Computational Physics

Practical exercises

David Buscher

Version 3.1 February 2022

Contents

Introduction

These exercises are designed to help you learn some `Python`, understand some physics and learn a little more about numerical programming. You should attempt the exercises one per week, in the order given. Demonstrators will be on hand during the practical sessions to assist.

The speed at which people can program varies widely. It even varies for individuals day by day — all programmers will sympathise with how much time can be wasted trying to fix an obscure bug. So do keep an eye on the clock, and do seek assistance. Talk to your colleagues so you can share experience and spot each others' mistakes. The goal of the exercises is for you to learn – and you can only do this by experience.

Each exercise is split into tasks. The *core task or tasks* are ones that I hope you will all be able to achieve in the class in the time available. If you hand in a working solution to this you will get approximately 60–70% of the credit for the exercise. There is also one or more *supplementary tasks* per exercise, which take the problem further, or in a different direction. I hope that many of you will achieve some or most of these, and they count for the remaining fraction of the credit.

I have given structured hints to each problem. In addition, you should look at the code examples in the lectures and, importantly, **consult the relevant documentation for the libraries** such as `scipy` and `numpy` — finding and learning from online documentation and code examples is an important skill. There are some starting links from the course web-page or just use Google/Bing/DuckDuckGo.

Assessment

Over the 4 weeks you should carry out three exercises: Exercise 1, Exercise 2, and either Exercise 3A *or* 3B. The marks available are a maximum of 6 for exercises 1 and 2, and a maximum of 8 for exercise 3A or 3B, for a total of 20 marks.

You should upload your solution to the TiS. The **deadline** is posted on the course web site and the TiS. You are encouraged to submit your work as soon as it is in a reasonable state, and move on. So don't spend too much time on the tasks, but don't spend too little time either.

Note that the procedures explained below assume that you are completing the exercises in Python, which is strongly recommended. If you want to submit C++ code instead of Python then please **email me as soon as possible to make alternative arrangements.**

For each exercise you should submit a Jupyter notebook file (a `.ipynb` file) which is structured into tasks as explained in the next section. This notebook will contain Python code, plots illustrating relevant results, and text commentary. The text commentary describes very briefly what you did, any major problems, mentions any numerical answers required and describes the accompanying plots.

It is possible in Jupyter notebooks to execute your code cells in any order. You should make sure that, in your submitted notebook, the code will run correctly when you restart the Python interpreter and run all the cells in the order that they appear (for example using the "Resart and run all" function under the "Runtime" menu in Google Colab). This is so that the logical order that the code should be run is clear to someone reading the code. Breaking your code up into functions will help with the organisation and ordering of your code especially when you need to do different variations of the same calculation.

Note that the usual rules with respect to plagiarism apply: you are of course allowed to discuss the work with others and to refer to online or other sources, but you must clearly state where any of the work you submit is not your own work, or is joint work, and explicitly attribute the authors/sources of that work.

Code quality

In the first two exercises, the main emphasis of the marking will be on successful completion of the tasks. In the last exercise a percentage of the marks will be given for code quality (this will also be true for the optional project). High-quality code will include at minimum:

- Structured code: tasks broken down into sensible functions;
- Meaningful function names;
- Meaningful variable names (this is less important than for naming functions: single-letter variable names can be meaningful if the meaning can be inferred from context, e.g. loop counters);
- Appropriate levels of commenting (at minimum a Python "docstring" identifying what each function does);
- Sensible use of whitespace to indicate code structure.

Remember, code quality is important because (a) it helps to make the code easier to understand and debug for the person writing the code and (b) it makes the code easier to understand and debug for others who may have to modify the program later on.

Writing high-quality code is somewhat like writing high-quality prose or mathematics: clearly structuring and explaining your thoughts for someone else can help clarify your own thinking, and this clear explanation is what code quality is all about.

Getting started

Using Google Colab

We will be using the Jupyter notebooks provided by Google Colab at https://colab.research.google.com. It is best to log in with your University of Cambridge account: you can do this by logging out of any other Google accounts you may be logged into and then logging back in using `crsid@cam.ac.uk` as your username — if you are then prompted you about a choice of accounts associated with this email address, always choose the "Google workspace account" rather than the "Personal account". You can also use Colab with any other Google account you may have, but this may give rise to problems when you want to get access to any shared files or to share notebooks with others, e.g. demonstrators.

There are plenty of tutorials available in Google Colab and elsewhere about how to use Jupyter notebooks, but an effective way to learn for many people is to just open a new notebook and start experimenting. To do this, click on the "New notebook" link in Colab. The first thing you should do after creating a new notebook is to rename the notebook by clicking on the existing title (usually something like "Untitled0.ipynb") and editing it to give it a descriptive name such as "Exercise 1".

Structuring the notebook

One feature of Jupyter notebooks we will be making use of in the exercises is the use of text cells as well as code cells. Code cells are the default cell type and contain the Python code you are developing. You will use text cells to put in section headings to separate different tasks, and also to put in text commentary and conclusions.

The text cells use a format called "markdown" which allows you to type in plain text into the cell and get various "rich text" effects such as section headings, bold, italic, and even nicely-formatted mathematics when the cell is "run". You can find out more about markdown online, for example at https://colab.research.google.com/notebooks/markdown_guide.ipynb.

With this in mind, you should insert a text cell (you can use the `+ Text` menu item near the top of the screen for this) at the beginning of your notebook that gives it a section heading, for example:

```
# Core task 1

Some text describing the task (optional - this can be used to
remind yourself what you are aiming to do).
```

When you press the arrow to "run" the cell it will turn it into a bold-font first-level heading, plus any text you typed in.

After that you should probably insert your first code cell. A suggested first cell is one which just imports all the relevant modules, e.g.

```
import matplotlib.pyplot as plt
import numpy as np
import scipy
```

You can then run this cell and in the next code cell start writing the first part of your code.

At the end of each task, you should add a text cell with a conclusions sub-heading and text in the form

```
## Conclusions

Some brief words here about what you did and what
you achieved in this task.
Any numerical results should be stated here.
```

You can of course add additional text cells in the notebook to annotate your work for yourself and/or the assessors. This section structure should be repeated for all the core and supplementary tasks. You should submit one notebook per exercise, containing the code and results for all the tasks in that exercise.

Plotting your results

You will want to present many of your results in the form of graphs made using pyplot, which will appear below the code cell when the cell is run. **Remember to add captions for the axes and a title.** Here is a simple Python code fragment which plots a sine function with axes.

```
plt.plot(np.sin(np.linspace(-10,10,100)))
plt.xlabel("Distance along axis (m)")
plt.ylabel("Intensity (arbitrary units)")
plt.title("Diffraction pattern of a double slit");
```

Note the use of a semicolon on the last line to suppress any printed output — not essential but makes the notebook look cleaner. You can also use the "object-oriented" pyplot plotting style used in places in the lectures, which can be helpful when you are plotting multiple plots in the same figure.

It is usually a good idea to do the long-running part of any calculation in one code cell and then do the plotting in a subsequent code cell. This way, you can change the plotting code and re-run it to improve the appearance of the plot and add labels etc without re-running all of the calculations.

If you are doing very-long-running calculations you may want to save the results to a file to analyse/-plot later. You can use the python "pickle" module as one way to do this.

Uploading your notebook to the TiS

You should use one notebook per exercise. When you are finished the exercise and are ready to submit it for assessment, you can use the "Download .ipynb" item under the "File" menu in Colab. You can then upload the file from your computer to the TiS.

Exercise 1: The Driven Pendulum

Goal

To explore the physics of a non-linear oscillator (a damped, driven pendulum) by accurate integration of its equation of motion.

Physics

The pendulum comprises a bob of mass m on a light rod of length l and swings in a uniform gravitational field g. If there is a resistive force equal to αv where the bob speed is v, and a driving sinusoidal couple G at frequency Ω_D, we can write

$$m\,l^2\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -m\,g\,l\,\sin(\theta) - \alpha\,l\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + G\sin(\Omega_D t) \tag{1}$$

Rearranging:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\,\sin(\theta) - q\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + F\sin(\Omega_D t) \tag{2}$$

where we have defined $q \equiv \alpha/(m\,l)$ and $F \equiv G/(m\,l^2)$.

For the purposes of this exercise, take $l = g$, so the natural period for small oscillations should be 2π seconds. Also let the driving angular frequency be $\Omega = 2/3\mathrm{rad\,s^{-1}}$. This leaves q and F, and the initial conditions, to be varied. For all of these problems, we will start the pendulum from rest i.e. $\dot\theta = 0$ at $t = 0$, but we will vary the initial displacement θ_0. The parameter space to explore then is in the three values q, F and θ_0.

Tasks

Core Task 1 First re-write this second-order differential equation Equation 2 as a pair of linked first-order equations in the variables $y_0 = \theta$ and $y_1 = \omega = \dot\theta$. Now write a program that will integrate this pair of equations using a suitable algorithm, from a given starting point $\theta = \theta_0$ and $\omega = \omega_0$ at $t = 0$. I recommend using `scipy` rather than implementing your own Runge-Kutta or other technique.

Test the code by setting $q = F = 0$ and starting from $(\theta_0, \omega_0) = (0.01, 0.0)$, and plotting the solution for 10, 100, 1000…natural periods of oscillation. Overlay on your plot the expected theoretical result for small-angle oscillations — make sure they agree!

Test how well your integrator conserves energy: run the code for, say, 10,000 oscillations and plot the evolution of energy with time.

Now find how the amplitude of undriven, undamped oscillations depends affects the period. Plot a graph of the period T versus θ_0 for $0 < \theta_0 < \pi$.

> **Include in your notebook:** Source code, appropriately-scaled plots showing how well energy is conserved in at least one case and a plot of period versus amplitude, conclusions text containing a couple of sentences summarising what you managed to achieve, and the value of the period for $\theta_0 = \pi/2$

Core task 2 Now turn on some damping, say $q = 1, 5, 10$, plot some results, and check that the results make sense. Now turn on the driving force, leaving $q = 0.5$ from now on, and investigate with suitable plots the behaviour for $F = 0.5, 1.2, 1.44, 1.465$. What happens to the period of oscillation? *Note that the period of oscillation is best observed in the angular velocity rather than angular position, to avoid problems with wrap-around at* $\pm\pi$.

> **Include in your notebook:** Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the displacement θ and the angular velocity $\frac{\mathrm{d}\theta}{\mathrm{d}t}$ versus time.

Supplementary task 1 Investigate the sensitivity to initial conditions: compare two oscillations, one with $F = 1.2, \theta_0 = 0.2$ and one with $F = 1.2, \theta_0 = 0.20001$. Integrate for a 'long time' to see if the solutions diverge or stay the same.

> **Include in your notebook:** Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the behaviour.

Supplementary Task 2 Try plotting angle versus angular speed for various solutions, to compare the type of behaviour in various regimes: you can investigate chaotic behaviour using this simple code — have a look in the books or a web site for examples. There is a nice demo for example at http://www.myphysicslab.com. There's lots more physics to be explored here — experiment if you have time!

Include in your notebook: Source code, a sentence or two in the conclusions sub-section explaining what you see in the solutions, and illustrative plots of the behaviour.

Hints

1. Hopefully rewriting the equation as 2 ODEs is straightforward: if not, ask!
2. I recommend that you use a "canned" ODE integrator, for example the `scipy.integrate.solve_ivp()` function, rather than writing your own.
3. You will need to write a function that evaluates the derivatives of the ODEs at a given time given the current values of the variables. You can look at the example code solving the spinning ring problem discussed in lectures. You can perhaps adapt some of that code if you get stuck.
4. Think about the time window you choose for your plots: in some cases, plotting less than the full set of data you have computed may make for clearer visualisation of what is going on.
5. Many of the ODE integrator functions in `scipy.integrate` and elsewhere are adaptive-stepping algorithms, and have options which are set using "keyword arguments" to control this behaviour. Firstly, you can set a desired accuracy for the solution: setting a higher accuracy may make the program run more slowly because it will typically take shorter steps. Secondly, the functions usually have the capability to return an interpolated value of the solution at user-specified times, and not just at the time steps used to integrate the ODE. You can choose a time sampling to give appropriately smooth plots. It pays to experiment with these values to see if they have any effect on your results, especially when investigating "chaotic" behaviour.
6. To find the period versus amplitude relationship, a simple (and just about acceptable) way is to measure the period off a suitable plot, and do this for several values of θ_0. However, it is much better to alter your code to estimate the period directly. You can then loop over θ_0 values and plot the period versus amplitude relation. Two obvious approaches spring to mind. First, you can find when θ first goes negative. This is when the time is approximately $T/4$ where T is the period. How accurate would this result be? You could also find several zero crossings by considering when y changes sign or becomes exactly zero after a step is taken; by counting many such zero crossings and recording the time between them you can get a more accurate value for T.
7. Note that you need to think about what happens when the pendulum goes "over the top" and comes down the other side — you need to think about the 2π ambiguities involved, and what this means in terms of the "period" of an oscillation.

Exercise 2: Integration and Vectorisation

Goal

To learn how to evaluate integrals numerically, using (a) a self-written Monte-Carlo method and (b) a general purpose integrator from `scipy`, and to learn how to write vectorised Python code.

Background Theory

Suppose we have a function f of n parameters which we can regard as an n-dimensional vector $\vec{r}$. We want to integrate this over a multidimensional volume V. We can estimate this by taking N samples distributed at random points $\vec{r}_i$ throughout the volume V, as follows:

$$\int f\, dV \approx V \langle f \rangle \pm \sigma \tag{3}$$

where

$$\langle f \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f(\vec{r}_i) \tag{4}$$

and

$$\sigma \approx V \left(\frac{\langle f^2 \rangle - \langle f \rangle^2}{N} \right)^{1/2} \tag{5}$$

with

$$\langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f^2(\vec{r}_i) \tag{6}$$

Note that the error estimate in equation (5) is not guaranteed to be very good, and is not Gaussian-distributed: treat it as indicative only of the error. In this exercise you will estimate the error by a robust Monte-Carlo approach and compare it with the theoretical error.

Tasks

Core Task 1 : Write a program to find an approximate value of this integral and an associated error estimate:

$$10^6 \int_0^s \int_0^s \int_0^s \int_0^s \int_0^s \int_0^s \int_0^s \sin(x_0 + x_1 + x_2 + x_3 + x_4 + x_5 + x_6 + x_7) \mathrm{d}x_0 \mathrm{d}x_1 \mathrm{d}x_2 \mathrm{d}x_3 \mathrm{d}x_4 \mathrm{d}x_5 \mathrm{d}x_6 \mathrm{d}x_7$$

where

$$s = \frac{\pi}{8}$$

using Monte-Carlo techniques.

Show that the error in the integral falls off as $N^{-1/2}$ where N is the number of Monte-Carlo samples. In this task you should estimate the error on the value for a given N from the standard deviation of several independent estimates, and plot a suitable graph.

Compare your error derived from the scatter of your Monte-Carlo simulations with the theoretical predication of the error given by equation (5), using a suitable plot.

(In case you are wondering, there is an analytic answer in this case, but that's not the point! The answer is in fact $10^6 \times (70 - 16\sin(\pi/8) + 56\sin(\pi/4) - 112\sin(3\pi/8)) \approx 537.1873411$.)

> **Include in your notebook:** Source code, relevant graphical plots, and a conclusions subsection containing a couple of sentences summarising what you managed to achieve, and the integral's value and the error estimate. Specifically, include your integral value, error estimate and N value for the largest value of N for which you can compute results in about a minute. Comment on the agreement between the theoretical error prediction and the experimental scatter of your data.

Core Task 2 : Write a program to evaluate the Fresnel integrals *accurately* using a standard integration routine from the `scipy`, and use this to make a plot of the Cornu spiral using `pyplot`. Do not use a Monte-Carlo routine — they are not efficient for low-dimensional integrals — instead use a standard quadrature technique. One version of the Fresnel integrals can be written

$$C(u) = \int_0^u \cos\left(\frac{\pi\, x^2}{2}\right)\, dx$$

$$S(u) = \int_0^u \sin\left(\frac{\pi\, x^2}{2}\right)\, dx$$

Note that there is a function `scipy.special.fresnel()` whose only purpose is to evaluate this integral! However using this special-purpose function to compute your answer negates the point of understanding how to use a general-purpose integrator and will not be looked on favourably when assessing your work.

> **Include in your notebook:** source code, relevant plots.

Supplementary Task 1 Compare the speed of a *vectorised* function for computing the above Monte-Carlo integral versus one where there is less vectorisation, i.e. where there are more explicit loops in the code. For example, you can compute the argument to the `sin()` in the integrand by using the `numpy.sum()` function on an array of random numbers or instead by using a Python `for` loop.

To take the vectorisation a step further, you can compute multiple values of the integrand in a single line of code by generating a 2-dimensional array of random numbers where one of the dimensions ("axes") is length 8 and then using the `axis=` keyword argument for the `numpy.sum()` function to compute the sum only over that axis. This will result in a 1-dimensional array, and you can take the sine of the resulting array.

If your code was already as vectorised as you could make it when you did Core Task 1, then do the inverse — write a code using explicit loops to try and see how much slower it is when you remove one or more levels of vectorisation.

Use the `%timeit` "ipython magic function" in the Jupyter notebook to compare the execution times of a vectorised and less-vectorised version of the code for a suitably large value of N. If there is a large improvement in speed of the vectorised code compared with the code you used in core task 1, you may want to recompute the result for the integral derived in that task using larger values of N.

Include in your notebook: source code, quantitative statements in your conclusions subsection about the speed improvement from vectorisation.

Supplementary Task 2 :

A pendulum swinging with amplitude θ_m has $\ddot{\theta} = -(g/l)\sin(\theta)$. The period of the swing can be expressed as an elliptical integral

$$T = \sqrt{\frac{8l}{g}} \int_0^{\theta_m} \frac{d\theta}{\sqrt{\cos(\theta) - \cos(\theta_m)}},$$

Use a quadrature integrator to calculate and plot the period as a function of the amplitude of the swing. Note that the integrand contains a variable parameter θ_m. You can pass this parameter via, for example, the `args` parameter in your call to the `scipy.integrate.quad()` function. This allows you to pass arguments through the integrator which end up as additional arguments to the integrand function, in a similar way as was used in the ODE integrator in the lectures.

Compare the results to the relevant results you derived using the ODE integrator in Exercise 1, and comment on the levels of accuracy achieved by the two integrators. It would help to read the documentation for the relevant integrator functions in scipy regarding accuracy.

Include in your notebook: source code, relevant plot(s), commentary on accuracy

Hints

Monte-Carlo integration

1. The first goal is to write a function that returns an estimate of the integral's value using N sample points using Equation 3. It is sensible to make N an argument (perhaps the only argument) of this function, and make the function return the value of the estimate. This function can then be called over and over to get estimates of the integral.
2. In Python, the value of π is held in `numpy.pi`. You can simplify your life a little by having a `from numpy import pi` statement at the start of your code, then you only need to type `pi` thereafter to get its value.
3. You will need to be able to generate random points in 8-dimensional (or generally $n-$dimensional) space. Use a suitable random number generator from `numpy.random` for this. Look in the example code from the lectures to see how this can be done.
4. The basic logic you require now is to loop over increasing values of N; and for each N value, estimate the integral several times, and find a best value and error by looking at the mean and spread of estimates returned. How big should N be? Experiment! And think about the error you might want to achieve. To debug your code, a small value is good, but when you are sure things are working well you should set N to be as large as possible.
5. For a given N, estimate the integral n_t times. ($n_t = 25$ might be a sensible choice). (Make sure your random numbers are different for each trial – they will be as long as you do not re-seed the random number generator). From these n_t independent estimates you can estimate the best value (from the mean of the values) and the error in the value (from the rms of the values). You could either store these independent samples in an array so that you can compute mean and standard deviations at the end, or you can compute the mean and mean square values as you go through your loop over n_t.
6. Make your program output the estimates and errors, and plot them. For the longer runs it is likely best to save the results to a file and read them back in to a separate plotting program. You could perhaps use the `plt.errorbar()` function in `pyplot` to illustrate error bars. Also don't forget the logarithmic plotting capabilities in `plt.loglog()` for example.

Numerical Integration with `scipy`

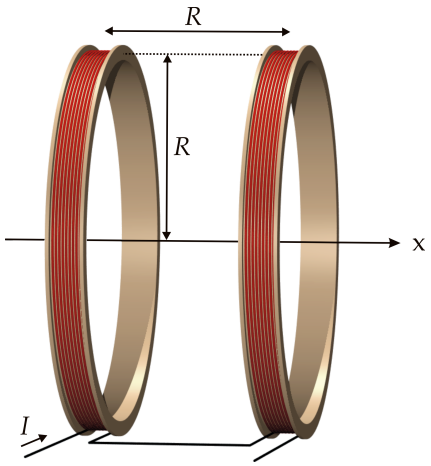
1. You will need to choose a `scipy` routine to do the integration. The `scipy.integrate.quad()` function is a good general purpose integrator. Example code is available online: search for "scipy numerical integration examples".
2. Write suitable functions that evaluate the two integrands. Make sure the functions have the correct parameters and return value types to match the function required by the relevant integration function.

3. Evaluate the integrals for various values of s and use `pyplot` to plot the spiral.
4. Plotting the diffraction pattern should now be straightforward. Remember to think carefully about the coordinates associated with your calculated pattern.

Exercise 3A: Helmholtz Coils

Goal

Write a program to calculate the magnetic field caused by Helmholtz coils. Check your solution agrees with the on-axis analytical result. Investigate with suitable plots the uniformity of the field near the centre of the system.



Physics

The physics of this exercise is straightforward: you will all know that a small length of wire $\vec{dl}$ carrying a current I creates a magnetic field

$$d\vec{B} = \frac{\mu_0}{4\pi} \frac{I\, \vec{dl} \wedge \vec{r}}{r^3} \tag{7}$$

at a location $\vec{r}$ with respect to the current element. So by breaking up any wire into short elements we can simply add up all the contributions to find the total magnetic field. Note that in this problem you are expected to do simply this. There are of course more accurate ways to compute integrals numerically — but this exercise is more about how to organise your code than how to evaluate integrals to high accuracy.

Recall that the axial field on the axis of a single coil is given by

$$B = \frac{\mu_0 I R^2}{2(R^2 + x^2)^{3/2}} \tag{8}$$

Helmholtz coils produce a fairly uniform field close to the centre of the system. They consist of two co-axial coils carrying parallel and equal currents I. The separation D of the coils centres is set equal to their radius R. For coils with axes aligned with the x axis we can place the coil centres at $(\pm(R/2), 0, 0)$. For the purposes of this problem set $R = 1.0\mathrm{m}$ and it is convenient to set the current so it has a (very large!) value of $(1/\mu_0)$ amps, to get easy-to-interpret numbers (although somewhat high field strengths!). The field expected at the coil centres is then

$$B = \left(\frac{4}{5}\right)^{3/2} \frac{\mu_0 I}{R}$$

Tasks

Core Task 1 Write a program which computes the magnetic field from a single coil of radius $R = 1\mathrm{m}$ carrying a current $I = 1/\mu_0$ amps. Put the coil centre at $(0, 0)$ and let the coil's axis run in the x direction. Calculate the field in the $x - y$ plane. First check your field on axis ($y = z = 0$) agrees with theory by plotting your result against the theoretical one. Then calculate the field on a $x - y$ grid and plot your results. To compare theory with your calculation, you should plot the *difference* between theory and calculation along the x axis, as well as the actual values.

You will need to break the wire up into a series of straight line elements, and using the Biot-Savart law, you can then add up the total field. How many elements should you break your circle into? You should experiment with different values in your code. And how does the final accuracy of your calculation depend on this value?

Core Task 2 Now adapt your program to calculate the field from Helmholtz coils arranged as in the diagram. Plot the field strength near the centre. How uniform is the field? The coil centres should be set to be $(\pm 0.5\mathrm{m}, 0, 0)$ with the coils' axes pointing in the x direction. Calculate and plot the magnetic field strength in the vicinity of $(0, 0, 0)$. Show that the field is quite uniform in this region. To quantify this uniformity, find the maximum percentage deviation of the field magnitude from that at $(0, 0, 0)$ within a *cylinder* which has diameter 10cm, length 10cm, and is coaxial with the coils. The centre of the cylinder should be at $(0, 0, 0)$. Because of the symmetry, it is of course sufficient to calculate and plot the field for $z = 0$ i.e. plot $\vec{B}(x, y, 0)$, with x and y extending to ± 5cm.

Supplementary Task Extend or modify your code to investigate the field caused by a series of N coaxial coils with uniform spacing carrying the same current. Make plots to show the effects of varying N while keeping the distance between the outermost coils constant at $D = 10R$.

Hints

There are several ways to approach this problem. Do some thinking before you start coding. First, you have to decide how to deal with the vector fields, both for the position and magnetic field vectors. One possibility is to have three arrays representing the Cartesian x, y and z components of the vectors respectively. Alternatively, a higher-dimensional array where one dimension of the array is of length three may be the way forward. The choice is yours.

You then need to think about how to represent the wire. I recommend that you write a program that is general, i.e. that can calculate the field due to an arbitrary wire. You could define the wire by a set of N line segments defined by $N + 1$ points in space, each section of wire running from (x_i, y_i, z_i) to $(x_{i+1}, y_{i+1}, z_{i+1})$, i.e. from $\vec{r}_i$ to $\vec{r}_{i+1}$ using vectors.

You can now write a function that calculates the contribution $d\vec{B}$ from each section using equation Equation 7, and add them up to find the total field of the wire. You need to evaluate the cross product of course. Be careful close to the wires where r gets small.

Now you need to create a wire which represents a single coil suitably, and evaluate the field it produces by stepping over a 2-d grid of points in the (x, y) plane.

You can try vectorising the program for greater speed, for example using the `numpy.sum()` function to do the integration over all the wire segments. Another opportunity for vectorisation is the loop over all the grid-points where you calculate the field (`numpy.mgrid` and related functions may help with this).

You can plot the 2-dimensional variation of the magnitude of the field as an image using the `plt.imshow()` function used in the lectures, but you will likely find that using `plt.contourf()` gives a visualisation which is more easy to interpret quantitatively. You can plot the vector field using `plt.streamplot()` or `plt.quiver()` functions.

Exercise 3B: Diffraction by the FFT

Goal

Write a program to calculate the near and far-field diffraction patterns of an arbitrary one-dimensional complex aperture using the Fast Fourier Transform technique. Test this program by using simple test

apertures (a slit) for which the theoretical pattern is known. Investigate more complicated apertures for which analytical results are difficult to compute.

Physics

Plane monochromatic waves, of wavelength λ, arrive at normal incidence on an aperture, which has a complex transmittance $A(x)$. The wave is diffracted, and the pattern is observed on a screen a distance D from the aperture and parallel to it. We want to calculate the pattern when the screen is in the far-field of the aperture (Fraunhofer diffraction) and also in the near-field (Fresnel).

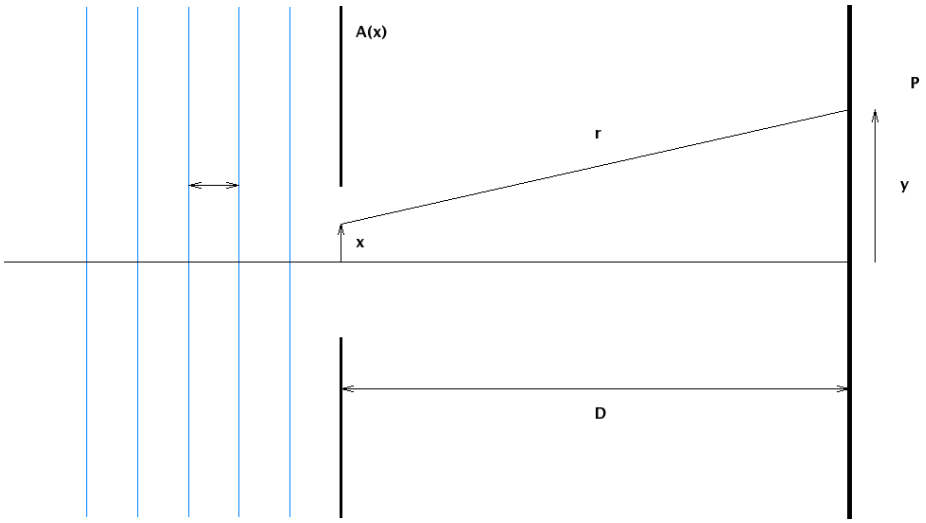


Figure 1: Geometry for diffraction calculation

Using Huygen's construction, we can write the disturbance at a point P on the screen, a distance y from the axis, as

$$\psi(y) \propto \int \frac{A(x)\ \exp(ikr)}{r}\,\mathrm{d}x$$

where $k = 2\pi/\lambda$. We have assumed that all angles are small:

$$x, y \ll D$$

so that we are close to the straight-through axis and can therefore neglect terms like $\cos(\theta)$ which

appear if we are off-axis. We now expand the path length r in powers of x/r:

$$r^2 = D^2 + (y-x)^2$$

$$r \approx D + \frac{y^2}{2D} - \frac{xy}{D} + \frac{x^2}{2D} + \mathcal{O}\left(\frac{(y-x)^4}{D^3}\right)$$

If we now neglect the variation in r in the denominator of the integral, setting $r \approx D$, which is adequate for $x, y \ll D$, then we can write

$$\psi(y) \propto \frac{\exp(ikD)}{D} \exp\left(\frac{iky^2}{2D}\right) \int A(x) \exp\left(\frac{ikx^2}{2D}\right) \exp\left(\frac{-ikxy}{D}\right) \mathrm{d}x \tag{9}$$

The diffraction pattern is thus the Fourier-transform of the modified aperture function A':

$$\psi(y) \propto \exp\left(\frac{iky^2}{2D}\right) \int A'(x) \exp\left(\frac{-ikxy}{D}\right) \mathrm{d}x \tag{10}$$

with

$$A'(x) = \exp\left(\frac{ikx^2}{2D}\right) A(x) \tag{11}$$

In the far-field (Franhofer limit) we have $kx^2/(2D) \ll \pi$ so that $A' \approx A$ for all values of x in the aperture where $A(x)$ is non-zero, i.e. the familiar result

$$d \gg \frac{x_{\mathsf{max}}^2}{\lambda}.$$

The distance $x_{\mathsf{max}}^2/\lambda$ is the Fresnel distance. In this case, the diffraction pattern is just the Fourier transform of the aperture function.

Note that we can calculate the near-field (Fresnel) pattern also if we include a step to modify the aperture function according to Equation 11 *before we take its Fourier transform*.

Note that if we are only interested in the pattern's intensity, we can ignore the phase prefactor in Equation 10.

Finally, we can discretize Equation 10, by sampling the aperture evenly at positions x_j

$$\psi(y) \propto \Delta \sum_{j=0}^{N-1} A'(x_j) \exp\left(\frac{-ikx_jy}{D}\right) \tag{12}$$

where Δ is the distance between the aperture sample positions x_j.

One convenient definition of the sample points in x is

$$x_j = (j - (N/2))\Delta, \tag{13}$$

where N is the number of sample points in the aperture. Note that this definition of the x-coordinate is equivalent to applying a coordinate transform equivalent to the "fftshift" operation described in the lectures, and results in Fourier phases which are closer to zero compared to a more simple linear relationship between x_j and j.

Tasks

Core Task 1 : Write a program that will calculate the diffraction pattern of a general 1-dimensional complex aperture in the far field of the aperture using FFT techniques. The program should calculate the intensity in the pattern across the screen, which you should plot using the correct y coordinates (in metres or microns for example). **Label your coordinates.**

Test this program for the specific case of a slit in the centre of an otherwise blocked aperture: take the single slit to have width d in the centre of an aperture of total extent L. For definiteness, use $\lambda = 500\,$nm, $d = 100\,\mu$m, $D = 1.0\,$m and $L = 5\,$mm. Overlay on your plot the theoretical value of the intensity pattern expected.

Core Task 2 : Now calculate and plot the Fraunhofer diffraction pattern of a *sinusoidal phase grating*. This grating is a slit of extent $d = 2$mm, outside of which the transmission is zero. Within $|x| < d/2$, the transmission amplitude is 1.0, and the phase of A is

$$\phi(x) = (m/2)\sin(2\pi x/s)$$

where s is the spacing of the phase maxima, and can be taken as 100microns for this problem. For this calculation, use $m = 8$. The Fresnel distance d^2/λ is 8 m, so calculate the pattern on a screen at $D = 10\,$m. What do you notice about the resulting pattern?

Core task 3 : Now modify your program so that the calculation is accurate even in the near-field by adding a phase correction to the aperture function as defined by Equation 11. Repeat your calculations in the previous two tasks for $D = 5\,$mm for the slit, and $D = 0.5\,$m for the phase grating, and plot the results. Do the intensity patterns look sensible?

Supplementary Task : Calculate and plot the diffraction pattern intensity on a screen at a distance $D = 5\,$mm for a set of (a) 2 slits and (b) 3 slits, where the slits all have widths of $100\,\mu$m and centre-to-centre spacings of $200\,\mu$m (note that the slits are clear, in other words they do not have a sinusoidal phase grating in them).

Hints

Recall the FFT definition:

$$H_j = \sum_{m=0}^{N-1} h_m \, e^{2\pi \, i \, m \, j/N} \tag{14}$$

which maps N time-domain samples h_m into N frequencies, which are

$$f_j = \frac{j}{N\Delta} \tag{15}$$

You can think of frequencies $(j/N) \times (1/\Delta)$, running from $j = 0$ to $(N - 1)$, with

- $j = 0$ is zero frequency.
- For $1 \leq n \leq (N/2)$, we have positive frequencies $(j/N) \times (1/\Delta)$.
- For $(N/2) + 1 \leq j \leq (N - 1)$ we have *negative frequencies* which we compute as $((j/N) - 1) \times (1/\Delta)$. (Remember the sequences are periodic).

Of course in this case we don't have time-domain samples, but we can still use the FFT to carry out the transform.

The complex aperture function will be represented by an array of N discrete complex values along the aperture, encoding the real and imaginary parts of $A(x)$. Each complex value represents the aperture's transmittance over a small length Δ of the aperture, so that $N\Delta$ is the total extent of the aperture.

Choose appropriate values for N and Δ to make sure you can represent the whole aperture of maximum extent L well enough. Bear in mind that Fast Fourier Transform calculations are fastest when the transform length is a power of 2, and that you want Δ to be small enough to resolve the features of the aperture. (Computers are fast these days though; so you can use small values of Δ and correspondingly large values of N. In practice, for such a small problem, the use of N as a power of 2 is not necessary, but it is important if performance is critical.)

For the slit problem you can use the `numpy.zeros()` function to set up an array of the appropriate size filled with zeros and then set the locations where the slit is transparent by assigning non-zero values to "slices" e.g. `a[5:10]=1.0`.

You now need a routine to calculate the fast Fourier transform (FFT). You can use the `numpy.fft.fft()` function, which is straightforward to use (see the code examples from the lectures). It accepts a real or a complex input and produces a complex output. You will need to compute the intensity of the pattern. You might also want to plot the aperture function to make sure you have calculated it correctly.

Now think carefully about the coordinates associated with your calculated pattern. The discussion at the beginning of this section reminds you about how the frequencies appear in the FFT'ed data. Can

you understand the form of the intensity pattern you have derived?

To plot the intensity on the screen as a function of actual distance y, you need to work out how to convert the pixels in the Fourier transform into distances on the screen y. To do this you first need to compare carefully Equation 12 and Equation 14 (also referring to Equation 13) which should tell you how to derive y at each pixel value. In addition, by interpreting the second half of the transform as negative frequencies (or y values in this case) you should be able to plot the intensity pattern as a function of y for positive and negative y, and plot over this the matching sinc function for a slit. **If you are having difficulties with this step it may help to revise your notes from the lectures concerning the location of negative frequencies in the output of an FFT**.