Ray tracing

Tom Spencer

15:12, Wednesday 29th September, 2021

1   Introduction

Aiming to make a program for tracing light rays through various optical components such as lenses, prisms and mirrors. Hope to implement it in Python first, then re-implement it in C++ and finally get Python running it in C++ (might need to use Cython or C to do this).

It allows me to explore various object orientated programming concepts in C++ such as polymorphism and virtual functions.

2   Program implementation

We define a base class called `component` from which each type of component derives. E.g. we have derived classes including `mirror` and `lens`. The base class declares the methods `test_hit()` which tests if a light ray interacts with the component and `hit()`, which performs the interaction. In Python we also have a `plot()` method that plots the component to the current plot.

All components present in an optical system are simply stored in a Python list.

2.1   C++ complications

In C++ there isn't a simple way of storing all our components in a list like container such as `std::vector` as all elements must be of the same type. Without cheating and using something like `boost::any`, this can be overcome by making use of polymorphism.

Any sort of component (such as a mirror or lens) is described by a class (`Mirror` and `Lens`) that derives from a `Component` base class. The `Component` class defines the `test_hit()` and `hit()` methods as virtual functions. These are then overridden in the derived classes.

C++ allows us to create a pointer `ptr` of type `*Component` to a derived class such as `Mirror`. Suppose `ptr` points at an instance of `Mirror`. Since `hit()` is a virtual finction, when we call it using `ptr->hit()`, the `hit()` method in the `Mirror` class is called. This is because the `hit()` defined in `Mirror` is 'more derived' than the `hit()` in `Component` and when a virtual function is called, the implementation that is executed corresponds to the 'most derived.' Equivalently if `ptr` was pointing at an instance of `Lens`, it would call the `hit()` method defined in `Lens`.

We then store all the components in a `std::vector` of type `*Component`. All elements in the vector are then the same type, even though they point to different types of component. To avoid having to manage the memory it would probably be a good idea to use a smart pointer like `std::unique_ptr` rather than a raw pointer.

2.2 Memory structures

We know now how to store and describe the system of optical components we wish to trace the rays through. Now we need to work out how to structure the rays themselves.

In the original pure Python implementation, we used a 3 dimensional `numpy` array, with the first index identifying the ray, the second the interaction point and the third discriminating x and y coordinate. `numpy` allows us to easily slice the array to extract specific components. In C++, it doesn't seem we can do this. There would seem to be two options:

- Create a class to define a 3d array

  - Has the advantage memory is all in one place

  - Can directly interact with a `numpy` array when using python

- Create a `Ray` class that contains vectors of x and y coordinates

  - Memory is all over the place

  - Would need to copy everything into a `numpy` array if we wanted to do that

It would be easier to create a `Ray` class to keep track of everything and then wrap it in `Cython`. We can always add a function to convert it to a `numpy` array and we're never going to be dealing with large numbers of rays, so memory probably isn't an issue. It also means we just need to pass a reference to the ray to any function and the function can work out get whatever it wants.

We can create a `Ray` class, with member variables describing the current direction of the ray as a `std::array<double, 2>` and the interaction points as a `std::vector<std::array<double, 2>>`. This has the benefit that a single mathematical vector is described as a `std::array<double, 2>`, and this can easily be passed to functions. Also the position data of a single ray is contiguous. All the rays are then stored in a vector.

When initialising the `Ray` class, we should reserve the memory required for the vector, depending on how many tracings we want to do. When we compute the new position, we can then just `push_back` the answer into the vector.

2.3 Final

Each ray is described by an instance of a `Ray` class. This includes:

- A `std::array<double, 2>` used to describe the direction of the ray

- A vector of `std::array<double, 2>` used to describe the initial and interaction points of the ray

Each component ultimately derives from the `Component` class. This declares:

- A `test_hit()` method that tests whether a ray interacts with the component. Takes a reference to a instance of `Ray` as an argument. Returns positive infinity if the ray does not interact, otherwise the time to interaction

- A `hit()` method that performs the interaction. Takes a pointer to a instance of `Ray` and maximum number of interactions to perform as arguments. Doesn't return anything, modifies the existing `Ray` instance.

- Several helper functions for performing common tasks:

– Rotate function. Takes a `std::array<double, 2>` and rotates it by $-\theta$. Equivalent to rotating to a coordinate system rotated by θ

– A method `compute_t()` which returns the time for a ray to reach a certain position. Assumes the position does lie along the ray's path. Uses the larger of v_x or v_y in magnitude so it so numerically stable.

From this three helper classes are derived. These aren't meant to be initialised directly and are still abstract classes, but help in the construction of classes that are. They are:

- `Plane`

  – Describes a simple planar component

  – Declares `start` and `end` member variables to define the start and end points of the plane

  – Defines a constructor to initialise the class with given start and end points

  – Implements the `test_hit()` method to test for a ray hitting the plane

- `Spherical`

  – Describes a circular component or arc

  – Declares member variables for describing centre of circle/arc, its radius, and start/end angles for defining arc

  – Defines constructor for these variables

  – Implements the `test_hit()` method to test for a ray hitting the circle/arc

- `Complex_Component`

  – Describes a complex component composed of several other components

  – Declares a vector of type `std::shared_ptr<Component>` to store the sub-components of this component. We use `std::shared_ptr<Component>` as Cython wrappers also need to hold onto a pointer

  – Implements both the `test_hit()` and `hit()` method:

    * For `test_hit()`, we simply need to iterate over all the sub-components, return positive infinity if there are no interactions, otherwise the smallest positive time

    * For `hit()`, we wish to trace a ray through these sub-components with a maximum number of interactions

    * This is the same as the original problem, so we can just call the `trace_ray()` function that will do this for us

From these there are five basic components implemented:

- `Mirror` - plane mirror

- `Mirror_Sph` - spherical mirror

- `Refract_Plane` - refraction at a planar boundary

- `Refract_Sph` - refraction at a spherical boundary

- `Screen_Plane` - a planar, absorbing screen

3

Once these basic components have been implemented, complex components such as prisms or lenses should be straightforward to implement. All that would need to be implemented would be a constructor that constructs the sub-components and adds them to the vector of sub-components.

To perform the tracing, there are two relevant functions:

- `trace_ray()` traces a single ray through a number of components, up to a maximum number of iteration, n. Takes a reference to a `Ray` instance, vector of `std::shared_ptr<Component>` and maximum number of iterations as arguments. Works directly on the ray instance

- `trace()` takes a vector of rays and traces them through the components given using `trace_ray()`

3 Determining the intersection of a ray with a surface

We shall always consider the light ray in question to start at $\boldsymbol{r}_0 = x_0\hat{\mathbf{i}} + y_0\hat{\mathbf{j}}$ and follow a path,

$$\boldsymbol{r}(t) = \boldsymbol{r}_0 + \boldsymbol{v}t, \tag{1}$$

where $\boldsymbol{v}$ is a unit vector indicating the direction of travel. t is a time-like quantity and for an interaction we must have $t > 0$ for the interaction to be in the future. If there are multiple components that could interact we select the one with the smallest t.

We first determine expressions for computing where a ray interested with either a planar or spherical surface, since all components possess a combination of planar and spherical surfaces.

3.1 Ray intersection with a planar surface

We define a plane as starting at $\boldsymbol{r}_1$ and ending at $\boldsymbol{r}_2$. It is described by,

$$\boldsymbol{r}_{\mathrm{m}}(t') = \boldsymbol{r}_1 + (\boldsymbol{r}_2 - \boldsymbol{r}_1)t', \tag{2}$$

where $0 \leq t' \leq 1$ for a point to be on the mirror.

Furthermore, we define two unit vectors. We define $\hat{\boldsymbol{D}}$ as a unit vector pointing along the plane from its start point to its end point,

$$\hat{\boldsymbol{D}} = \frac{\boldsymbol{r}_2 - \boldsymbol{r}_1}{|\boldsymbol{r}_2 - \boldsymbol{r}_1|}. \tag{3}$$

From this we define the normal unit vector $\hat{\boldsymbol{n}}$ as,

$$\hat{\boldsymbol{n}} = -D_y\hat{\mathbf{i}} + D_x\hat{\mathbf{j}}, \tag{4}$$

where D_x and D_y are the x and y components of $\hat{\boldsymbol{D}}$ respectively. These are shown in Fig. 1.

We can derive where a ray intersects as obeying,

$$t' = \frac{v_y(x_1 - x_0) - v_x(y_1 - y_0)}{v_x(y_2 - y_1) - v_y(x_2 - x_1)}, \qquad t = \frac{x_0(-y_1 + y_2) - x_1 y_2 + x_2 y_1 + y_0(x_1 - x_2)}{v_x(y_1 - y_2) + v_y(-x_1 + x_2)} \tag{5}$$

Remember we require both a unique solution for t to exist and $0 \leq t' \leq 1$ for it to intersect the plane. We have to check the denominator is not zero in both cases.
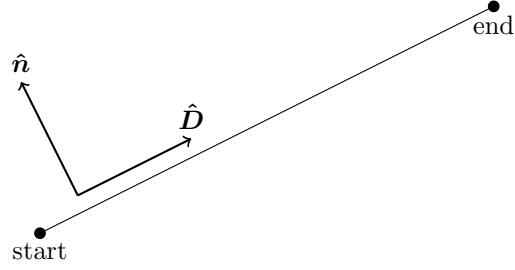
4

Figure 1: Structure of a plane, along with the unit vectors $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{n}}$.
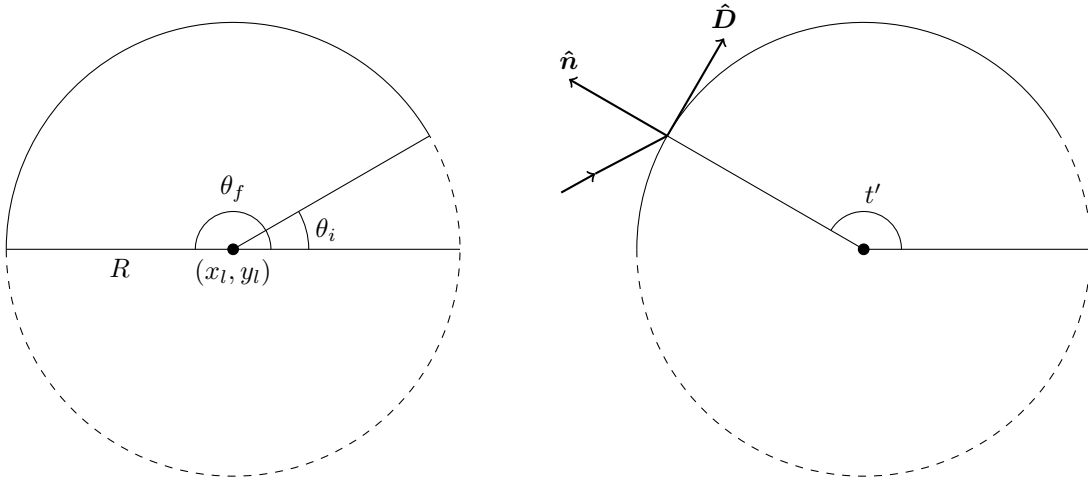

Figure 2: Structure of a circular arc, along with the unit vectors $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{n}}$ as defined for a circular arc for a given ray intersection.

3.2 Ray intersection with a circular surface

We consider a circle with centre (x_l, y_l) and radius R as shown in Fig. 2. In general, only a portion of its surface is modelled with $\theta_i \leq t' \leq \theta_f$. Here t' is a parameter in that takes the place of the polar angle. Its surface can be described by,

$$\boldsymbol{r}_{\rm s}(t') = (x_l + R\cos t')\hat{\mathbf{i}} + (y_l + R\sin t')\hat{\mathbf{j}}. \tag{6}$$

We define a normal unit vector $\hat{\boldsymbol{n}}$ that points radially outwards. We define the unit vector $\hat{\boldsymbol{D}}$ to be orthogonal to $\hat{\boldsymbol{n}}$, in a manner consistent with that of a plane and Eq. 4, namely,

$$\hat{\boldsymbol{D}} = n_y\hat{\mathbf{i}} - n_x\hat{\mathbf{j}}. \tag{7}$$

When determining if a ray intersects an arc, we have the simultaneous equations,

$$x_0 + v_x t = x_l + R\cos t' \tag{8}$$
$$y_0 + v_y t = y_l + R\sin t'. \tag{9}$$

Letting $\Delta x = x_0 - x_l$, $\Delta y = y_0 - y_l$ we get,

$$\Delta x + v_x t = R\cos t' \tag{10}$$
$$\Delta y + v_y t = R\sin t'. \tag{11}$$

We can solve for either t or t'. Originally we solved for t', but it is more efficient to solve for t instead.

3.2.1 Solving for t

We can eliminate t' by squaring Eqs. 10 & 11, and utilising $\sin^2\theta + \cos^2\theta = 1$. This produces the quadratic,

$$0 = (v_x^2 + v_y^2)t^2 + 2(v_x\Delta x + v_y\Delta y)t + \Delta x^2 + \Delta y^2 - R^2, \tag{12}$$
$$= t^2 + 2\gamma t + \Delta x^2 + \Delta y^2 - R^2, \tag{13}$$

where we have uses the fact $\boldsymbol{v}$ is normalised and defined $\gamma = v_x\Delta x + v_y\Delta y$. The solutions for t can be found as,

$$t = -\gamma \pm \sqrt{\gamma^2 + R^2 - \Delta x^2 - \Delta y^2}. \tag{14}$$

We can the compute the position of intersection, if there is one, and then use `atan2()` to compute t'. Trigonometric functions are only called when the ray has intersections in the future ($t > 0$). By comparison, arccos is called for all possible intersections when solving for t'. sin may also be called there could be spurious t' solutions that don't lie along the ray's path, at least using the method below. This leads to solving for t being a more efficient method.

If we do not need the value of t', we can avoid the call to `atan2()` and improve performance further. The new (non-rotation) methods for performing rotation and refraction do not require t', allowing for a reasonable improvement. First consider the start and end points of the arc,

$$r_{\rm start} = R\cos\theta_i\hat{\mathbf{i}} + R\sin\theta_i\,\hat{\mathbf{j}}, \qquad\qquad r_{\rm end} = R\cos\theta_f\hat{\mathbf{i}} + R\sin\theta_f\hat{\mathbf{j}}. \tag{15}$$

We can then rotate our coordinate system clockwise by θ_i, with coordinates (x', y'). We can test if the intersection point, $\boldsymbol{P}$, lies on the arc by comparing its components in this coordinate system to that of $\boldsymbol{r}_{\rm end}$. There are two cases as illustrated in Fig. 3.
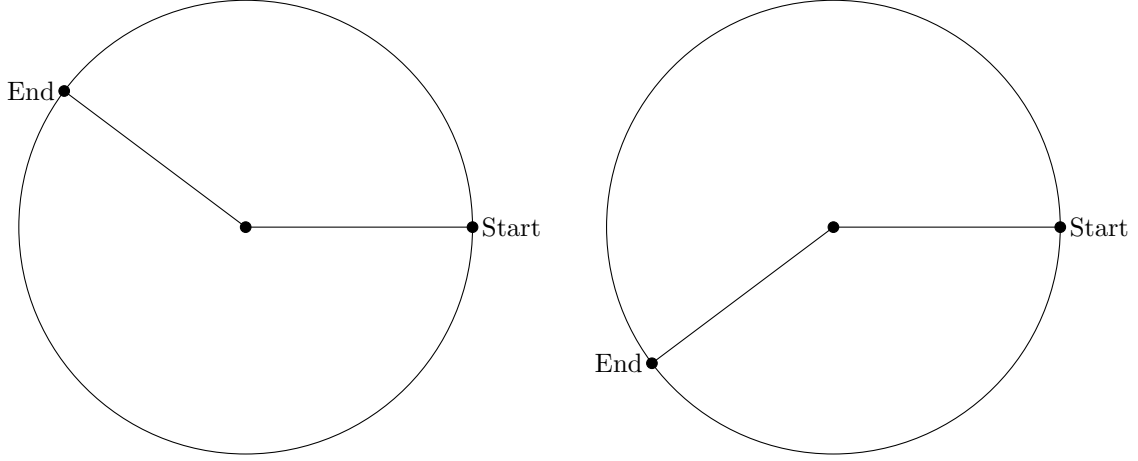
Figure 3: The two cases to consider when testing if a point is between θ_i and θ_f.

In the first case, we have $r'_{\text{end},y} \geq 0$. If $P_y \geq 0$ and $P_x \geq r'_{\text{end},x}$ then the ray does intersect the arc, otherwise it does not.

In the second case, we have $r'_{\text{end},y} < 0$. $P_y \geq 0$ is sufficient to know the ray does intersect the arc. If $P_y < 0$, we must have $P_x \leq r'_{\text{end},x}$ for intersection.

3.2.2   Solving for t'

This section is kept for reference with earlier versions of the code. Eliminating t from Eqs. 10 & 11 produces,

$$v_y \Delta x - v_x \Delta y = R v_y \cos t' - R v_x \sin t'. \tag{16}$$

Defining $\delta = v_x \Delta y - v_y \Delta x$, we can eliminate $\sin t'$,

$$-\left(\delta + R v_y \cos t'\right)^2 = R^2 v_x^2 \sin^2 t' \tag{17}$$
$$= R^2 v_x^2 (1 - \cos^2 t'). \tag{18}$$

This is a quadratic in $\cos t'$ with coefficients,

$$a = R^2 \left(v_x^2 + v_y^2\right), \tag{19}$$
$$b = 2 R v_y \delta, \tag{20}$$
$$c = \delta^2 - R^2 v_x^2, \tag{21}$$

and solutions,

$$\cos t' = \frac{1}{R}\left(-v_y \delta \pm \sqrt{\delta^2\left(v_y^2 - 1\right) + (v_x R)^2}\right), \tag{22}$$

This will generate four solutions; two due to the $\pm$ and a further two since if t' is a solution, then so is $2\pi - t'$. We will need to determine to correct solution for the intersection. The solution we are after, if one exists must satisfy the following conditions:

- $\cos t'$ must be in the range $-1 \leq \cos t' \leq 1$, otherwise the ray never intersects the surface

- t' are in the necessary range for the surface, $\theta_i \leq t' \leq \theta_f$

- Determine which solutions are true solutions and lie along the light ray's path

- If there are two solutions left, choose the solution with the smaller but non-negative t parameter

4 Components

Ultimately, we would like to provide implementations for the following components:

- Mirrors, plane and spherical

- Refraction components, plane and spherical

- Lenses

- Prisms

- Absorbing screen

We will first provide five basic components from which most other components we might be interested in can be described. These will be two that perform reflection/refraction at a planar surface and two that do the same but at a spherical (circular) surface.

5 Reflecting basic components

5.1 Plane mirrors

We define a plane mirror as starting at $\boldsymbol{r}_1$ and ending at $\boldsymbol{r}_2$. It is reflecting on both sides and is described by,

$$\boldsymbol{r}_{\mathrm{m}}(t') = \boldsymbol{r}_1 + (\boldsymbol{r}_2 - \boldsymbol{r}_1)t', \tag{23}$$

where $0 \leq t' \leq 1$ for a point to be on the mirror. Once we have determined that a ray will intersect with a plane mirror, we need to determine how the ray will reflect. We can compute the new starting position for the ray using the standard equation describing the ray as we have t. To compute the new direction, we have that the angle of incidence is equal to the angle of reflection. We have two methods: the rotation method and vector method. Originally the code used the rotation method and its details are kept for reference. The vector method is now used as it is faster.

5.1.1 Rotation method

To find the new velocity, we compute the angle, α, the plane makes with the x axis,

$$\tan \alpha = \frac{y_2 - y_1}{x_2 - x_1}. \tag{24}$$

Now we must be careful with the cases of $\alpha = \frac{\pi}{2}, \frac{3\pi}{2}$ since the tangent is infinite in these cases. Moreover as tangent is π periodic, arctan will give the correct answer up to $\pm \pi$. Fortunately `numpy` has the function `arctan2(y, x)` which specifically accounts for these two cases, and also will always return the correct sign. In C++, the `cmath` header provides `atan2()`.

We can now perform the reflection by rotating our coordinates by α to coordinates (x', y'). In these coordinates the ray is normal to the plane so the reflection is then simply a case of negating

the y' component. The standard rotation matrix to rotate a point anticlockwise from (x, y) to $x', y')$ by θ is,

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}. \tag{25}$$

However, this rotates a point anticlockwise, we want to rotate our coordinates, so we require a minus sign for our matrix,

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}. \tag{26}$$

Once we've negated the y' component, we can then rotate back using $R(-\theta)$ to get the new direction of the ray.

The disadvantage of this method is performance. It involves call to trigonometric functions which are slow, especially `atan2(y, x)`. Fortunately we can avoid this.

5.1.2 Vector method

We don't really need to rotate the plane each time we perform a reflection. All we really want to do is reflect the component of $\boldsymbol{v}$. We can make use of $\hat{\boldsymbol{D}}$, the normal vector that points along the plane. The normalisation of $\hat{\boldsymbol{D}}$ does involve square rooting, but it only needs to be computed once. The component of $\boldsymbol{v}$ perpendicular plane is $\boldsymbol{v} - (\boldsymbol{v} \cdot \hat{\boldsymbol{D}})\hat{\boldsymbol{D}}$ and so the reflections ca be performed as,

$$\boldsymbol{v} \to (\boldsymbol{v} \cdot \hat{\boldsymbol{D}})\hat{\boldsymbol{D}} - \boldsymbol{v}. \tag{27}$$

We could use $\hat{\boldsymbol{n}}$ instead and derive a similar formula.

A disadvantage of this method is $|\boldsymbol{v}|$ drifts from 1 over time. As all the maths assumes $|\boldsymbol{v}| = 1$, this quickly becomes catastrophic after as little as 15-20 interactions. This is easily resolved by re-normalising $\boldsymbol{v}$ after each interaction. Since we expect $|v - 1| \ll 1$, we can avoid calls to `hypot()` by using a Taylor expansion. Denoting $S = v_x^2 + v_y^2$, expanding around $S = 1$ produces,

$$\frac{1}{\sqrt{S}} = 1 - \frac{1}{2}(S - 1) + \mathcal{O}\left((S - 1)^2\right). \tag{28}$$

Working to $\mathcal{O}(S - 1)$ should be sufficient.

5.2 Spherical mirror

To reflect at a spherical surface, we follow a broadly similar method to that of a planar surface.

5.2.1 Old method

First we determine the t, t' values where the ray intersects the surface. The intersection location is given by,

$$x = x_s + R\cos t', \qquad\qquad y = y_s + R\sin t', \tag{29}$$

for a surface with centre (x_s, y_s).

We must then determine the angle α as before. We can easily compute α with the help of a little geometry as we know the angle t' where the intersection occurs. We then have,

$$\alpha = t' - \frac{\pi}{2}. \tag{30}$$

After that it is identical to the plane mirror. We use Eq. 26 to rotate the axis by α, negate the y' component, and finally rotate back to get the new direction.

5.2.2 New method

The new method applies a similar method to the vector method for a plane. We use the normal vector of the circle to write the reflections as,

$$\boldsymbol{v}_f = \boldsymbol{v}_i - 2(\boldsymbol{v}_i \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}. \tag{31}$$

6 Refracting basic components

Refraction is based on Snell's law, that the angle of incidence, θ_i, and the angle of refraction, θ_f, obey,

$$n_i \sin \theta_i = n_f \sin \theta_f, \tag{32}$$

where both angles are measured from the normal of the surface. n_i and n_f are the refractive indices of the initial and final media. A complication of refraction is the possibility of total internal reflection. This occurs when,

$$\sin \theta_f = \frac{n_i \sin \theta_i}{n_f} > 1. \tag{33}$$

The value of θ_i for which $\sin \theta_f = 1$ is known as the critical angle. It is,

$$\theta_c = \arcsin \frac{n_f}{n_i}. \tag{34}$$

6.1 Refraction at a planar surface

We define the refractive indices as n_1 and n_2, with n_1 on the side $\hat{\boldsymbol{n}}$ points towards. In our old method, we rotated our system as it was convenient, though slower. We rotate our coordinates round by α (the angle of the plane) so that the normal lies along the y-axis. We then have,

$$\theta_i = \frac{\pi}{2} - \theta', \tag{35}$$

where θ' is the angle of $\boldsymbol{v}'$, that is $\boldsymbol{v}$ in the rotated coordinates. Applying Snell's law to compute θ_f, the new direction of the ray in the rotated coordinates is,

$$\boldsymbol{v}' = \cos\left(\frac{\pi}{2} - \theta_f\right)\hat{\boldsymbol{i}}' + \sin\left(\frac{\pi}{2} - \theta_f\right)\hat{\boldsymbol{j}}'. \tag{36}$$

Finally we then just have to rotate back to our original coordinates. If total internal reflection is determined to happen, then we simply need to reflect the ray at the point of intersection.

6.1.1 New method

The old method calls trigonometric functions including `sin()` and `atan2()`. If we were working in three dimensions we could write,

$$n_{\mathrm{i}}\hat{\boldsymbol{n}} \times \boldsymbol{v}_{\mathrm{i}} = n_{\mathrm{f}}\hat{\boldsymbol{n}} \times \boldsymbol{v}_{\mathrm{f}}, \tag{37}$$

which produces Snell's law when we take it's magnitude. Working in the xy plane, Eq. 37 produces,

$$n_{\mathrm{i}}(n_x v_{\mathrm{iy}} - n_y v_{\mathrm{ix}}) = n_{\mathrm{f}}(n_x v_{\mathrm{fy}} - n_y v_{\mathrm{fx}}). \tag{38}$$

Imposing the condition $|\boldsymbol{v}_\text{f}| = 1$ and defining $\gamma = \frac{n_\text{i}}{n_\text{f}}(n_x v_\text{iy} - n_y v_\text{ix})$ produces the simultaneous equations,

$$n_x v_\text{fy} - n_y v_\text{fx} = \gamma, \tag{39}$$
$$v_\text{fx}^2 + v_\text{fy}^2 = 1, \tag{40}$$

which have the solutions,

$$v_\text{fx} = -n_y \gamma \pm n_x \sqrt{1 - \gamma^2}, \tag{41}$$
$$v_\text{fy} = n_x \gamma \pm n_y \sqrt{1 - \gamma^2}. \tag{42}$$

Now consider when the discriminant $1 - \gamma^2$ is less than zero. This occurs for $\gamma > 1$ and $\gamma < -1$. Taking the magnitude of Eq. 37, γ can be written in terms of the angle the incident ray makes to the normal of the plane as $\gamma = \frac{n_\text{i}}{n_\text{f}} \sin\theta$. We therefore do not have a real solution for $\boldsymbol{v}_\text{f}$ when,

$$\sin\theta > \frac{n_\text{f}}{n_\text{i}} \quad \text{or} \quad \sin\theta < -\frac{n_\text{f}}{n_\text{i}}. \tag{43}$$

This corresponds to total internal reflection occurring. Therefore if the discriminant $1 - \gamma^2 < 0$, we should perform reflection instead of refraction.

Now we determine how to assign n_i and n_f from the n_1 and n_2 parameters of the plane. The definition of $\hat{\boldsymbol{n}}$ in Eq. 4 means $\hat{\boldsymbol{n}}$ points to the 'left' of the plane when viewed from its start point towards its end. We can determine which side the ray originates on by considering $\boldsymbol{v}_\text{i} \cdot \hat{\boldsymbol{n}}$,

$$\boldsymbol{v}_\text{i} \cdot \hat{\boldsymbol{n}} > 0 \implies n_\text{i} = n_2, n_\text{f} = n_1, \tag{44}$$
$$\boldsymbol{v}_\text{i} \cdot \hat{\boldsymbol{n}} < 0 \implies n_\text{i} = n_1, n_\text{f} = n_2. \tag{45}$$

This leaves the case of $\boldsymbol{v}_\text{i} \cdot \hat{\boldsymbol{n}} = 0$ where the ray travels along the plane. This should be considered in future.

We must still determine which of the possible solutions in Eqs. 41 & 42 are physical. We start by checking $\boldsymbol{v}_\text{f}$ is normalised. Taking the same sign produces,

$$|\boldsymbol{v}_\text{f}|^2 = (-n_y \gamma \pm n_x \sqrt{1 - \gamma^2})^2 + (n_x \gamma \pm n_y \sqrt{1 - \gamma^2})^2, \tag{46}$$
$$= (n_x^2 + n_y^2)\gamma^2 + (n_x^2 + n_y^2)(1 - \gamma^2), \tag{47}$$
$$= n_x^2 + n_y^2, \tag{48}$$
$$= 1. \tag{49}$$

Therefore choosing both positive or both negative will always be correctly normalised. If we choose mixed signs, the result is,

$$|\boldsymbol{v}_\text{f}|^2 = (-n_y \gamma \mp n_x \sqrt{1 - \gamma^2})^2 + (n_x \gamma \pm n_y \sqrt{1 - \gamma^2})^2, \tag{50}$$
$$= (n_x^2 + n_y^2)\gamma^2 + (n_x^2 + n_y^2)(1 - \gamma^2) \pm 4 n_x n_y \gamma \sqrt{1 - \gamma^2}, \tag{51}$$
$$= 1 \pm 4 n_x n_y \gamma \sqrt{1 - \gamma^2}. \tag{52}$$

Choosing mixed solutions is only correctly normalised if the second term is zero. It is zero if $n_x = 0$, or $n_y = 0$ or $\gamma = 1$. These three cases cause the second term in Eqs. 41 & 42 to be zero meaning the mixed sign solutions reduce to the same sign solutions. We do not need to consider these cases any further.

$4n_x n_y \gamma \sqrt{1-\gamma^2}$ is also zero when $\gamma = 0$. Remembering $\gamma = \frac{n_i}{n_f}|\hat{\boldsymbol{n}} \times \boldsymbol{v}_i|$, $\gamma = 0$ implies the ray travels perpendicular to the plane initially and so we must have $\boldsymbol{v}_f = \pm\hat{\boldsymbol{n}}$. Hence the solution we seek involves either both positive or both negative.

This leaves us with two possible solutions. To distinguish which of the solutions is the one we are after, we note the sign of $\boldsymbol{v} \cdot \hat{\boldsymbol{D}}$ and $\boldsymbol{v} \cdot \hat{\boldsymbol{n}}$ must be unchanged after refraction.

6.2 Refraction at a spherical surface

At a spherical surface we already have a method for determining where a ray intersects the surface.

6.2.1 Old method

The angle we must rotate through is given again by Eq. 30,

$$\alpha = t' - \frac{\pi}{2}. \tag{53}$$

Other than the different equation for computing α, the method is very similar to that of a plane.

6.2.2 New method

The new method is very similar to that of a plane, but the definitions of $\hat{\boldsymbol{n}}$ and $\hat{\boldsymbol{D}}$ are different.

7 Screen

We also include a component intended to represent a screen. When a ray hits a screen, we stop tracing it any further. We will implement only a planar screen named `Screen_Plane`.

8 Complex components

A complex component is one that is composed of other sub-components. These include objects such as lenses and prisms that can be described in terms of basic sub-components or a objects such as system of lenses, which are composed themselves of complex components. Equally, a complex component could be created from a mixture of basic and complex components.

8.1 Lenses

We shall first consider a lens to be composed of four components. Two will always be planar boundaries corresponding to the top and bottom planes in Fig. 4. The remaining two will be either arcs of planes depending on the type of lens, e.g. biconvex or plano-convex. A generic lens can then be described as a complex component composed of these four objects.

To distinguish between convex, concave and plane lens faces, we take the following convention:

- A convex component arc has a **positive** radius of curvature

- A concave component arc has a **negative** radius of curvature

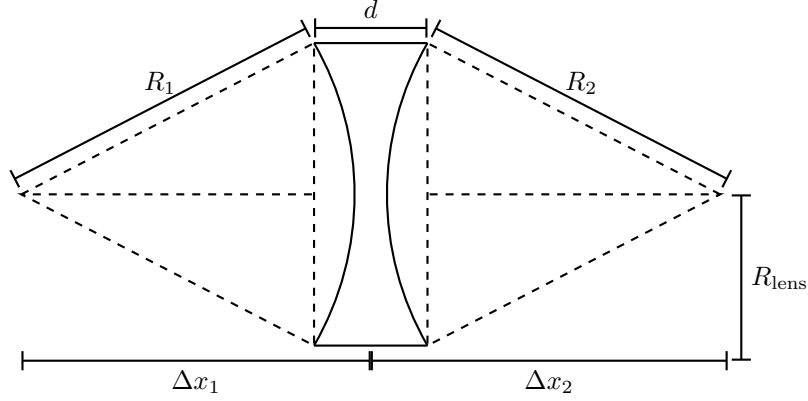- A plane component arc sets the radius of curvature to be **zero**

Figure 4: Left: Parameters used to describe a lens, in this case a biconcave. R_1 and R_2 are the radii of curvature on the left and right sides respectively. Δx_1 and Δx_2 give the horizontal offset between the centre of the lens and its respective arc. R_{lens} give the radius of the lens and d its thickness.

This means any radius of curvature where $R < |R_{\text{lens}}|$ but $R \neq 0$ is invalid. We note we must be careful with how the interior and exterior refractive indices of the lens, N_{in} and n_{out}, correspond to what refractive parameters are passed to each individual component. Changing a lens arc from convex to concave or swapping from left to right will require care with these refractive parameters.

Note we can compute Δx as,

$$\Delta x = \frac{d}{2} + \sqrt{R^2 - R_{\text{lens}}^2}. \tag{54}$$

Finally we should check for self intersection if one or both of the lens arcs are concave ($R < 0$). Strange behaviour might result otherwise.

9   Calling from Python

To call from Python we wrap the C++ using Cython. We wrap each C++ class to produce a similarly named class as that in C++ but roughly with "Py" at the front. Class equivalents are detailed in Tab. 1. Similarly we also produce a tracing function that takes in Python components and traces them in C++ named `PyTrace`.

Properties such as the start and end points of planes are exposed as properties. In most cases points are returned as a `numpy` view. As such they can be edited element wise. The exception to this is `PyRay.pos` which returns a copy due to the way `Ray.pos` is stored in C++. Cython classes are initialised using a `__cinit__()` initialiser. Unfortunately this cannot be overloaded with the current version of Cython, though I do not believe this is an issue.

Furthermore, we have an additional class, `PyCC_Wrap`. `PyCC_Wrap` is intended to allow easy creation of complex components from Python by simply inheriting from `PyCC_Wrap` and passing the Cython components needed to describe the component to the `PyCC_Wrap` initialiser. For example, a triangular prism defined through the positions of its vertices and interior/exterior refractive indices can be created using:

```
class Prism(PyCC_Wrap):

    def __init__(self, a, b, c, n_in, n_out=1.0):
```

C++	Cython
Component	_PyComponent
Complex_Component	PyComplex_Complex
Plane	_PyPlane
Spherical	_PySpherical
Mirror_Plane	PyMirror_Plane
Refract_Plane	PyRefract_Plane
Mirror_Sph	PyMirror_Sph
Refract_Sph	PyRefract_Sph
Screen_Plane	PyScreen_Plane
Ray	PyRay

Table 1: C++ classes and their equivalent Cython versions.

```
comps = [
        PyRefract_Plane(a, b, n_out, n_in),
        PyRefract_Plane(b, c, n_out, n_in),
        PyRefract_Plane(c, a, n_out, n_in)
    ]

# Must remember to call super __init__()
super().__init__(comps)
```

Like other components, `PyCC_Wrap` provides a `Plot()` method useful for plotting the component.