Getting Ready to Work with IPSII (Version 0.2)

May 25, 2021

This document is intended to help you learn the background you need to do research with me in Interference Pattern Structured Illumination Imaging (IPSII). It assumes that you are able to do calculations and plots in Python (see the "Python tank challenge" if you need an introduction to Python). If you get stuck on anything or if you have any questions, be sure to let me know right away so I can help you (don't just get frustrated!).

1 Introduction to Fourier Series

Watch the video below, and then do the exercises listed below. Note that the exercises use a more common notation for Fourier series than the one used in the video, but you should be able to adapt from one to the other.

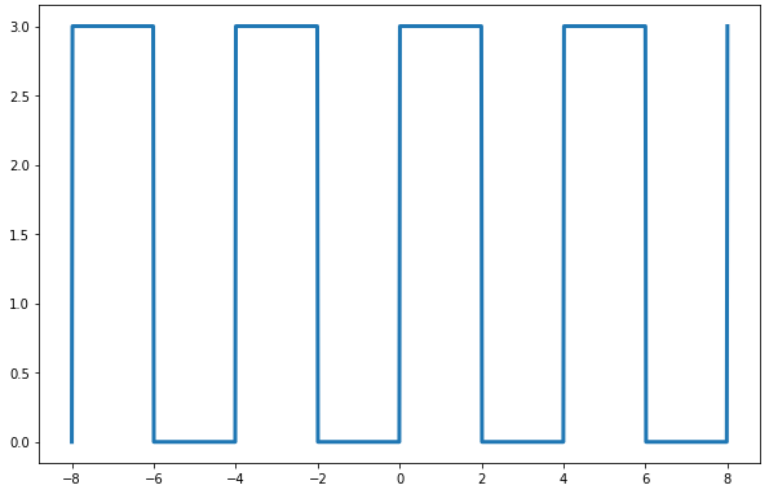
`https://youtu.be/6MpmxZ6YB7U`

More information about Fourier series can be found at `https://en.wikipedia.org/wiki/Fourier_series`

1.1 Exercises

- Figure out what k_0 should be, and then find equations for the coefficients a_n and b_n in the equation below such that when you plot the equation, you would get the plot shown below the equation. Make sure you solve the integrals and simplify your expressions for a_n and b_n.

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nk_0x) + \sum_{n=1}^{\infty} b_n \sin(nk_0x)$$



note that the equations you use to calculate the Fourier coefficients a_n and b_n are

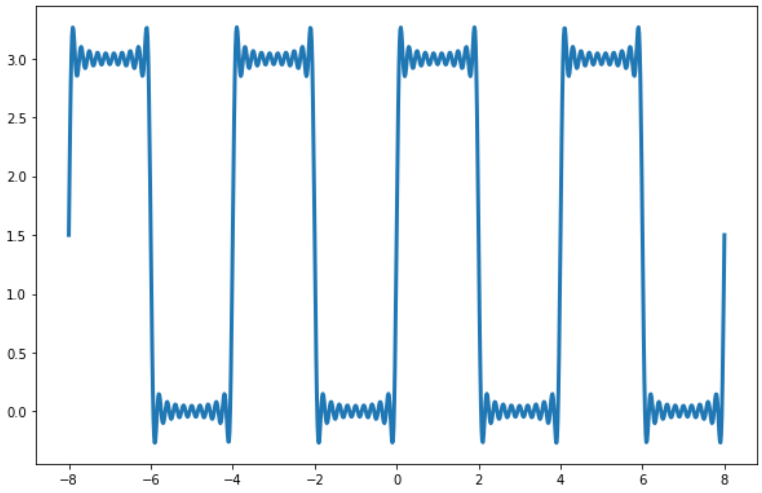
$$a_n = \frac{2}{L}\int_0^L f(x)\cos(nk_0x)dx$$

and

$$b_n = \frac{2}{L}\int_0^L f(x)\sin(nk_0x)dx$$

where $k_0 = 2\pi/L$. Use the techniques shown in the video to solve the integrals.

- Now use Python to plot the function from $x = -8$ to $x = 8$, summing up to $n = 20$ using the equations you found for a_n and b_n. If you've done this correctly, it should look like the plot below.



1.2 Calculating integrals numerically

Often, rather than solving the integrals analytically as you did on the last exercise, we simply approximate them numerically. We do that by noting that the integral

$$\int f(x)dx$$

is just a sum of tiny slices of size $f(x)dx$ over different values of x. Each of those slices has a value of zero, because dx is infinitesimally small. But there are an infinite number of them, so they can add up to something finite. An easy way to approximate an integral numerically is to just use a finite number of tiny slices. In the limit as the slices get smaller and smaller, the result will get closer to the correct answer.

For example, the integral

$$\int_0^1 x^2 dx = \frac{1}{3}x^3\Big|_{x=0}^{1} = \frac{1}{3} - 0 = \frac{1}{3} = 0.33333333333.$$

We can numerically approximate the integral by choosing some small but finite value of dx and then adding together the slices, as is done in the code below:

```
import numpy as np
dx = 0.00001
y = 0
for x in np.arange(0,1,dx):
y += (x**2)*dx
print(y)
```

For $dx = 0.1$, we get a result of 0.285. This is close-ish. By reducing dx to 0.001, we get 0.3328335, which is correct to three significant digits. If we make dx really small, say 10^{-7}, we get 0.33333328333333545 which is correct to seven significant figures. You can often get a feel for how accurate your answer is by cutting dx in half and seeing how much the result changes. If it doesn't change much, you probably have a pretty accurate result.

Note that Python is slow at doing loops. When I timed the loop in the code above (with dx set to 1e-7), it took about 8 seconds to run. We can often improve things by making code which is more "Pythonic," or, in other words, code which avoids loops. For example, in the code below, rather than going through a loop, we have numpy create an array of the different values for x. Then we have numpy calculate an array containing the integrand for each slice. Finally, we have numpy sum up all of the slices. Because numpy is doing these calculations rather than having Python do a loop, the calculation is much faster. For $dx = 10^{-7}$ it took my computer just over $1/10^{\text{th}}$ of a second to do the calculation — a speedup by a factor of 80.

```
import numpy as np
```

```
dx = 1e-7
x = np.arange(0,1,dx)
dy = (x**2)*dx
y = np.sum(dy)
print(y)
```

1.3 Exercises

- Write Python code to numerically approximate the integral

$$\int_{-2}^{2} x^2 \cos(x) dx.$$

  See how the result changes when you make dx smaller. By the way, the correct answer (to 16 sig figs) is 0.3080150149255876.

1.4 Fourier series in other variables

Before moving on, it is important to note that a Fourier series can be applied using any variable. For example, instead of summing up sines and cosines of nk_0x to make a function $f(x)$, we could also write a function of time $f(t)$ as a Fourier series simply by replacing k_0x with $\omega_0 t$ everywhere. Noting that just as $k_0 = 2\pi/L$, where L is the distance before the function $f(x)$ repeats, we can analogously see that $\omega_0 = 2\pi/T$, where T is the time we have to wait before $f(t)$ repeats.

1.5 Fourier series of non-repeating functions

In the video above, it showed how you can apply a Fourier series to a finite-length guitar string. More generally, we can describe a function $f(x)$ on the interval $-L/2 < x < L/2$ using a Fourier series:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nk_0x) + \sum_{n=1}^{\infty} b_n \sin(nk_0x)$$

where, as before, $k_0 = 2\pi/L$, and

$$a_n = \frac{2}{L} \int_{-L/2}^{L/2} f(x) \cos(nk_0x) dx$$

and

$$b_n = \frac{2}{L} \int_{-L/2}^{L/2} f(x) \sin(nk_0x) dx.$$

Outside of this interval, the Fourier series will not follow $f(x)$, but will repeat periodically.

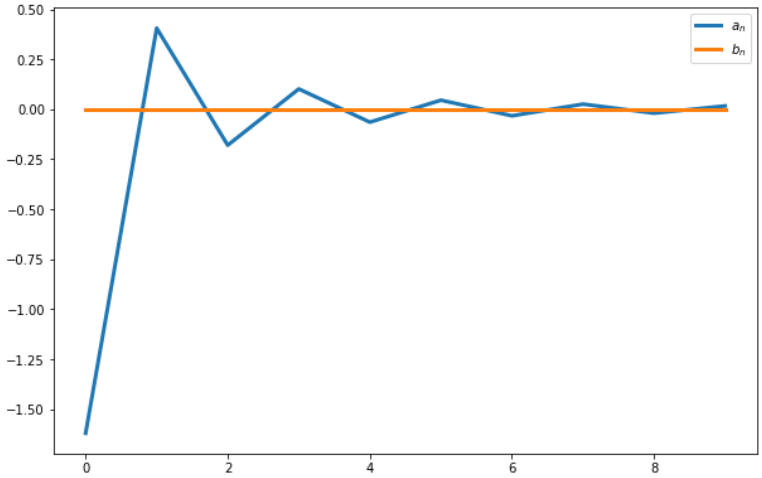
1.6 Exercises

- Write Python code that numerically calculates a_0 and a_n and b_n for $n = 1$ to 10 for the function

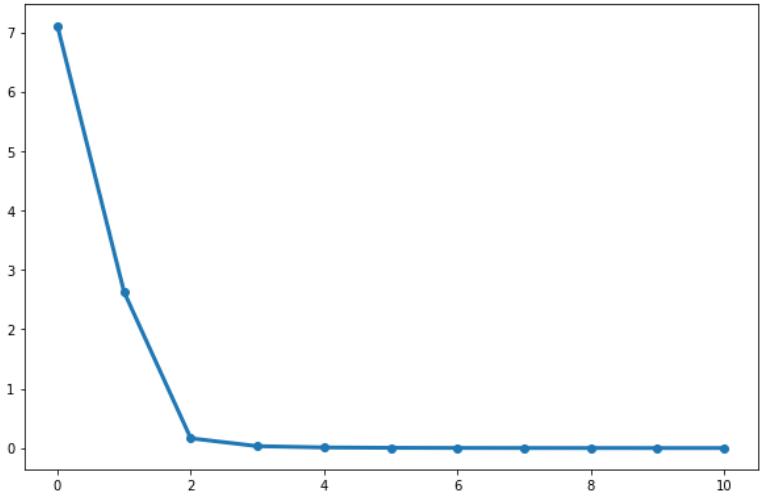
$$f(x) = x^2 dx$$

  over the interval $-2 \leq x \leq 2$. Note that the size of this interval is $L = 4$. Use an appropriately small dx to get your coefficients to 3 sig figs, but make dx large enough that your code executes in a reasonable time. To check your code, note that, to 3 sig figs, $a_0 = 2.67$, $a_1 = -1.62$, $b_1 = 0$, and $a_2 = 0.405$. Try to write the code on your own. But, if you get stuck, there is example code in appendix A.

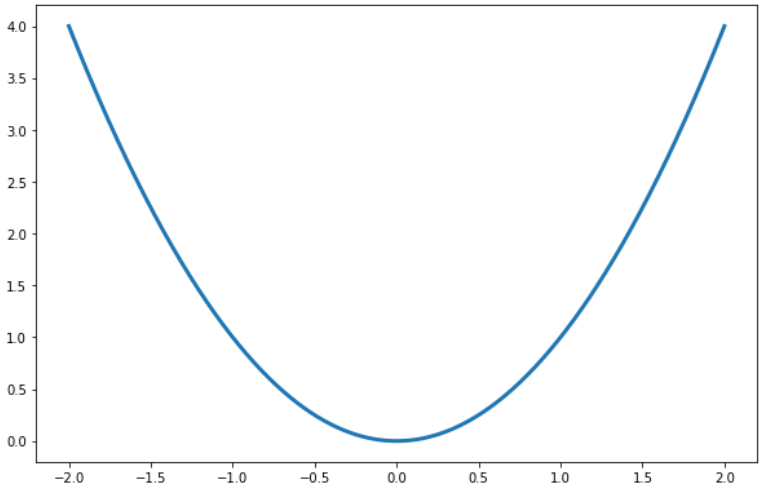
- Make a plot of a_n and b_n as a function of n. For reference, the plot I made is below.



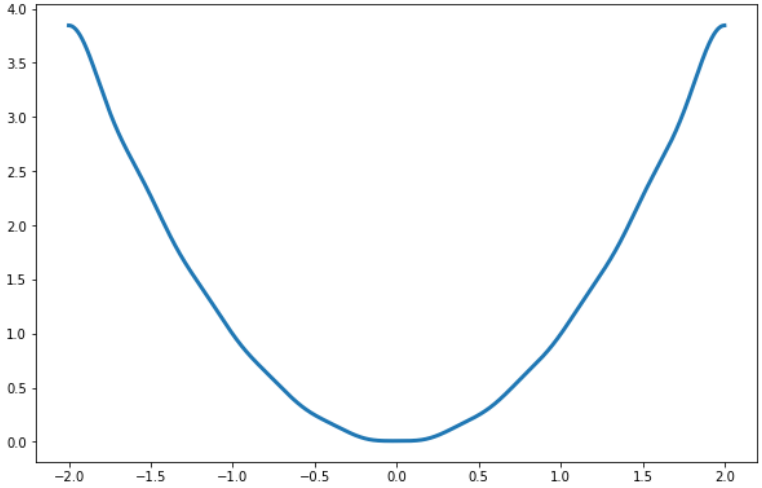
- You can also plot the spectrum, which shows the power as a function of frequency. The power is proportional to the amplitude squared. If we have both sine and cosine terms at a given frequency, we just add their powers together. In other words, plot $A_n^2 = a_n^2 + b_n^2$ as a function of n. For reference, when I did this I got the plot below.



- From the symmetry arguments given in the video, what would we expect b_n to be for this function?
- Now add to your Python code to calculate $f(x)$ from a Fourier series summing up to $n = 10$ using the coefficients you found. Then plot the function from $x = -2$ to 2. For reference, the actual function looks like

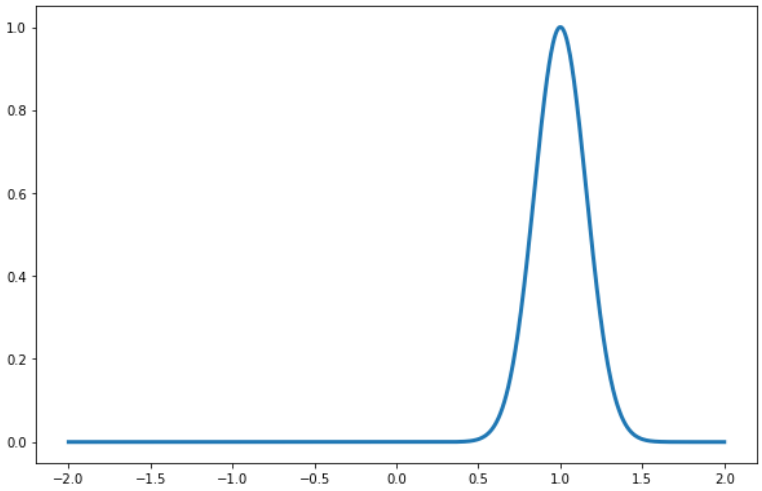


and the plot that I got from the Fourier series going up to $n = 10$ looked like

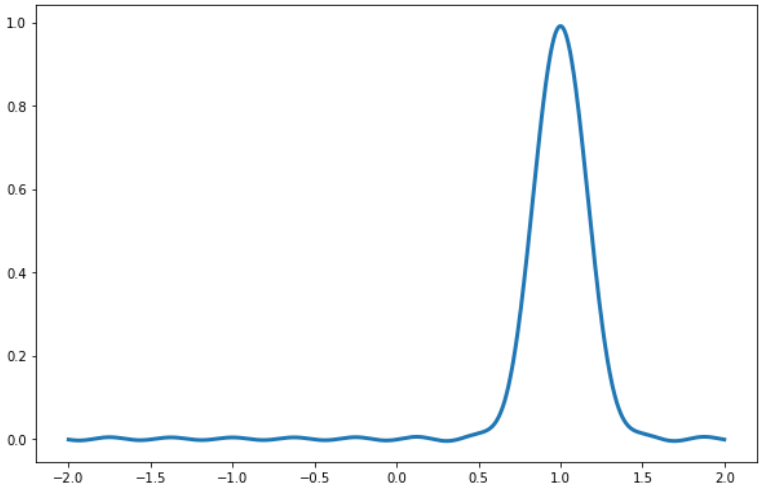


To get the exact function, we have to sum up to $n = \infty$. But, practically, that's not possible. However, as we add more terms, we tend to get a closer approximation to the function.

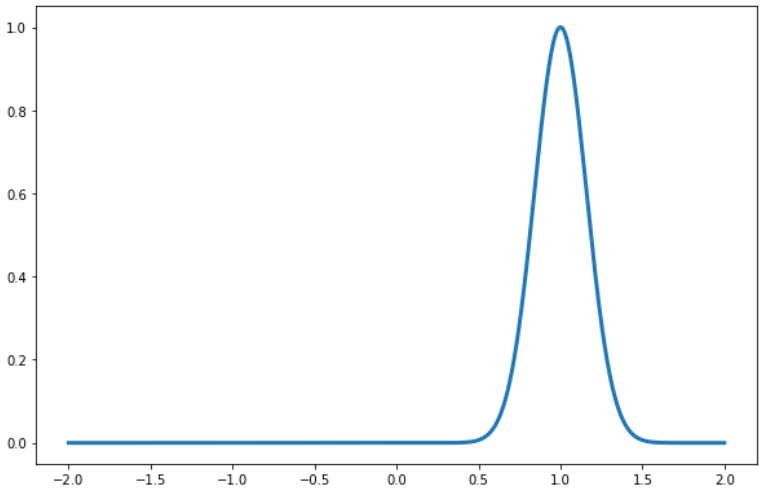
- See what happens if you only sum up to $n = 1$, 2, 5.
- See what happens if you plot your series from $x = -8$ to 8. Does it look like x^2? Does it look the way you expect?
- Now find the coefficients up to $n = 10$ for the function $f(x) = e^{-20(x-1)^2}$ and plot the Fourier series. For reference, a plot of the function looks like the image below.



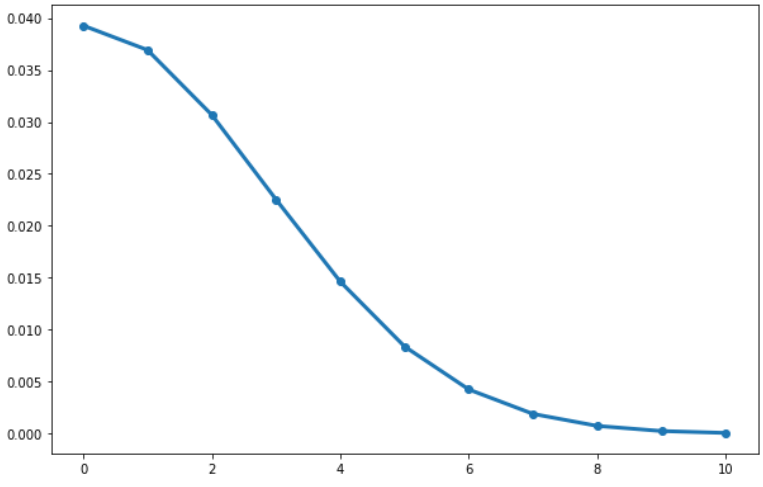
When I summed my Fourier series up to $n = 10$, I got the plot below.



When I summed up to $n = 100$, I got the plot below.



- Plot the spectrum of the function $f(x) = e^{-20(x-1)^2}$. For reference, my plot is below.



2 Review of Complex Numbers

Take a look at this video to review complex numbers: `https://youtu.be/K1WkflAH5eA`. Now review the following ideas which were not discussed in the video. Finally make sure you understand complex numbers by completing the exercises below.

Idea 1: You can add/subtract complex numbers simply by adding/subtracting the real and imaginary parts. For example,

$$(2+4i)+(3-7i)=5-3i,$$
$$(1+i)-(4+i)=-3,$$
$$(-3+3i)+(5-4i)=2-i.$$

Idea 2: You can multiply complex numbers by treating i just like you would any other number or variable. For example, remembering that $i^2=-1$,

$$(2+4i)(3-5i)=6-10i+12i-20i^2=6-10i+12i+20=26+2i,$$
$$(-3+3i)(-2-i)=6+3i-6i-3i^2=6+3i-6i+3=9-3i,$$
$$4(3+2i)=12+8i.$$

2.1 Exercises

- Use Python to plot the following numbers on an Argand diagram, assuming that $\tilde{z}=-3-4i$ and that $\tilde{q}$ is a complex number with a magnitude of 3 and a phase angle of 2.5 radians. (a) $\tilde{z}$, (b) $\tilde{q}$, (c) $\tilde{z}^*$, (d) $\tilde{q}^*$, (e) $\tilde{z}\tilde{q}$, and (f) $\tilde{z}^*\tilde{z}$. For reference, my plot of this is shown in figure 1 in appendix C.

3 Introduction to Complex Exponentials

It turns out that exponentials are closely related to sines and cosines. Euler's formula states that

$$\boxed{e^{i\theta}=\cos(\theta)+i\sin(\theta)},$$

where θ is in radians. If you are interested in how you can prove this, see appendix D.

This relationship between exponentials and trig functions can be very useful. For example, as shown in appendix E, you can use it to quickly derive various trig identities.

3.1 Complex exponentials as polar notation

As discussed previously, any complex number can be represented by its real and imaginary part *or* by its magnitude and phase angle. Using Euler's formula, we see that, if A and ϕ are both real numbers,

$$Ae^{i\phi}=A\cos(\phi)+iA\sin(\phi).$$

If we were to plot this number on the complex plane, the x or real component would be $A\cos(\phi)$, and the y or imaginary component would be $A\sin(\phi)$. That describes a point which is a distance A away from the origin at a phase angle of ϕ. As such, complex exponentials are a great way to write complex numbers in polar notation. For example, if I have a complex number with a magnitude of 5.5 and a phase angle of 1.23 radians, I can write that number as

$$\tilde{z}=5.5e^{i\,1.23}.$$

3.2 Using complex exponentials instead of sine waves

In IPSII imaging, we use illumination patterns which are sinusoidal. But exponentials are easier to work with than sines and cosines. As such, we usually represent the patterns using complex exponentials. For example, if I have a complex exponential with an argument of $\theta = \omega t + \phi$, I can separate that into the product of two exponentials, separating the part that varies in time (ωt) from the part that doesn't (ϕ):

$$e^{i(\omega t+\phi)} = e^{i\omega t}e^{i\phi}.$$

With a sine or a cosine, you can't do that:

$$\sin(\omega t + \phi) \neq \sin(\omega t)\sin(\phi).$$

One example which shows the utility of this property is adding two sound waves together. Imagine I play a sine wave through two different speakers. Because the speakers are a each a different distance from me, the sine wave from each speaker will have a different phase at my location. If I add two sine waves at the same frequency but with different phases, they can add constructively to make a large wave, or destructively to make a small one, depending on their relative phases. I can use the trig identity we used in Physics 2210 and which is derived in appendix E, to show that

$$A\sin(\omega t + \phi_1) + A\cos(\omega t + \phi_2) = 2A\cos\left(\frac{\omega t + \phi_1 - (\omega t + \phi_2)}{2}\right)\sin\left(\frac{\omega t + \phi_1 + \omega t + \phi_2}{2}\right)$$

$$= 2A\cos\left(\frac{\omega\phi_1 - \phi_2}{2}\right)\sin\left(\omega t + \frac{\phi_1\phi_2}{2}\right).$$

As such, the two waves combine to make a single sine wave at the frequency ω which has a phase equal to the average of the two phases, and which has an amplitude of

$$2A\cos\left(\frac{\phi_1 - \phi_2}{2}\right).$$

But what if the two waves had different amplitudes?

$$A_1\sin(\omega t + \phi_1) + A_2\sin(\omega t + \phi_1) = ???$$

Well, we can use complex exponentials to figure this out. First, I'm going to write functions for the sound wave from each speaker individually:

$$y_1(t) = A_1\sin(\omega t + \phi_1)$$

and

$$y_2(t) = A_2\sin(\omega t + \phi_2).$$

Now, I'm going to make up two new functions

$$\tilde{y}_1(t) = A_1 e^{i(\omega t+\phi_1)}$$

and

$$\tilde{y}_2(t) = A_2 e^{i(\omega t+\phi_2)}.$$

These are different functions from $y_1(t)$ and $y_2(t)$. But if I take the imaginary part of each of those functions, I get $y_1(t)$ and $y_2(t)$. So the imaginary part of $\tilde{y}_1(t) + \tilde{y}_2(t)$ must be equal to $y_1(t) + y_2(t)$:

$$A_1\sin(\omega t + \phi_1) + A_2\sin(\omega t + \phi_1) = \Im\left[A_1 e^{i(\omega t+\phi_1)} + A_2 e^{i(\omega t+\phi_2)}\right]$$

$$= \Im\left[A_1 e^{i\omega t}e^{i\phi_1} + A_2 e^{i\omega t}e^{i\phi_2}\right].$$

Note that both terms have an $e^{i\omega t}$ in them which we can factor out:

$$A_1\sin(\omega t + \phi_1) + A_2\sin(\omega t + \phi_1) = \Im\left[\left(A_1 e^{i\phi_1} + A_2 e^{i\phi_2}\right)e^{i\omega t}\right].$$

Before we analyze this any further, let's take another look at $\tilde{y}_1(t)$ and $\tilde{y}_2(t)$. Note that I can rewrite them as

$$\tilde{y}_1(t) = A_1 e^{i(\omega t+\phi_1)} = A_1 e^{i\phi_1}e^{i\omega t} = \tilde{A}_1 e^{i\omega t}$$

and

$$\tilde{y}_2(t) = A_2 e^{i(\omega t + \phi_2)} = A_2 e^{i\phi_2} e^{i\omega t} = \tilde{A}_2 e^{i\omega t},$$

where

$$\tilde{A}_1 = A_1 e^{i\phi_1}$$

and

$$\tilde{A}_2 = A_2 e^{i\phi_2}.$$

The constants $\tilde{A}_1$ and $\tilde{A}_2$ are complex numbers which contain information about both the amplitude and phase of each wave. We call these constants the *complex amplitude* of each wave. Note, that when we added the two waves together using our complex exponential notation, we found that

$$A_1 \sin(\omega t + \phi_1) + A_2 \sin(\omega t + \phi_1) = \Im\left[\left(A_1 e^{i\phi_1} + A_2 e^{i\phi_2}\right) e^{i\omega t}\right].$$

But this is just equal to

$$\Im\left[\left(\tilde{A}_1 + \tilde{A}_2\right) e^{i\omega t}\right].$$

So when we add the two waves together, we end up just adding their complex amplitudes together! If we go one step further and take the fact that we have to take the imaginary part to get the actual function that happens in real life and we shove that to the very back of our brain, and start just thinking of the waves as if they were actually complex exponentials, then the problem of adding two different waves with different amplitudes and phases just becomes the simple process of adding their complex amplitudes.

Now, with that quick intro, watch the following video about complex exponentials, then do the exercises below
`https://youtu.be/b5hykAni1UA`

3.3 Exercises

- Write code in Python that plots $\sin(x)$ from $x = -10$ to 10. Then write code that plots the imaginary part e^{ix} from $x = -10$ to 10. Note that in Python the imaginary number is 1j. The numpy function that gives you the imaginary part of an array is np.imag(). So the imaginary part of e^{ix} would be written as np.imag(np.exp(1j*x)).

- Write code in Python that plots $\cos(x)$ from $x = -10$ to 10. Then write code that plots the real part e^{ix} from $x = -10$ to 10. The numpy function that gives you the real part of an array is np.real().

- Write code that makes an array of angles from 0 to 2π in steps of $\pi/100$. Then, for each angle θ, calculate the real and imaginary parts of $e^{i\theta}$. Then make an Argand diagram (a plot in the complex plane) of the different points. You can check your work by looking at figure 2 in appendix C. Then try plotting only a small range of values for θ and make sure you know which points correspond with which values of theta.

- Now modify your code so that instead of plotting $e^{i\theta}$, it plots $3.5e^{i\theta}$. Does the plot you get make sense?

- Now imagine what the function $f(t) = te^{it}$ would look like on an Argand diagram. Then write code to make the plot from $t = 0$ to 4π, and see if it is what you expected. For reference, my plot of this is figure 3 in appendix C. Make sure that everything makes sense, including the angle at which the plot starts, the position at which it ends, etc.

- Now imagine what the function $f(t) = Rte^{i(\omega t + \phi)}$ would look like if $R = 7.3$ m/s, $\omega = \pi$ rad/s, and $\phi = \pi/2$. Then modify your code to plot this from $t = 0$ to 5 seconds, and see if it is what you expected. For reference, my plot of this is figure 4in appendix C.

- Finally, imagine what the imaginary part of the function in the last exercise would look like if you plotted it versus time. Modify your code to plot it and see if you were right. For reference, my plot of this is figure 5 in appendix C.

4 Complex Fourier Series

Because of the close relationship of sines and cosines with complex exponentials, you probably won't be surprised to know that there is a complex exponential version of a Fourier series. In section 1.5 we saw that you can represent any function over the interval $-L/2 < x < L/2$ with a sum of sines and cosines:

$$f(x) = \frac{a_0}{2} + \sum_{n=1}^{\infty} a_n \cos(nk_0 x) + \sum_{n=1}^{\infty} b_n \sin(nk_0 x).$$

The complex version of this is

$$\boxed{f(x) = \sum_{-\infty}^{\infty} c_n e^{ink_0 x}}.$$

One weird thing to note is that this sum includes negative values of n. In other words, it includes *negative* frequencies (or, in the case of a Fourier series in x, it includes negative wavenumbers). What does it even mean to have a negative frequency or wavenumber? Well, for real things it doesn't make any sense at all: $\cos(-\omega t)$ is just equal to $\cos(\omega t)$, and $\sin(-\omega t)$ is just equal to $-\sin(\omega t)$. So changing ω to $-\omega$ doesn't really put another frequency into our sum. Likewise, changing k to $-k$ won't add a new wavenumber to our sum. But negative frequencies and wavenumbers do have meaning for complex Fourier series. Note that if you plot the real or imaginary part of $e^{i\omega t}$ vs $e^{-\omega t}$ you will get functions that oscillate at the same frequency. But if you make an Argand diagram of $e^{i\omega t}$ and $e^{-i\omega t}$, you will see that they rotate in opposite directions. Note that, unlike sines and cosines, which are two different functions with two different sets of coefficients a_n and b_n, the complex Fourier series only has one function and one set of coefficients, c_n. As such, maybe it's not surprising that we need negative values of n. Since we only have one set of coefficients, to contain the same information, you need twice as many of them in your set!

You can find the Fourier coefficients for a complex Fourier series using the formula

$$\boxed{c_n = \frac{1}{L}\int_{-L/2}^{L/2} f(x) e^{-ink_0 x}.}$$

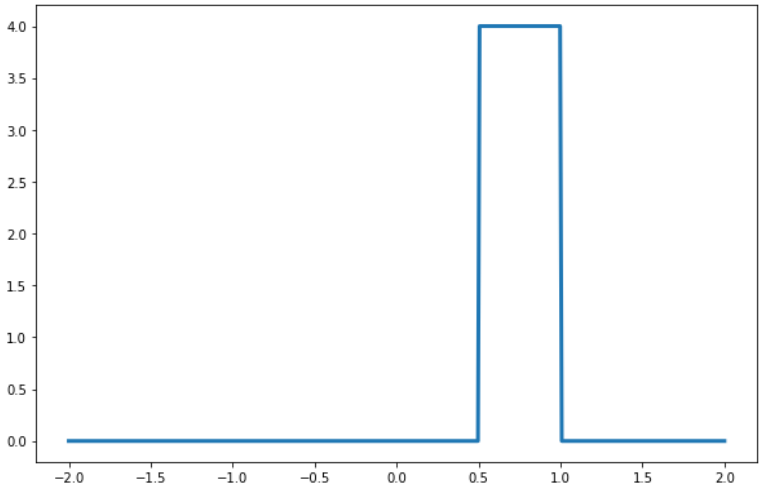
4.1 Complex vs. real functions

Note that a Fourier series can represent not just real functions, but imaginary and complex functions as well. For a sine/cosine Fourier series to represent a complex function, we simply allow the constants a_n and b_n to be complex.

For a complex Fourier transform the coefficients are generally complex, even if the function $f(x)$ is real. If the function is real, however, each negative n coefficient will just be equal to the complex conjugate of the corresponding positive n coefficient (this is derived in appendix F).

4.2 Exercises

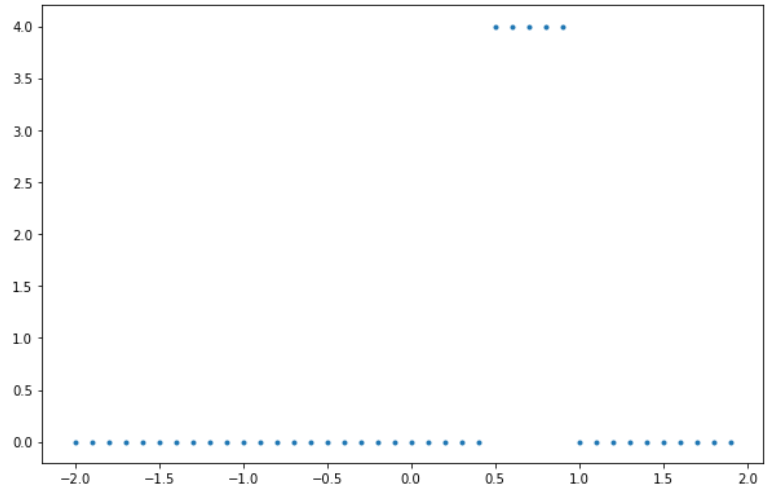
- Write code to calculate the Fourier coefficients c_n from $n = -10$ to $n = 10$ for the function $4x^2 + x^3$. Plot the real and imaginary parts of the Fourier coefficients as a function of n. Plot the real part in green and the imaginary part in red. Verify that the negative n coefficients are the complex conjugate of the corresponding positive n coefficients. For reference, the code I used and the plot it generated can be found in appendix G.
- Now add code to calculate and plot the Fourier series from $x = -L/2$ to $L/2$. Plot the real part in green and the imaginary part in red. Also directly calculate $4x^2 + x^3$ and plot that in black. Note that the imaginary part should essentially be zero because $4x^2 + x^3$ is real. My code and plot are in appendix G.
- Increase the number of coefficients and see how the plot of the Fourier series changes.
- Write code to calculate the Fourier coefficients for the function shown below. Then calculate the Fourier series and plot it. Make dx small enough and calculate enough coefficients such that your Fourier series is a reasonable representation of the function.



4.3 Discrete Fourier transforms

Real data is taken at discrete locations or moments in time. We can still write discrete data as a Fourier series, but a data set with a finite amount of information and can be represented exactly with a finite number of Fourier coefficients.

Discrete Fourier transforms usually involve data that is spaced regularly in space or time. For example, let's consider the square pulse from the previous exercise. Let's imagine that I sample this function from $x = -2$ to 2 meters in steps of 0.1 meter, giving me the data shown below



Now I'll calculate my Fourier coefficients. But, since I only know the data at points spaced by 0.1 meters, when I calculate the integral numerically, it makes sense to set dx =0.1 meters. But, because it is finite in size, let's call it Δx rather than dx. When I do that, my sum which is approximating the integral becomes

$$c_n = \frac{1}{L}\sum_{m=0}^{N-1} f_m e^{-ink_0 x}\Delta x,$$

where N is the number of data points and f_m is the m^{th} data point. Noting that the number of data points is just $L/\Delta x$ and that $x = m\Delta x$ and $k_0 = 2\pi/L = 2\pi/(N\Delta x)$, we can rewrite this as

$$\boxed{c_n = \frac{1}{N}\sum_{m=0}^{N-1} f_m e^{-i2\pi nm/N}}.$$

Here are a few things to note about this formula. First, when $n = N/2$, this becomes

$$c_{N/2} = \frac{1}{N}\sum_{m=0}^{N-1} f_m e^{-i\pi m}$$

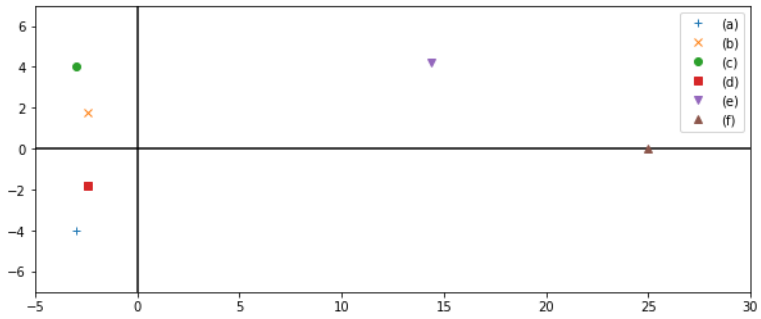


Figure 1: Various complex numbers plotted on an Argand diagram.

A Code to calculate Fourier coefficients

```
import numpy as np
dx = 1e-5
L = 4
ko = 2*np.pi/L
Nmax = 10
x = np.arange(-L/2,L/2,dx)
dy = (2/L)*(x**2)*dx
ao = np.sum(dy)
print("ao = %f" % ao)
a = np.zeros(Nmax)
b = np.zeros(Nmax)
for i in range(Nmax):
n = i+1
dy = (2/L)*(x**2)*np.cos(n*ko*x)*dx
a[i] = np.sum(dy)
print("a%d = %f" % (n,a[i]))
dy = (2/L)*(x**2)*np.sin(n*ko*x)*dx
b[i] = np.sum(dy)
print("b%d = %f" % (n,b[i]))
```

B Code to plot Fourier series

```
import matplotlib.pyplot as plt
x = np.arange(-L/2,L/2,0.001)
y = ao/2 for i in range(Nmax):
n = i+1
y += a[i]*np.cos(n*ko*x)
y += b[i]*np.sin(n*ko*x)
plt.figure(figsize=(10,6.5))
plt.plot(x,y,linewidth=3)
plt.show()
```

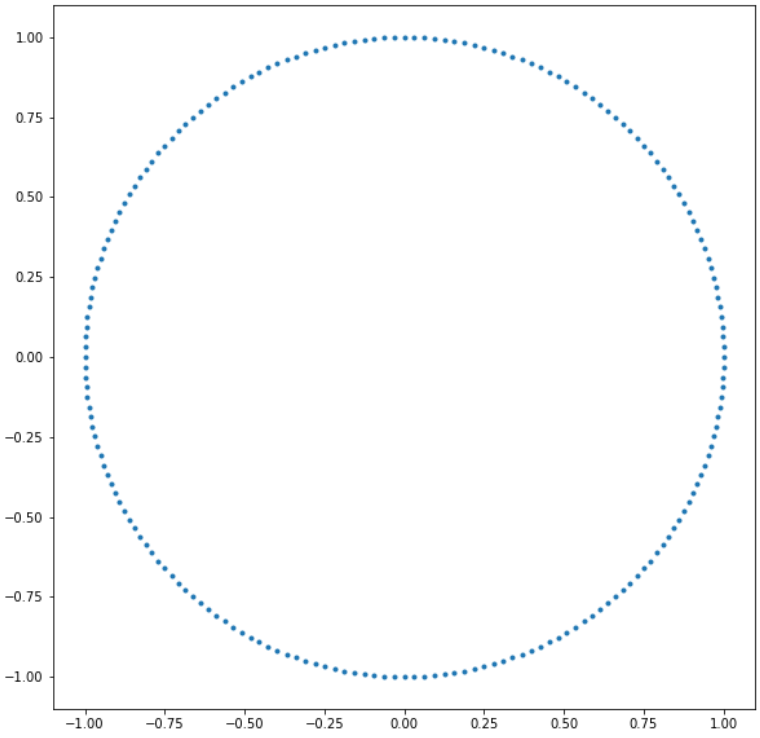


Figure 2: An Argand diagram of $e^{i\theta}$for different values of θ.

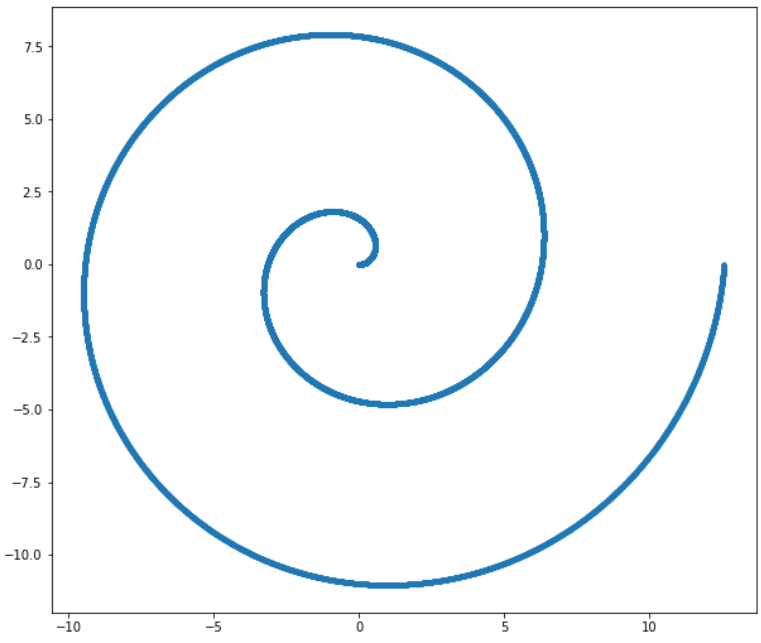


Figure 3: An Argand diagram of te^{it} for $0 \leq t \leq 4\pi$.

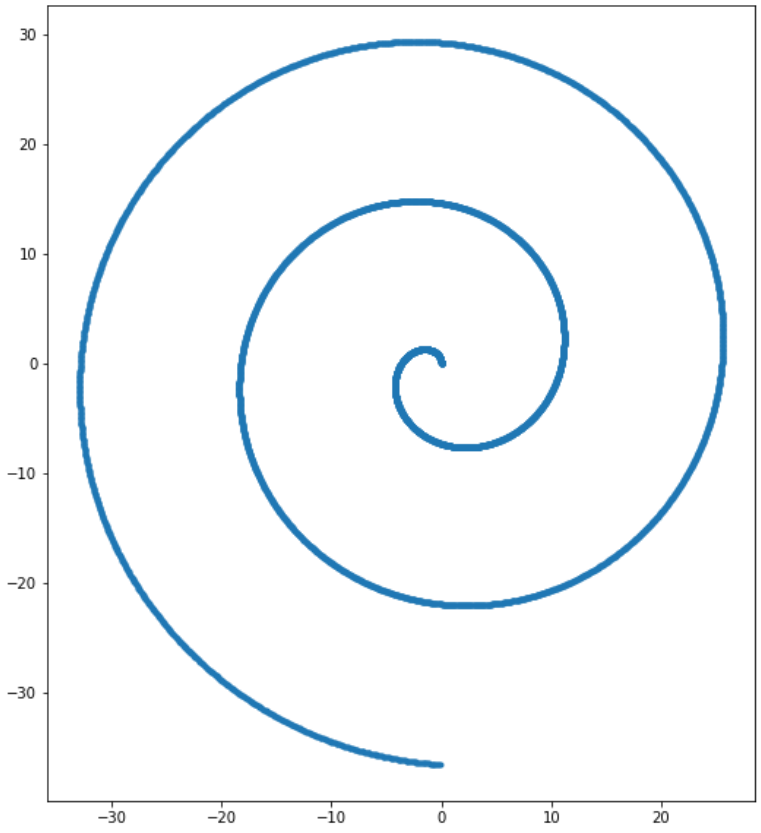


Figure 4: An Argand diagram of $Rte^{i(\omega t+\phi)}$ for $0 \leq t \leq 4$.

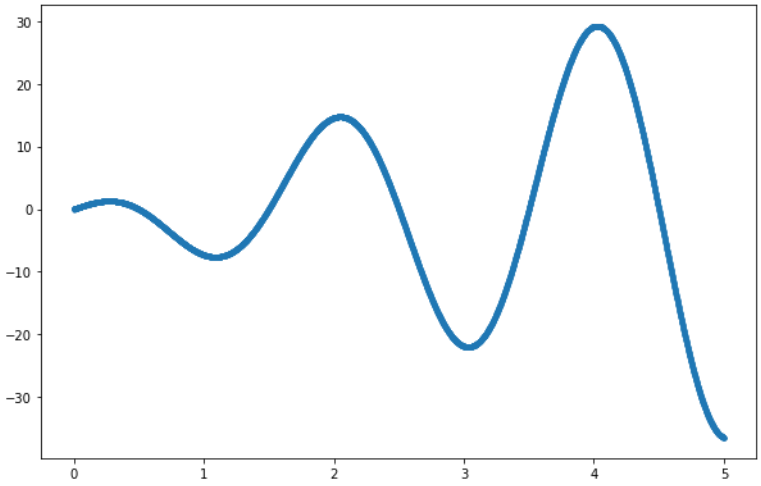


Figure 5: A plot of the imaginary part of $Rte^{i(\omega t+\phi)}$ vs time.

C Figures and Code on Complex Numbers

D Proving Euler's formula

You can prove Euler's formula using a Mauriac series (a Taylor series about $\theta = 0$):

$$e^x = 1 + x + \frac{x^2}{2} + \frac{x^3}{3!} + ... = \sum_{n=0}^{\infty} \frac{x^n}{n!} \quad \Rightarrow$$

$$e^{i\theta} = \sum_{n=0}^{\infty} \frac{i^n \theta^n}{n!} = 1 + i\theta - \frac{\theta^2}{2} - i\frac{\theta^3}{3!} + ... = \sum_{n=\text{even}} \frac{(-1)^{n/2}\theta^n}{n!} + i \sum_{n=\text{odd}} \frac{(-1)^{(n-1)/2}\theta^n}{n!}$$

$$\cos(x) = 1 - \frac{x^2}{2} + \frac{x^4}{4!} + ... = \sum_{n=\text{even}} \frac{(-1)^{n/2}x^n}{n!} \quad \Rightarrow$$

$$\cos(\theta) = \sum_{n=\text{even}} \frac{(-1)^{n/2}\theta^n}{n!}$$

$$\sin(x) = x - \frac{x^3}{3!} + \frac{x^5}{5!} + ... = \sum_{n=\text{odd}} \frac{(-1)^{(n-1)/2}x^n}{n!} \quad \Rightarrow$$

$$i\sin(\theta) = i \sum_{n=\text{odd}} \frac{(-1)^{(n-1)/2}\theta^n}{n!}$$

$$\cos(\theta) + i\sin(\theta) = \sum_{n=\text{even}} \frac{(-1)^{n/2}\theta^n}{n!} + i \sum_{n=\text{odd}} \frac{(-1)^{(n-1)/2}\theta^n}{n!} = e^{i\theta}.$$

E Using Euler's formula to derive trig identities

This subsection isn't essential to understand IPSII, so you can skip it if you wish. But I find it interesting, and very useful, that you can use Euler's formula to find all kinds of trig identities. For example, writing

$$e^{i(a+b)} = e^{ia}e^{ib}$$

and then applying Euler's formula to this, we get

$$\cos(a+b) + i\sin(a+b) = (\cos(a) + i\sin(a))\,(\cos(b) + i\sin(b))$$

$$= \cos(a)\cos(b) - \sin(a)\sin(b) + i\sin(a)\cos(b) + i\sin(b)\cos(a).$$

Then, noting that the real part of the left side must be equal to the real part of the right side, and the imaginary part of the left must equal the imaginary part of the right, we can separate this into the two equations

$$\cos(a+b) = \cos(a)\cos(b) - \sin(a)\sin(b),$$

$$\sin(a+b) = \sin(a)\cos(b) + \sin(b)\cos(a),$$

which are the sum identities for sine and cosine.

We can also write a pure sine or cosine in terms of complex exponentials. If

$$e^{i\theta} = \cos(\theta) + i\sin(\theta),$$

then

$$e^{-i\theta} = \cos(-\theta) + i\sin(-\theta) = \cos(\theta) - i\sin(\theta).$$

That means that

$$e^{i\theta} + e^{-i\theta} = 2\cos(\theta) \quad \Rightarrow \quad \boxed{\cos(\theta) = \frac{e^{i\theta} + e^{-i\theta}}{2}},$$

and

$$e^{i\theta} - e^{-i\theta} = 2i\sin(\theta) \;\Rightarrow\; \boxed{\sin(\theta) = \frac{e^{i\theta} - e^{-i\theta}}{2i}}.$$

With Euler's formula and these relations, we can derive the trig identity that we used to understand standing waves and beats in Physics 2210:

$$\cos(a)\sin(b) = \left(\frac{e^{ia} + e^{-ia}}{2}\right)\left(\frac{e^{ib} - e^{-ib}}{2i}\right) = \frac{e^{ia}e^{ib} - e^{ia}e^{-ib} + e^{-ia}e^{ib} - e^{-ia}e^{-ib}}{4i}$$

$$= \frac{e^{ia}e^{ib} - e^{-ia}e^{-ib}}{4i} + \frac{e^{-ia}e^{ib} - e^{ia}e^{-ib}}{4i} = \frac{e^{i(a+b)} - e^{-i(a+b)}}{4i} + \frac{e^{i(b-a)} - e^{-i(b-a)}}{4i}$$

$$= \frac{1}{2}\sin(a+b) + \frac{1}{2}\sin(b-a) \;\Rightarrow$$

$$2\cos(a)\sin(b) = \sin(a+b) + \sin(b-a).$$

Then, if we define $A = a + b$ and $B = b - a$, such that $a = (A - B)/2$ and $b = (A + B)/2$, this becomes

$$\boxed{\sin(A) + \sin(B) = 2\cos\left(\frac{A-B}{2}\right)\sin\left(\frac{A+B}{2}\right)}.$$

F Fourier coefficients for a real function

As discussed above, on an Argand diagram negative and positive wavenumbers or frequencies will rotate in opposite directions with time. But if $f(x)$ is real at every value of x in the domain, each frequency component that we add together must also be real. As such, when we add together the n and the $-n$ term, the imaginary parts must cancel out:

$$\Im\left[c_n e^{ink_0x} + c_{-n}e^{-ink_0x}\right] = 0.$$

Using Euler's formula, we can write

$$c_n e^{ink_0x} + c_{-n}e^{-ink_0x} = c_n\cos(nk_0x) + ic_n\sin(nk_0x) + c_{-n}\cos(nk_0x) - ic_{-n}\sin(nk_0x)$$

$$= (c_n + c_{-n})\cos(nk_0x) + i(c_n - c_{-n})\sin(nk_0x).$$

The imaginary part of this is

$$\Im\left[c_n + c_{-n}\right]\cos(nk_0x) + \Re\left[c_n - c_{-n}\right]\sin(nk_0x).$$

For this to be zero at all values of x, both the imaginary part of $c_n + c_{-n}$ and the real part of $c_n - c_{-n}$ have to be zero. This means that the imaginary part of c_n has to be the negative of the imaginary part of c_{-n}, and the real parts of $c_n$and c_{-n} must be the same. So, that means that if $f(x)$ is a real function, then

$$c_{-n} = c_n^*.$$

As such, if we know that our function is real, we can simply replace all of our negative n coefficients with the complex conjugate of the corresponding positive n coefficient.

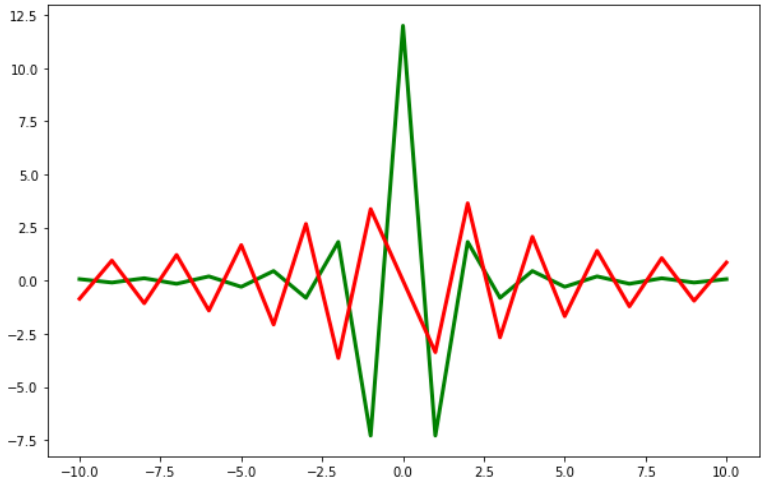
G Plots and code for complex Fourier series

Here is the code I used to calculate and plot the Fourier coefficients

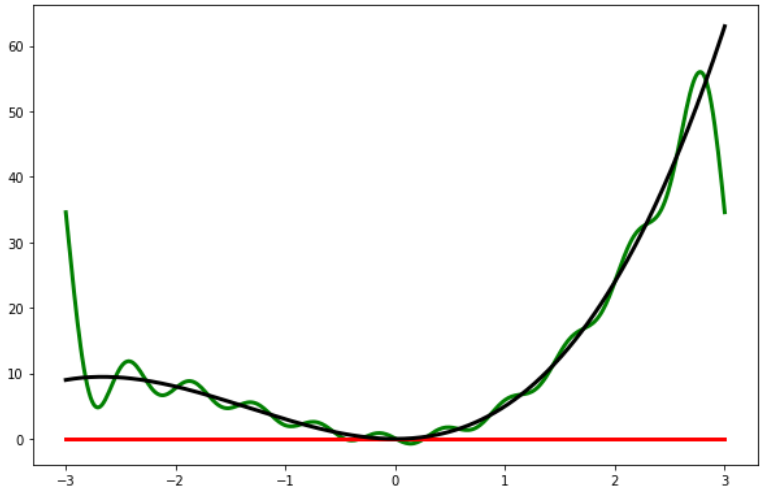
```
import numpy as np
import matplotlib.pyplot as plt

def f(x):
return(4*x**2+x**3)

L = 6
```

```
Nmax = 10
dx = 0.0001
ko = 2*np.pi/L
x = np.arange(-L/2,L/2,dx)
c = np.zeros(Nmax*2+1).astype(np.complex)
n = np.zeros(Nmax*2+1)
for i in range(Nmax*2+1):
n[i] = -Nmax+i
y = f(x)*np.exp(-1j*n[i]*ko*x)*dx
c[i] = np.sum(y)
c *= 1/L
plt.figure(figsize=(10,6.5))
plt.plot(n,np.real(c),'g',linewidth=3)
plt.plot(n,np.imag(c),'r',linewidth=3)
plt.show()
```

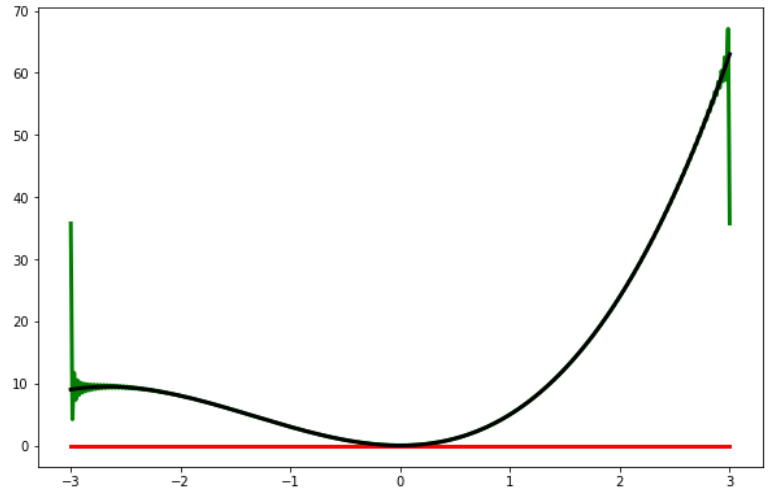
Here is the plot this generated.



Here is the additional code to calculate and plot the series along with the original function.



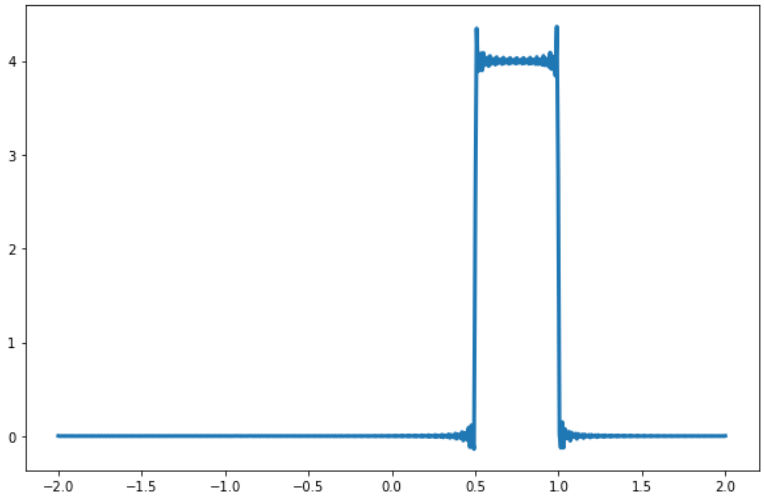
Here's what I got when I calculated coefficients from $n = -200$ to 200.



Here's the code for the next exercise.

```
import numpy as np
import matplotlib.pyplot as plt
L = 4
ko = 2*np.pi/L
dx = 0.0001
Nmax = 200
x = np.arange(0.5,1,dx)
c = np.zeros(2*Nmax+1).astype(np.complex)
n = np.zeros(2*Nmax+1)
for i in range(2*Nmax+1):
n[i] = -Nmax+i
y = 4*np.exp(-1j*n[i]*ko*x)*dx
c[i] = np.sum(y)
c /= L

x = np.linspace(-L/2,L/2,500)
y = np.zeros(x.shape).astype(np.complex)
for i in range(2*Nmax+1):
y += c[i]*np.exp(1j*n[i]*ko*x)
plt.figure(figsize=(10,6.5))
plt.plot(x,np.real(y),linewidth=3)
plt.show()
```

Here's the plot it generated.


4
3
2
1
0
−2.0
−1.5
−1.0
−0.5
0.0
0.5
1.0
1.5
2.0