Introduction to Programming in Python

Main Project

(Module: PHYS1101: Discovery Skills in Physics – 2015-2016)

Submission Instructions

To submit your program, start DUO, go to the Discovery Skills in Physics module, click on *Assignments* in the left-hand side column and select *Programming Main Project*. Attach one (and only one) file. This should be your main project module file containing the function(s) that you have written. **Call the file ray_main.py**. Please do not write any comments in the comment box. When you are ready to submit your work, click on the *Submit* button at the bottom of the page. If you later improve your module you can submit an updated version. (**Beware**: The last version you submit will be marked).

At the top of your file (after your module documentation string), you must put two variables with your name and CIS login as follows:

USER = “Put your name here as a string”

USER_ID = “Put your CIS login here as a string”

The two variable names must be exactly as shown (upper case). These variables will be read and used by the marking program. The mini-project is due by **14.00 on Thursday 3 March** (please check for updates to this date if there is CIS network maintenance during the previous weekend).

Assessment

The main project will be assessed **summatively.**

Since part of the marking process is automated, it is important to name functions exactly as instructed in this script or they will not be found by the automatic marking process and you may lose marks.

Arguments to functions should be accepted **in the same order as defined in this script** or they will be incompatible with the automatic marking process and you will lose marks.

The marking pro-forma is available on DUO. Please look at it so that you are aware of the marking criteria. Your marks and feedback will be returned to you on this pro-forma before the end of term.

Level 1 python main project: Ray Tracing

Mini-project (recap)

This section is a recap of the mini-project task (included for reference). You should already have completed this task and you should have a working function called `refraction_2d().` The main project builds on this so, if you did not complete the task successfully, you should refer to the simplified model solution on DUO (in the mini-project folder) and ensure you understand it.

Mini-project Task

You wrote a function that evaluates the horizontal and vertical positions of the points of intersection between an array of incident rays and a specified surface. The function also evaluates the clockwise angles in radians with respect to the vertical of the refracted rays.

The function should return a 2-dimensional `numpy` array of floats that should contain, for each specified light ray (in order), the horizontal coordinate of the point of intersection with the surface (in metres), the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the refracted ray. The returned array must be ordered with these three values specified by the second index, and with the number of the ray specified by the first index (following the order of the incident rays). If *any* of the incident rays exceeds the critical angle then your function should raise an exception.

The function name, inputs and outputs should be defined as follows:

```
def refraction_2d (incident_rays, planar_surface):
    '''Docstring'''
# Your code here, which inserts real values
# into the output array refracted_rays.
    return refracted_rays
```

The arguments and outputs are defined as follows:

- `incident_rays` – a 2-dimensional numpy array of floats, with the individual incident light ray specified by the first index, and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of origin of the ray, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the incident ray.

- `planar_surface` – a 1-dimensional numpy array of floats, specifying (in order): the horizontal coordinate (in metres) of one end of the line (the 2-dimesional “surface”), the corresponding vertical coordinate, the horizontal coordinate of the other end of the line, the corresponding vertical coordinate, the refractive index of the medium on the incident side of the surface, and the refractive index of the medium on the other side of the surface.

- `refracted_rays` – a 2-dimensional numpy array of floats, with the individual refracted light ray specified by the first index (in the same order as the incident rays) and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of intersection of the corresponding incident ray with the specified planar surface, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the refracted ray.

Main Project

Your overall task is to produce an optical ray tracing module with a function for dealing with 'spherical' surfaces as well as planar surfaces, and also with opaque detector surfaces. You will also write functions capable of tracing rays through a series of such surfaces (an 'optical design') and of plotting both the ray paths and surfaces, together with an illustration of the distribution of rays at the output (a '1D image'). There will also be a function for evaluating the 'image quality' of a given optical design and finally a function for optimizing the image quality by adjusting the radii of curvature of some of the spherical surfaces. You will therefore have produced a usable optical design package. Because the optimizing function will need to evaluate the image quality several times, your functions should be written to run efficiently (i.e., in a short time) and there will be marks for this.

Task 1

Write and test a new function `refraction_2d_sph()`, which should be broadly similar to `refraction_2d()` from the mini project, but adapted to work with a "spherical" surface (such as the surface of a lens), rather than a planar surface. As we are dealing with a 2d surface, the "spherical surface" will actually be an arc of a circle, of course. The `incident_rays` argument and the returned `refracted_rays` will have exactly the same meaning as for the original `refraction_2d()`. The argument `planar_surface` will however become `spherical_surface` and this will have a different meaning in order to specify the spherical surface (this will be an arc of a circle in our 2d representation). The specification for calling the new function is as follows:

```
def refraction_2d_sph (incident_rays, spherical_surface):
    '''Docstring'''
# Your code here, which inserts real values
# into the output array refracted_rays.
    return refracted_rays
```

The arguments and outputs are defined as follows:

- `incident_rays` – a 2-dimensional numpy array of floats, with the individual incident light ray specified by the first index, and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of origin of the ray, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the incident ray.

- `spherical_surface` – a 1-dimensional numpy array of floats, specifying (in order): the horizontal coordinate (in metres) of one end of the arc (the 2-dimesional spherical "surface"), the corresponding vertical coordinate, the horizontal coordinate of the other end of the arc, the corresponding vertical coordinate, the refractive index of the medium on the incident side of the surface, and the refractive index of the medium on the other side of the surface, and finally the radius of curvature in metres of the surface. A positive radius of curvature means that a ray travelling in the positive x direction will intersect a convex surface, while a negative value means that such a ray will intersect a concave surface.

- `refracted_rays` – a 2-dimensional numpy array of floats, with the individual refracted light ray specified by the first index (in the same order as the incident rays) and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of intersection of the corresponding incident ray with the specified planar surface, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the refracted ray.

Task 2

Write and test a new function `refraction_2d_det()`, which should be broadly similar to `refraction_2d()` from the mini project, but adapted to work with a "detector" surface, which is an opaque planar vertical surface. As we are dealing with a 2d surface, the vertical surface will actually be a straight vertical line, specified by its x coordinate `x_det`, a float value in metres. The `incident_rays` argument and the returned `refracted_rays` will have exactly the same meaning as for the original `refraction_2d()`, except that only the y points of intersection need to be evaluated in the output array, as the x values will be those of the detector position, and the refracted ray directions will not be meaningful as the detector is opaque (and no refractive index is given in the arguments).

```
def refraction_2d_det (incident_rays, x_det):
    '''Docstring'''
# Your code here, which inserts real values
# into the output array refracted_rays.
    return refracted_rays
```

The arguments and outputs are defined as follows:

- `incident_rays` – a 2-dimensional numpy array of floats, with the individual incident light ray specified by the first index, and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of origin of the ray, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the incident ray.

- `x_det` – a float value specifying the x position of the vertically (y) oriented detector surface.

- `refracted_rays` – a 2-dimensional numpy array of floats, with the individual refracted light ray specified by the first index (in the same order as the incident rays) and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of intersection of the corresponding incident ray with the specified detector surface (this is always the same as x_det so need not be calculated), the corresponding vertical coordinate (required), and the clockwise angle (in radians), (this need not be calculated and may be always set to zero, for example, as the surface is opaque).

Task 3

Write a function `trace_2d()` to trace the rays through a series of surfaces, which can be either planar or spherical. The inputs to this function will be the `incident_rays` (specified as before), and a list of surfaces, each element of which will itself be a list consisting of a string containing ‘`PLA`’ or ‘`SPH`’, or ‘`DET`’ to indicate whether the specified surface is planar or spherical or a detector, followed by a 1-dimensional numpy array of floats to specify the surface (in the case of ‘`PLA`’ or ‘`SPH`’) or a single float value to specify the x detector position in the case of ‘DET’. The function will return a 3-dimensional numpy array , containing the points of intersection and refracted angles (in the same format as previously) for each surface in turn. The specification for calling the new function is as follows:

```
def trace_2d (incident_rays, surface_list):
    '''Docstring'''
# Your code here, which inserts real values
# into the 3D output array refracted_rays.
    return refracted_ray_paths
```

The arguments and outputs are defined as follows:

- `incident_rays` – a 2-dimensional numpy array of floats, with the individual incident light ray specified by the first index, and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of origin of the ray, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the incident ray.

- `surface_list` – a list containing the specification of each surface in turn. Each element of the list is itself a list, the first element of which is a string containing either 'PLA' or 'SPH', or 'DET'. The second element is either a float value specifying the x position in the case DET', or a numpy array specifying either a planar or spherical surface in the same formats as used for refraction_2d and refraction_2d_sph respectively.

- refracted_ray_paths – a 3 dimensional array containing the refracted_rays output for each surface in turn. It is therefore a 3-dimensional numpy array of floats, with the refracted_rays for each surface specified by the first index (in the same order as the surfaces in surface_list).. The final surface is likely to be a 'DET, detector surface and the horizontal intersection coordinates and output ray angles need not be calculated for such a surface.

Task 4

Write a function plot_trace_2d() to plot the output from trace_2d() showing the surfaces and ray paths, and also illustrating the 1-D 'image' produced on the final surface by the incident rays. This final surface may be taken as the opaque surface of a detector, and the final refracted rays from this plane should be ignored. This task allows relatively wide flexibility, and you are intended to design a plot using your own ideas about how to best display the data. You should use the online plotting tutorials and documentation (see workshop 6) to discover how to produce this plot. The specification for calling this new function is as follows:

```
def plot_trace_2d (incident_rays, refracted_ray_paths,
surface_list):
    '''Docstring'''
# Your code here, which produces the plot
# There is no return value
```

The arguments are defined as follows:

- incident_rays – this has the same form and meaning as it does for all of the previously defined functions above.

- refracted_ray_paths – this has the same format as the output returned by trace_2d().

- surface_list – this has the same format as the corresponding argument to trace_2d().

Task 5

Write a function `evaluate_trace_2d()` to measure the image quality in the output from `trace_2d()` on the final surface, which will always be a detector. You need to return a single float value containing the fraction of rays arriving at the final surface that fall within the 'circle' specified by a given radius `r`, in metres. The centre of this 'circle' is the mean vertical position of all arriving rays.

```
def evaluate_trace_2d (refracted_ray_paths, r):
    '''Docstring'''
# Your code here, which inserts a float value
# into the output variable frac.
    return frac
```

The arguments and outputs are defined as follows:

- `refracted_ray_paths` - this has the same format as the output returned by `trace_2d()`.
- `r` – a float value specifying the radius of the 1-dimensional 'circle' on the detector for which the fraction of rays arriving within it must be calculated. The circle is centred around the mean vertical arrival position for all the rays
- `frac` – a float value specifying the calculated fraction of rays within the specified circle.

Task 6

Write a function `optimize_surf_rad_2d()` to calculate the optimum radius of curvature values for the (spherical) surfaces specified by the numpy array `n_surf`, with the given `incident_rays`, `surface_list`, and evaluation radius, `r`. The optimum radius of curvature values are the ones that give the highest return values from `evaluate_trace_2d()` (when called with the evaluation radius `r`). The surfaces whose radii are to be optimized are specified by the 1-dimensional numpy array of integers, `n_surf`, which contains the index values into the given `surface_list`. Each specified radius of curvature may be varied by up to 10% of its originally specified value in order to find the optimal values.

```
def optimize_surf_rad_2d (incident_rays, surface_list, r, n_surf):
    '''Docstring'''
# Your code here, which inserts float value(s)
# into the output numpy array rad_opt.
    return rad_opt
```

The arguments and outputs are defined as follows:

- `incident_rays` – a 2-dimensional numpy array of floats, with the individual incident light ray specified by the first index, and the following values specified by the second index for each ray (in order): the horizontal coordinate (in metres) of the point of origin of the ray, the corresponding vertical coordinate, and the clockwise angle (in radians) with respect to the positive vertical direction of the direction of the incident ray.

- `Surface_list` – this has the same format as the corresponding argument to `trace_2d()`.

- `r` – a float value specifying the radius of the 'circle' on the detector for which the fraction of rays arriving within it must be calculated. The circle is centred around the mean vertical arrival position for the rays.

- `n_surf` – a 1-dimensional numy array of integers which specify the the index values of the surfaces in surface_list, the radius of curvature values of which is to be optimized.

- `rad_opt` – a 1 dimensional numpy array of float values containing the calculated optimal radius values for the specified spherical surfaces in `n_surf and` following the same order.

Simplifications

You do NOT have to worry about:

1. Rays that miss a surface. All of the test rays specified in the marking program will hit the specified surfaces.

2. Rays that have clockwise angles to the positive vertical axis of greater than or equal to pi radians. All of the test rays will all have angles less than this.

3. Planar Surfaces or rays with infinite gradients. All spherical surfaces will be symmetrical about the x-axis and may be taken as the surfaces of spherical lenses. All detector surfaces will be vertical.

Testing

Test your code thoroughly before submitting your work. Do this by calling your function with sample input numbers and confirming that the output is what you expect.

Note that your code should behave as a module. If it is imported, no code in the module should be executed unless a function in the module is explicitly called. If you want to include code that runs when the python file is executed you should do this in a "test block" as follows:

```
if __name__=='__main__':
     # Your test code indented here
```

You may submit your module with test code included, as long as it is in a test block. Your test code will not be marked.

Please note that your project will be marked by an automatic marking program that will import your module and call your function with some test values. It will test the execution speed and the accuracy of the returned results.

Main Project Marking Scheme

Results and Correctness (marked automatically) 30/100

Does the module import correctly?

Does the module produce correct results for the various stages of the project?

Efficiency (marked automatically) 20/100

Has the module been written efficiently?

How quickly does it run?

Features and Techniques Used (marked manually) 25/100

Has the student used appropriate features of the Python language?

Are the techniques used appropriate to the problem?

Style (marked manually) 25/100

Is the module well structured?

Were function and variable names clear and well chosen?

Is the program clearly documented through the use of Python document strings and in-line comments?

The Main Project is worth 20% of the Discovery Skills module