Ray tracing

Tom Spencer

12:36, Friday 13th August, 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 component 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 `hit_test()` 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 `hit_test()` 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 Python, 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

I think it would be easier to create a `Ray` class to keep track of everything and then wrap it in `Cython`. Could 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 I just need to pass a reference to the ray to any function and the function can work out get whatever it wants.

I think the best way forward is to 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` used to describe the direction of the ray
- A vector of `std::array` 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 -1.0 if the ray does not interact, otherwise the time to interaction
- A `hit()` method that performs the interaction. Takes a reference 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, 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::unique_ptr<Component>` to store the sub-components of this component
  - Implements both the `test_hit()` and `hit()` method:
    * For `test_hit()`, we simply need to iterate over all the sub-components, return -1.0 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 four basic components implemented:

- `Mirror` - plane mirror
- `Mirror_Sph` - spherical mirror
- `Refract_Plane` - refraction at a planar boundary
- `Refract_Sph` - refraction at a spherical boundary

Once these basic components have been implemented, complex components such as prisms or lenses should be exceedingly easy 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::unique_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 reflecting on both sides and 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.

We can derive where the ray interests as obeying,

$$t' = \frac{v_y(x_1 - x_0) - v_x(y_1 - y_0)}{v_x(y_2 - y_1) - v_y(x_2 - x_1)}, \qquad t = \frac{x_0\left(-y_1 + y_2\right) - x_1y_2 + x_2y_1 + y_0\left(x_1 - x_2\right)}{v_x\left(y_1 - y_2\right) + v_y\left(-x_1 + x_2\right)} \tag{3}$$

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.

3.2 Ray intersection with a spherical surface

We consider a circle with centre (x_l, y_l) and radius R. In general, only a portion of its surface is modelled with $\theta_i \leq t' \leq \theta_f$. Here t' is a parameter in the range $0 \leq t' < 2\pi$ that takes the place of the polar angle. Its surface can be described by,

$$\boldsymbol{r}_{\mathrm{s}}(t') = (x_l + R\cos t')\hat{\mathbf{i}} + (y_l + R\sin t')\hat{\mathbf{j}}. \tag{4}$$

This gives us two simultaneous equations, namely,

$$x_0 + v_xt = x_l + R\cos t' \tag{5}$$
$$y_0 + v_yt = y_l + R\sin t'. \tag{6}$$

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

$$\Delta x + v_xt = R\cos t' \tag{7}$$
$$\Delta y + v_yt = R\sin t'. \tag{8}$$

Eliminating t and substituting $\tan\theta = v_y/v_x$ produces,

$$\Delta y - \Delta x \tan\theta + R\tan\theta\cos t' = R\sin t'. \tag{9}$$

Defining $\gamma = \Delta y - \Delta x\tan\theta$ and $u = \cos t'$ produces,

$$\left(\gamma + Ru\tan\theta\right)^2 = R^2\left(1-u^2\right). \tag{10}$$

The a, b, c components of this quadratic are,

$$a = R^2\left(1+\tan^2\theta\right), \tag{11}$$
$$b = 2R\gamma\tan\theta, \tag{12}$$
$$c = \gamma^2 - R^2. \tag{13}$$

The solution we have is,

$$u = \cos t' = \frac{-2R\gamma\tan\theta \pm \sqrt{4R^2\gamma^2\tan^2\theta - 4R^2\left(1+\tan^2\theta\right)\left(\gamma^2-R^2\right)}}{2R^2\left(1+\tan^2\theta\right)}. \tag{14}$$

This will actually 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 work out how to determine to correct solution for the intersection. Now we need to be careful in case v_x is small, as in this case $\tan\theta \to \infty$. Substituting back in $\tan\theta$ and multiplying top and bottom by v_x^2 produces,

$$\cos t' = \frac{-2R\gamma v_x v_y \pm \sqrt{4R^2\gamma^2 v_x^2 v_y^2 - 4R^2\left(v_x^4 + v_x^2 v_y^2\right)\left(\gamma^2-R^2\right)}}{2R^2\left(v_x^2+v_y^2\right)}, \tag{15}$$

$$= \frac{-\gamma v_x v_y \pm v_x\sqrt{\gamma^2 v_y^2 - \left(\gamma^2-R^2\right)}}{R}, \tag{16}$$

where the second line follows from $v_x^2 + v_y^2 = 1$ since $\boldsymbol{v}$ is normalised. γ also has a dependence on $\tan\theta$, so we need to be careful again:

$$\cos t' = \frac{1}{R}\left(-v_y\delta \pm \sqrt{\delta^2\left(v_y^2-1\right) + \left(v_x R\right)^2}\right), \tag{17}$$

where $\delta = (\Delta y v_x - \Delta x v_y)$. This form should be numerically stable with no problems when v_x is small. We shall denoted the two solutions to $\cos t'$ as u and v.

Now we must work out which of the four possible solutions is the one we are after. The solution can be determined by testing each possible solution in the following way,

1. First if u (v) are in the range $-1 \le u \le 1$ ($-1 \le v \le 1$)
2. Then determine which, (if any) t' are in the necessary range for the surface, $\theta_i \le t' \le \theta_f$
3. Determine which solutions are true solutions and lie along the light ray's path
4. Finally if there are two solutions left, choose the solution with the smaller t parameter

4 Components

We would like to provide implementations for the following components:

- Mirrors, plane and spherical
- Lenses
- Prisms
- Absorbing screen

We will first provide four 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,

$$\boldsymbol{x}_{\mathrm{m}}(t') = \boldsymbol{x}_1 + (\boldsymbol{x}_2 - \boldsymbol{x}_1)t', \tag{18}$$

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. The basic law is that the angle of incidence is equal to the angle of reflection.

We can compute the new starting position for the ray using the standard equation describing the ray as we have t. To find the new velocity, we compute the angle, α, the plane makes with the x axis,

$$\tan\alpha = \frac{y_2 - y_1}{x_2 - x_1}. \tag{19}$$

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 `atan2(y, x)` which specifically accounts for these two cases, and also will always return the correct sign. In C++ we will have to be careful here and possibly implement our own function for this purpose.

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,

$$\begin{pmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{pmatrix}. \tag{20}$$

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

$$R(\theta) = \begin{pmatrix} \cos\theta & \sin\theta \\ -\sin\theta & \cos\theta \end{pmatrix}. \tag{21}$$

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

5.2 Spherical mirror

To reflect at a spherical surface, we follow a broadly similar method to that of a planar surface. First we determine the t, t' values where the ray intersects the surface. The intersection location is given by,

$$x = x_s + R\cos t', \qquad y = y_s + R\sin t', \tag{22}$$

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,

$$\alpha = t' - \frac{\pi}{2}. \tag{23}$$

After that it is identical to the plane mirror. We use Eq. 21 to rotate the axis by α, negate the y' component, and finally rotate back to get the new direction.

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,

$$n_i \sin\theta_i = n_f \sin\theta_f, \tag{24}$$

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,

$$\sin\theta_f = \frac{n_i \sin\theta_i}{n_f} > 1. \tag{25}$$

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

$$\theta_c = \arcsin\frac{n_f}{n_i}. \tag{26}$$

So once we compute the angle of incidence we ought to compute the critical angle to check if we will actually have total internal reflection.

We will also need some way of describing the refractive index on each side. We can construct a vector normal to the surface that points in a consistent direction.

6.1 Refraction at a planar surface

Once again we want to rotate our system so that it is convenient. We will rotate our coordinates round by α (angle of the plane) so that normal lies along the y-axis. We then have,

$$\theta_i = \frac{\pi}{2} - \theta', \tag{27}$$

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,

$$\boldsymbol{v}' = \cos\left(\frac{\pi}{2} - \theta_f\right)\hat{\mathbf{i}}' + \sin\left(\frac{\pi}{2} - \theta_f\right)\hat{\mathbf{j}}'. \tag{28}$$

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.

6.2 Refraction at a spherical surface

At a spherical surface we already have a method for determining where a ray intersects the surface. The angle we must rotate through is given again by Eq. 23,

$$\alpha = t' - \frac{\pi}{2}. \tag{29}$$

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

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. 1. 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,

$$\Delta x = \frac{d}{2} + \sqrt{R^2 - R_{\text{lens}}^2}. \tag{30}$$

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

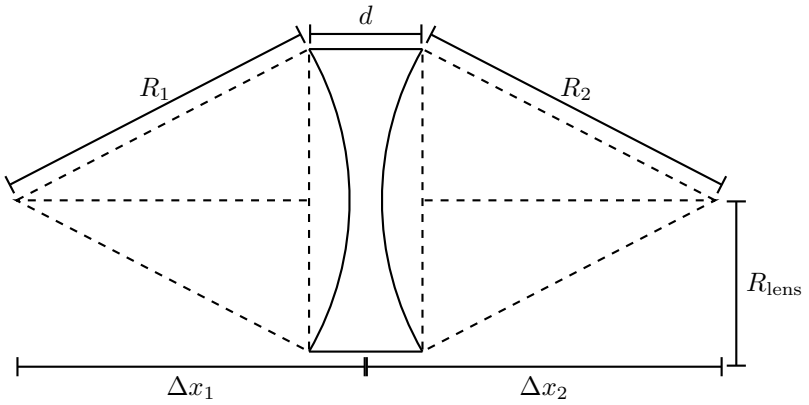

Figure 1: Left: Parameters used to describe a lens, in this case a biconcave. R_1 and R_2 are the radii of curvature on the left and right sides respectively. Δx_1 and Δx_2 give the horizontal offset between the centre of the lens and its respective arc. R_{lens} give the radius of the lens and d its thickness.

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`.

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 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):
        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.