Computational Physics, 2nd edition

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. 7.3:

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 online 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 a rough estimate by eye 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 with a higher amplitude than the others around it, meaning that there is a large sine-wave term with this frequency. This term 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.

d) In the online resources you will find another file called `signal.txt`, which contains a set of 1024 measurements of a certain signal. It is not immediately apparent, but there is, in fact, a clear, periodic signal buried within these data. Use a Fourier transform to find this signal. If the data points in the file correspond to measurements made at a rate of 10 kHz, what is the frequency of the buried signal in hertz?

Exercise 7.3: Fourier transforms of the sound of musical instruments

In the online 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. 7.3. Note that you will have to use a fast Fourier transform for the calculation because there are too many samples 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 you 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 44100 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 1: The musical note corresponds to the first peak in the spectrum, which is not necessarily the highest peak. Hint 2: The note middle C has a frequency of 261.6 Hz.)

Exercise 7.4: Fourier filtering and smoothing

In the online resources you will 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 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 the variation over time of the Dow Jones Industrial Average stock market index (colloquially called "the Dow"). 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 online resources there is another file called `dow2.txt`, which also contains data on the Dow but for a different time period, from 2013 until 2024. Over this period the value changed considerably from a starting level around 23 000 to a final level around 38 000.

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 function. These occur because the smoothed version of 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 the discrete cosine transform. You can use the functions in the file `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. These 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 is true that 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 is there all the same. To reduce this effect, audio compression schemes 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. This approach is known technically as a *modified discrete cosine transform* or MDCT.

Exercise 7.7: Diffraction gratings:

Exercise 5.22 (page 208) looked at the physics of diffraction gratings, calculating the intensity of the diffraction patterns they produce from the equation

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

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 at that point. In Exercise 5.22 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 basically 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

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

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 5.22 we had $\lambda = 500\,\text{nm}$, $f = 1\,\text{m}$, and $w = 200\,\mu\text{m}$, and the screen was 10 cm wide, which means that $\lambda f/w = 2.5\,\text{mm}$ 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 is not changed. 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 5.22), with slits of width $20\,\mu\text{m}$ [meaning that $\alpha = \pi/(20\,\mu\text{m})$] and parameters as above: $w = 200\,\mu\text{m}$, $W = 10w = 2\,\text{mm}$, incident light of wavelength $\lambda = 500\,\text{nm}$, a lens with focal length 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 5.22, check to make sure your answers to the two exercises agree.

Exercise 7.8: Image compression:

In this problem you will write your own program to do JPEG-style compression of a photographic image.

In the online resources you will find a file called `house.txt`, which contains data for a picture of a house in simple grid form—a two-dimensional array of numbers representing the intensity of pixels in the image.

a) Write a Python program that reads the data in the file into a two-dimensional array and then makes a density plot of the array, showing the picture on the screen. You should use the gray-scale color scheme for your density plot, so you get a sensible looking black-and-white photograph.

b) Now create another two-dimensional array of floats of the same size as the picture array and initially empty. Go through the picture array in 16×16 blocks and perform a 2D discrete cosine transform of the data in each block, producing a 16×16 array of (real) Fourier coefficients, and then store those coefficients in the corresponding block of the new array. You can perform the

DCTs using the function `dct2` from the file `dcst.py`. To get the 16×16 blocks you will need to do two-dimensional "slicing" on the arrays—see Section 2.4.5 on page 62 if you want a reminder of how to do this. When you are finished with all the blocks, you will have a new array of the same size as the old one, filled with Fourier coefficients.

c) Now go through the Fourier coefficients one by one and set to zero every coefficient whose absolute value is less than 0.1. In other words, every coefficient whose value lies between -0.1 and $+0.1$ should get set to zero.

d) When we store a picture in a JPEG file we store the Fourier coefficients, not the picture itself, and we only need to store the coefficients that are nonzero. Count how many coefficients get set to zero in your calculation and use this to calculate and print a figure for how much you have compressed the image—how much smaller is the set of numbers you would have to store than the original set from the file `house.txt`? This figure is called the *compression ratio*.

e) When you want to view a picture stored in JPEG format you perform inverse Fourier transforms to recover the image. Add lines to your program to go once more through the array of Fourier coefficients in 16×16 blocks and perform an inverse 2D DCT on each one, storing the results back in the original data array again (or in a third, new array, if you prefer). This is the decompressed image. You can use the function `idct2` for the inverse DCTs.

f) Make a density plot of the decompressed image. You should find that it is virtually indistinguishable from the original picture, even though the image was compressed quite a lot—a significant number of the Fourier coefficients were discarded by setting them to zero.

g) Increase the threshold value below which the coefficients get set to zero. Instead of 0.1, try 0.2, or 0.5, or 1 or more. Have your program print out the compression ratio and display the resulting picture for each value you try. See how large a compression ratio you can achieve and still have the picture look essentially the same. Hint: You may wish to include the options "`vmin=0,vmax=1`" when you use `imshow`, to ensure that the gray-scale is the same for each image you display.

Exercise 7.9: Image deconvolution

You have 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 will have the opportunity to 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 us assume we are working with a black-and-white photograph, so that the picture can be represented by a single function $a(x, y)$ which specifies 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 us switch temporarily to a one-dimensional equivalent of our problem. Once we work out the details in 1D we will 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, Eq. (7.10). 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

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

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 usually we do not know the point spread function. Typically we have to experiment with different ones until we find something that works. For many cameras it is 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 do not 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 will 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 or sideways. 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 will 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 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.

   One thing you will 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 will get an error (because you can't divide by zero) or just a very large number (because you are dividing by something small). A workable compromise is to place a lower limit on the magnitude of the values in the Fourier transform of the point spread function, bearing in mind that those values are in general complex. That is, if a value in the Fourier transform has complex magnitude smaller than a certain amount ϵ, set the magnitude to exactly ϵ. The choice of ϵ is not very critical but a reasonable value seems to be around 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 defect and was returning blurry photos—scientists managed to partially correct the blurring using Fourier transform techniques.

Exercise 7.10: Differentiating by integrating:

In Exercise 5.25 on page 211 we saw how one can calculate multiple derivatives of a function by using the Cauchy derivative formula and approximating its contour integral using the trapezoidal rule in

the complex plane. For example, when calculating derivatives of a function $f(z)$ at the origin and integrating over N sample points this leads to

$$\left(\frac{\mathrm{d}^m f}{\mathrm{d}z^m}\right)_{z=0} \simeq \frac{m!}{N} \sum_{k=0}^{N-1} f(z_k)\, \mathrm{e}^{-\mathrm{i}2\pi km/N},$$

where $z_k = \mathrm{e}^{\mathrm{i}2\pi k/N}$. But the sum in this expression is none other than the discrete Fourier transform of $f(z)$, so we can evaluate it using the fast Fourier transform methods of this chapter. In most cases we are interested in calculating the derivatives of a function that is real on the real line, in which case it is convenient to take the complex conjugate to get

$$\left(\frac{\mathrm{d}^m f}{\mathrm{d}z^m}\right)_{z=0} \simeq \frac{m!}{N} \sum_{k=0}^{N-1} f^*(z_k)\, \mathrm{e}^{\mathrm{i}2\pi km/N},$$

which turns the calculation into an *inverse* discrete Fourier transform. This is useful because we can now use the function `irfft` from `numpy`, which takes complex inputs and returns real answers—exactly what we need in this case.

Write a program to calculate and print the first 20 derivatives of the function $f(z) = \mathrm{e}^{2z}$ (the same one as in Exercise 5.25) using this method with $N = 10\,000$. It is straightforward to show in this case that the derivatives are just the powers of two—2, 4, 8, 16, and so on. Check that your program gives the right answers. Hint: Bear in mind that the function `irfft` requires only the first half of the inputs, from $k = 0$ to $\frac{1}{2}N$, since it knows that the second half are complex conjugates of the first half.