```
# -*- coding: utf-8 -*-
"""
Created on Mon Feb 25 10:15:51 2019

@author: irg16

Description: This module contains the classes to be used to design an optical ray
tracer.

"""

import numpy as np
import math
import matplotlib.pyplot as plt

"""
Aims of the module:
   - design/write optical ray tracer using OOP
   - test/verify operation of ray tracer
   - use ray tracer to investigate imaging performance of simple lenses
   - use ray tracer to optimise design of biconvex lens
"""

"""
Designing class to represent optical ray:
    - ray represented by a point and a direction (vector)
    - use 3-element NumPy arrays to store both positions and directions
    - Cartesian representation
"""

"""
Refraction at a surface:
    - Snells law: n1sinθ1=n2sinθ2
    - write fn to implement snell's law
    - parameters:
        - incident direction (unit vector)
        - surface normal (unit vector)
        - refractive indices, n1, n2
    - if ray subject to total internal reflection:
        - (sinθ1 > n2/n1)
        - return None
"""
def SnellsLaw(incident_direction, surface_normal, n1, n2):
    ki_mag = np.linalg.norm(incident_direction) #magnitude of direction vector
    norm_mag = np.linalg.norm(surface_normal) #magnitude of normal vector
    ki_unitvector = incident_direction/ki_mag #set as a unit vector
    if norm_mag == 0:
        norm_mag = 1
```

```
    norm_unitvector = surface_normal/norm_mag #set as a unit vector
    cos1 = -np.dot(ki_unitvector, norm_unitvector) #use dot product
    if cos1 < 0.:
        norm_unitvector = -norm_unitvector
        cos1 = -np.dot(ki_unitvector, norm_unitvector)
    sin1 = np.arccos(cos1)
    if sin1 > n2/n1: #ray subject to total internal reflection
        return None
    else:
        n_ratio = n1/n2
        cos2 = np.sqrt(1 - (1-cos1**2)*n_ratio**2)
        snell = n_ratio* ki_unitvector + (n_ratio*cos1 - cos2)*norm_unitvector
        return snell


class Ray:
    """
    A class to represent an optical ray.
    """

    def __init__(self, point, direction): #creates variable
        #variable set at origin
        self._all_points = [point] #variable used to recall all points of a ray
        self._p = np.array(point)
        self._k = np.array(direction)
    def p(self): # returning the current point
        return self._p
    def k(self): # returning the current direction
        return self._k
    def append(self,point,direction): # setting a new point and direction
        self._all_points.append(point) #adds new point to dictionary of the points
        self._p = np.array(point)
        self._k = np.array(direction)
    def vertices(self): #returns the dictionary of points for that object
        return self._all_points

"""

"""

class RayBundle:
    """
    A class to represent a bundle of optical rays.
    """

    def __init__(self, point = [0,0,0], direction = [0,0,1], bundle_radius = 2,
ring_no = 4):
        #creates variable
```

```
        self._br = bundle_radius
        self._rngs = ring_no
        self._RayN = [Ray(point,direction)] #ray at the centre of bundle
        self._points = 1 #max number of points that a ray will propagate through
        self._ray_no = 0 #number of rays in the bundle
        for i in range(0, self._rngs): #loop through the rings
            ring_rad = i*self._br/(self._rngs-1) #the radius of the current ring
            length = self._br/(self._rngs-1) #the distance between the points
            maxrays = int(2*np.pi*ring_rad/length) #maximum number of rays in the
ring (made integer value)
            if maxrays == 0:
                maxrays = 1
            self._ray_no = self._ray_no + maxrays
            angle = 2*np.pi/maxrays #angle between the rays in the ring
            for j in range (0, maxrays): #creating all the rays in the ring
                ring_ray =
Ray([point[0]+ring_rad*np.cos(j*angle),point[1]+ring_rad*np.sin(j*angle),point[2]],
direction)
                #position of each ray in the ring
                self._RayN.append(ring_ray)

    def FireBundle(self, lens): #propagate the bundle of rays through a lens or to
the output plane
        for Ray in self._RayN:
            lens.propagate_ray(Ray)
        self._points += 1

    def plotbundle(self): #plot the rays at different points

        plt.figure()
        for ray in self._RayN:
            x, y, z = zip(*ray.vertices()) #use this function to create a
dictionary of values from the array
            plt.plot(x[0],y[0], 'bo') #this plots the input bundle of rays
            # Add title and axis names
            plt.title('Input Bundle of Rays')
            plt.xlabel('x plane /mm')
            plt.ylabel('y plane /mm')


        plt.figure()
        for ray in self._RayN:
            x, y, z = zip(*ray.vertices())
            if len(y) == self._points: #rays that have gone through all available
lenses
                plt.plot(x[-1],y[-1], 'ro')
                #this plots the spot diagram for the bundle of rays at the paraxial
focal plane
```

```
                plt.title('Spot Diagram at the Paraxial Focal Plane')
                plt.xlabel('x plane /mm')
                plt.ylabel('y plane /mm')

        plt.figure()
        for ray in self._RayN:
            x, y, z = zip(*ray.vertices())
            plt.plot(z, y, 'o-') #this plots the path of the ray bundle
            plt.title('Ray Paths')
            plt.xlabel('z plane /mm')
            plt.ylabel('y plane /mm')

    def RMS(self): #find the RMS of the spot radius at the output, from the optical
axis
        RMSvar1=0
        for ray in self._RayN:
            x, y, z = zip(*ray.vertices())
            if len(y) == self._points:
                centredistsq=((x[-1])**2+(y[-1])**2)
                #the distance of the individual point from the optical axis,
squared
            RMSvar1 = RMSvar1 + centredistsq #adding the distances together
        RMSvar2 = RMSvar1/self._ray_no #finding the mean
        RMS = np.sqrt(RMSvar2) #square root of the mean
        return RMS


"""
Representing optical system using optical elements:
    - e.g. refracting surfaces, lenses, etc
In sequential ray tracer:
    - ray propagated through each optical element in turn
    - the object representing the optical element will be:
        responsible for propagating the ray through it
Coding method:
    - have general base class: OpticalElement
    - give it method: propogate_ray
    - all optical elements will inheret this class
    - each will be required to implement propagate_ray according to their own
properties
"""

class OpticalElement: #base class for all optical elements
    def propagate_ray(self, ray): #method to describe motion of ray through
elements
        "propagate a ray through the optical element"
        raise NotImplementedError() #this forces subclasses to define the method
themselves
```

```
        #if subclass does not define this method, gives error

"""
Refracting Surfaces:

    - Design a class to represent a spherical refracting surface
    - initially restrict to surface centred on optical axis (z axis)
    - surface has 5 parameters:
        - intercept of surface with z-axis
        - curvature:
            - magnitude 1/(radius of curvature)
            - z>z(0) gives +ve (convex)
            - z<z(0) gives -ve (concave)
            - plane surface = zero curvature
        - refractive indices either side of surface
        - aperture radius (max extent of surface from optical axis)
"""

class SphericalRefraction(OpticalElement): #represents a spherical refracting
surface
    def __init__(self, interceptz, curvature, refractiveindex1, refractiveindex2,
ap_radius):
        self._z0=interceptz #intercept of surface with z-axis
        self._curv=curvature #the curvature of the surface
        self._n1=refractiveindex1 #the refractive index on front side of the
surface
        self._n2=refractiveindex2 #the refractive index on back side of the surface
        self._ar=ap_radius #aperture radius - the maximum extent of the surface
from the optical axis
        if self._curv != 0:
            if self._ar > 1/np.absolute(self._curv):
                raise ValueError("You have set aperture radius to be larger than
lens.")
    def intercept(self, ray):
        """
        Method to calculate first valid intercept of ray with surface.
        Line can have 2 intercepts with sphere - choose appropriate one.
        For no valid intercept, return None.
        Create exceptional case for zero curvature.
        """
        if self._curv != 0: #when the curvature is NOT zero
            #find radius of sphere (=1/ar)
            radius = 1/np.absolute(self._curv)
            #find centre of sphere
            # if curv +ve centre is z intercept - radius
            # if curv -ve centre is z intercept + radius
            if self._curv > 0: #concave
                Centre = self._z0 - radius
```

```
            if self._curv < 0: #convex
                Centre = self._z0 + radius
            Centre_array = np.array([0,0,Centre])
            #define starting point as latest point of ray
            startingP=ray.p()
            kmag = np.linalg.norm(ray.k())
            #find the distance between the centre and the starting point
            #can write like this as they are all in vector form
            dist1 = (startingP - Centre_array)
            #distance in vector form #direction as shown in diagram (centre to
start)
            d = np.dot(dist1,(ray.k()/kmag))**2
            c = np.linalg.norm(dist1)**2 - radius**2 #second part of eqn
            b = d-c
            if b > 0:
                if self._curv > 0: #concave
                        distance = -np.dot(dist1,(ray.k()/kmag)) + np.sqrt(b)
                        distvect = distance*ray.k()/kmag
                if self._curv < 0: #convex
                        distance = -np.dot(dist1,(ray.k()/kmag)) - np.sqrt(b)
                        distvect = distance*ray.k()/kmag
                interceptP = startingP + distvect
                interceptradius = np.sqrt(interceptP[0]**2 + interceptP[1]**2)
                if interceptradius > self._ar:
                    return None
                else:
                    return interceptP
            else:
                return None
        elif self._curv == 0: #case when curvature is zero
                lamda =  (self._z0 - ray.p()[2])/ray.k()[2]
                interceptP = ray.p() + lamda*ray.k()
                #raise TypeError("This has not been coded yet")
                interceptradius = np.sqrt(interceptP[0]**2 + interceptP[1]**2)
                if interceptradius > self._ar:
                    return None
                else:
                    return interceptP

    #for testing intercept:
        #ensure for concave, z values are < z intercept
        #ensure for convex, z values are > z intercept
        #ensure that a line running parallel along centre of lens gives [0,0,z0]
    def propagate_ray(self, ray):
        if self.intercept(ray) is None: #when there is not a valid intercept
            return None #return None to terminate the ray
        new_point = self.intercept(ray)
        incident_k = ray.k()
```

```
        if self._curv != 0:
            radius = 1/np.absolute(self._curv)
            if self._curv > 0: #concave
                Centre = self._z0 - radius
            if self._curv < 0: #convex
                Centre = self._z0 + radius
            Centre_array = np.array([0,0,Centre])
            normal_vect = new_point - Centre_array
            normal_mag = np.linalg.norm(normal_vect)
            normal_unit = normal_vect/normal_mag
        if self._curv == 0:
            Centre_array = np.array([0,0,self._z0])
            normal_unit = [0,0,-1]
        new_k = SnellsLaw(incident_k,normal_unit,self._n1, self._n2)
        ray.append(new_point,new_k)


"""
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
"""
class OutputPlane(OpticalElement): #represents a spherical refracting surface
    """
    This is the output plane that the rays will end at.
    This is made so that plotting the rays can be done.
    """
    def __init__(self, interceptz):
        self._z0 = interceptz # only need z intercept
    def intercept (self, ray):
        lamda2 =  (self._z0 - ray.p()[-1])/ray.k()[-1]
        interceptP2 = ray.p() + lamda2*ray.k()
        return interceptP2
    def propagate_ray (self, ray):
        new_point2 = self.intercept(ray)
        new_k2 = ray.k()  #no refracting here
        ray.append(new_point2, new_k2)
```