COMPUTATIONAL PHYSICS

EXERCISES FOR CHAPTER 7

Exercise 7.1: Fourier transforms of simple functions

Write Python programs to calculate the coefficients in the discrete Fourier transforms of the following periodic functions sampled at $N = 1000$ evenly spaced points, and make plots of their amplitudes similar to the plot shown in Fig. 7.4:

a) A single cycle of a square-wave with amplitude 1
b) The sawtooth wave $y_n = n$
c) The modulated sine wave $y_n = \sin(\pi n/N)\sin(20\pi n/N)$

If you wish you can use the Fourier transform function from the file `dft.py` as a starting point for your program.

Exercise 7.2: Detecting periodicity

In the on-line resources there is a file called `sunspots.txt`, which contains the observed number of sunspots on the Sun for each month since January 1749. The file contains two columns of numbers, the first representing the month and the second being the sunspot number.

a) Write a program that reads the data in the file and makes a graph of sunspots as a function of time. You should see that the number of sunspots has fluctuated on a regular cycle for as long as observations have been recorded. Make an estimate of the length of the cycle in months.
b) Modify your program to calculate the Fourier transform of the sunspot data and then make a graph of the magnitude squared $|c_k|^2$ of the Fourier coefficients as a function of k—also called the *power spectrum* of the sunspot signal. You should see that there is a noticeable peak in the power spectrum at a nonzero value of k. The appearance of this peak tells us that there is one frequency in the Fourier series that has a higher amplitude than the others around it—meaning that there is a large sine-wave term with this frequency, which corresponds to the periodic wave you can see in the original data.
c) Find the approximate value of k to which the peak corresponds. What is the period of the sine wave with this value of k? You should find that the period corresponds roughly to the length of the cycle that you estimated in part (a).

This kind of Fourier analysis is a sensitive method for detecting periodicity in signals. Even in cases where it is not clear to the eye that there is a periodic component to a signal, it may still be possible to find one using a Fourier transform.

Exercise 7.3: Fourier transforms of musical instruments

In the on-line resources you will find files called `piano.txt` and `trumpet.txt`, which contain data representing the waveform of a single note, played on, respectively, a piano and a trumpet.

a) Write a program that loads a waveform from one of these files, plots it, then calculates its discrete Fourier transform and plots the magnitudes of the first 10 000 coefficients in a manner similar to Fig. 7.4. Note that you will have to use a fast Fourier transform for the calculation because there are too many samples in the files to do the transforms the slow way in any reasonable amount of time.

   Apply your program to the piano and trumpet waveforms and discuss briefly what one can conclude about the sound of the piano and trumpet from the plots of Fourier coefficients.

b) Both waveforms were recorded at the industry-standard rate of 44 100 samples per second and both instruments were playing the same musical note when the recordings were made. From your Fourier transform results calculate what note they were playing. (Hint: The musical note middle C has a frequency of 261 Hz.)

Exercise 7.4: Fourier filtering and smoothing

In the on-line resources you'll find a file called `dow.txt`. It contains the daily closing value for each business day from late 2006 until the end of 2010 of the Dow Jones Industrial Average, which is a measure of average prices on the US stock market.

Write a program to do the following:

a) Read in the data from `dow.txt` and plot them on a graph.

b) Calculate the coefficients of the discrete Fourier transform of the data using the function `rfft` from `numpy.fft`, which produces an array of $\frac{1}{2}N + 1$ complex numbers.

c) Now set all but the first 10% of the elements of this array to zero (i.e., set the last 90% to zero but keep the values of the first 10%).

d) Calculate the inverse Fourier transform of the resulting array, zeros and all, using the function `irfft`, and plot it on the same graph as the original data. You may need to vary the colors of the two curves to make sure they both show up on the graph. Comment on what you see. What is happening when you set the Fourier coefficients to zero?

e) Modify your program so that it sets all but the first 2% of the coefficients to zero and run it again.

Exercise 7.5: If you have not done Exercise 7.4 you should do it before this one.

The function $f(t)$ represents a square-wave with amplitude 1 and frequency 1 Hz:

$$f(t) = \begin{cases} 1 & \text{if } \lfloor 2t \rfloor \text{ is even,} \\ -1 & \text{if } \lfloor 2t \rfloor \text{ is odd,} \end{cases} \tag{1}$$

where $\lfloor x \rfloor$ means x rounded down to the next lowest integer. Let us attempt to smooth this function using a Fourier transform, as we did in the previous exercise. Write a program that creates an array of $N = 1000$ elements containing a thousand equally spaced samples from a single cycle of this square-wave. Calculate the discrete Fourier transform of the array. Now

set all but the first ten Fourier coefficients to zero, then invert the Fourier transform again to recover the smoothed signal. Make a plot of the result and on the same axes show the original square-wave as well. You should find that the signal is not simply smoothed—there are artifacts, wiggles, in the results. Explain briefly where these come from.

Artifacts similar to these arise when Fourier coefficients are discarded in audio and visual compression schemes like those described in Section 7.3.1 and are the primary source of imperfections in digitally compressed sound and images.

Exercise 7.6: Comparison of the DFT and DCT

This exercise will be easier if you have already done Exercise 7.4.

Exercise 7.4 looked at data representing the variation of the Dow Jones Industrial Average, colloquially called "the Dow," over time. The particular time period studied in that exercise was special in one sense: the value of the Dow at the end of the period was almost the same as at the start, so the function was, roughly speaking, periodic. In the on-line resources there is another file called `dow2.txt`, which also contains data on the Dow but for a different time period, from 2004 until 2008. Over this period the value changed considerably from a starting level around 9000 to a final level around 14000.

a) Write a program similar to the one for Exercise 7.4, part (e), in which you read the data in the file `dow2.txt` and plot it on a graph. Then smooth the data by calculating its Fourier transform, setting all but the first 2% of the coefficients to zero, and inverting the transform again, plotting the result on the same graph as the original data. As in Exercise 7.4 you should see that the data are smoothed, but now there will be an additional artifact. At the beginning and end of the plot you should see large deviations away from the true smoothed function. These occur because the function is required to be periodic—its last value must be the same as its first—so it needs to deviate substantially from the correct value to make the two ends of the function meet. In some situations (including this one) this behavior is unsatisfactory. If we want to use the Fourier transform for smoothing, we would certainly prefer that it not introduce artifacts of this kind.

b) Modify your program to repeat the same analysis using discrete cosine transforms. You can use the functions from `dcst.py` to perform the transforms if you wish. Again discard all but the first 2% of the coefficients, invert the transform, and plot the result. You should see a significant improvement, with less distortion of the function at the ends of the interval. This occurs because, as discussed at the end of Section 7.3, the cosine transform does not force the value of the function to be the same at both ends.

It is because of the artifacts introduced by the strict periodicity of the DFT that the cosine transform is favored for many technological applications, such as audio compression. The artifacts can degrade the sound quality of compressed audio and the cosine transform generally gives better results.

The cosine transform is not wholly free of artifacts itself however. It's true it does not force the function to be periodic, but it does force the gradient to be zero at the ends of the interval (which the ordinary Fourier transform does not). You may be able to see this in your calculations for part (b) above. Look closely at the smoothed function and you should see

that its slope is flat at the beginning and end of the interval. The distortion of the function introduced is less than the distortion in part (a), but it's there all the same. To reduce this effect, audio compression schemes often use overlapped cosine transforms, in which transforms are performed on overlapping blocks of samples, so that the portions at the ends of blocks, where the worst artifacts lie, need not be used.

Exercise 7.7: Fast Fourier transform

Write your own program to compute the fast Fourier transform for the case where N is a power of two, based on the formulas given in Section 7.4.1. As a test of your program, use it to calculate the Fourier transform of the data in the file `pitch.txt`, which can be found in the on-line resources. A plot of the data is shown in Fig. 7.3. You should be able to duplicate the results for the Fourier transform shown in Fig. 7.4.

This exercise is quite tricky. You have to calculate the coefficients $E_k^{(m,j)}$ from Eq. (7.43) for all levels m, which means that first you will have to plan how the coefficients will be stored. Since, as we have seen, there are exactly N of them at every level, one way to do it would be to create a two-dimensional complex array of size $N \times (1 + \log_2 N)$, so that it has N complex numbers for each level from zero to $\log_2 N$. Then within level m you have 2^m individual transforms denoted by $j = 0 \ldots 2^m - 1$, each with $N/2^m$ coefficients indexed by k. A simple way to arrange the coefficients would be to put all the $k = 0$ coefficients in a block one after another, then all the $k = 1$ coefficients, and so forth. Then $E_k^{(m,j)}$ would be stored in the $j + 2^m k$ element of the array.

This method has the advantage of being quite simple to program, but the disadvantage of using up a lot of memory space. The array contains $N \log_2 N$ complex numbers, and a complex number typically takes sixteen bytes of memory to store. So if you had to do a large Fourier transform of, say, $N = 10^8$ numbers, it would take $16N \log_2 N \simeq 42$ gigabytes of memory, which is much more than most computers have.

An alternative approach is to notice that we do not really need to store all of the coefficients. At any one point in the calculation we only need the coefficients at the current level and the previous level (from which the current level is calculated). If one is clever one can write a program that uses only two arrays, one for the current level and one for the previous level, each consisting of N complex numbers. Then our transform of 10^8 numbers would require less than four gigabytes, which is fine on most computers.

(There is a third way of storing the coefficients that is even more efficient. If you store the coefficients in the correct order, then you can arrange things so that every time you compute a coefficient for the next level, it gets stored in the same place as the old coefficient from the previous level from which it was calculated, and which you no longer need. With this way of doing things you only need one array of N complex numbers—we say the transform is done "in place." Unfortunately, this in-place Fourier transform is much harder to work out and harder to program. If you are feeling particularly ambitious you might want to give it a try, but it's not for the faint-hearted.)

Exercise 7.8: Diffraction gratings

Exercise 5.19 (page 206) looked at the physics of diffraction gratings, calculating the intensity

of the diffraction patterns they produce from the equation

$$I(x) = \left| \int_{-w/2}^{w/2} \sqrt{q(u)}\, \mathrm{e}^{\mathrm{i}2\pi xu/\lambda f}\, \mathrm{d}u \right|^2,$$

where w is the width of the grating, λ is the wavelength of the light, f is the focal length of the lens used to focus the image, and $q(u)$ is the intensity transmission function of the diffraction grating at a distance u from the central axis, i.e., the fraction of the incident light that the grating lets through. In Exercise 5.19 we evaluated this expression directly using standard methods for performing integrals, but a more efficient way to do the calculation is to note that the integral is in fact just a Fourier transform. Approximating the integral, as we did in Eq. (7.13), using the trapezoidal rule, with N points $u_n = nw/N - w/2$, we get

$$\begin{aligned} \int_{-w/2}^{w/2} \sqrt{q(u)}\, \mathrm{e}^{\mathrm{i}2\pi xu/\lambda f}\, \mathrm{d}u &\simeq \frac{w}{N} \mathrm{e}^{\mathrm{i}\pi wx/\lambda f} \sum_{n=0}^{N-1} \sqrt{q(u_n)}\, \mathrm{e}^{\mathrm{i}2\pi wxn/\lambda fN} \\ &= \frac{w}{N} \mathrm{e}^{\mathrm{i}\pi k} \sum_{n=0}^{N-1} y_n\, \mathrm{e}^{\mathrm{i}2\pi kn/N}, \end{aligned}$$

where $k = wx/\lambda f$ and $y_n = \sqrt{q(u_n)}$. Comparing with Eq. (7.15), we see that the sum in this expression is equal to the complex conjugate c_k^* of the kth coefficient of the DFT of y_n. Substituting into the expression for the intensity $I(x)$, we then have

$$I(x_k) = \frac{w^2}{N^2} |c_k|^2,$$

where

$$x_k = \frac{\lambda f}{w}\, k.$$

Thus we can calculate the intensity of the diffraction pattern at the points x_k by performing a Fourier transform.

There is a catch, however. Given that k is an integer, $k = 0 \ldots N-1$, the points x_k at which the intensity is evaluated have spacing $\lambda f/w$ on the screen. This spacing can be large in some cases, giving us only a rather coarse picture of the diffraction pattern. For instance, in Exercise 5.19 we had $\lambda = 500\,\mathrm{nm}$, $f = 1\,\mathrm{m}$, and $w = 200\,\mu\mathrm{m}$, and the screen was 10 cm wide, which means that $\lambda f/w = 2.5\,\mathrm{mm}$ and we have only forty points on the screen. This is not enough to make a usable plot of the diffraction pattern.

One way to fix this problem is to increase the width of the grating from the given value w to a larger value $W > w$, which makes the spacing $\lambda f/W$ of the points on the screen closer. We can add the extra width on one or the other side of the grating, or both, as we prefer, but—and this is crucial—the extra portion added must be opaque, it must not transmit light, so that the physics of the system does not change. In other words, we need to "pad out" the data points y_n that measure the transmission profile of the grating with additional zeros so as to make the grating wider while keeping its transmission properties the same. For example, to increase the width to $W = 10w$, we would increase the number N of points y_n by a factor of ten, with the extra points set to zero. The extra points can be at the beginning, at the end, or

split between the two—it will make no difference to the answer. Then the intensity is given by

$$I(x_k) = \frac{W^2}{N^2}|c_k|^2,$$

where

$$x_k = \frac{\lambda f}{W}\, k.$$

Write a Python program that uses a fast Fourier transform to calculate the diffraction pattern for a grating with transmission function $q(u) = \sin^2 \alpha u$ (the same as in Exercise 5.19), with slits of width $20\,\mu$m [meaning that $\alpha = \pi/(20\,\mu\text{m})$] and parameters as above: $w = 200\,\mu$m, $W = 10w = 2\,$mm, incident light of wavelength $\lambda = 500\,$nm, a lens with focal length of 1 meter, and a screen 10 cm wide. Choose a suitable number of points to give a good approximation to the grating transmission function and then make a graph of the diffraction intensity on the screen as a function of position x in the range $-5\,\text{cm} \le x \le 5\,\text{cm}$. If you previously did Exercise 5.19, check to make sure your answers to the two exercises agree.

Exercise 7.9: Image deconvolution

You've probably seen it on TV, in one of those crime drama shows. They have a blurry photo of a crime scene and they click a few buttons on the computer and magically the photo becomes sharp and clear, so you can make out someone's face, or some lettering on a sign. Surely (like almost everything else on such TV shows) this is just science fiction? Actually, no. It's not. It's real and in this exercise you'll write a program that does it.

When a photo is blurred each point on the photo gets smeared out according to some "smearing distribution," which is technically called a *point spread function*. We can represent this smearing mathematically as follows. For simplicity let's assume we're working with a black and white photograph, so that the picture can be represented by a single function $a(x, y)$ which tells you the brightness at each point (x, y). And let us denote the point spread function by $f(x, y)$. This means that a single bright dot at the origin ends up appearing as $f(x, y)$ instead. If $f(x, y)$ is a broad function then the picture is badly blurred. If it is a narrow peak then the picture is relatively sharp.

In general the brightness $b(x, y)$ of the blurred photo at point (x, y) is given by

$$b(x,y) = \int_0^K \int_0^L a(x',y') f(x-x', y-y')\, \mathrm{d}x'\, \mathrm{d}y',$$

where $K \times L$ is the dimension of the picture. This equation is called the *convolution* of the picture with the point spread function.

Working with two-dimensional functions can get complicated, so to get the idea of how the math works, let's switch temporarily to a one-dimensional equivalent of our problem. Once we work out the details in 1D we'll return to the 2D version. The one-dimensional version of the convolution above would be

$$b(x) = \int_0^L a(x') f(x-x')\, \mathrm{d}x'.$$

The function $b(x)$ can be represented by a Fourier series as in Eq. (7.5):

$$b(x) = \sum_{k=-\infty}^{\infty} \tilde{b}_k \exp\left(\mathrm{i}\frac{2\pi k x}{L}\right),$$

where

$$\tilde{b}_k = \frac{1}{L}\int_0^L b(x) \exp\left(-\mathrm{i}\frac{2\pi k x}{L}\right) \mathrm{d}x$$

are the Fourier coefficients. Substituting for $b(x)$ in this equation gives

$$\begin{aligned}\tilde{b}_k &= \frac{1}{L}\int_0^L \int_0^L a(x') f(x-x') \exp\left(-\mathrm{i}\frac{2\pi k x}{L}\right) \mathrm{d}x' \,\mathrm{d}x \\ &= \frac{1}{L}\int_0^L \int_0^L a(x') f(x-x') \exp\left(-\mathrm{i}\frac{2\pi k (x-x')}{L}\right) \exp\left(-\mathrm{i}\frac{2\pi k x'}{L}\right) \mathrm{d}x' \,\mathrm{d}x.\end{aligned}$$

Now let us change variables to $X = x - x'$, and we get

$$\tilde{b}_k = \frac{1}{L}\int_0^L a(x') \exp\left(-\mathrm{i}\frac{2\pi k x'}{L}\right) \int_{-x'}^{L-x'} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X \,\mathrm{d}x'.$$

If we make $f(x)$ a periodic function in the standard fashion by repeating it infinitely many times to the left and right of the interval from 0 to L, then the second integral above can be written as

$$\begin{aligned}\int_{-x'}^{L-x'} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X &= \int_{-x'}^{0} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X \\ &\qquad + \int_0^{L-x'} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X \\ &= \exp\left(\mathrm{i}\frac{2\pi k L}{L}\right) \int_{L-x'}^{L} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X + \int_0^{L-x'} f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X \\ &= \int_0^L f(X) \exp\left(-\mathrm{i}\frac{2\pi k X}{L}\right) \mathrm{d}X,\end{aligned}$$

which is simply L times the Fourier transform $\tilde{f}_k$ of $f(x)$. Substituting this result back into our equation for $\tilde{b}_k$ we then get

$$\tilde{b}_k = \int_0^L a(x') \exp\left(-\mathrm{i}\frac{2\pi k x'}{L}\right) \tilde{f}_k \,\mathrm{d}x' = L\,\tilde{a}_k \tilde{f}_k.$$

In other words, apart from the factor of L, the Fourier transform of the blurred photo is the product of the Fourier transforms of the unblurred photo and the point spread function.

Now it is clear how we deblur our picture. We take the blurred picture and Fourier transform it to get $\tilde{b}_k = L\,\tilde{a}_k \tilde{f}_k$. We also take the point spread function and Fourier transform it to get $\tilde{f}_k$. Then we divide one by the other:

$$\frac{\tilde{b}_k}{L\tilde{f}_k} = \tilde{a}_k$$

which gives us the Fourier transform of the *unblurred* picture. Then, finally, we do an inverse Fourier transform on $\tilde{a}_k$ to get back the unblurred picture. This process of recovering the unblurred picture from the blurred one, of reversing the convolution process, is called *deconvolution*.

Real pictures are two-dimensional, but the mathematics follows through exactly the same. For a picture of dimensions $K \times L$ we find that the two-dimensional Fourier transforms are related by

$$\tilde{b}_{kl} = KL\tilde{a}_{kl}\tilde{f}_{kl}\,,$$

and again we just divide the blurred Fourier transform by the Fourier transform of the point spread function to get the Fourier transform of the unblurred picture.

In the digital realm of computers, pictures are not pure functions $f(x,y)$ but rather grids of samples, and our Fourier transforms are discrete transforms not continuous ones. But the math works out the same again.

The main complication with deblurring in practice is that we don't usually know the point spread function. Typically we have to experiment with different ones until we find something that works. For many cameras it's a reasonable approximation to assume the point spread function is Gaussian:

$$f(x,y) = \exp\left(-\frac{x^2+y^2}{2\sigma^2}\right),$$

where σ is the width of the Gaussian. Even with this assumption, however, we still don't know the value of σ and we may have to experiment to find a value that works well. In the following exercise, for simplicity, we'll assume we know the value of σ.

a) On the web site you will find a file called `blur.txt` that contains a grid of values representing brightness on a black-and-white photo—a badly out-of-focus one that has been deliberately blurred using a Gaussian point spread function of width $\sigma = 25$. Write a program that reads the grid of values into a two-dimensional array of real numbers and then draws the values on the screen of the computer as a density plot. You should see the photo appear. If you get something wrong it might be upside-down. Work with the details of your program until you get it appearing correctly. (Hint: The picture has the sky, which is bright, at the top and the ground, which is dark, at the bottom.)

b) Write another program that creates an array, of the same size as the photo, containing a grid of samples drawn from the Gaussian $f(x,y)$ above with $\sigma = 25$. Make a density plot of these values on the screen too, so that you get a visualization of your point spread function. Remember that the point spread function is periodic (along both axes), which means that the values for negative x and y are repeated at the end of the interval. Since the Gaussian is centered on the origin, this means there should be bright patches in each of the four corners of your picture, something like this:

c) Combine your two programs and add Fourier transforms using the functions `rfft2` and `irfft2` from `numpy.fft`, to make a program that does the following:

   i) Reads in the blurred photo
   ii) Calculates the point spread function
   iii) Fourier transforms both
   iv) Divides one by the other
   v) Performs an inverse transform to get the unblurred photo
   vi) Displays the unblurred photo on the screen

   When you are done, you should be able to make out the scene in the photo, although probably it will still not be perfectly sharp.

   Hint: One thing you'll need to deal with is what happens when the Fourier transform of the point spread function is zero, or close to zero. In that case if you divide by it you'll get an error (because you can't divide by zero) or just a very large number (because you're dividing by something small). A workable compromise is that if a value in the Fourier transform of the point spread function is smaller than a certain amount ϵ you don't divide by it—just leave that coefficient alone. The value of ϵ is not very critical but a reasonable value seems to be 10^{-3}.

d) Bearing in mind this last point about zeros in the Fourier transform, what is it that limits our ability to deblur a photo? Why can we not perfectly unblur any photo and make it completely sharp?

We have seen this process in action here for a normal snapshot, but it is also used in many physics applications where one takes photos. For instance, it is used in astronomy to enhance photos taken by telescopes. It was famously used with images from the Hubble Space Telescope after it was realized that the telescope's main mirror had a serious manufacturing flaw and was returning blurry photos—scientists managed to partially correct the blurring using Fourier transform techniques.