Introduction to Programming in Python

Ray Tracing Main Project
Frequently Asked Questions Version 2.0

Q1: Do I need to put all of my functions for the various tasks into one file?
YES, only one file can be submitted via duo. If you try to import any other files of your own, then this will fail when tested by the marking program, because your other files will not be present.
During marking, you can only rely on being able to import the standard python library modules (as described in python.org), and numpy, matplotlib and scipy.
Q2: Do I need to include a working function refraction_2d for a planar surface in my submitted file?
Yes
Q3: Can I use the model mini project solution?
Yes, you will not loose marks for doing this (though it will clearly not be possible for that part of your model solution to run faster than the model solution – see Q10).
Q4: Can I use the scipy library?
Yes. Although the course notes say you will not need to use scipy, this simply means that it is not compulsory. it is fine to use it if you want to.
Q5: Related questions: [Are the spherical surfaces all symmetrical about the x-axis? Does that mean that the centre of curvature is always on the x-axis? Does it mean that the x values for both ends of the spherical surfaces are always identical? Does it mean that the y value for one end of the spherical surface is the same as the other end, but with the opposite sign?]. What kind of physics problem will we test the program with? What will be the order of the surfaces?
[Yes, Yes, Yes, Yes]
This illustrates the kind of physical ray tracing problem we will try to address:
https://en.wikipedia.org/wiki/Tessar#/media/File:Tessar.png
This shows a 1902 camera lens (a Carl Zeiss Tessar) composed of a series of spherical lenses. We will not try to optimize anything quite as complex as this, but we almost could! We could not model the vertical planar surfaces without a small modification of our code, and we could not model the aperture stop in the centre of the design without another small modification. We would not need to worry about the straightened edges of the lenses, as our test rays will not pass through these areas.
Just like this illustration, all our spherical surfaces will form spherical lenses, which are symmetrical about the x axis. All of the rays will start at the left of the design and proceed to the right, passing through all the surfaces in order.
We can also have prisms composed of two planar surfaces.
The last surface on the right will always be a detector surface, and no rays can pass beyond this point.
The order of the surfaces listed in the function parameters will always be left to right (i.e., the direction of increasing x coordinates).
Q6: [more advanced question] Why do we have apparently redundant parameters for the spherical surfaces, which would allow us to shift them off axis and to tilt them (when we don't intend to do this).

This is to illustrate the good practice of allowing for a future upgrade when designing "software interfaces" (the function parameters in this case). The currently redundant parameters that would allow us to implement advanced tilt-shift lenses do not cost us anything computationally and would avoid an awkward and error-prone re-ordering of the parameters if we ever did upgrade the code capabilities in this way.

Q7: How do we set up the output array for task 3?

Here is ONE way:

```
# create an array to hold the refracted
# rays from each surface in turn.
refracted_ray_paths = numpy.zeros([len(surface_list),
incident_rays.shape[0], incident_rays.shape[1]])
```

as the `refracted_rays` array from each surface is calculated, it can then be stored successively in this 3-dimensional ray:

```
    refracted_ray_paths[i] = refracted_rays
```

where `i` is the number of the surface (in left to right order, starting at zero).

Q8: Do we need to use a `for` loop in task 3 (to examine the successive surfaces)?

Task 3 was designed so that a `for` loop would be required. This is in order to provide practice with situations where faster numpy operations could not be easily used. There are ways to reduce the impact of this in order to increase efficiency, but please see Q10 below before attempting this.

Q9: can we optimize the radii of the specified surfaces in task 6 independently (i.e., one at a time)?

No, you cannot assume that you can do this. The optimum value for one surface may well depend on the current value of anther, so you will need to test combinations of values.

Q10: How do we tell if our code is running fast enough to get the marks for computational efficiency? How much effort should we put into this? How do we achieve efficiency?

The project is marked out of 100 in total, and if your code runs as fast as the reference implementation, then you will get 16 of the 20 marks allocated to efficiency. Task 4 (the plotting function) will NOT be marked for efficiency.

It is important to note from the outset that, although there are a few marks allocated to beating the speed of the reference implementation (4/100), IT IS NOT WORTH SPENDING A LONG TIME ON THIS and it should not be attempted until everything else is working. There are far more marks for a working slow implementation than a fast implementation that does not work. If you have the time and interest, then looking at faster implementations would be very useful experience in the long term: efficiency techniques are often of very high importance in computational physics.

It is difficult to specify an absolute time required to equal or beat the reference implementation. This is because the absolute execution time of a function will depend on the processor architecture and clock speed, cache implementation, operating system and configuration, system load, network activity, etc. The marker program therefore works by comparing your program speeds with the reference implementation. You can do approximately the same by assuming that the reference implementation will be very much at the same level of efficiency as the mini project model solution: it will use numpy arrays for efficiency where this is straightforward, and will use a simple for-loop implementation of task 3. If you follow the same

approach you will get most of the marks allocated for efficiency. If you ARE interested in going further and beating the reference implementation, then read on…
The main lesson will be that there is no single answer to achieving efficiency and you may need to research (using WWW resources, say), implement and test various approaches, You can basically speed up your implementation in two ways: (a) increase the speed of each evaluation of tasks 1, 2, 3, and 5, and (b) make the optimization operation in task 6 use fewer evaluations of these tasks. In all cases, you need to time the effectiveness (relative speedup) of anything you try (using, for example, the python time module or a more sophisticated execution time profiling package).
Some starting hints:
The speed of executing task 6 MIGHT be improved by finding a way of doing some of the interpretation of the surface list normally required in each call of task 3 only once at the start of an optimization run.
It will probably be possible to home in on the right solution in task 6 without laboriously evaluating all combinations. There are optimization algorithms in scipy that will enable this.

Q11: Will the marker program test my functions with many rays
Yes, many rays will be used to test your functions, particularly for task 6.

Q12: The surfaces seem very odd in the supplied test program. Will the marker program use a surface list like this?

No. The email that came with the test program said:
"Please note that this test program does not use a set of surfaces that correspond to a realistic optical design, nor the arrangements of surfaces that will be used to mark your program. It is simply a test of the interfaces. The recommendation is that you adapt the test program to use sets of surfaces (such as simple lenses and prisms), where you know what the plot will look like, and where can easily calculate (for the lenses) what the optimal configuration will be."

See also the response to Q5.

Q12: SO what surfaces should I use to test my program (especially task 6)?
A good example would be a single simple symmetrical thin convex lens (2 spherical surfaces), where you could use the lens maker formula to calculate the focal length from the refractive index and radius of curvature. You know that a set of rays (use lots!) traveling in parallel (from infinity) parallel to the x axis will focus at the detector surface if it is situated at the focal distance from the centre of the lens. By slightly adjusting the detector away from this position and optimising the two radii of the lens surfaces, the lens should refocus itself with a good peak in the returned `frac`, if a tight (small) value is used for `r`.

Q13: How do I go about task 6?

Firstly, you do not need to assume that there is any analytical formula available for task 6. It would become very complex as more surfaces, ray origins, directions and refractive index values are added. Such problems are rarely solved analytically these days.

This exercise is intended to practice using a numerical method which searches for a solution by repeatedly calling trace_2d and evaluate_trace_2d and finding the combination of radii of curvature which gives the highest value of frac. This exercise is in preparation for dealing with the large class of current physics research problems where no analytical solution is known.

You could do very well by having an n-dimensional "grid" of all the combinations of radius values covering the search range (+/-10%) and searching through those for the best solution. If you do this aim to get the radii within +/-1% of their realoptimal value. At this point you would have most of the marks. (the "+/-1%" tells you the step size required for your grid).

To make this go faster you could start with a sparser grid to find an approximate set of good radius values then "zoom in" with a more detailed search. But this would be getting harder than using scipy optimizer and there are not many marks available for going any faster

A couple of points to note if you do want to go ahead with an optimizer/minimizer

1. minimising rather than maximising a quantity is just a convention. You could just minimize `-1.0*frac` for example.

2. The Nelder-Mead method in generally gives good results for this type of problem, even though it is an unconstrained local optimizer.

Q13: So am I allowed to use an unconstrained local optimizer like Nelder and Mead?

YES, in fact it is encouraged, and can get you full marks for task 6. The +/-10% range only applies to the "grid" type searches referred to in the response to Q12.

Q14: how will the marking of task 6 work?

We will look at both execution speed and accuracy of results. A slow but very accurate task 6 will do very well. So would a very fast, less accurate task 6. Obviously a very slow and very inaccurate task 6 will do less well, but this is just one task and so there are not many marks available here anyway (see Q10).