Project A: An optical ray-tracer

Introduction

Optical ray tracing is a powerful technique that is at the heart of most optical design packages. In this project you will design and code your own optical ray tracer and use it to investigate quantitatively some optical systems.

Project Aims:

- design and write an optical ray tracer in Python using object-oriented programming
- test and verify the operation of the ray-tracer
- use the ray-tracer to investigate the imaging performance of simple lenses
- use the ray-tracer to optimise the design of a biconvex lens (optional)

Geometrical optics and optical ray tracing

Geometrical optics describes the propagation of light using rays. Although wave optics and ray optics appear to be very different, the ray optics formulation can be derived from the wave equation in the limit of the wavelength becoming small. An optical ray defines the propagation direction of the light. Rays are normal to the wavefront, in the direction of the wave vector of the optical waves. In this sense, ray optics is an approximation to the full wave optics model when the wavelength is non-zero, but a remarkable useful model.

In a sequential optical ray tracer, the optical system is modelled by propagating rays through the optical system using the principles of geometrical optics. When the ray encounters an interface between two dielectric media in the system, such as the surface of a lens, the ray is refracted (using Snell's law). Propagation continues to the next surface and so on until the ray reaches the output of the system. It is possible to include different types of optical surfaces including reflecting surfaces.

Designing an optical ray tracer

We shall build a simple 3-D optical ray tracer and use it to model the behaviour or some simple optical systems. We will use an object-oriented approach to designing the model.

We will consider rays and optical elements in the system as objects.

A ray object will model the optical ray. A ray can be represented by a series of points that describe the path through the optical system and a direction of propagation.

We shall model the optical elements of the system as objects. We shall primarily be concerned with elements comprised of refracting surfaces. In our model, an optical surface object will have a number of properties including refractive indices, and geometrical properties of surface shape. It will also have methods for determining where rays intercept with it and for calculating the effect of the surface on a ray, such as refraction or reflection. (Of course the interaction of an optical ray

with an optical surface involves both objects, so in principle we have a choice about how we deal with this.)

The construction of the ray tracer can be broken down into the following tasks.

Optical Rays

Task 1: Begin by creating a new module. Name your module something informative, such as raytracer.py. Put some short documentation (as a docsctring) in your empty module.

We shall begin by designing a class to represent an optical ray. The ray will be represented by a point and a direction (vector). We suggest you use 3-element NumPy arrays to store both positions and directions in Cartesian representation.

Task 2: Design a `Ray` class that can be used to represent an optical ray. `Ray` should have methods that return the current point and direction of the ray. You should should be able to initialise `Ray` objects with a starting point and direction.

Since your ray will be propagated through the system, it should have a method to add a new point and direction — you could store the points internally as a list. It should also have a method that returns all the points along the ray.

As well as initialization, we suggest your class therefore has the following methods:

`p`()
: return the current point of the ray

`k`()
: return the current ray direction

`append`(*p*, *k*)
: append a new point and direction to the ray

`vertices`()
: return all the points along the ray

Test the behaviour of your class. Can you make instances of it? Can you initialize it? Check that the method functions work sensibly. Can you add new points to the ray and does it record them properly? Make sure you include appropriate documentation for your class.

Optical Elements

We shall represent our optical system using optical elements such as refracting surfaces, lenses etc. In our sequential ray tracer, a ray will be propagated through each optical element in turn. The object representing the optical element will be responsible for propagating the ray through it. We shall begin with a general base class, `OpticalElement` that has a method `propagate_ray`. All optical

elements will inherit this class and will be required to implement **`propagate_ray`** according to their own properties. Create the following simple base class:

```
class OpticalElement:
  def propagate_ray(self, ray):
    "propagate a ray through the optical element"
    raise NotImplementedError()
```

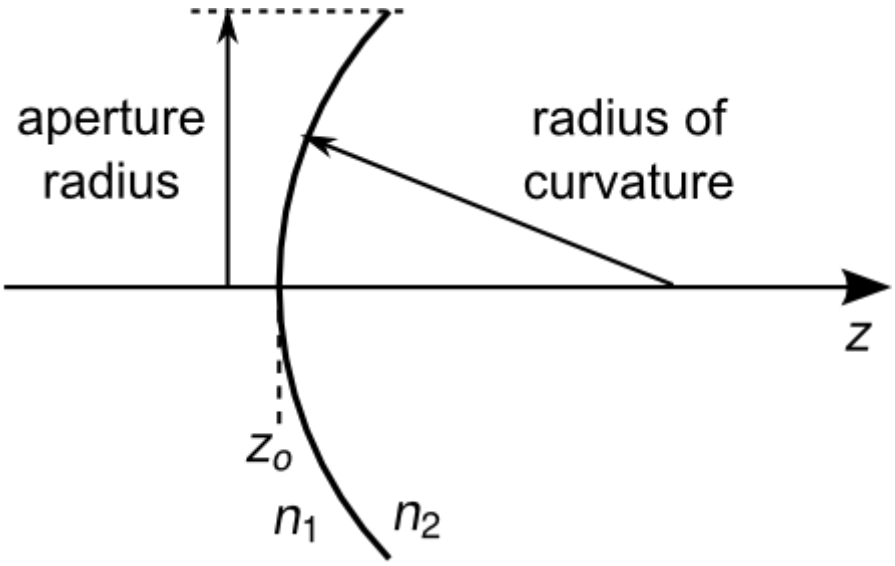
Then for a **`Ray`** object, and a list of different optical elements (all of which will have a **`propagate_ray`** method) one might propagate the ray through the system thus:

```
for elem in elements:
  elem.propagate_ray(ray)
```

Refracting Surfaces

Most optical surfaces are spherical (or planar). We will now design a class to represent a spherical refracting surface. We shall initially restrict ourselves to spherical surfaces centred on the optical axis (which by convention we shall take to be the z-axis). Then a spherical refracting surface can be represented by five parameters:

- z_0 - the intercept of the surface with the z-axis
- curvature - the curvature of the surface
- n_1, n_2 - the refractive indices either side of the surface
- aperture radius - the maximum extent of the surface from the optical axis

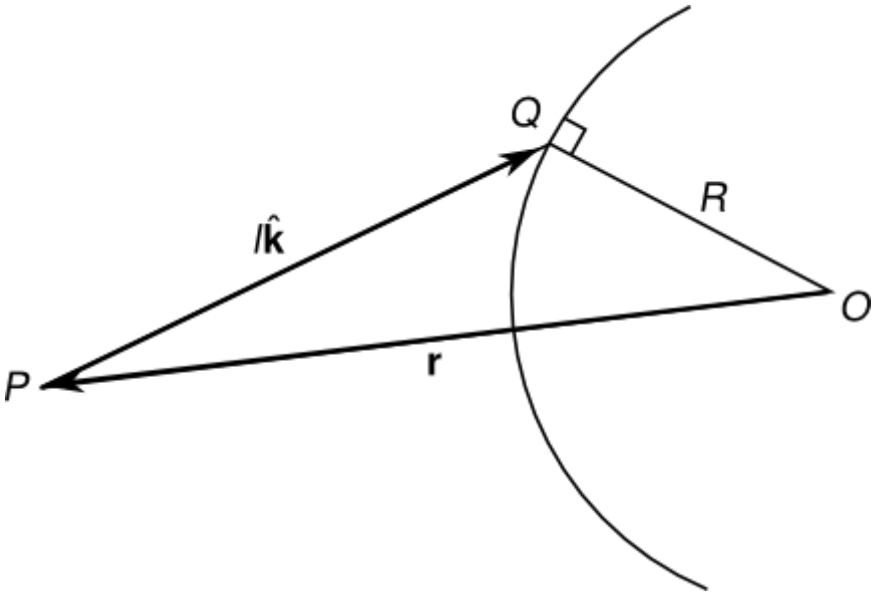


The curvature is a signed quantity with magnitude 1/(radius of curvature) We shall use the convention that a positive curvature corresponds to the centre of curvature at $z > z_0$ and negative curvature corresponds to a centre of curvature at $z < z_0$. A curvature of zero corresponds to a plane surface (i.e., infinite radius of curvature.)

Task 3: Create a class **`SphericalRefraction`** that you will use to represent a spherical refracting surface. It should have **`OpticalElement`** as a base class. It should be initialized with the five parameters described above.

Intersection of a ray with a spherical surface

For a ray starting at point *P* with a sphere of radius R centred at *O*, $\mathbf{r}$ is the vector from the sphere centre to the ray point and $\hat{\mathbf{k}}$ is the unit vector in the direction of the ray.



Then the distance l to the intercept *Q* can be found by solving the quadratic equation

$$\left|\mathbf{r} + l\hat{\mathbf{k}}\right|^2 = R^2$$

which gives

$$l = -\mathbf{r} \cdot \hat{\mathbf{k}} \pm \sqrt{(\mathbf{r} \cdot \hat{\mathbf{k}})^2 - (|\mathbf{r}|^2 - R^2)}$$

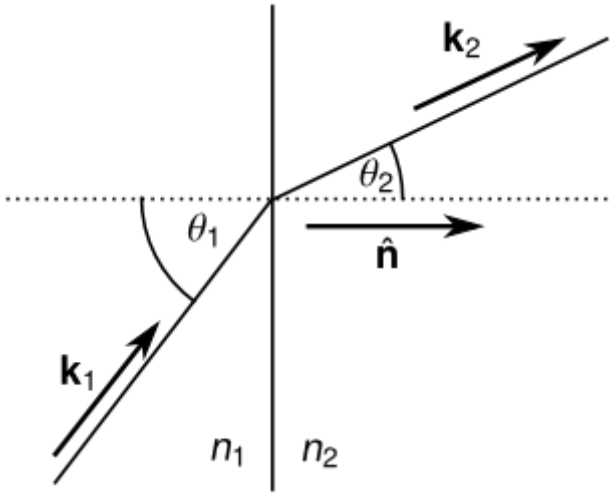
Task 4: Create a method `intercept`(*ray*) for your `SphericalRefraction` class that calculates the first valid intercept of a ray with the spherical surface. Remember a line can have two intercepts with a sphere — make sure you get the correct intercept for your surface. This should be the first intercept along the forward direction of the ray that lies within the aperture of the surface. Note: you may need to treat the case where the curvature is zero as special case. If there is no valid intercept, then your method should return `None`.

Refraction at a surface

Once a ray is intercepted by a surface it is refracted according to Snell's law.

$$n_1 \sin\theta_1 = n_2 \sin\theta_2$$

where n_1 and n_2 are the refractive indices either side of the surface and θ_1 and θ_2 are the angles of the rays with respect to the surface normal $\hat{\mathbf{n}}$.



Task 5: Write a function to implement Snell's law refraction. Your function should take an incident direction and a surface normal (both as unit vectors) and the refractive indices n_1 and n_2 as parameters. It should return the refracted ray direction (a unit vector). (Hint: the refracted ray should be in the same plane as the incident ray and the surface normal.) If the ray is subject to total internal reflection (i.e., $\sin\theta_1 > n_2/n_1$) your function should return `None`.

Task 6: Implement the method **`propagate_ray`** for **`SphericalRefraction`**. It should take a **`Ray`** object as parameter and should propagates it to the surface and refract it. It will need to find the intercept of the ray by the surface, calculate the direction of the refracted ray and update the ray to its new position and direction using **`append`**.

You will need to decide what to if there is not a valid intercept or if there is not a valid refracted ray direction. You could mark the ray as terminated (perhaps add appropriate methods to your **`Ray`** class to deal with this), or you could decide to signal via the return value of **`propagate_ray`**. Ensure you document the behaviour appropriately.

Task 7: Test your code. Create a refracting surface and a ray and check **`propagate_ray`** correctly propagates and refracts the ray. Try a range of initial rays to check your refracting object behaves as you expect.

Output plane

The final element in your sequence of optical elements should be the output plane. This should propagate rays to where they intersect with the output plane, but should not perform any refraction.

Task 8: Write a class **`OutputPlane`**, that is an **`OpticalElement`**. Implement methods **`intercept`** and **`propagate_ray`**.

Using your model

You should now be in a position to use your model to investigate optical systems.

Getting started - Modelling a single spherical refracting surface

Start with a simple system comprising a single spherical refracting surface an output plane. Consider a spherical surface at $z = 100$ with curvature 0.03 and refractive indices $n_1 = 1.0$ and $n_2 = 1.5$ and an output plane at $z = 250$.

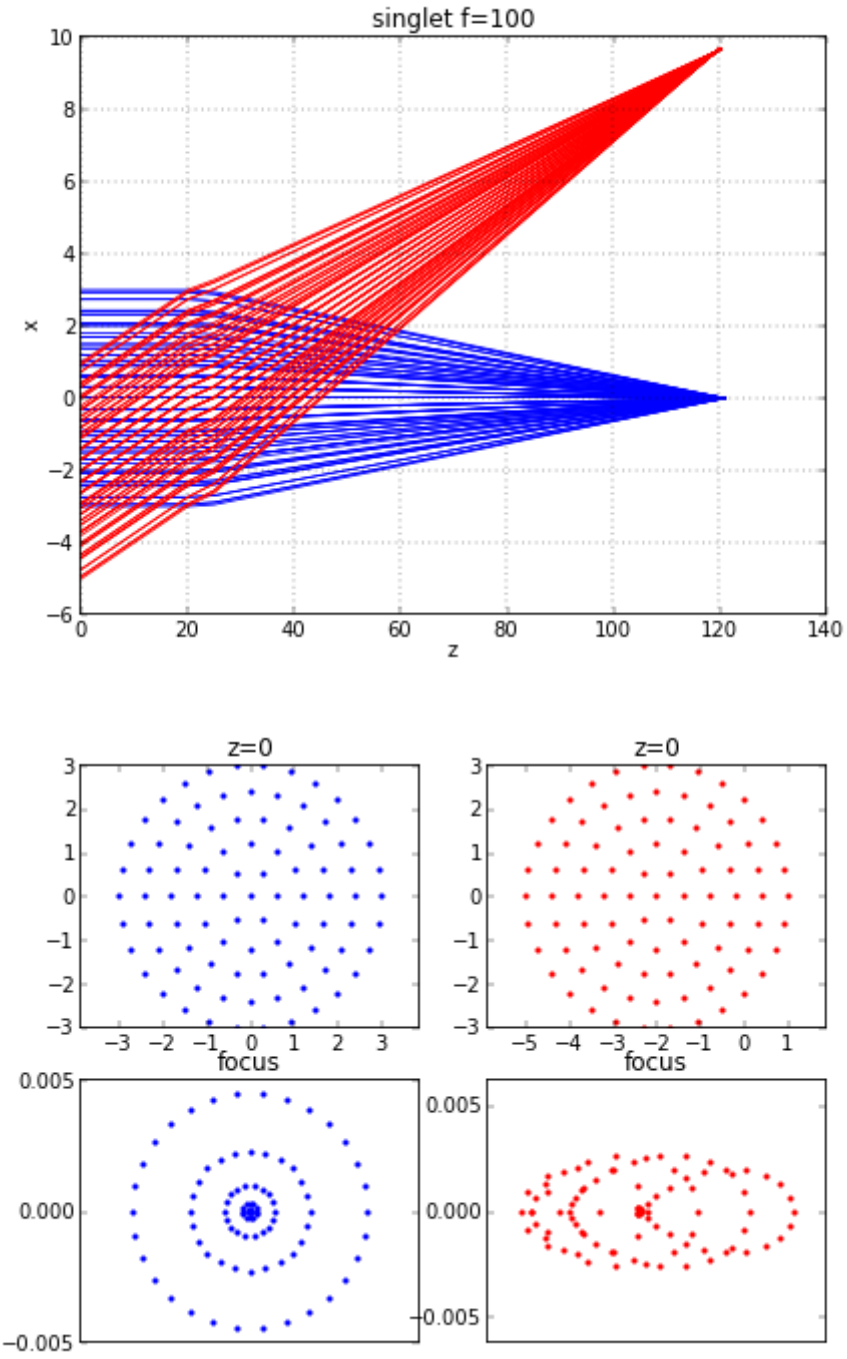
Try tracing a few example rays from different points in the input plane ($z = 0$) to the output plane. When you have traced rays through the optical system, you can plot their paths using **`matplotlib`** (remember: the method **`vertices`** should return the points along the ray.)

An incident plane wave, corresponding to light from a point object at infinity, can be represented by rays that are parallel to each other in the input. Trace a few rays parallel to the optical axis through your system. In an ideal imaging system, all such rays would meet at the output focal plane. In practice this does not happen because of spherical aberration — rays deviate from the ideal (paraxial) behaviour as the distance of the input ray from the optical axis increases. The paraxial focus is the ideal focus in the limit of a narrow input beam.

Estimate the position of the paraxial focus using a ray close to the optical axis (say 0.1mm). Once you have found the paraxial focus, adjust the position of the output plane in your model to the paraxial focus.

Now trace a bundle of rays for a collimated beam for a larger diameter, e.g., 5mm to the paraxial focal plane. You may find it useful to write code to generate bundles of rays to input to your optical system. You can use a Python list to store a bundle of rays, or you could write your own class to represent a bundle. You should find that your module **`genpolar`** (Task 15 from Worksheet 1) is particularly handy for generating suitable sets of coordinates corresponding to a circular pupil.

The xy-positions of the rays in the output form a spot diagram. Plot the spot diagram for your bundle of rays at the paraxial focal plane. An estimate of the size of the geometrical focus is given by the spread of the xy-positions of the rays in the focal plane. Calculate the RMS spot radius (i.e., the RMS deviation of the ray positions from the optical axis) at the paraxial focus for the bundle of rays and hence estimate the size of the geometrical focus. How does it compare with the diffraction limit? (The diffraction scale is $\lambda f/D$, where λ is the wavelength, f the focal length and D the aperture diameter or diameter of the input beam in this case).



Example of a biconvex lens: ray vertices (projected on the xz-plane) and spot diagrams

Modelling a planoconvex singlet lens

One of the simplest of lenses is the planoconvex lens. It has two optical surfaces: one plane, one convex.

Consider a planoconvex lens which has a convex surface with a curvature of 0.02/mm and a plane surface. The separation of the surfaces on the optical axis is 5mm and the refractive index of the glass is 1.5168 at a wavelength of 588nm.

Investigate the performance of the lens for focusing collimated light with a beam diameter of 10mm. Model the lens first with the plane surface facing the input, then with the convex surface facing the input. In each case, first find an estimate of the position of the paraxial focus using a ray very close to the optical axis (e.g., 0.1mm). Then calculate the geometrical spot size (the RMS radius of the rays as above) at the paraxial focus for bundles of rays corresponding to a range of beam diameters up to 10mm. Do this this for both orientations of the lens. Is the lens diffraction limited when used in either of these configurations?

Lens optimization (optional)

Although the plano-convex lens performs reasonably when used the correct way round, it is not optimal for a singlet lens. A best-form lens is a singlet which has two spherical surfaces with curvatures that have been chosen to optimize the performance of the lens for given object and image distances.

Can you design a best form lens for the above configuration. Rather than specify a plane and a convex surface, allow the two curvatures to be parameters in an optimization. Design the lens to minimize the geometrical RMS spot radius for a bundle of rays for a beam of diameter 10mm. One approach would be to use an optimizer from `scipy.optimize`, for example `fmin_tnc`. You will need to write a function that takes as parameters the curvatures of the two surfaces and returns an RMS spot radius as an optimization metric. How does the performance compare with the plano-convex system?

Advanced extensions

If you have time, you might like to consider any of these extensions (ordered by increasing level of challenge.) **Do not attempt these unless you have completed the previous tasks.**

Reflection

Can you model reflections in your ray-tracer? You could create a class to model a spherical reflecting surface, in a similar fashion to the `SphericalRefraction` class. Can you use the system to model a reflecting spherical mirror?

Dispersive materials

Optical materials are dispersive: the refractive index depends on wavelength. Could you model materials with wavelength-dependent refractive indices?

Rainbow

Could you model the physics of a rainbow? You'll need to account for reflections, refraction and dispersion. Your optical system could be represented as a single complete sphere of water, but rather than discard total internal reflection, you will need to include it as a reflection in your system.

Document Changes