geometrical-ray-tracing

Tom Spencer

12:22, Tuesday 14th June, 2022

1 Introduction

`geometrical-ray-tracing` is a Python library that performs geometrical ray tracing in two dimensions. It provides several basic components such as a planar mirror as well as the ability to create new components composed of other components. The ray tracing itself is implemented in C++, with a Cython wrapper to interface with Python.

This work makes extensive use of Cython and NumPy [1, 2]. The examples provided also make use of the Matplotlib and SciPy libraries [3, 4].

2 Program implementation

In this section, we consider the only C++ details of the implementation. The details on the Python side are discussed in Section 7.

2.1 Rays

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
- A `reset()` method to reset the `Ray` to its initial position and direction of travel, optionally allowing a change of initial position.

2.2 Components

We define an abstract base class `Component` from which all components ultimately derive. This declares:

- A `test_hit()` method that tests whether a ray interacts with the component. It 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. It takes a reference to a instance of `Ray` and a maximum number of interactions to perform as arguments. It has no return value as it modifies the existing `Ray` instance.

From this two helper classes are derived. These aren't meant to be initialised directly and are still abstract classes. They are:

- `Plane`
  - Describes a simple planar component
  - Declares to define the start and end points of the plane and a unit vector pointing from the plane's start to its end
  - 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, the end angle should be larger than the start angle
  - Defines constructor for these variables
  - Implements the `test_hit()` method to test for a ray hitting the circle/arc

Note in this work "spherical" and "arc" are taken to be synonymous. From these there are five basic components implemented:

- `Mirror_Plane` - 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

Finally we have the `Complex_Component` class for describing a component composed of several other components. These can include the five components above as well as other instances of `Complex_Component`.

- 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 of C++ component classes also need to hold onto a pointer.
- Implements both the `test_hit()` and `hit()` methods:
  - 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 described below.

`Complex_Component` should make components such as prisms or lenses 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.

2.3 Tracing functions

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>`, maximum number of iterations and whether the `Ray` should be "filled up" to the maximum 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 dimensionless quantity and for 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 positive 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/or 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,

$$t' = \frac{v_y(x_1 - x_0) - v_x(y_1 - y_0)}{v_x(y_2 - y_1) - v_y(x_2 - x_1)}, \qquad t = \frac{x_0\,(-y_1 + y_2) - x_1y_2 + x_2y_1 + y_0\,(x_1 - x_2)}{v_x\,(y_1 - y_2) + v_y\,(-x_1 + x_2)}. \tag{5}$$

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.

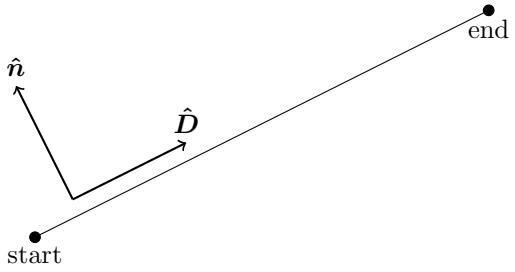

Figure 1: Structure of a plane, along with the unit vectors $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{n}}$.

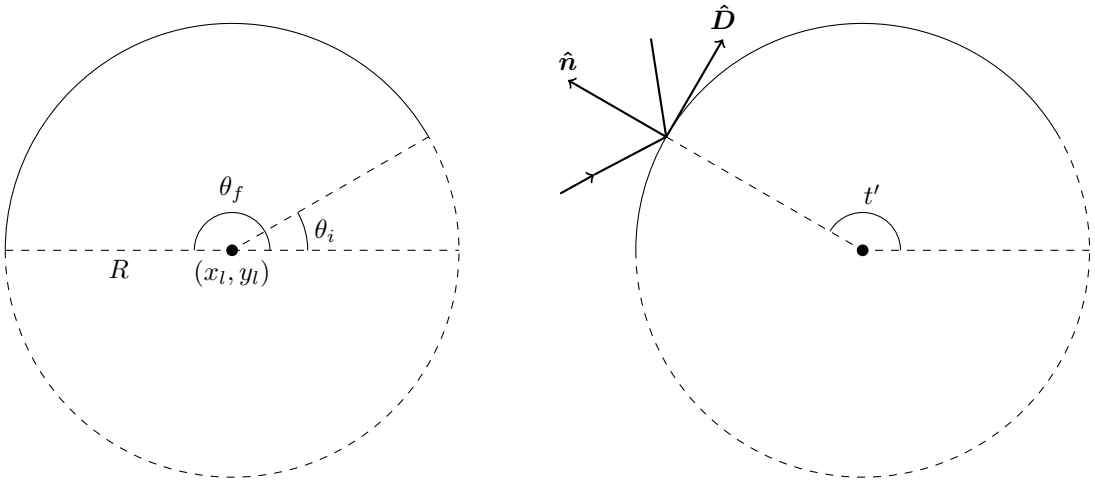

Figure 2: Structure of a circular arc, along with the unit vectors $\hat{\boldsymbol{D}}$ and $\hat{\boldsymbol{n}}$ as defined for a circular arc for a given ray intersection.

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,

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

We define a normal unit vector $\hat{\boldsymbol{n}}$ that points radially outwards at the point of the ray's intersection. 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. We can eliminate t' by squaring Eqs. 10 & 11, and utilising the identity $\sin^2\theta + \cos^2\theta = 1$. 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 used 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 do not need to call any trigonometric functions to determine t unlike if we solve for t'. This helps solving for t to be a more efficient method.

If we do not need the value of t', we can avoid the call to `atan2()` that would be used to determine it and improve performance further. The new (non-rotation) methods for performing reflection and refraction do not require t'. To check for an interaction, first consider the start and end points of the arc,

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

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 $\boldsymbol{r}_{\mathrm{end}}$. There are two cases as illustrated in Fig. 3.

- In the first case, we have $r'_{\mathrm{end},y} \geq 0$. If $P_y \geq 0$ and $P_x \geq r'_{\mathrm{end},x}$ then the ray does intersect the arc, otherwise it does not.
- In the second case, we have $r'_{\mathrm{end},y} < 0$. $P_y \geq 0$ is sufficient to know the ray does intersect the arc. If $P_y < 0$, we must have $P_x \leq r'_{\mathrm{end},x}$ for intersection.

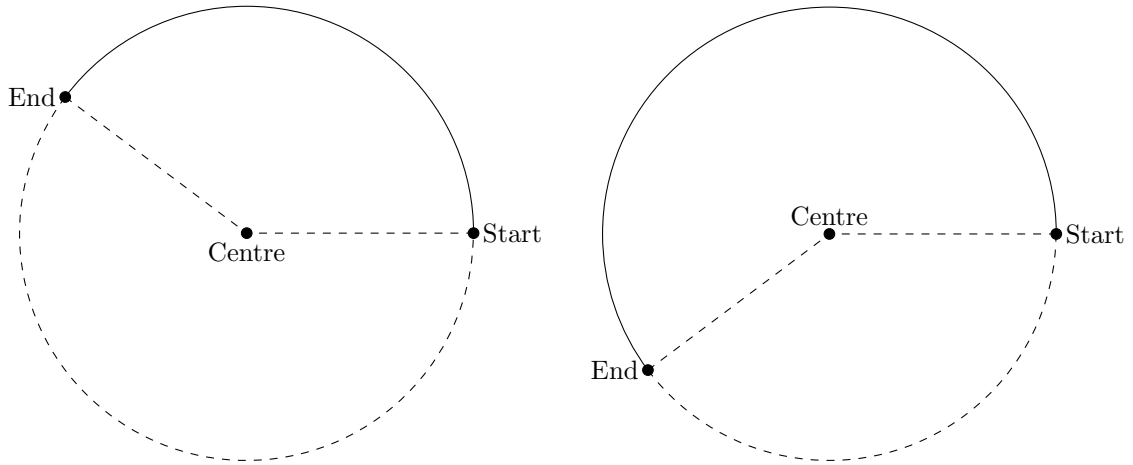

Figure 3: The two cases to consider when testing if a point is between θ_i and θ_f.

4 Reflecting basic components

4.1 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,

$$\boldsymbol{r}_{\mathrm{m}}(t') = \boldsymbol{r}_1 + (\boldsymbol{r}_2 - \boldsymbol{r}_1)t', \tag{16}$$

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.

4.1.1 Rotation method

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

$$\tan\alpha = \frac{y_2 - y_1}{x_2 - x_1}. \tag{17}$$

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 the `cmath` header provides `atan2()` which specifically accounts for these two cases and also will always return the correct sign.

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,

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}. \tag{18}$$

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

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}. \tag{19}$$

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()`, each time we process a ray interaction. Fortunately we can avoid this.

4.1.2 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 $\boldsymbol{v}$ normal to the plane. We can make use of $\hat{\boldsymbol{D}}$, the unit 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 to the plane is $\boldsymbol{v} - (\boldsymbol{v} \cdot \hat{\boldsymbol{D}})\hat{\boldsymbol{D}}$ and so the reflections ca be performed as,

$$\boldsymbol{v} \to (\boldsymbol{v} \cdot \hat{\boldsymbol{D}})\hat{\boldsymbol{D}} - \boldsymbol{v}. \tag{20}$$

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 $|v - 1| << 1$, we can avoid calls to `hypot()` by using a Taylor expansion. Denoting $S = v_x^2 + v_y^2$, expanding around $S = 1$ produces,

$$\frac{1}{\sqrt{S}} = 1 - \frac{1}{2}(S-1) + \mathcal{O}\left((S-1)^2\right). \tag{21}$$

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

4.2 Spherical mirror

To reflect at a spherical surface, we follow a broadly similar method to that of a planar surface. Again we originally followed a rotation based method similar to that of a plane and now use a similar method to the vector method for a plane. We use the normal vector of the circle at the point of intersection to write the reflections as,

$$\boldsymbol{v}_f = \boldsymbol{v}_i - 2(\boldsymbol{v}_i \cdot \hat{\boldsymbol{n}})\hat{\boldsymbol{n}}. \tag{22}$$

5 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,

$$n_i \sin\theta_i = n_f \sin\theta_f, \tag{23}$$

where both angles are measured from the normal of the surface and n_i, 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,

$$\sin\theta_f = \frac{n_i \sin\theta_i}{n_f} > 1. \tag{24}$$

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

$$\theta_c = \arcsin\frac{n_f}{n_i}. \tag{25}$$

5.1 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,

$$\theta_i = \frac{\pi}{2} - \theta', \tag{26}$$

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,

$$\boldsymbol{v}' = \cos\left(\frac{\pi}{2} - \theta_f\right)\hat{\mathbf{i}}' + \sin\left(\frac{\pi}{2} - \theta_f\right)\hat{\mathbf{j}}'. \tag{27}$$

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.

5.1.1 New method

The old method calls trigonometric functions including `sin()` and `atan2()`. If we were working in three dimensions we could write [5],

$$n_\mathrm{i}\hat{\boldsymbol{n}} \times \boldsymbol{v}_\mathrm{i} = n_\mathrm{f}\hat{\boldsymbol{n}} \times \boldsymbol{v}_\mathrm{f}, \tag{28}$$

which produces Snell's law when we take it's magnitude. Working in the xy plane, Eq. 28 produces,

$$n_\mathrm{i}(n_x v_\mathrm{iy} - n_y v_\mathrm{ix}) = n_\mathrm{f}(n_x v_\mathrm{fy} - n_y v_\mathrm{fx}). \tag{29}$$

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

$$n_x v_\mathrm{fy} - n_y v_\mathrm{fx} = \gamma, \tag{30}$$

$$v_\mathrm{fx}^2 + v_\mathrm{fy}^2 = 1, \tag{31}$$

which have the solutions,

$$v_\mathrm{fx} = -n_y\gamma \pm n_x\sqrt{1-\gamma^2}, \tag{32}$$

$$v_\mathrm{fy} = n_x\gamma \pm n_y\sqrt{1-\gamma^2}. \tag{33}$$

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. 28, γ can be written in terms of the angle the incident ray makes to the normal of the plane as $\gamma = \frac{n_\mathrm{i}}{n_\mathrm{f}}\sin\theta_\mathrm{i}$. We therefore do not have a real solution for $\boldsymbol{v}_\mathrm{f}$ when,

$$\sin\theta_\mathrm{i} > \frac{n_\mathrm{f}}{n_\mathrm{i}} \quad \text{or} \quad \sin\theta_\mathrm{i} < -\frac{n_\mathrm{f}}{n_\mathrm{i}}. \tag{34}$$

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. Earlier, we defined n_1 to be on the side $\hat{\boldsymbol{n}}$ points towards. Therefore can determine which side the ray originates on by considering $\boldsymbol{v}_\mathbf{i} \cdot \hat{\boldsymbol{n}}$,

$$\boldsymbol{v}_\mathbf{i} \cdot \hat{\boldsymbol{n}} > 0 \implies n_\mathrm{i} = n_2, n_\mathrm{f} = n_1, \tag{35}$$

$$\boldsymbol{v}_\mathbf{i} \cdot \hat{\boldsymbol{n}} < 0 \implies n_\mathrm{i} = n_1, n_\mathrm{f} = n_2. \tag{36}$$

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

We must still determine which of the possible solutions in Eqs. 32 & 33 are physical. We start by checking $\boldsymbol{v}_\mathrm{f}$ is normalised. Taking the same sign in each of Eqs. 32 & 33 produces,

$$|\boldsymbol{v}_\mathrm{f}|^2 = (-n_y\gamma \pm n_x\sqrt{1-\gamma^2})^2 + (n_x\gamma \pm n_y\sqrt{1-\gamma^2})^2, \tag{37}$$

$$= (n_x^2 + n_y^2)\gamma^2 + (n_x^2 + n_y^2)(1-\gamma^2), \tag{38}$$

$$= n_x^2 + n_y^2, \tag{39}$$

$$= 1. \tag{40}$$

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

$$|\boldsymbol{v}_\mathrm{f}|^2 = (-n_y\gamma \mp n_x\sqrt{1-\gamma^2})^2 + (n_x\gamma \pm n_y\sqrt{1-\gamma^2})^2, \tag{41}$$

$$= (n_x^2 + n_y^2)\gamma^2 + (n_x^2 + n_y^2)(1-\gamma^2) \pm 4n_x n_y \gamma\sqrt{1-\gamma^2}, \tag{42}$$

$$= 1 \pm 4n_x n_y \gamma\sqrt{1-\gamma^2}. \tag{43}$$

Choosing mixed solutions is only correctly normalised if the second term is zero. There are four possible ways for it to be zero. The first three are $n_x = 0$, or $n_y = 0$ or $\gamma = 1$. These three cases cause the second term in Eqs. 32 & 33 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 $\gamma = 0$. Eqs. 32 & 33 give $\boldsymbol{v}_\mathrm{f}$ as being parallel to the plane. However, remembering $\gamma = \frac{n_\mathrm{i}}{n_\mathrm{f}}|\hat{\boldsymbol{n}} \times \boldsymbol{v}_\mathrm{i}|$, $\gamma = 0$ implies the ray travels perpendicular to the plane initially and so we must have $\boldsymbol{v}_\mathrm{f} = \pm\hat{\boldsymbol{n}}$. Hence we can conclude $4n_x n_y \gamma\sqrt{1-\gamma^2}$ cannot reasonably be zero and only the solutions of Eqs. 32 & 33 that are either both positive or both negative are of interest.

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.

5.2 Refraction at a spherical surface

At a spherical surface we already have a method for determining where a ray intersects the surface. Again, we originally utilised a rotation based method but now use a vector based method for improved performance. This new method is very similar to that of a plane, but the definitions of $\hat{\boldsymbol{n}}$ and $\hat{\boldsymbol{D}}$ are different.

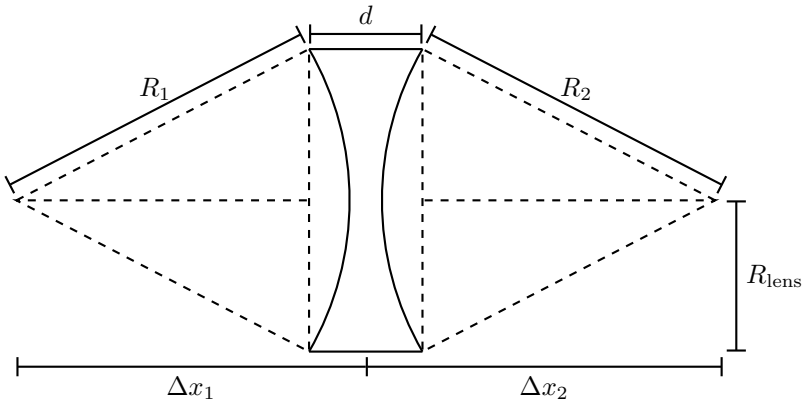

Figure 4: Left: Parameters used to describe a lens, in this case a biconcave. R_1 and R_2 are the radii of curvature on the left and right sides respectively. Δx_1 and Δx_2 give the horizontal offset between the centre of the lens and each arc respectively. R_{lens} give the radius of the lens and d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

6 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 themselves composed of complex components. Equally, a complex component could be created from a mixture of basic and complex components.

6.1 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**

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,

$$\Delta x = \frac{d}{2} + \sqrt{R^2 - R_{\text{lens}}^2}. \tag{44}$$

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

7 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`.

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.

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 may be edited element wise if the view is writable. The notable exception to this is `PyRay.pos` which returns a copy due to the way `Ray.pos` is stored in C++.

Furthermore, we have some additional classes. `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):
        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.

In addition to these classes, we also provide `PyLens` and `PyBiConvexLens` to model a generic lens and a bi-convex lens.

References

[1] S. Behnel, R. Bradshaw, C. Citro, L. Dalcin, D. S. Seljebotn, and K. Smith, "Cython: The best of both worlds," *Computing in Science & Engineering*, vol. 13, no. 2, pp. 31–39, 2011.

[2] C. R. Harris, K. J. Millman, S. J. van der Walt, R. Gommers, P. Virtanen, D. Cournapeau, E. Wieser, J. Taylor, S. Berg, N. J. Smith, R. Kern, M. Picus, S. Hoyer, M. H. van Kerkwijk, M. Brett, A. Haldane, J. F. del Río, M. Wiebe, P. Peterson, P. Gérard-Marchant, K. Sheppard, T. Reddy, W. Weckesser, H. Abbasi, C. Gohlke, and T. E. Oliphant, "Array programming with NumPy," *Nature*, vol. 585, pp. 357–362, Sept. 2020.

[3] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007.

[4] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors, "SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python," *Nature Methods*, vol. 17, pp. 261–272, 2020.

[5] E. Hecht, *Optics*. Pearson, 2014.