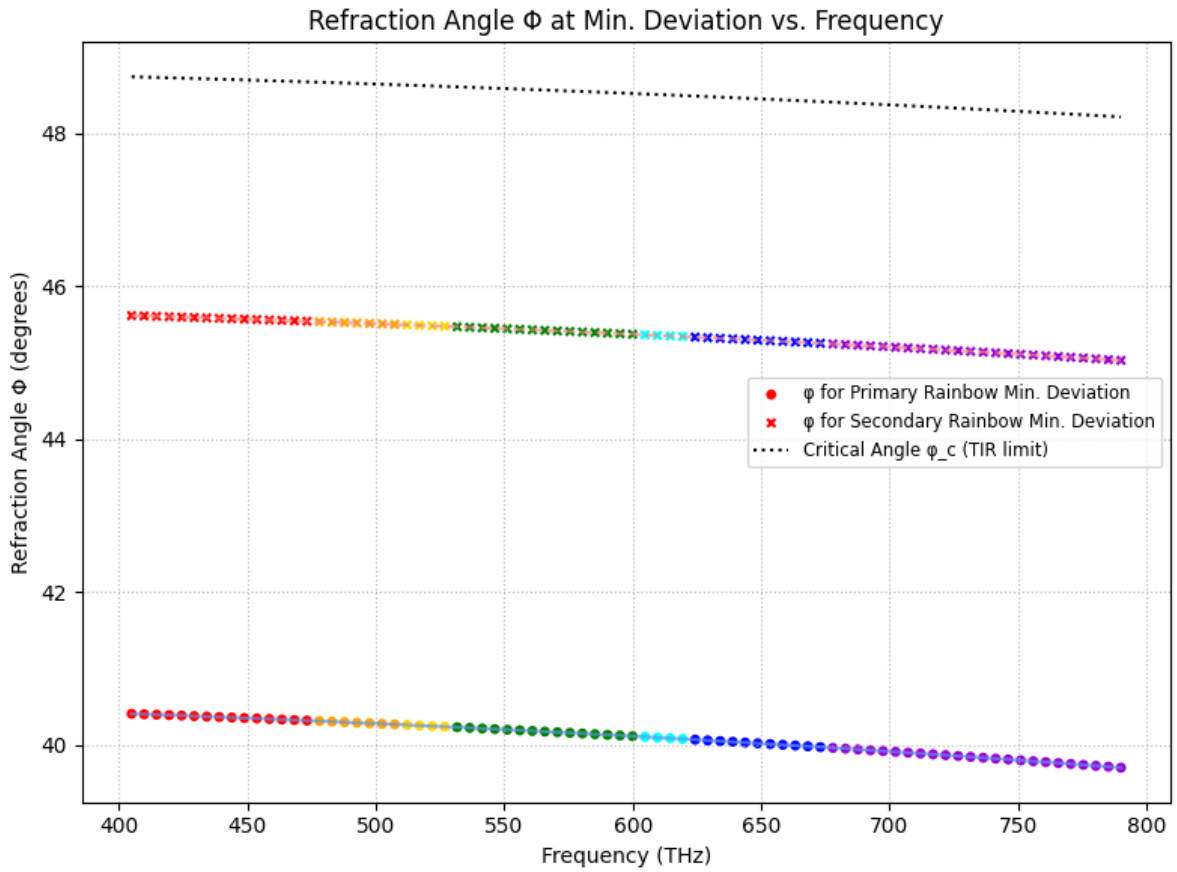


Figure 8: Task 11c Graph

11.4 Part d

For this subtask we were challenged to plot an interactive model of a primary and secondary rainbow based on the elevation of the sun above the horizon. In order to do this we used the fact that the rainbow circle of a particular frequency of light can be described by

$$\text{radius} = r \sin\epsilon \cos\alpha$$

$$\text{distance centre is below horizon} = r \sin\epsilon \cos\alpha - r \sin(\epsilon - \alpha)$$

Where α is the elevation angle of the sun, and ϵ is the elevation angle of the rainbow, which is found by using the following equations

$$\epsilon = 4 \sin^{-1}(\frac{\sin\theta}{n}) - 2\theta$$

$$\epsilon = \pi - 6 \sin^{-1}(\frac{\sin\theta}{n}) + 2\theta$$

$$\theta = \sin^{-1}\sqrt{\frac{4-n^2}{3}}$$

$$\theta = \sin^{-1}\sqrt{\frac{9-n^2}{8}}$$

$$n = \sqrt{1 + \sqrt{\frac{1}{1.731 - 0.261(\frac{f}{10^{15}\text{Hz}})^2}}}$$

Using this information the correct circle for both primary and secondary rainbows was plotted for each frequency of light. We made this interactive by having a slider to change the sun elevation angle, and created an animation, playable on the website, to display how the graph changes when the elevation angle changes.

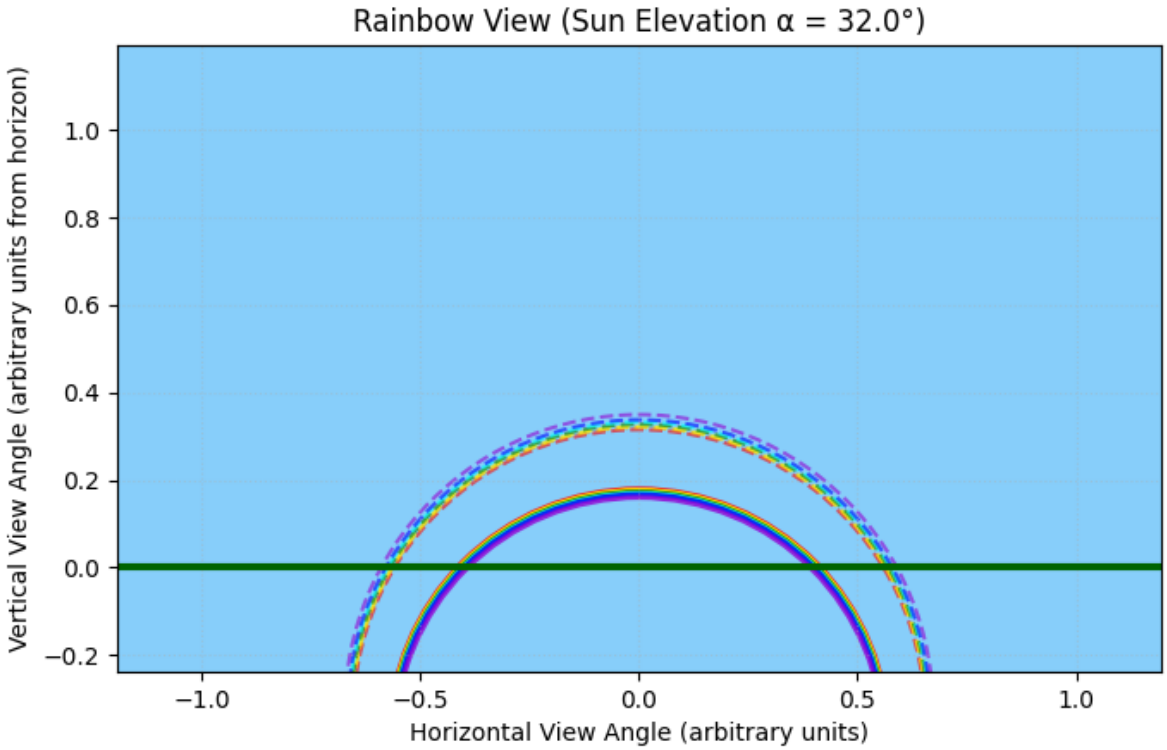


Figure 9: One Frame of the Task 11d Animation

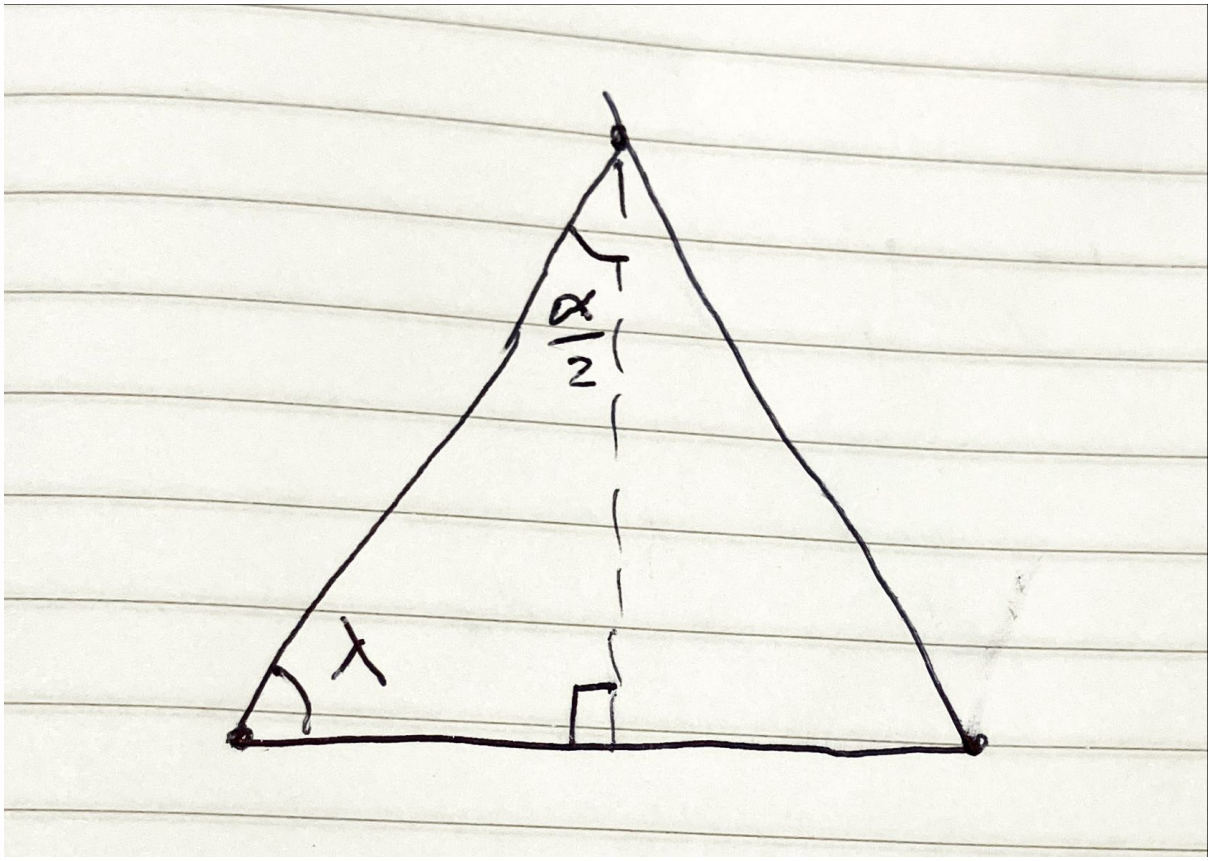
12 Task 12

12.1 Part a

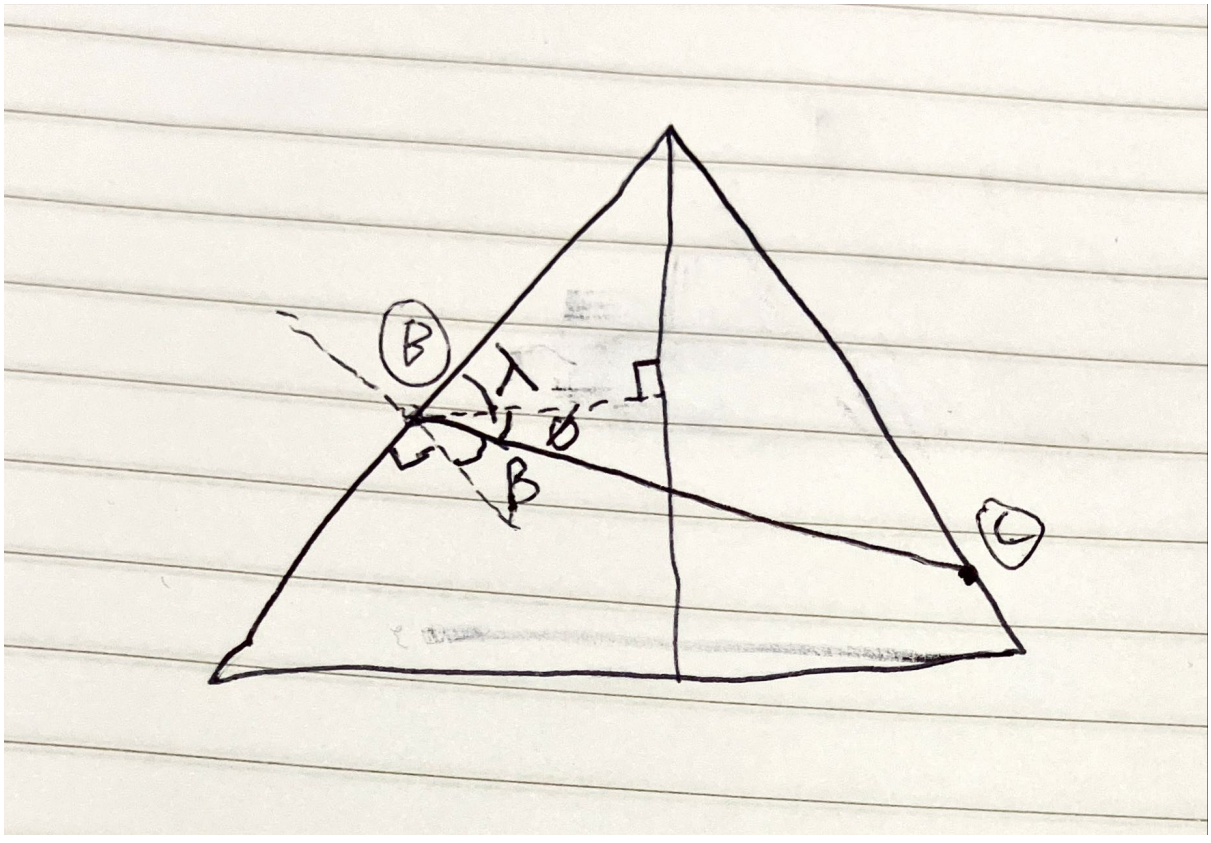
For part (a) of this task we were challenged to create a dynamic model as white light goes through a triangular prism. In order to do this we can find the coordinates of all relevant points, based on the refractive index, which is in turn based on the frequency, we can then use a linspace of wavelengths, and plot the correct line for each one

First let's try to find the coordinates of point C.

By Snell's law of refraction we know $n_a \sin\theta_i = n_g \sin\beta$, and so we have $\beta = \arcsin(\frac{n_a \sin\theta_i}{n_g})$.



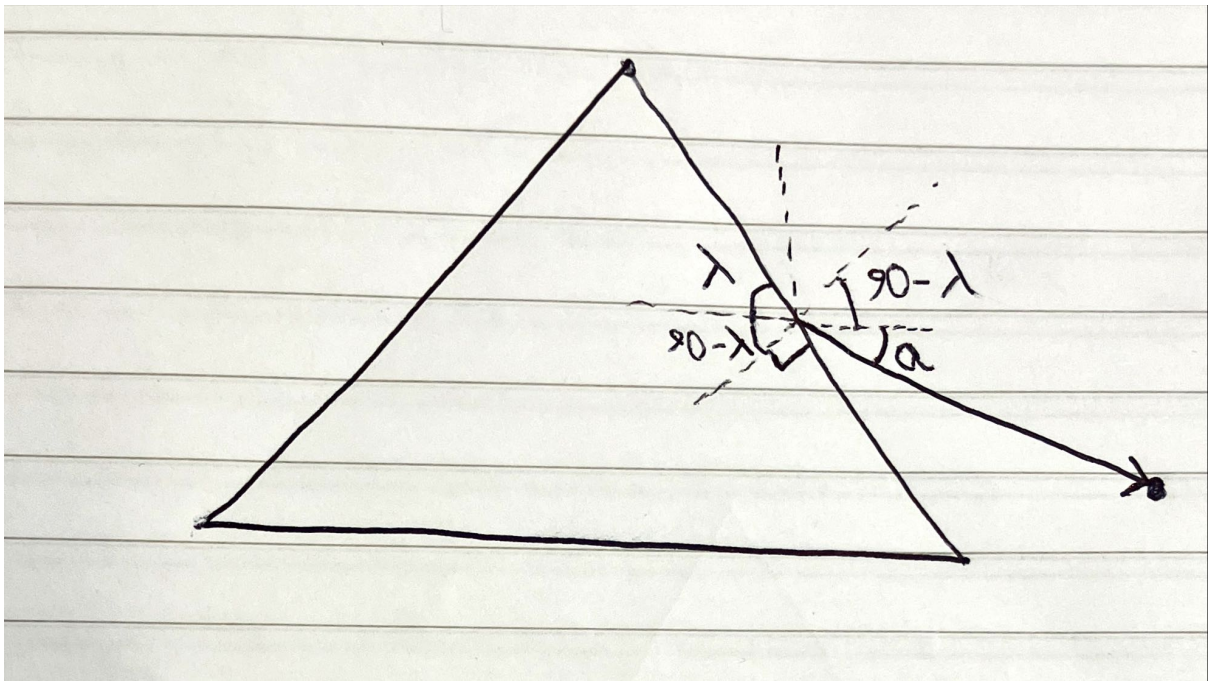
We define λ as shown, it then follows that $\lambda = 90 - \frac{\alpha}{2}$.



We define ϕ as shown, it then follows that $\phi = 90 - \lambda - \beta = \frac{\alpha}{2} - \beta$.

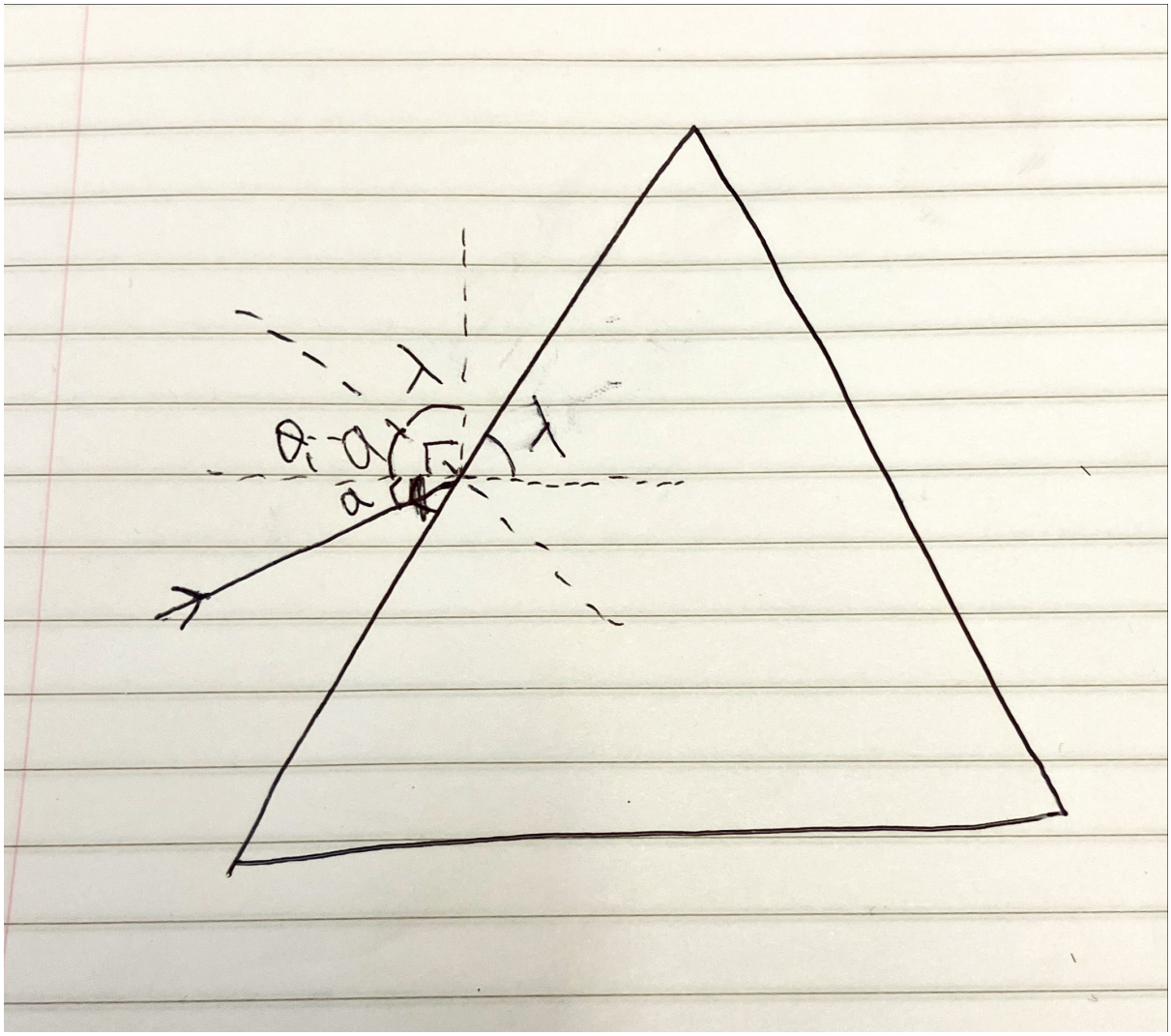
Now let's consider the line which connects B to C. The gradient of this line is $\frac{\triangle y}{\triangle x} = -\frac{\text{opposite}}{\text{adjacent}} = -\tan\phi$. Now let the origin of these axes be the midpoint of the base of the triangle. If we let the top two sides of the triangle have length 1, then the coordinates of point B are $(-\frac{1}{2}\cos\lambda, \frac{1}{2}\sin\lambda)$. The equation of this line is then given by $y - \frac{1}{2}\sin\lambda = -\tan\phi(x - -\frac{1}{2}\cos\lambda)$, which simplifies to $y = -\tan(\phi)x - \frac{1}{2}\cos\lambda\tan\phi + \frac{1}{2}\sin\lambda$. Now let's consider the right side of the triangle, this line has gradient $\frac{\triangle y}{\triangle x} = -\frac{\text{opposite}}{\text{adjacent}} = -\tan\lambda$. This line goes through the top of the triangle which has coordinates $(0, \sin\lambda)$. This means the equation of this line is given by $y - \sin\lambda = -\tan(\lambda)x$, which simplifies to $y = -\tan(\lambda)x + \sin\lambda$.
The point C is the intersection of these two lines, and so we can find its coordinates: $-\tan(\lambda)x + \sin\lambda = -\tan(\phi)x - \frac{1}{2}\cos\lambda\tan\phi + \frac{1}{2}\sin\lambda$, we can simplify to $(\tan\lambda - \tan\phi)x = \sin\lambda + \frac{1}{2}\cos\lambda\tan\phi - \frac{1}{2}\sin\lambda$, and so $x = \frac{\frac{1}{2}\sin\lambda + \frac{1}{2}\cos\lambda\tan\phi}{\tan\lambda - \tan\phi}$, by using the second line we see $y = -\tan(\lambda)x + \sin\lambda$. Now we have our coordinates of point C, we will call these X_C and Y_C.

Now let's find where the light will be after it leaves the prism.



From the diagram above we can see that $\theta_t = 90 - \lambda + a$, and so $a = \theta_t + \lambda - 90$, and $\lambda = 90 - \frac{\alpha}{2}$, substituting this in we see $a = \theta_t - \frac{\alpha}{2}$. If we let the length of the line be 1 (in order to simplify calculations) we see that the new x coordinate is $X_C + \frac{\cos(\theta_t - \frac{\alpha}{2})}{1} = X_C + \cos(\theta_t - \frac{\alpha}{2})$, and the new y coordinate is $Y_C - \frac{\sin(\theta_t - \frac{\alpha}{2})}{1} = Y_C - \sin(\theta_t - \frac{\alpha}{2})$.

Now let's find where the light will be before it goes into the prism, we will use this in order to plot the white ray of light.



From the diagram above we can see that $\theta_i - a + \lambda = 90$, and so $a = \theta_i + \lambda - 90$, now we know that $\lambda = 90 - \frac{\alpha}{2}$, substituting this in gives $a = \theta_i + 90 - \frac{\alpha}{2} - 90 = \theta_i - \frac{\alpha}{2}$. If we let the length of the line be 1 (in order to simplify calculations) we see that the new x coordinate is $-\cos(\theta_i - \frac{\alpha}{2}) - \frac{1}{2}\cos\lambda$, and the new y coordinate is $-\sin(\theta_i - \frac{\alpha}{2}) + \frac{1}{2}\sin\lambda$.

After finding the coordinates of all relevant points lines were drawn connecting them for each colour, using LineCollection from Matplotlib.collections. We also allowed the values for α and θ_i to be changed using a slider, using the ipywidgets library, and converted to JavaScript sliders in the website.

Keypress controls use the W and Q keys to increase and decrease the incident angle respectively, and S and A keys to increase and decrease the apex angle of the prism respectively. This functionality makes the simulation interactive and user-friendly. Banned slider functionality is used to prevent invalid combinations of input values, to stop the incident ray from being above the horizontal.

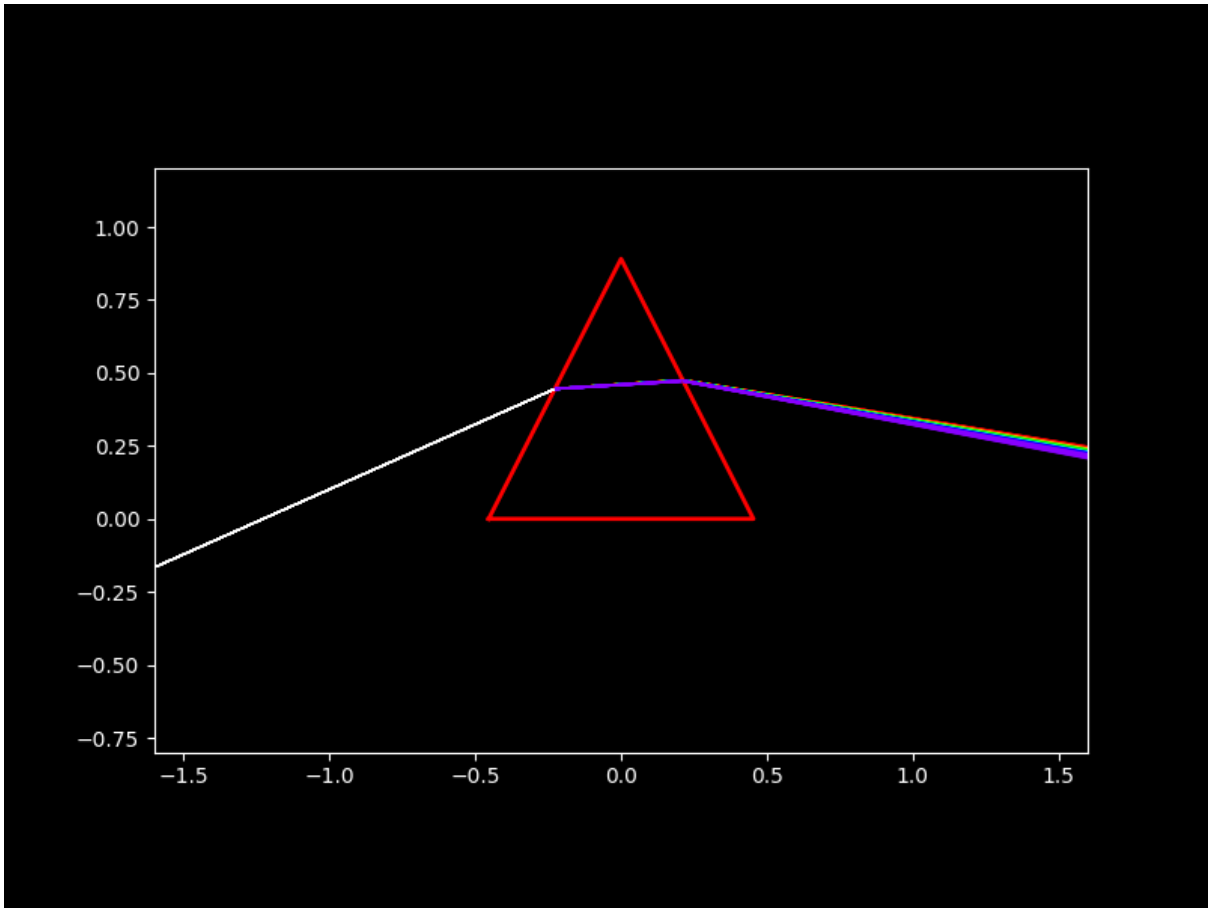


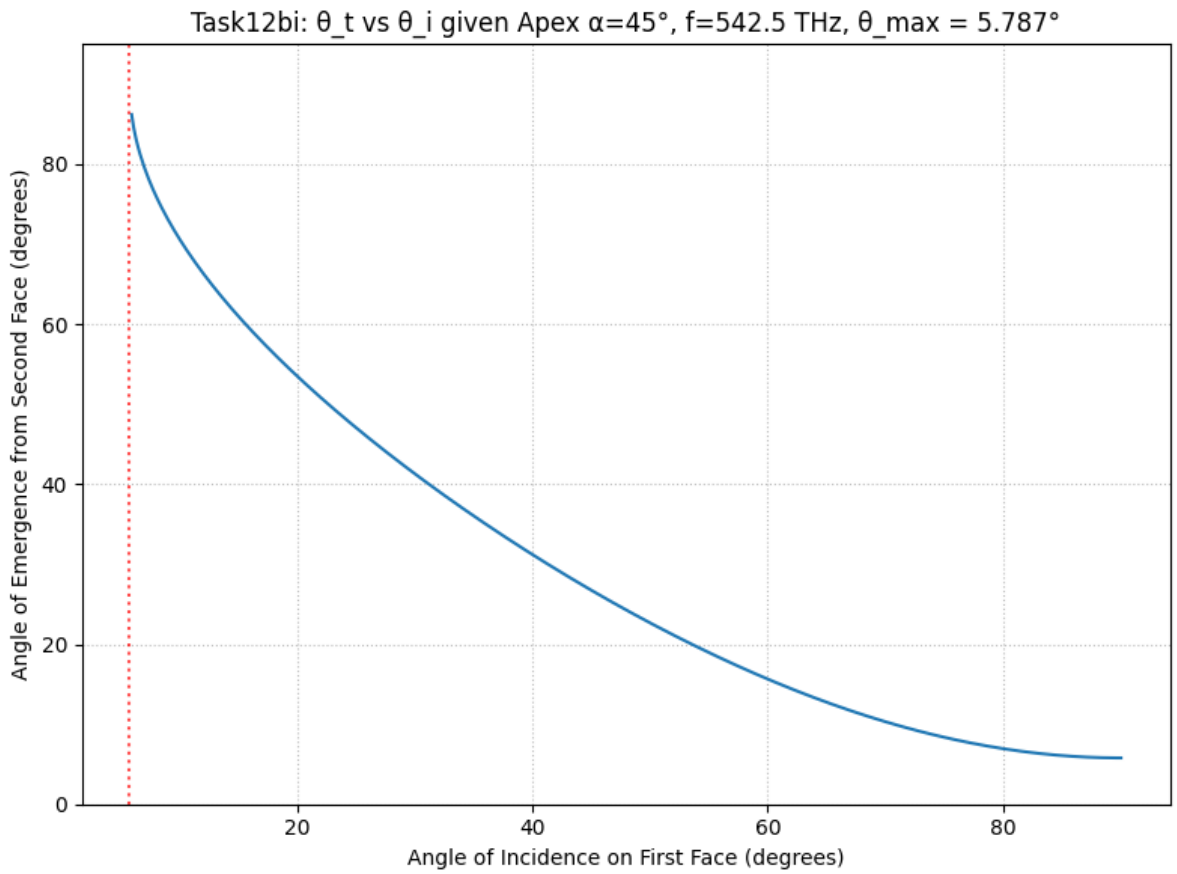
Figure 10: Task 12a Simulation

12.2 Part b

This subtask has three parts to it.
For the first task we were challenged to plot θ_t vs θ_i using the equations

$$\sin\theta_t = \sqrt{n^2 - \sin^2\theta_i} - \sin(\theta_i)\cos\alpha$$
$$n = \sqrt{1 + \sum_k \frac{a_k\lambda^2}{\lambda^2 - b_k}}$$
$$a = [1.03961212, 0.231792344, 1.01146945]$$
$$b = [0.00600069867, 0.0200179144, 103.560653]$$

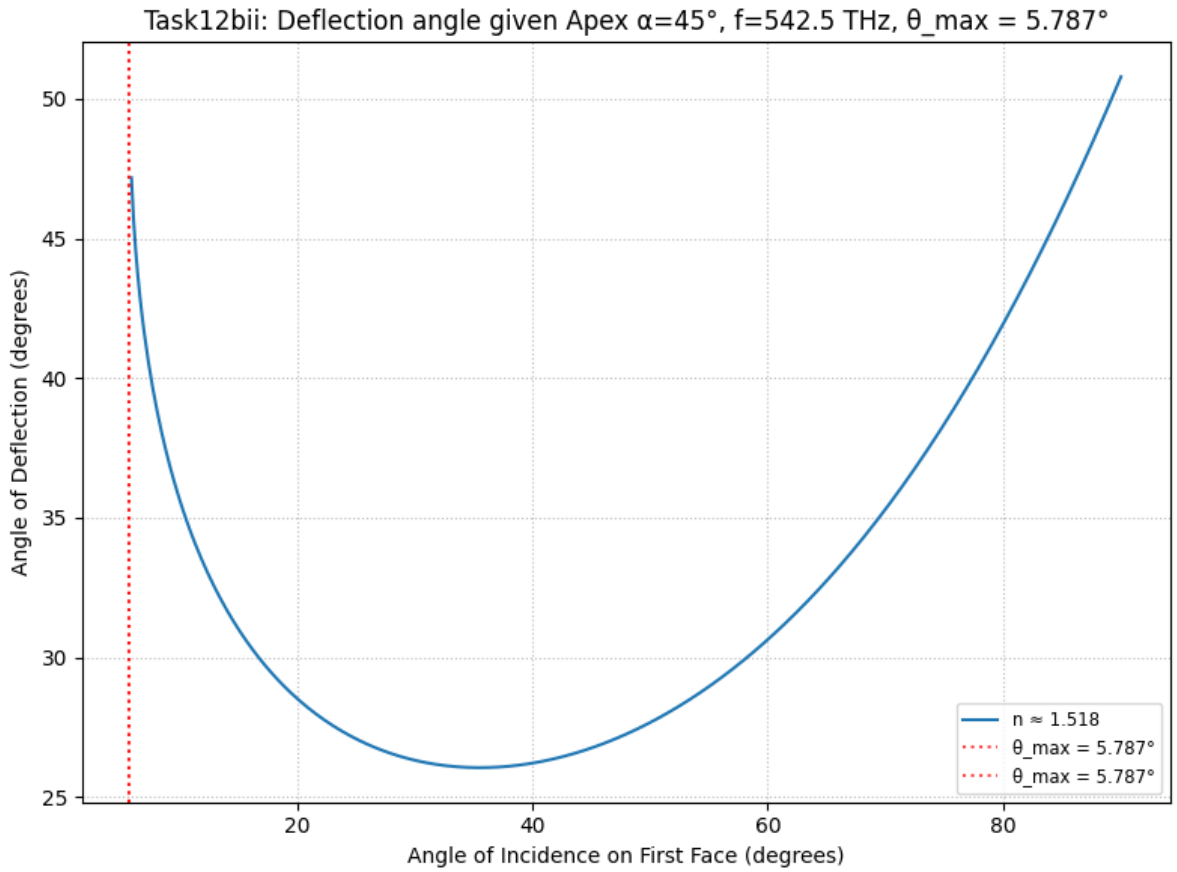
Where $\alpha = 45°$ and frequency $= 542.5$THz. From our graph it can be observed that θ_t reaches its maximum value of $90°$ at $\theta_i = 5.787°$



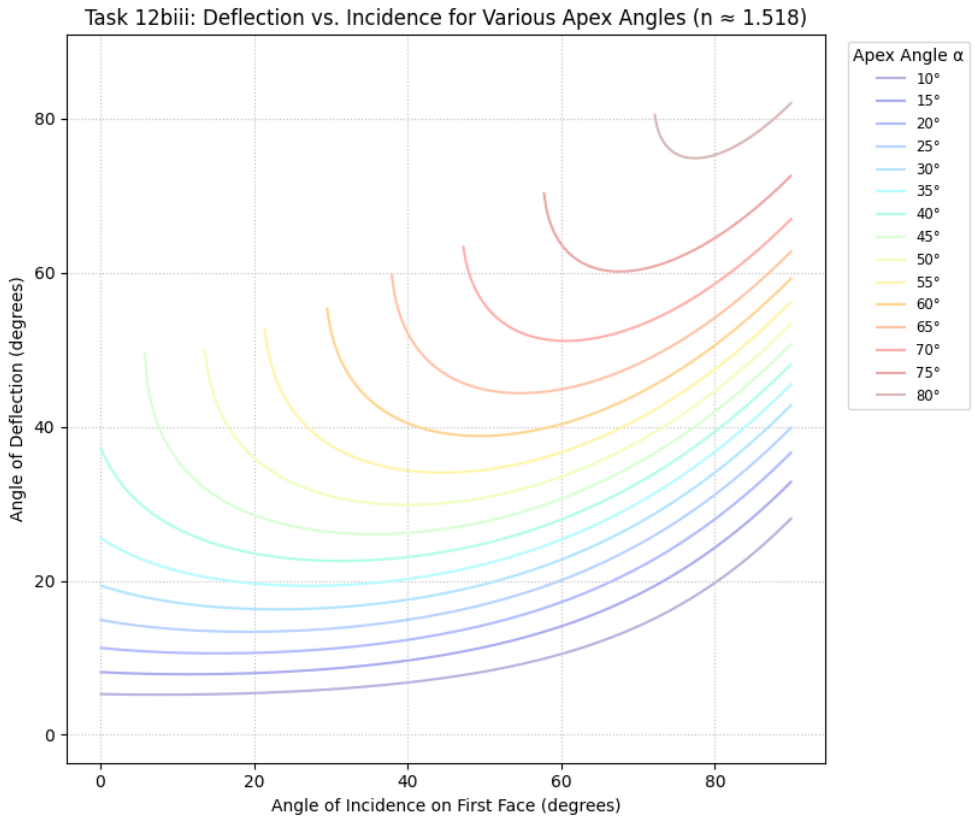
For the second task we were challenged to plot δ vs θ_i using the same equations as in the previous part as well as the equation

$$\delta = \theta_i + \theta_t - \alpha$$

Where $\alpha = 45°$ and $f = 542.5$THz.This was another simple graph plotting task, achieved using Matplotlib



For the third task we were challenged to do the same thing, for α values from 10° to 80° in 5° increments. These were also colour-coded to improve clarity. It can be noticed that as α increases δ also increases, and the θ_i which results in the maximum θ_t also increases.



13 Vision Simulator Website and App Development

13.1 Introduction

We have created the **Vision Optics Simulator and Eye Test Game**, a web application designed to provide an interactive demonstration of the principles of vision correction. The project is built using the Flask web framework and HTML, CSS, and JavaScript. The application features two primary components: a detailed **Vision Simulator** that visualises how lenses correct vision defects, and an **Eye Test Game** that challenges users to diagnose a simulated patient's prescription. The project is self-contained, with all rendering and physics simulations performed directly on the client's browser via the HTML5 Canvas API.

We also created an Android app from this website, which can be downloaded: instructions to access it are listed on the github at [10].

13.2 Libraries and Technologies

The application is built upon a set of core technologies and libraries:

- **Flask Framework**: The backend is powered by Flask, a lightweight Python web framework. Its role is primarily to serve the static HTML, CSS, and JavaScript files and to handle the game's logic through a few API endpoints. All application management logic is contained within a single `main.py` file.
- **HTML5, CSS, and JavaScript**: The entire user interface, interactivity, and visual rendering are built with standard web technologies. There is no server-side plot generation with Matplotlib, unlike in our website which contains our solutions to this year's tasks. Instead:

- **HTML** (`templates/*.html`): Defines the structure for the home page, simulator, and game. It uses Jinja2 templating to receive initial constants from the Flask backend.
- **CSS** (`static/css/style.css`): Provides styling for a clean and responsive user interface.
- **JavaScript** (`static/js/*.js`): Contains all the client-side logic. This is the core of the application, handling user interactions, real-time physics calculations, and rendering all graphics on HTML5 Canvas elements.

13.3 Architecture

The application follows a client-heavy architectural model. The Python backend is lightweight, while the browser-based frontend handles the complex tasks of simulation and rendering.

- **Backend (`main.py`)**:
  - **Routing**: Defines URL routes using Flask's `@app.route` decorator to serve the different pages (`/`, `/simulator`, `/game`).
  - **Game Logic API**: Provides endpoints for the Eye Test Game.
    - `/game/ask_patient`: Receives two test lens powers, compares their effectiveness against a secret, randomly generated patient error, and returns text-based feedback.
    - `/game/submit_guess`: Receives the user's final prescription guess, compares it to the ideal value, calculates a score, and returns the final results.
  - **Session Management**: Uses Flask's session to store game-specific state, such as the patient's actual vision error and the game's start time.
  - **Initial Data Injection**: Passes essential optical constants (e.g., retina distance) to the HTML templates upon initial page load.
- **Frontend (JavaScript)**:
  - **Modularity**: The JavaScript code is logically divided into separate files for maintainability:
    - `simulator_controls.js` & `simulator_draw.js`: Manage the Vision Simulator.
    - `game_logic.js` & `game_diagram.js`: Manage the Eye Test Game.
    - `common.js`: Contains shared utility functions.
  - **Client-Side Rendering**: All visualisations, including ray tracing diagrams, are drawn dynamically on `<canvas>` elements. The JavaScript code calculates the physics and translates the results into graphical representations in real-time. This approach offloads all computational work from the server to the client.
  - **Asynchronous Communication**: The Eye Test Game uses the `fetch` API to communicate with the Flask backend asynchronously, allowing the game to get patient feedback or submit results without a full page reload.

13.4 Functionality: Vision Simulator

The simulator provides a detailed, interactive ray-tracing diagram to demonstrate how vision works.

- **Interface**: The user is presented with controls to adjust the *Simulated Eye's Inherent Error* (in diopters, representing myopia or hyperopia), the *Object Distance*, and the *Correction Mode* (Uncorrected vs. Glasses).
- **Physics Engine (`simulator_draw.js`)**:

13.6.3 Tree 3: Eye Test Game (Client-Side)

14 Optics Tasks Website

14.1 Introduction

We have created a website which contains our solutions to this year's Optics Tasks. Our website is available at [11]. Our website is composed of a single main.py code file as well as some supplementary images. In order to contain our solutions to the tasks, our website incorporates backend logic with Flask, data manipulation with NumPy, and sophisticated plot generation with Matplotlib, all rendered within HTML structures styled with CSS and animated with JavaScript.
We also created an Android app from this website, which can be downloaded: instructions to access it are listed on the github at [11].

14.2 Libraries

The application is built upon several key python libraries:

Flask Framework: The application uses Flask, a lightweight WSGI web application framework in Python. Flask handles routing, request processing, and template rendering. The choice of Flask is suitable for such an application, allowing for rapid development and flexibility, as well as the ability to make the website within a python file which is convenient for displaying matplotlib graphs. All routes, from the main index page to specific task pages and plot generation endpoints, are defined within this single Python file using Flask decorators (app.route).

Matplotlib for Plotting: We have used matplotlib to generate detailed scientific plots for every task. For each task, dedicated Python functions (generate_taskX_plot) create figures, populate them with data (lines, scatter plots, images, patches like polygons and circles), and then save these figures into in-memory byte buffers (io.BytesIO). These buffers are subsequently sent to the client as PNG images. This server-side rendering of plots is especially useful when complex computations are involved, because it allows the website to run at a similar speed regardless of the user's device.

NumPy: Given the scientific nature of the optics tasks, NumPy is employed for efficient numerical computations. This includes creating arrays for plot coordinates (e.g., np.linspace), performing mathematical operations on these arrays (e.g., trigonometric functions, Sellmeier equation calculations), and handling image data as NumPy arrays. NumPy arrays are useful for doing these operations efficiently, which helps make the interactive parts of the website easier to work with due to the reduction in processing times.

Embedded HTML, CSS, and JavaScript: Instead of using separate template files (e.g., .html files in a templates directory, which is standard Flask practice), the HTML, CSS, and JavaScript code for the main page, subtask pages, static plot pages, and the highly interactive task pages are embedded directly into the Python script as multi-line strings (main_page_template, interactive_template, etc.). Flask's render_template_string function is then used to inject dynamic data (like slider configurations) into these string-based templates.

14.3 Architecture

The HTML (Hyper-Text Markup Language) defines the structure of the web pages, including containers for sliders, plot images, navigation buttons, and an image upload form.

The CSS (Cascading Style Sheets) within the <style> tags provides styling for visual presentation, aiming for a clean user interface and incorporating some responsive design principles using media queries to adapt to different screen sizes. The JavaScript is crucial for the interactive aspects. It handles:

- Fetching slider values.

- Debouncing requests to the backend to avoid overwhelming the server during rapid slider changes (requestTimer).
- Updating the plot image dynamically by changing the src attribute of an ¡img¿ tag, appending slider values and a unique request ID as query parameters.
- Displaying loading spinners (spinner, loadingText) during plot generation.
- Client-side validation for certain tasks (e.g., prevent the image overlapping the focus in Task 6, prevent the incident ray hitting the bottom of the prism in Task 12a). This is to prevent the user from entering an invalid combination of slider values.
- Keyboard controls for sliders (arrow keys for sliders, and specific QWAS keys for Task 12a).
- An animation feature for "playable" tasks (e.g., Task 11d), which iteratively updates a slider and refreshes the plot.

Task-Based Structure: We organise our website around the provided tasks. These tasks are defined in two main Python dictionaries:

- task_overview: Contains metadata for all tasks, including titles, descriptions, subtask relationships, and button text. This dictionary drives the navigation on the main page and subtask pages, and the user interface for this navigation.
- interactive_tasks: Specifically defines the configuration for tasks that feature interactive sliders. This includes slider parameters (ID, label, min, max, initial value, step, unit) and the backend endpoint for fetching the plot.

14.4 Initialisation and Configuration

Upon starting, the Flask application is initialised.

Essential configurations like upload folders (UPLOAD_FOLDER, STATIC_FOLDER) and the maximum manual upload dimension (MAX_DIMENSION). Dummy logo and optics images are created in the static folder if they don't exist, ensuring the UI has its visual assets. A default image (DEFAULT_IMAGE_PATH) is loaded and processed using load_and_process_image. This function is central to handling image inputs, normalising them to RGBA format with float values between 0 and 1, and resizing them if necessary. The processed image data is stored in global variables (global_image_rgba, img_height, img_width, img_aspect_ratio).

14.5 Homepage and Navigation

The root route (/) serves the main page, which lists all available tasks as buttons. Clicking a task button navigates the user to either a page for that specific task, or a subtask page if the main task has further divisions. The main page also features an image upload form. Image Upload and Processing:

Users can upload an image via the form on the main page. The /upload route handles this. A unique folder ID is generated per session and uploaded files are stored in a session-specific subdirectory within the UPLOAD_FOLDER. This is a good practice for isolating user data.

The uploaded image is processed by load_and_process_image, updating the global image variables. After a new image is loaded, the configurations for interactive tasks that depend on image dimensions (Tasks 5, 6, and 9) are dynamically updated (e.g., slider max values, step sizes).

14.6 Task Display

Static Tasks: For tasks defined in the static_tasks dictionary (e.g., 1a, 1b, 2), selecting them leads to a page (static_plot_page_template) that displays a pre-rendered plot. The plot itself is generated on demand when the /plot/¡task_id¿ endpoint is hit for that static task.

Interactive Tasks: For tasks in interactive_tasks (e.g., 3, 5, 6, 12a), selecting them loads an interactive page (interactive_template). This page includes:

- Sliders defined by the task's configuration.
- An image element that initially loads a plot based on default slider values.
- JavaScript event listeners that detect slider changes.
- Interactive plot updates:
  - When a user adjusts a slider, the client-side JavaScript captures the new value.
  - To prevent excessive requests, a short delay (debouncing via a 150ms setTimeout call) is used.
  - After the delay, the script constructs a URL for the /plot/¡task_id¿ endpoint, appending all current slider values and a unique request ID (_req_id) as query parameters.
  - The src attribute of the plot image is updated to this new URL, triggering a request to the server.
- A loading spinner is shown while awaiting the new plot from the relevant function.
- Interrupt Handling:
  - The backend /plot/¡task_id¿ route uses a global dictionary active_requests and a check_interrupt function.
  - If a new request for the same task arrives before an older one has finished generating its plot, the older computation is aborted by raising an exception.
  - This ensures that only the plot corresponding to the latest slider settings is eventually displayed, saving server resources and improving responsiveness.

14.7 Plot Generation (Backend)

The `/plot/<task_id>` route is the main function for generating visuals.

It parses slider values from the request arguments.

It then calls the specific generate_taskX_plot function associated with the task_id, providing it the slider values. These generation functions perform the necessary physical simulations and calculations (e.g., ray tracing, lens equations, refractive index calculations) and use matplotlib to draw the plot.

For tasks involving user-uploaded images (Tasks 5, 6+7, 8, 9, 10), the plot generation functions access the global_image_rgba data in order to access the most recently uploaded image by the user.

The generated Matplotlib figure is saved to an io.BytesIO buffer as a PNG image and sent back to the client using send_file.

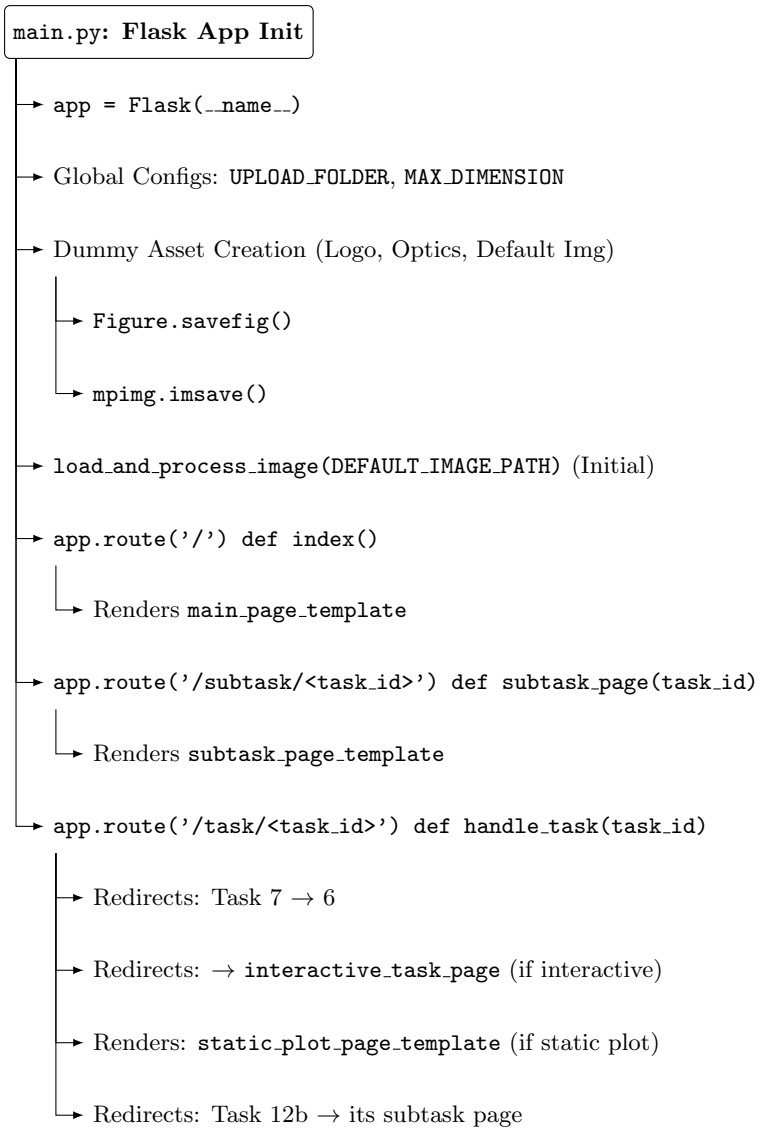
If an error occurs during plot generation, or if the request is interrupted, a generic "Cancelled / Error" image (generate_blank_image()) is returned. During development, the specific error could be identified from the log outputs.

14.8 Deployment

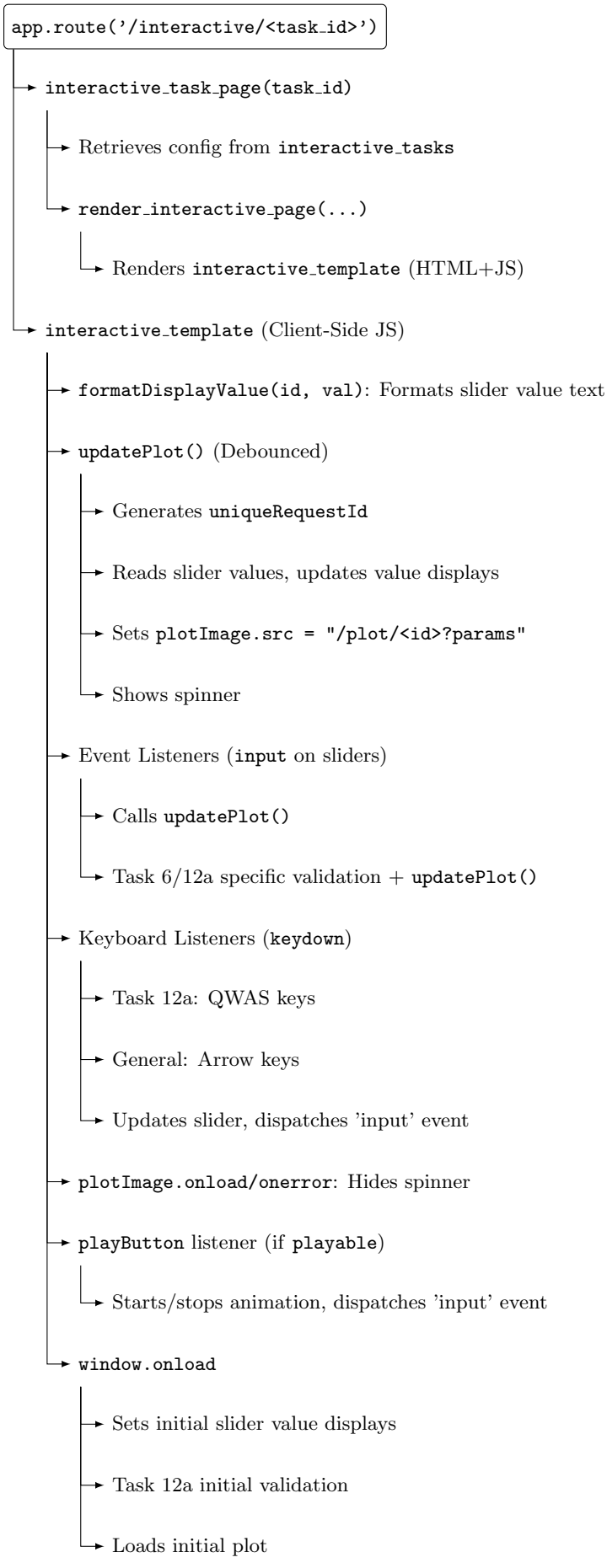
The website was first developed and tested locally, then deployed to render [7]. Now, it has been deployed using railway [8]: this increased the processing power which reduced response time for many task solutions.

14.9 Functionality Forest

14.9.1 Tree 1: Core Application Setup & Main Navigation Routes



14.9.3 Tree 3: Interactive Task Page & Client-Side Logic

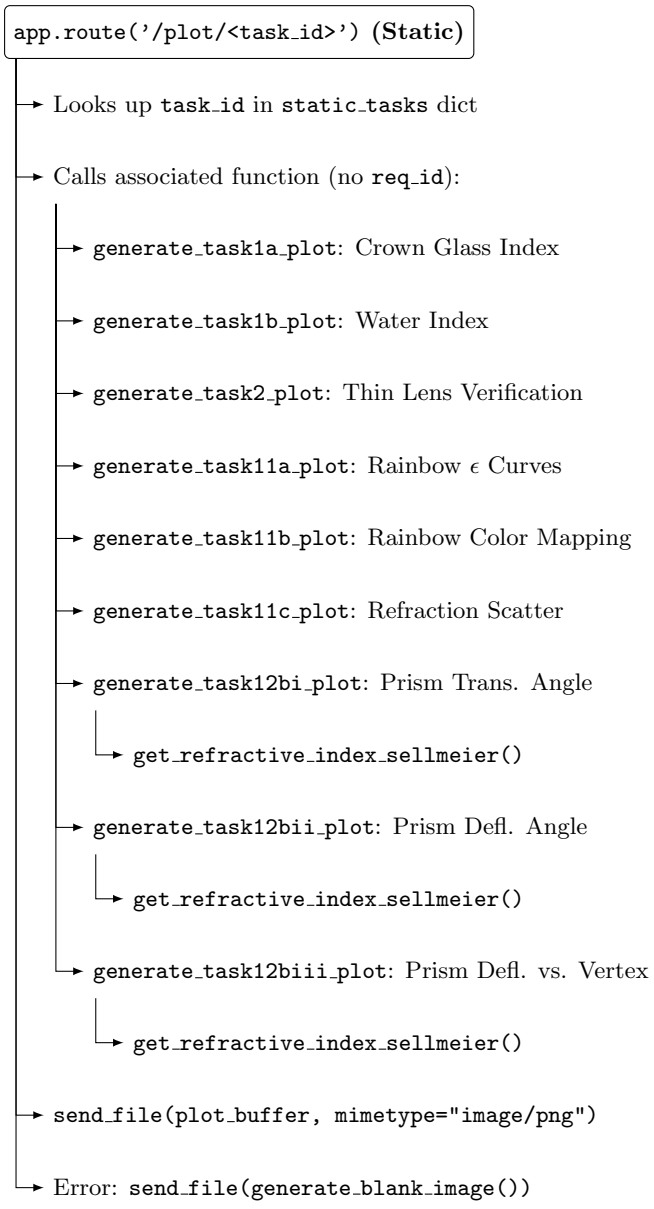


14.9.4 Tree 4: Backend Plot Generation (Interactive Tasks via `/plot/<task_id>`)

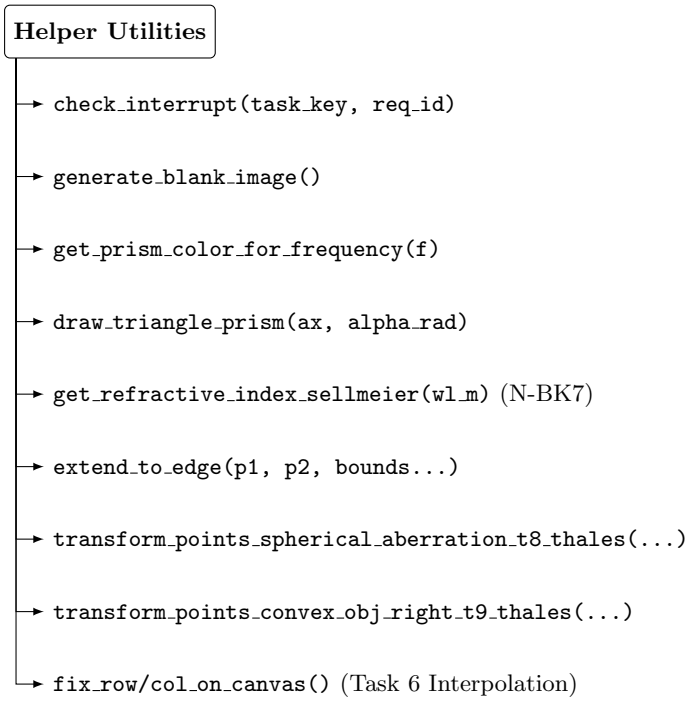
`app.route('/plot/<task_id>')` **(Interactive)**

- Extracts `_req_id`, updates `active_requests`
- Parses slider values from `request.args`
- Calls specific `generate_taskX_plot(..., req_id)`
- `check_interrupt(task_key, req_id)` (Common first call)
- `generate_task3_plot`: Reflection time
- `generate_task4_plot`: Refraction time
- `generate_task5_plot`: Virtual image (uses `global_image_rgba`)
- `generate_task6_plot`: Converging lens (uses `global_image_rgba`)
- `fix_row/col_on_canvas()` (Interpolation)
- `generate_task8_plot_new`: Concave mirror (uses `global_image_rgba`)
  - `transform_points_spherical_aberration_t8_thales()`
- `generate_task9_plot_new`: Convex mirror (uses `global_image_rgba`)
  - `transform_points_convex_obj_right_t9_thales()`
- `generate_task10_plot`: Anamorphic (uses `global_image_rgba`)
- `generate_task11d_plot`: Interactive Rainbows
  - `calculate_rainbow_params()` (inline helper)
- `generate_task12a_plot`: Prism Dispersion
  - `get_refractive_index_sellmeier()`
  - `get_prism_color_for_frequency()`
  - `extend_to_edge()`
  - `draw_triangle_prism()`
- `send_file(plot_buffer, mimetype="image/png")`
- Error/Interrupt: `send_file(generate_blank_image())`

14.9.5 Tree 5: Backend Plot Generation (Static Tasks via `/plot/<task_id>`)



14.9.6 Tree 6: Key Helper Utilities



14.10 Evaluation

The website operates efficient solutions to all tasks, utilising solutions described in earlier sections. The maximum processing time for any task is between 5 and 10 seconds: this allows a quick response. To improve this, we could purchase a premium plan from a cloud-based hosting service to increase the amount of computing power we would have access to. One notable improvement, for similar projects in the future, is in the architecture of the website. Instead of putting all of our code into one file, we should have split this code up into different files for each plotting function, and main files for the UI and interaction management. This would increase efficiency in debugging and implementing new features. Additionally, such an arrangement would allow us to utilise Large Language Models such as GPT-o3-mini and Gemini-2.5-Pro-Experimental for development more efficiently – with our current arrangement, it became very difficult to implement sections using them starting when the code was over 2000 lines long. In our current project, we utilised these models for parts of the software development, but had to refactor the code multiple times, manually debugging and rewriting certain sections. Our main uses for Large Language Models were in implementing core HTML, CSS and Javascript interactions, as well as converting our sliders from `ipywidgets` to HTML sliders. Creating detailed specifications for each section that could be fed into Large Language Models would have significantly increased the speed of development and reduced the accural of bugs within the code.

14.11 Website Links

- Railway (recommended): `https://bpho-computational-challenge-production.up.railway.app/`
- Render (not recommended): `https://bpho-computational-challenge.onrender.com/`
- GitHub: `https://github.com/AnangoPrabhat/BPhO-Computational-Challenge`

15 References

1. Hunter JD. Matplotlib: A 2D Graphics Environment. Computing in Science Engineering 2007; 9:90–5. DOI: 10.1109/MCSE.2007.55
2. Harris CR, Millman KJ, Walt SJ van der, Gommers R, Virtanen P, Cournapeau D, Wieser E, Taylor J, Berg S, Smith NJ, et al. Array programming with NumPy. Nature 2020; 585:357–62
3. Ronacher A and Pallets team the. Flask. https://flask.palletsprojects.com/. 2010
4. team TJW. ipywidgets: Interactive Widgets for the Jupyter Notebook. https://ipywidgets.readthedocs.io/en/latest/. 2015
5. Clark A and Pillow contributors the. Pillow (PIL Fork). https://python-pillow.org/. 2010
6. Walt S van der, Schönberger JL, Nunez-Iglesias J, Boulogne F, Warner JD, Yager N, Gouillart E, and Yu T. scikit-image: image processing in Python. PeerJ 2014; 2:e453. DOI: 10.7717/peerj.453. Available from: https://doi.org/10.7717/peerj.453
7. Render. https://render.com. Accessed: 2025-05-25. 2025
8. Railway. https://railway.app. Accessed: 2025-06-22. 2025
9. Capacitor: Cross-platform native runtime for web apps. Accessed on June 15, 2025. Available from: https://capacitorjs.com/ [Accessed on: 2025 Jun 15]
10. Prabhat A and Swanson T. Vision Optics Simulator Eye Test Game. https://github.com/AnangoPrabhat/BPhO-Computational-Challenge. Accessed: 2025-06-22. 2025
11. Prabhat A and Swanson T. BPhO-Computational-Challenge. https://github.com/AnangoPrabhat/BPhO-Computational-Challenge. Accessed: 2025-06-22. 2025