**Exercise 1: Time Domain and Frequency Domain representation using Python.**

# **Motivation**

In this chapter, we will start to introduce you the Fourier method that named after the French mathematician and physicist Joseph Fourier, who used this type of method to study the heat transfer. The basic idea of this method is to express some complicated functions as the infinite sum of sine and cosine waves. We saw this in the previous chapters, that we can decompose a function using the Taylor series, which express the function with an infinite sum of polynomials.

The Fourier method has many applications in engineering and science, such as signal processing, partial differential equations, image processing and so on. The Fast Fourier Transform is chosen as one of the 10 algorithms with the greatest influence on the development and practice of science and engineering in the 20th century in the January/February 2000 issue of *Computing in Science and Engineering*. In this chapter, we take the Fourier transform as an independent chapter with more focus on the signal processing, which we will encounter in many problems in science and engineering. By the end of this chapter, you should be able to know the basics of Fourier transform, as well as how to do simple signal analysis with it.

# **The Basics of Waves**

There are many types of waves in our life, for example, if you throw a rock into a pond, you can see the waves form and travel in the water. Of course, there are many more examples of waves, some of them are even difficult to see, such as such as sound waves, earthquake waves, microwaves (that we use to cook our food in the kitchen). But in physics, a wave is a disturbance that travels through space and matter with a transferring energy from one place to another. It is important to study waves in our life to understand how they form, travel and so on. In this chapter, we will cover a basic tool that help us to understand and study the waves - the **Fourier Transform**. But before we proceed, let’s first get familiar how do we actually model the waves and study it.

# **Model a wave using mathematical tools**

We can model a single wave as a field with a function F(x,t), where x is the location of a point in space, while t is the time. One simplest case is the shape of a sine wave change over x.

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

plt.style.use('seaborn-poster')

%**matplotlib** inline

x = np.linspace(0, 20, 201)

y = np.sin(x)

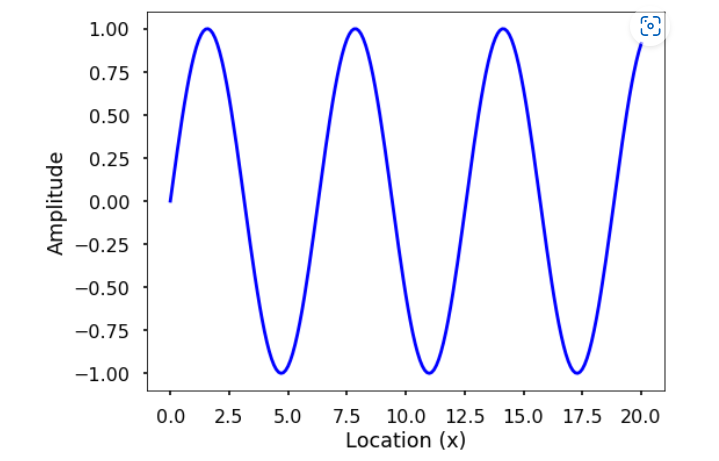
plt.figure(figsize = (8, 6))

plt.plot(x, y, 'b')

plt.ylabel('Amplitude')

plt.xlabel('Location (x)')

plt.show()



We can think of the sine wave can change both in time and space. If we plot the changes at various locations, each time snapshot will be a sine wave changes with location. See the following figure with a fix point at x=2.5 showing as a red dot. Of course, you can see the changes over time at specific location as well, you can plot this by yourself.

fig = plt.figure(figsize = (8,8))

times = np.arange(5)

n = len(times)

**for** t **in** times:

plt.subplot(n, 1, t+1)

y = np.sin(x + t)

plt.plot(x, y, 'b')

plt.plot(x[25], y [25], 'ro')

plt.ylim(-1.1, 1.1)

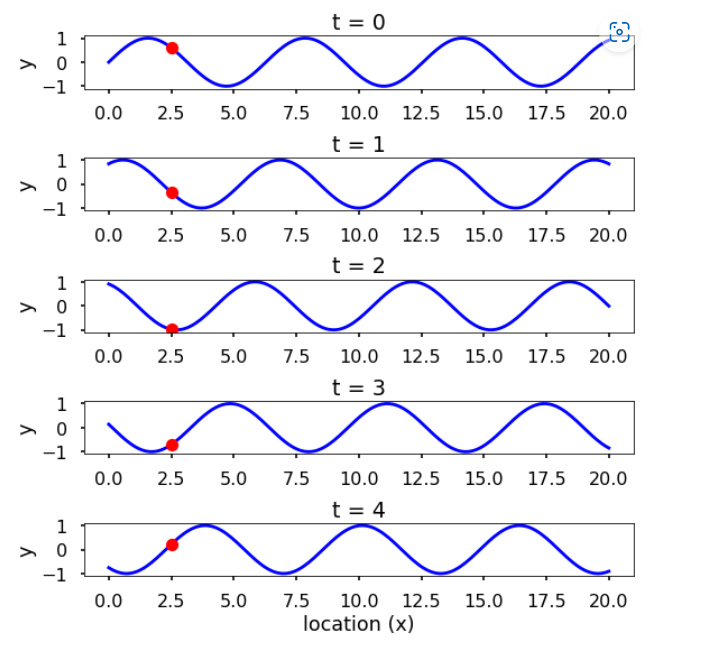
plt.ylabel('y')

plt.title(f't = *{*t*}*')

plt.xlabel('location (x)')

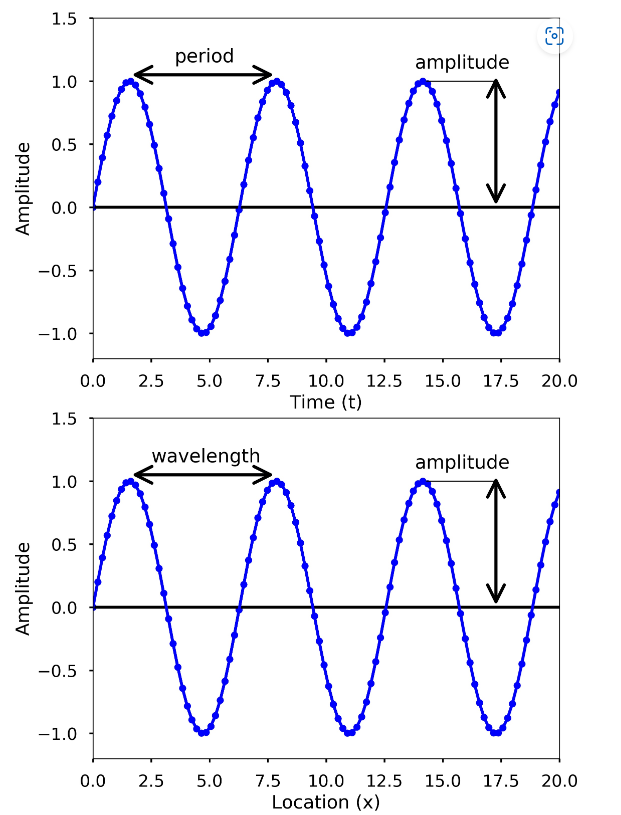
plt.tight\_layout()

plt.show()



# **Characteristics of a wave**

We can see waves can be a continuous entity both in time and space. But in reality, many times we discrete the time and space at various points. For example, we can use sensors such as accelerometers (can measure the acceleration of a movement) at different locations on the Earth to monitor the earthquakes, which is spatial discretization. Similarly, these sensors usually record the data at certain times which is a temporal discretization. For a single wave, it has different characteristics. See the following two figures.



**Amplitude** is used to describe the difference between the maximum values to the baseline value (see the above figures). A sine wave is a periodic signal, which means it repeats itself after certain time, which can be measured by **period**.

Period of a wave is time it takes to finish the complete cycle, in the figure, we can see that the period can be measured from the two adjacent peaks.

**Wavelength** measures the distance between two successive crests or troughs of a wave.

**Frequency** describes the number of waves that pass a fixed place in a given amount of time. Frequency can be measured by how many cycles pass within 1 second. Therefore, the unit of frequency is cycles/second, or more commonly used **Hertz** (abbreviated **Hz**). Frequency is different from period, but they are related to each other. Frequency refers to how often something happens while period refers to the time it takes to complete something, mathematically,

period=1/frequency

From the two figures, we can also see that blue dots on the sine waves, these are the discretization points we did both in time and space. Therefore, only at these dots, we have sampled the value of the wave. Usually when we record a wave, we need to specify how often we sample the wave in time, this is called **sampling**. And this rate is called **sampling rate**, with the unit Hz. For example, if we sample a wave at 2 Hz, it means that every second we sample two data points. Since we understand more about the basics about a wave, now let’s see a sine wave more carefully. A sine wave can be represented by the following equation:

y(t)=Asin(ωt+ϕ)

where A is the amplitude of the wave, ω is the **angular frequency**, which specifies how many cycles occur in a second, in radians per second. ϕ is the **phase** of the signal. If T is the period of the wave, and f is the frequency of the wave, then ω has the following relationship to them:

ω=2π/T=2πf

**TRY IT!** Generate two sine waves with time between 0 and 1 seconds and frequency is 5 Hz and 10 Hz, all sampled at 100 Hz. Plot the two waves and see the difference. Count how many cycles in the 1 second.

*# sampling rate*

sr = 100.0

*# sampling interval*

ts = 1.0/sr

t = np.arange(0,1,ts)

*# frequency of the signal*

freq = 5

y = np.sin(2\*np.pi\*freq\*t)

plt.figure(figsize = (8, 8))

plt.subplot(211)

plt.plot(t, y, 'b')

plt.ylabel('Amplitude')

freq = 10

y = np.sin(2\*np.pi\*freq\*t)

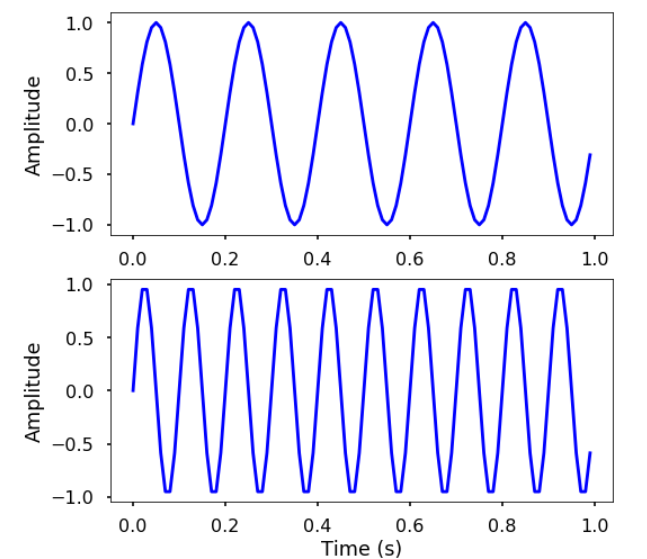
plt.subplot(212)

plt.plot(t, y, 'b')

plt.ylabel('Amplitude')

plt.xlabel('Time (s)')

plt.show()



**TRY IT!** Generate two sine waves with time between 0 and 1 seconds. Both waves have frequency 5 Hz and sampled at 100 Hz, but the phase at 0 and 10, respectively. Also the amplitude of the two waves are 5 and 10. Plot the two waves and see the difference.

*# frequency of the signal*

freq = 5

y = 5\*np.sin(2\*np.pi\*freq\*t)

plt.figure(figsize = (8, 8))

plt.subplot(211)

plt.plot(t, y, 'b')

plt.ylabel('Amplitude')

y = 10\*np.sin(2\*np.pi\*freq\*t + 10)

plt.subplot(212)

plt.plot(t, y, 'b')

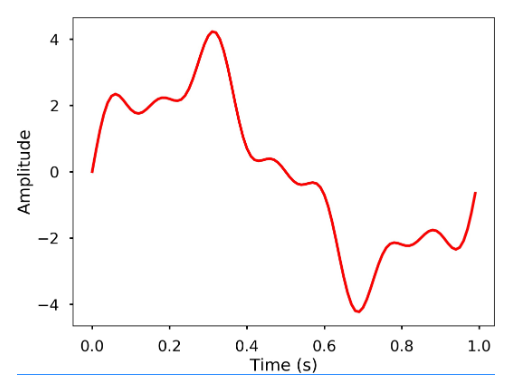
plt.ylabel('Amplitude')

plt.xlabel('Time (s)')

plt.show()

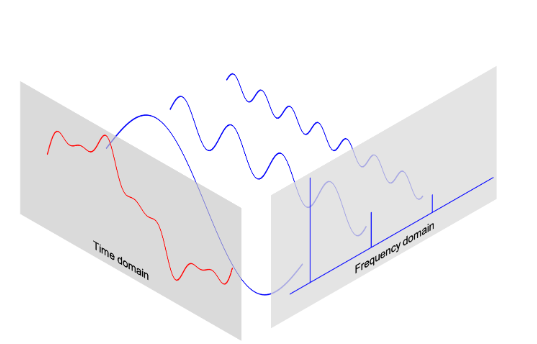
# **Discrete Fourier Transform (DFT)**

From the previous section, we learned how we can easily characterize a wave with period/frequency, amplitude, phase. But these are easy for simple periodic signal, such as sine or cosine waves. For complicated waves, it is not easy to characterize like that. For example, the following is a relatively more complicate waves, and it is hard to say what’s the frequency, amplitude of the wave, right?



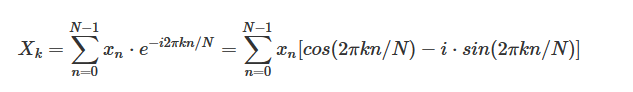
There are more complicated cases in real world, it would be great if we have a method that we can use to analyze the characteristics of the wave. The **Fourier Transform** can be used for this purpose, which it decompose any signal into a sum of simple sine and cosine waves that we can easily measure the frequency, amplitude and phase. The Fourier transform can be applied to continuous or discrete waves, in this chapter, we will only talk about the Discrete Fourier Transform (DFT).

Using the DFT, we can compose the above signal to a series of sinusoids and each of them will have a different frequency. The following 3D figure shows the idea behind the DFT, that the above signal is actually the results of the sum of 3 different sine waves. The time domain signal, which is the above signal we saw can be transformed into a figure in the frequency domain called DFT amplitude spectrum, where the signal frequencies are showing as vertical bars. The height of the bar after normalization is the amplitude of the signal in the time domain. You can see that the 3 vertical bars are corresponding the 3 frequencies of the sine wave, which are also plotted in the figure.



# **DFT**

The DFT can transform a sequence of evenly spaced signal to the information about the frequency of all the sine waves that needed to sum to the time domain signal. It is defined as:



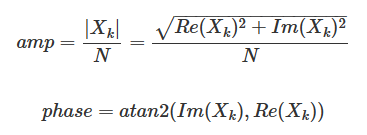
where

* N = number of samples
* n = current sample
* k = current frequency, where k∈[0,N−1]
* xn = the sine value at sample n
* Xk = The DFT which include information of both amplitude and phase

Also, the last expression in the above equation derived from the Euler’s formula, which links the trigonometric functions to the complex exponential function: eix=cosx+i⋅sinx

Due to the nature of the transform, X0=∑N−1n=0xnX0=∑n=0N−1xn. If N is an odd number, the elements X1,X2,...,X(N−1)/2 contain the positive frequency terms and the elements X(N+1)/2,...,XN−1 contain the negative frequency terms, in order of decreasingly negative frequency. While if N is even, the elements X1,X2,...,XN/2−1 contain the positive frequency terms, and the elements XN/2,...,XN−1 contain the negative frequency terms, in order of decreasingly negative frequency. In the case that our input signal x is a real-valued sequence, the DFT output Xn for positive frequencies is the conjugate of the values Xn for negative frequencies, the spectrum will be symmetric. Therefore, usually we only plot the DFT corresponding to the positive frequencies.

Note that the Xk is a complex number that encodes both the amplitude and phase information of a complex sinusoidal component ei⋅2πkn/N of function xnxn. The amplitude and phase of the signal can be calculated as:



where Im(Xk) and Re(Xk)) are the imagery and real part of the complex number, atan2atan2 is the two-argument form of the arctan function.

The amplitudes returned by DFT equal to the amplitudes of the signals fed into the DFT if we normalize it by the number of sample points. Note that doing this will divide the power between the positive and negative sides, if the input signal is real-valued sequence as we described above, the spectrum of the positive and negative frequencies will be symmetric, therefore, we will only look at one side of the DFT result, and instead of divide NN, we divide N/2 to get the amplitude corresponding to the time domain signal.

Now that we have the basic knowledge of DFT, let’s see how we can use it.

**TRY IT!** Generate 3 sine waves with frequencies 1 Hz, 4 Hz, and 7 Hz, amplitudes 3, 1 and 0.5, and phase all zeros. Add this 3 sine waves together with a sampling rate 100 Hz, you will see that it is the same signal we just shown at the beginning of the section.

**import** **matplotlib.pyplot** **as** **plt**

**import** **numpy** **as** **np**

plt.style.use('seaborn-poster')

%**matplotlib** inline

*# sampling rate*

sr = 100

*# sampling interval*

ts = 1.0/sr

t = np.arange(0,1,ts)

freq = 1.

x = 3\*np.sin(2\*np.pi\*freq\*t)

freq = 4

x += np.sin(2\*np.pi\*freq\*t)

freq = 7

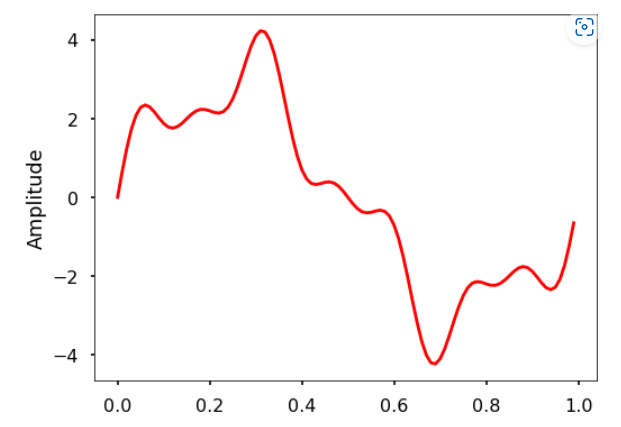
x += 0.5\* np.sin(2\*np.pi\*freq\*t)

plt.figure(figsize = (8, 6))

plt.plot(t, x, 'r')

plt.ylabel('Amplitude')

plt.show()



**TRY IT!** Write a function *DFT(x)* which takes in one argument, *x* - input 1 dimensional real-valued signal. The function will calculate the DFT of the signal and return the DFT values. Apply this function to the signal we generated above and plot the result.

**def** DFT(x):

*"""*

*Function to calculate the*

*discrete Fourier Transform*

*of a 1D real-valued signal x*

*"""*

N = len(x)

n = np.arange(N)

k = n.reshape((N, 1))

e = np.exp(-2j \* np.pi \* k \* n / N)

X = np.dot(e, x)

**return** X

X = DFT(x)

*# calculate the frequency*

N = len(X)

n = np.arange(N)

T = N/sr

freq = n/T

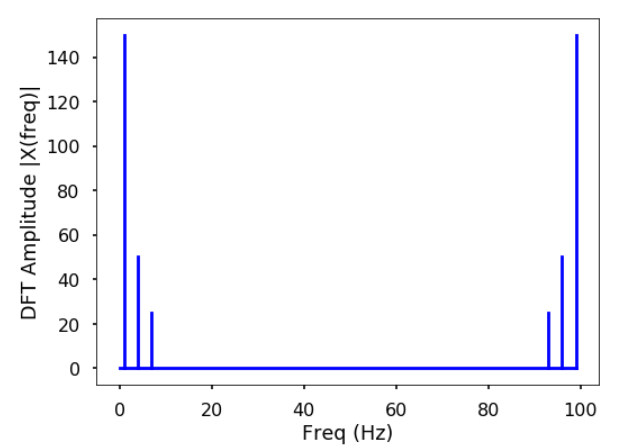
plt.figure(figsize = (8, 6))

plt.stem(freq, abs(X), 'b', markerfmt=" ", basefmt="-b")

plt.xlabel('Freq (Hz)')

plt.ylabel('DFT Amplitude |X(freq)|')

plt.show()



We can see from here that the output of the DFT is symmetric at half of the sampling rate (you can try different sampling rate to test). This half of the sampling rate is called **Nyquist frequency** or the folding frequency, it is named after the electronic engineer Harry Nyquist. He and Claude Shannon have the Nyquist-Shannon sampling theorem, which states that a signal sampled at a rate can be fully reconstructed if it contains only frequency components below half that sampling frequency, thus the highest frequency output from the DFT is half the sampling rate.

n\_oneside = N//2

*# get the one side frequency*

f\_oneside = freq[:n\_oneside]

*# normalize the amplitude*

X\_oneside =X[:n\_oneside]/n\_oneside

plt.figure(figsize = (12, 6))

plt.subplot(121)

plt.stem(f\_oneside, abs(X\_oneside), 'b', \

markerfmt=" ", basefmt="-b")

plt.xlabel('Freq (Hz)')

plt.ylabel('DFT Amplitude |X(freq)|')

plt.subplot(122)

plt.stem(f\_oneside, abs(X\_oneside), 'b', \

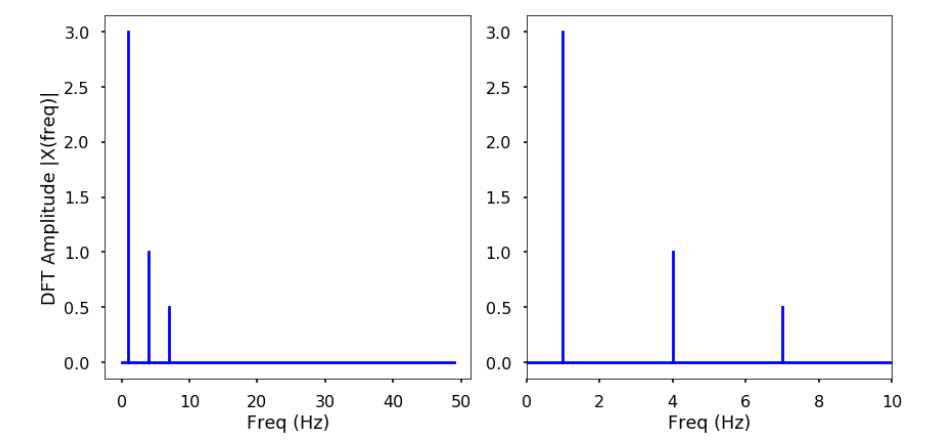
markerfmt=" ", basefmt="-b")

plt.xlabel('Freq (Hz)')

plt.xlim(0, 10)

plt.tight\_layout()

plt.show()



We can see by plotting the first half of the DFT results, we can see 3 clear peaks at frequency 1 Hz, 4 Hz, and 7 Hz, with amplitude 3, 1, 0.5 as expected. This is how we can use the DFT to analyze an arbitrary signal by decomposing it to simple sine waves.

# **Fast Fourier Transform (FFT)**

The **Fast Fourier Transform (FFT)** is an efficient algorithm to calculate the DFT of a sequence. It is described first in Cooley and Tukey’s classic paper in 1965, but the idea actually can be traced back to Gauss’s unpublished work in 1805. It is a divide and conquer algorithm that recursively breaks the DFT into smaller DFTs to bring down the computation. As a result, it successfully reduces the complexity of the DFT from O(n2) to O(nlogn), where nn is the size of the data. This reduction in computation time is significant especially for data with large N, therefore, making FFT widely used in engineering, science and mathematics.

# **FFT in Scipy**

**EXAMPLE:** Use fft and ifft function from scipy to calculate the FFT amplitude spectrum and inverse FFT to obtain the original signal. Plot both results. Time the fft function using this 2000 length signal.

**from** **scipy.fftpack** **import** fft, ifft

X = fft(x)

plt.figure(figsize = (12, 6))

plt.subplot(121)

plt.stem(freq, np.abs(X), 'b', \

markerfmt=" ", basefmt="-b")

plt.xlabel('Freq (Hz)')

plt.ylabel('FFT Amplitude |X(freq)|')

plt.xlim(0, 10)

plt.subplot(122)

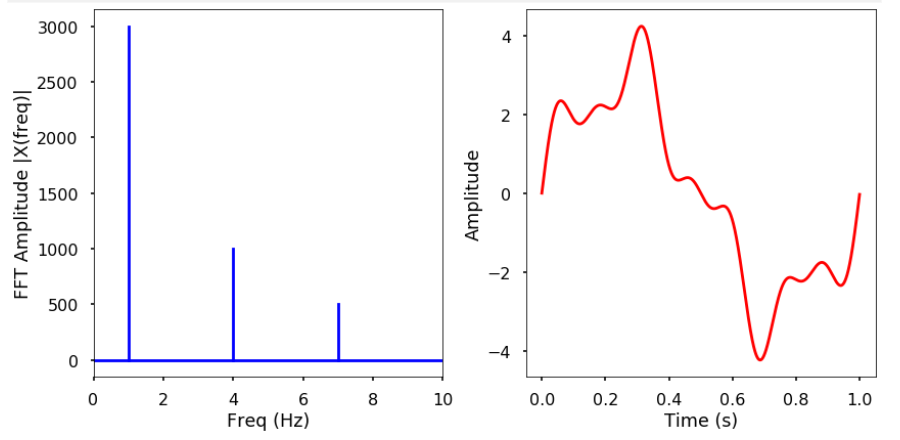
plt.plot(t, ifft(X), 'r')

plt.xlabel('Time (s)')

plt.ylabel('Amplitude')

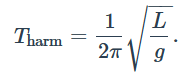
plt.tight\_layout()

plt.show()

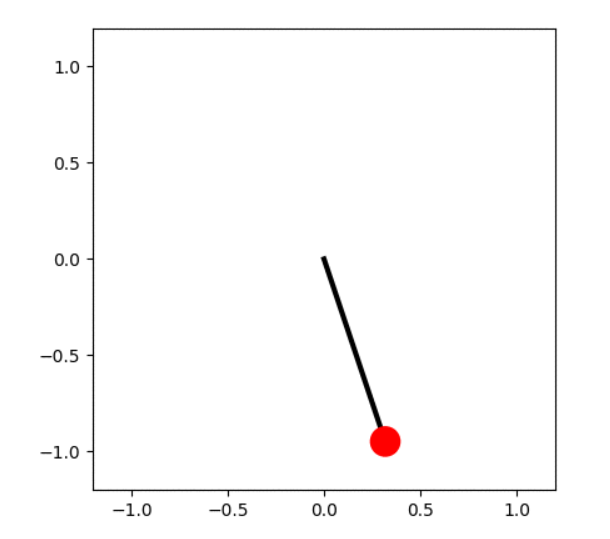


**Exercise 2: Pendulum Simulation using Python.**

In the code below the equation of motion for the pendulum is solved numerically and each frame of the animation shows an advance of the simulation by a small amount of time, dt. The small-angle approximation for the pendulum's motion gives a lower bound to the period,



This is used to determine when the pendulum has made one complete swing. The angular positions are calculated for a single period and then used to determine the position of the pendulum bob, which is drawn as a red circle.



import numpy as np

import matplotlib.pyplot as plt

from scipy.special import ellipk

import matplotlib.animation as animation

from matplotlib.animation import PillowWriter

from IPython.display import HTML

plt.style.use('seaborn')

# Bob mass (kg), pendulum length (m), acceleration due to gravity (m.s-2).

m, L, g = 1, 1, 9.81

# Initial angular displacement (rad), tangential velocity (m.s-1)

theta0, v0 = np.radians(60), 0

# Estimate of the period using the harmonic (small displacement) approximation.

# The real period will be longer than this.

Tharm = 2 \* np.pi \* np.sqrt(L / g)

# Time step for numerical integration of the equation of motion (s).

dt = 0.001

# Initial angular position and tangential velocity.

theta, v = [theta0], [v0]

old\_theta = theta0

i = 0

while True:

# Forward Euler method for numerical integration of the ODE.

i += 1

t = i \* dt

# Update the bob position using its updated angle.

old\_theta, old\_v = theta[-1], v[-1]

omega = old\_v / L

new\_theta = old\_theta - omega \* dt

# Tangential acceleration.

acc = g \* np.sin(old\_theta)

# Update the tangential velocity.

new\_v = old\_v + acc \* dt

if t > Tharm and new\_v \* old\_v < 0:

# At the second turning point in velocity we're back where we started,

# i.e. we have completed one period and can finish the simulation.

break

theta.append(new\_theta)

v.append(new\_v)

# Calculate the estimated pendulum period, T, from our numerical integration,

# and the "exact" value in terms of the complete elliptic integral of the

# first kind.

nsteps = len(theta)

T = nsteps \* dt

print('Calculated period, T = {} s'.format(T))

print('Estimated small-displacement angle period, Tharm = {} s'.format(Tharm))

k = np.sin(theta0/2)

print('SciPy calculated period, T = {} s'

.format(2 \* Tharm / np.pi \* ellipk(k\*\*2)))

def get\_coords(th):

"""Return the (x, y) coordinates of the bob at angle th."""

return L \* np.sin(th), -L \* np.cos(th)

Calculated period, T = 2.1550000000000002 s

Estimated small-displacement angle period, Tharm = 2.0060666807106475 s

SciPy calculated period, T = 2.152874666880516 s

# Initialize the animation plot. Make the aspect ratio equal so it looks right.

fig = plt.figure()

ax = fig.add\_subplot(aspect='equal')

# The pendulum rod, in its initial position.

x0, y0 = get\_coords(theta0)

line, = ax.plot([0, x0], [0, y0], lw=3, c='k')

# The pendulum bob: set zorder so that it is drawn over the pendulum rod.

bob\_radius = 0.08

circle = ax.add\_patch(plt.Circle(get\_coords(theta0), bob\_radius,

fc='r', zorder=3))

# Set the plot limits so that the pendulum has room to swing!

ax.set\_xlim(-L\*1.2, L\*1.2)

ax.set\_ylim(-L\*1.2, L\*1.2)

def animate(i):

"""Update the animation at frame i."""

x, y = get\_coords(theta[i])

line.set\_data([0, x], [0, y])

circle.set\_center((x, y))

nframes = nsteps

interval = dt \* 1000

ani = animation.FuncAnimation(fig, animate, frames=nframes, repeat=True,

interval=interval)

HTML(ani.to\_html5\_video())