Part II Computational Physics Exercises

David Buscher <db106@cam.ac.uk>

v2.3 - 2018/01/19

Contents

1 Introduction

These exercises are designed to help you learn some C++ and/or 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 in the MCS 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 browse the code examples available on the MCS computers (see §4.4), and, importantly **consult the relevant documentation for the libraries** — finding and learning from online documentation and code examples is an important skill. For the main library, GSL, FFTW and for gnuplot, Python, numpy, scipy and pyplot, refer to the online documentation via your browser — links from the course web-page or just use a search engine.

2 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.

As in the IB class, you should upload your solution to the exercises when complete each to your pigeonhole (see below). The exercises suggest what to submit: normally this is

1. Source code file: normally this will be just one or two `.cc` or `.py` files which compile and link (I will assume that `.cc` files require the `GSL` and `cavlib` and `FFTW` libraries to link while the `.py` should run using the MCS installation of `python3`, which includes the `numpy` and `scipy` modules).
2. a `readme.txt` text file which describes very briefly what you did, any major problems, and mentions any numerical answers required and describes any accompanying plots. An alternative is to submit a PDF file, called `readme.pdf` which contains the text mentioned above, but can also include the plots mentioned below. Do *not* submit Microsoft Word files or similar — only plain text or PDF files will be considered.
3. Plots from `gnuplot`/`pyplot` illustrating the problem. Plots must be in PDF format. See §4.3 on how to plot in PDF format. Your plots can be included in the `readme.pdf` file if desired, in which case additional plot files are not required.
   Remember to annotate your graph axes appropriately.

Please do not submit large files, e.g. data files or executables, to the pigeonholes. **Any files (with the exception of PDF files) larger than 1 MB will be deleted without looking at them.**

2.1 Submission of Work

You are required to submit your work electronically to what I have called your *pigeonhole*. You should upload your answer to each exercise to its own directory. These directories should have the names

`/ux/PHYSICS/PART_2/pigeonhole/CRSID/exercise1`

`/ux/PHYSICS/PART_2/pigeonhole/CRSID/exercise2`

and either `/ux/PHYSICS/PART_2/pigeonhole/CRSID/exercise3A`

or `/ux/PHYSICS/PART_2/pigeonhole/CRSID/exercise3B`

respectively, where `CRSID` should be substituted with your own crsid.

Let me know if you have problems using your pigeonhole or if it is missing. If you find you can't delete files once uploaded, and need to add a new version, just upload a new version with a number at the end, e.g. `readme2.txt`. Or you can create a whole new directory in the pigeonhole and put a whole new solution there; perhaps call it `bestSolution` or something like that.

2.2 Deadline

This 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.

3 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 identifying what each function does);
- Sensible use of whitespace to indicate code structure (somewhat obligatory in Python, less so in C++).

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.

4 Getting started

4.1 Setting up your workspace

I strongly recommend using a **new directory** for each project you do — this will make it much easier to look after your files. To do this, use the UNIX `mkdir` command. So for example from your home directory:

```
mkdir exercise1
cd exercise1
```

Then start work in this directory; next week, make `exercise2` etc.

4.2 Running your program

Python If you are running `Python` code then, once you have written a program and saved it in, say, `myprog.py` you can run it by typing

```
python3 myprog.py
```

and looking at the results. Note that if instead you were to type `python myprog.py`, this would invoke the `Python` 2 interpreter. In this course we are emphasising the use of `Python` 3, which is very similar to, but not the same as, `Python` 2.

C++ The easiest way to compile and link `C++` programs is using `Makefile` and the `make` command; you may be familiar with it from last year. There is an example `Makefile` which you can copy to your current directory by typing

```
cp /ux/PHYSICS/PART_2/examples/Makefile.simple Makefile
```

You can then say

```
make myprog.exe
```

and `make` will attempt to compile and link the code in `myprog.cc` to make an executable called `myprog.exe`.

You should see something like the following on your terminal when you do this:

```
g++ -g -pedantic -Wall -I/ux/PHYSICS/PART_2 myprog.cc /ux
   /PHYSICS/PART_2/cavlib/cavlib.a -lgsl -lgslcblas -
   lfftw3 -o myprog.exe
```

This will produce an *executable* binary file called `myprog.exe` which you can run by typing

```
./myprog.exe
```

Note that the extension `.exe` is just a convention used by the `Makefile` provided.

What is happening here? The `-I` flag tells the `g++` compiler where to find the include (or header) files which declare the functions available in `cavlib`. On the MCS system, the include files for GSL and FFTW are stored in the standard location `/usr/include` which is the default place the compiler looks for include files.

These include files are needed at *compile time*, when the compiler reads your code and tries to turn it into machine code (also called object files). After the compiling the compiler calls the *linker*, which the links together your own object files with that from the libraries, for example `cavlib` and the GSL library. Finally you run your program as indicated above, and this is of course called *run time*.

You can get errors at any stage of course — at compile, link or run time — but they will be of different kinds. Compile-time errors may be due to syntax errors in your code. Link-time errors may be due to missing library functions. Run-time errors are often caused by coding logic errors.

4.3 Plotting your results

It's often convenient to the normal `std::cout` stream or `print()` to write output from a program. You can catch this in a file for plotting with `gnuplot` or `pyplot` using UNIX re-direction: for example

```
./program.exe > output.dat
```

or equivalently

```
python3 myprog.py > output.dat
```

followed by, for example, `plot 'output.dat'u 1:2` in `gnuplot`. Be careful not to overwrite files you still need!

Please hand in plots in PDF format: in `gnuplot`, you just say

```
> set term pdf
> set output myplot.pdf
> plot sin(x)
```

Be sure to add captions for the axes and a title:

```
> set xlabel "Distance along axis (m)"
> set ylabel "Distance perpendicular to axis (m)"
> set title "Magnetic field strength"
```

In `pyplot` the equivalent would be:

```
import matplotlib.pyplot as plt
import numpy as np
plot(np.sin(np.linspace(-10,10,100)))
plt.xlabel("Distance along axis (m)")
plt.ylabel("Distance perpendicular to axis (m)")
plt.title("Magnetic field strength")
plt.savefig("myplot.pdf")
```

4.4 Example code

There are example programs which might be useful in the directory:
`/ux/PHYSICS/PART_2/examples`
Remember the `UNIX` command `grep` which searches through files for text — you could use this to search files in this directory for example.

5 Exercise 1: Integration and Random Numbers

To learn how to evaluate integrals numerically, using (a) a self-written Monte-Carlo method and (b) a general purpose integrator from the `GSL` or `scipy`.

5.1 Background Theory

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

$$\int f\,dV \approx V\,\langle f\rangle \pm \sigma \tag{1}$$

where

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

and

$$\sigma \approx V\left(\frac{\langle f^2\rangle - \langle f\rangle^2}{N}\right)^{1/2} \tag{3}$$

with

$$\langle f^2 \rangle \equiv \frac{1}{N} \sum_{i=0}^{N-1} f^2(\mathbf{r}_i) \tag{4}$$

Note that the error estimate in equation (3) 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.

5.2 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 with `gnuplot`/`pyplot`.

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

> Upload to the course pigeonhole: Source code (`.cc` or `.py`), relevant graphical plots, and a `readme.txt` 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 a reasonable amount of time.

Note: if you are writing in `Python`, then writing a *vectorised* program will allow you to reach much higher values of N than if you code all the loops explicitly. Try this if you have time.

Core Task 2 : Write a program to evaluate the Fresnel integrals *accurately* using a standard integration routine from the `GSL` or `scipy`, and use this to

make a plot of the Cornu spiral using `gnuplot`/`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

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

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

Upload to the course pigeonhole: source code, relevant plots.

Supplementary Task 1 Compare your error derived from the scatter of your Monte-Carlo simulations with the theoretical error estimate given by equation (3), using a suitable plot.

Upload to the course pigeonhole: source code, relevant plot.

Supplementary Task 2 : Use the previous work to calculate and plot the diffraction pattern of a slit of width $d = 10\,\mathrm{cm}$ illuminated at normal incidence by plane monochromatic waves of wavelength $\lambda = 1\,\mathrm{cm}$. The pattern is observed on a plane normal to the slit and the incoming waves. Calculate the pattern when the screen is at distances $D = 30, 50$ and $100\,\mathrm{cm}$ from the slit. Plot both the relative amplitude and the phase of the pattern using `gnuplot`/`pyplot`. (The absolute amplitude and phase are not important here).

Physics reminder: the complex amplitude in the near field on the axis of an open slit is given by

$$\Psi \propto \int_{x_0}^{x1} \left(\cos(\pi x^2/(\lambda D)) + i \sin(\pi x^2/(\lambda D))\right)\, dx$$

so you can use the Fresnel integrals directly scaling the length dimensions by $\sqrt{2/(\lambda D)}$. This result holds good for $D \gg \left(\frac{d^4}{\lambda}\right)^{1/3}$, And the Fraunhofer limit applies when $D \gg \frac{d^2}{\lambda}$.

Upload to the course pigeonhole: source code, relevant plot.

5.3 Step by step guide: 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 1. 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. If you are writing your program in `C++`, you may want to use the value of π from the `cavlib` library — it is `C::pi` in `cavlib/constants.hh`. This means you need to write `#include "cavlib/constants.hh"` at the top of your source file and you can then access it as `C::pi`. Or just type it in with plenty of precision...but this is not good practice.
   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 the `GSL` or `numpy.random` random number generators for this. Look in the example code from the lectures and in the examples directory 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.
   How many samples can you compute in a few minutes on the MCS? Does this makes sense, supposing the computer clock speed is about 3 GHz say? In `Python`, your code may run orders of magnitude faster if you think about how to *vectorise* your code, in other words use `numpy` functions such as `numpy.sum()` to act on a whole array of numbers in a single function call.
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 using `gnuplot` or `pyplot`. The simplest way is simply to print out the values you need one line at a time using `std::cout` or `print()` in your program, and then redirect the output of your program into a file using a command like `./your_prog.exe > output.txt`.
   You could perhaps use the `plot with errorbars` command in `gnuplot` or `errorbar()` in `pyplot` to illustrate error bars. Also don't forget the logarithmic plotting capabilities in `gnuplot`/`pyplot` — `set log xy` or `loglog()` respectively for example to get a log-log plot.
7. Save your plot to a `.pdf` file for submission to your pigeonhole.

5.4 Step by step guide: Numerical Integration with GSL or `scipy`

1. You will need to choose a `GSL` or `scipy` routine to do the integration. The routine `gsl_integration_qag` is a good general purpose integrator, as is `scipy.integrate.quad()`. Example code is available online: search for "GSL numerical integration examples" or "scipy numerical integration examples". There is full `C++` example on the MCS called `quad1.cc` which evaluates the pendulum period elliptical integral — this should guide you if you are stuck.
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 i.e. a function that looks like

   ```
   double func( double x, void* param )
   ```

   if you are using `GSL`. You probably won't need the param argument in this case.
3. Set up the interface to the integration routine — either look at the examples already mentioned, or if you using `C++` and are comfortable with classes, look at the example class interface in `cavlib` – the header file `gslint_integ.hh` has a simple class-based interface to the routine.
4. Evaluate the integrals for various values of s and use `gnuplot`/`pyplot` to plot the spiral.
5. Plotting the diffraction pattern should now be straightforward.

6 Exercise 2. ODEs: The Driven Pendulum

To investigate the physics of a damped, driven pendulum by accurate integration of its equation of motion.

6.1 Physics

The pendulum comprises a bob of mass m on a light cord 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

$$m\,l^2\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -m\,g\,l\,\sin(\theta) - \alpha\,l\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + G\sin(\Omega_D t) \tag{5}$$

Rearranging:

$$\frac{\mathrm{d}^2\theta}{\mathrm{d}t^2} = -\frac{g}{l}\,\sin(\theta) - q\,\frac{\mathrm{d}\theta}{\mathrm{d}t} + F\sin(\Omega_D t) \tag{6}$$

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.

6.2 Tasks

Core Task 1 First re-write this second-order differential equation Equation 6 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, for example a 4th order Runge-Kutta technique, from a given starting point $\theta = \theta_0$ and $\omega = \omega_0$ at $t = 0$. I recommend using the `GSL` library or `scipy` rather than implementing your own Runge-Kutta technique. The program should write out line by line the values of t, θ, ω and total energy at frequent enough intervals to follow the evolution of the equations. If your files get big, you should modify the program to write out the results less frequently.

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. Over-

lay 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 $0 < \theta_0 < \pi$. In your written answer file, state the period for $\theta_0 = \pi/2$.

> Upload to the course pigeonhole: Source code, `readme.txt` containing a couple of sentences summarising what you managed to achieve, and the value of the period for $\theta_0 = \pi/2$; include an appropriately-scaled plot showing how well energy is conserved in at least one case and a plot of period versus amplitude.

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* $\pm\pi$.

> Upload to the course pigeonhole: Source code, a sentence or two in the `readme.txt` 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.

> Upload to the course pigeonhole: Source code, a sentence or two in the `readme.txt` 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!

> Upload to the course pigeonhole: Source code, a sentence or two in the `readme.txt` explaining what you see in the solutions, and illustrative plots of the behaviour.

6.3 Hints

1. Hopefully rewriting the equation as 2 ODEs is straightforward: if not, ask!
2. I recommend that you use the fourth-order Runge-Kutta integrator in `GSL` or the `scipy.integrate.odeint()` function. Examples of their use can be found in the relevant manuals.
3. Both techniques are best done by first writing 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 programs in `ode_ring1.cc` or `ode_ring1_scipy.py` which solve the spinning ring problem discussed in lectures. You can perhaps adapt some of that code if you get stuck.
4. Use the ">" redirect symbol on the command line to send your output to a file. It is probably not necessary to write out every time step used by the integrator (this is not a problem with the `scipy` integrator as it only returns values at the ordinates you choose): you may want only to print out every, say, 10th or 100th value.
5. A note on plotting: you should make a text file containing columns of data — perhaps time, angle, angular speed, energy. Make sure you have spaces in between the columns. Then use `gnuplot`/`pyplot` as usual. You may need to set the ranges with `set xrange [0:100]` for example.
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? More sophisticatedly, you can 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.
8. This is a problem where it is really useful to be able to pass arguments to the program from the command line. For example you'd like to pass perhaps the values of q, F, dt etc. You might like to add this capability to your code — it could save you time (and it will be useful in future). There are examples of how to do this in the examples `ode5.cc` which is fairly simple, and a more advanced and sophisticated technique in `ode_ring2.cc` which also provides an example solution to the ring problem using a class interface. For `Python`, the best place to start is perhaps the tutorial for argparse at https://docs.python.org/3/howto/argparse.html.

7 Exercise 3A: Diffraction by the FFT

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.

7.1 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).

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

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

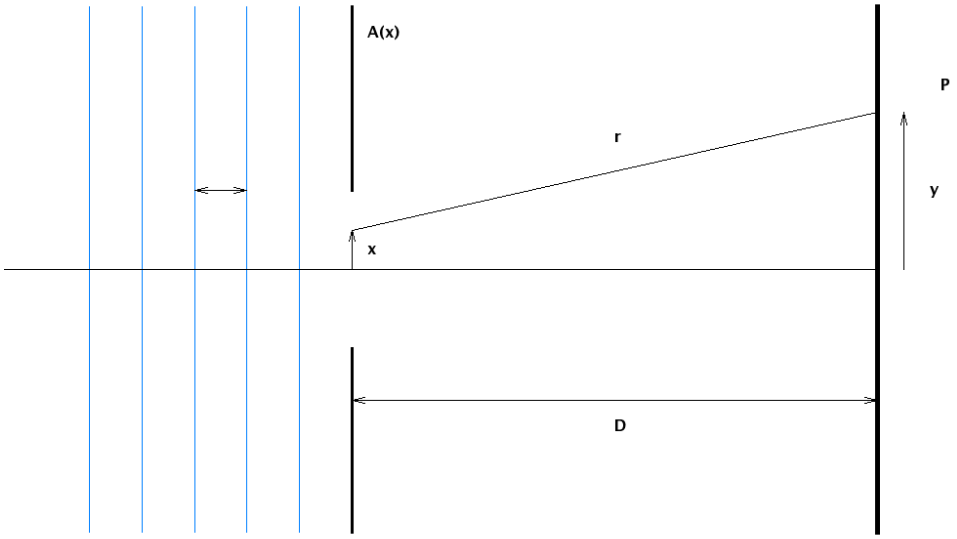


Figure 1: Geometry for diffraction calculation

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

$$\psi(y) \propto \frac{\exp(ikD)}{D} \exp\left(\frac{iky^2}{2D}\right) \int A(x) \exp\left(\frac{ikx^2}{2D}\right) \exp\left(\frac{-ikxy}{D}\right) \mathrm{d}x \quad (7)$$

The diffraction pattern is thus the Fourier-transform of the modified aperture

function A':

$$\psi(y) \propto \exp\left(\frac{iky^2}{2D}\right) \int A'(x)\ \exp\left(\frac{-ikxy}{D}\right) \mathrm{d}x \tag{8}$$

with

$$A'(x) = \exp\left(\frac{ikx^2}{2D}\right) A(x) \tag{9}$$

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

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

The distance $x_{\max}^2/\lambda$ 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 9 *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 8.

Finally, we can discretize Equation 8, by sampling the aperture evenly at positions x_j

$$\psi(y) \propto \Delta \sum_{j=0}^{N-1} A'(x_j)\ \exp\left(\frac{-ikx_jy}{D}\right) \tag{10}$$

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

One convenient definition of the sample points in x is

$$x_j = (j - (N/2))\Delta, \tag{11}$$

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.

7.2 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 pat-

tern across the screen, which you should plot using `gnuplot`/`pyplot`. The program should find and you should plot the pattern 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 microns, $D = 1.0$m and L =5mm. Overlay on your plot the theoretical value of the intensity pattern expected, using `gnuplot`/`pyplot` to plot the theoretical function.

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?

Supplementary Task : 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 9. 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?

7.3 Developing the Program

Recall the FFT definition:

$$H_j = \sum_{m=0}^{N-1} h_m \, e^{2\pi i m j/N} \tag{12}$$

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

$$f_j = \frac{j}{N\Delta} \tag{13}$$

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 must be represented by a set 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.)

In `C++` you will need $2N$ double precision values to store your discrete model of A. Use the conventional complex representation for where the real and imaginary parts are stored in consecutive storage cells of the array. The aperture can be set up as `vector<double> A(2*N);` in which `A[0]` contains the real part and `A[1]` the imaginary part of the first sample of the aperture function. You can also use normal C-style arrays if you prefer e.g. `double A[2*N];`. (You could also a class: if interested in this more advanced approach, look in `cavlib` at the classes in `complex_array1.hh`).

Using this storage, you can then set up an aperture representing the slit defined in the first task. The diffraction pattern will need a storage area of the same size. You might also want to declare arrays of length N to store the x and y coordinates of the aperture and diffraction pattern respectively.

In `Python`, explicit allocation for storage of the output arrays is not necessary, as these are created by the FFT function, but the input amplitude arrays need to be constructed. 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). If you

are coding in Python, 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.

If you are coding in C++ the **FFTW** library is recommended. (GSL only provides 1-dimensional FFTs at present, which while sufficient for this task, are not sufficient for one of the projects). There are clear examples to copy in the examples `fft1.cc` and `fft2.cc`. The latter does the FFT in a small function, which is neat and easy to use. You might want to use something like that.

If you are coding in C++, then the best thing to do is to call the FFT routine, and write out the result to a text file as a set of columns and then read it into `gnuplot`/`pyplot` for plotting. If you are coding in Python, then you can do the FFT and plotting in the same program if you prefer. Computing and plotting in the same program is not recommended if the computation time is long (more than a few seconds), but this should not be the case here.

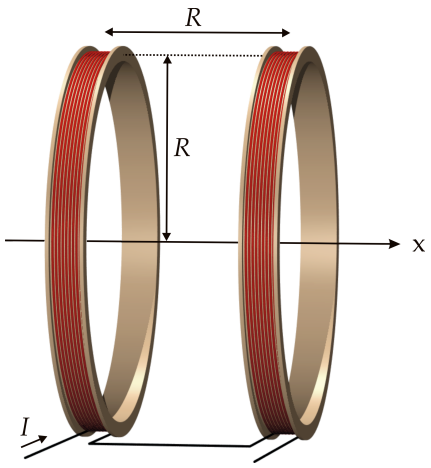
You will need to compute the intensity of the pattern (or you can do this in `gnuplot`/`pyplot`). For each $\psi(y)$ value, write out the y value, the real and imaginary parts of ψ, and the amplitude and phase (alternatively, you could compute the amplitude and phase from the real and imaginary parts when making your plots). You might want to also write out and 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 10 and Equation 12 (also referring to Equation 11) 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.**

8 Exercise 3B: Helmholtz Coils and the Biot-Savart law

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.



8.1 Physics

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

$$\mathbf{dB} = \frac{\mu_0}{4\pi} \frac{I\,\mathbf{dl} \wedge \mathbf{r}}{r^3} \tag{14}$$

at a location $\mathbf{r}$ 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 problems 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

$$B = \frac{\mu_0\, I\, R^2}{2(R^2 + x^2)^{3/2}} \tag{15}$$

Helmholtz coils produce a fairly uniform field close to the centre of the system. They consist of two co-axis 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

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

8.2 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, write the results to a column file and plot with `gnuplot`/`pyplot`. 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 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 $\mathbf{B}(x, y, 0)$, 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 $D = 10R$.

8.3 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 3-dimensional vector fields, both for the position and magnetic field vectors. Most simply, you can have three arrays representing the Cartesian x, y and z components of the vectors. Or you could use a class or structure that represents a point in 3-space, and use a vector of such points. In `Python`, a 4-dimensional array (or higher-dimensional as required) where three of the dimensions correspond to x, y and z respectively may be the way forward. The choice is yours.

(There is a suitable three-D vector class in `cavlib`, called `Vec3`, defined in `vec3.hh`: it stores a 3-vector, and provides for example the cross-product operator for `Vec3` objects. Look in the example `vector2.cc`.)

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 $\mathbf{r}_i$ to $\mathbf{r}_{i+1}$ using vectors.

You can now write a function that calculates the contribution $d\mathbf{B}$ from each section using equation Equation 14, 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 point in the xy plane.

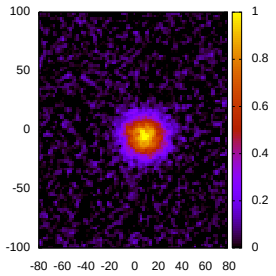
To plot your 2-d field with `gnuplot` you will need to write a column file with columns that represent the x and y values of the points where you have found $\mathbf{B}$, then the magnetic field. I suggest you write 4 columns for the magnetic field: B_x, B_y, B_z and the magnitude $|\mathbf{B}|$, so you now have 6 columns in all. It is important that you add one or more blank lines between each row (or column) of magnetic field values. For example, suppose you have a 2-D image of size $n \times m$ points. You need to write out n lines each containing xyz values, then

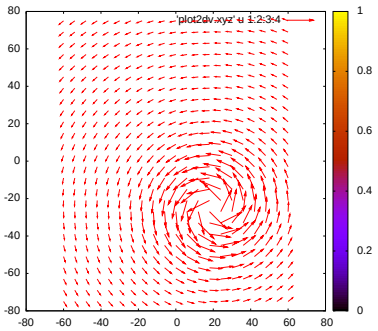
a blank line; then repeat this m times, so you end up with m blocks of data separated by blank lines. (It is also possible to interchange the order and write n blocks of m lines instead). This is to allow `gnuplot` to understand how to plot the data file.

In `gnuplot`, you now need to say something like

```
set pm3d
set view map # [this is optional]
splot 'myfile.dat' u 1:2:6 with pm3d
```

and this should produce a nice colour image of the magnitude of $\mathbf{B}$. There is an example to study called `plot2d.cc`, with a plot file called `plot2d.gp`, producing the coloured image below. By choosing other columns you can also plot each field component. In addition, using the 'with vectors' option you can plot the vector field with arrows — see the `gnuplot` documentation, and the examples `plot2dv.cc` and `plot2dv.gp` producing the plot on the right. Note these plots are not related to the problem under study in this exercise, they are just for illustration.





The `gnuplot` "xyz" file format can alternatively be read into `pyplot` using the `numpy.readtxt()` function. Note that the `numpy.readtxt()` function ignores any blank lines so it may be necessary to convert the data to a 2-d `numpy` array appropriately for plotting. In `Python` it may be simpler to compute your results and plot them in the same program, avoiding the overhead of writing out and then reading back in the data.

Now add another coil and investigate Helmholtz coils.