Table of Contents

[PI - ARCHIMEDES METHOD 1](#_Toc67163520)

[EULER’S CONSTANT 2](#_Toc67163521)

[THE GOLDEN RATIO 3](#_Toc67163522)

[THE LOGISTIC CURVE 4](#_Toc67163523)

[PENDULUMS & CHAOS 5](#_Toc67163524)

[PHASE SPACE 8](#_Toc67163525)

[MATRICES 12](#_Toc67163526)

[PYTHON LIBRARIES 12](#_Toc67163527)

[NUMPY 12](#_Toc67163528)

[SCIPY 13](#_Toc67163529)

[SECOND ORDER DIFFERENTIAL EQUATIONS 15](#_Toc67163530)

[MONTECARLO TECHNIQUES 16](#_Toc67163531)

[STOCHASTIC METHDODS 17](#_Toc67163532)

[RANDOM WALK 17](#_Toc67163533)

[PARTIAL DIFFERENTIAL EQUATIONS 19](#_Toc67163534)

[INTEREST RATES 19](#_Toc67163535)

[THE HEAT EQUATION: 19](#_Toc67163536)

[SCHRODINGER'S EQUATION 21](#_Toc67163537)

[THE WAVE EQUATION 23](#_Toc67163538)

[TAYLOR & MaCLAURiN SERIES applications 25](#_Toc67163539)

[appoximating FUNCTIONS 25](#_Toc67163540)

[solving DIFFERENTIAL EQUATIONS 28](#_Toc67163541)

[Pendulum Motion 28](#_Toc67163542)

[potential energy of an object 28](#_Toc67163543)

[FOURIER SERIES 29](#_Toc67163544)

[FOURIER analysis 32](#_Toc67163545)

[cONTINOUS FOURIER TRANSFORM 33](#_Toc67163546)

[WAVELET ANALYSIS 36](#_Toc67163547)

# PI - ARCHIMEDES METHOD

Let the number of sides of the polygon = n; length of each side of the polygon = p; and, the radius of the enclosed circle = r.

θ

θ

*r*

*r*

*p*

p

Perimeter of the polygon, where,

Circumference of the circle,

PYTHON: Refer to code ref: 09

RESULTS:

For value *θ =10- 93*

*π = 3.141592653589793115997963468544185161590576171875*

Value given after four iterations of the *Chudnovsky Algorithm*

*π = 3.1415926535897932384626433832795028841971693993750*

*error = 1.224646799147353177226065932275 x 10 -16*

But, there is an issue, python language works in radians, therefor requires *π* in conversion from degrees to radians.

# EULER’S CONSTANT

Z is a complex number:

Python function is :

import math

def eulerConstant(m):

e = (1+((complex(0,1)\*math.pi)/m))\*\*m

return e

for n in range(1,200,20):

print("m= ",m, " z = ", eulerConstant(n))

Results:

Iteration: m= 1 z = (1+3.141592653589793j)

Iteration: m= 21 z = (-1.2612864254825487+0.029174442792406885j)

Iteration: m= 41 z = (-1.1274859162512338+0.006907978896579237j)

Iteration: m= 61 z = (-1.0841404413646587+0.0030065259364327607j)

Iteration: m= 81 z = (-1.0627676205243795+0.0016726507178846939j)

Iteration: m= 101 z = (-1.050047387619564+0.0010632677771804516j)

Iteration: m= 121 z = (-1.0416119865698474+0.0007350012136457555j)

Iteration: m= 141 z = (-1.0356091224850832+0.0005382161335472575j)

Iteration: m= 161 z = (-1.0311194246078024+0.00041104219099277734j)

Iteration: m= 181 z = (-1.0276348964280082+0.0003241392179054821j)

Suggests that

Euler’s’ Equation

and

Euler’s formula, angle measured in radians

Euler’s number Also can also be described by the series:

In python

40 2.7182818284590452353602874713526624977572470936998

# THE GOLDEN RATIO

In python: from decimal import \*

#Sets decimal to 50 digits of precision

getcontext().prec = 50

phi = 11.28

for i in range(1,150):

phi = Decimal(1 + 1/phi)

print(i, phi)

Results:

Iteration No: 120 phi = 1.6180339887498948482045868343656381177203091798058

# THE LOGISTIC CURVE

Using the formulae below, we get the following graphs:

|  |  |
| --- | --- |
|  |  |
|  |  |
| In Python, Logistic Function, recursively: |  |

import matplotlib.pylab as plt

limit = 4.0; r = 1.0; x =0.01; delta\_r = 0.005

def chaos(x,r,limit):

if r >= limit:

return(0)

else:

n = r\*x\*(1-x)

x = n

r += delta\_r

plt.scatter(r,x, color='black', s=0.1)

return (chaos(x,r,limit))

z = chaos(x,r,limit)

plt.show()

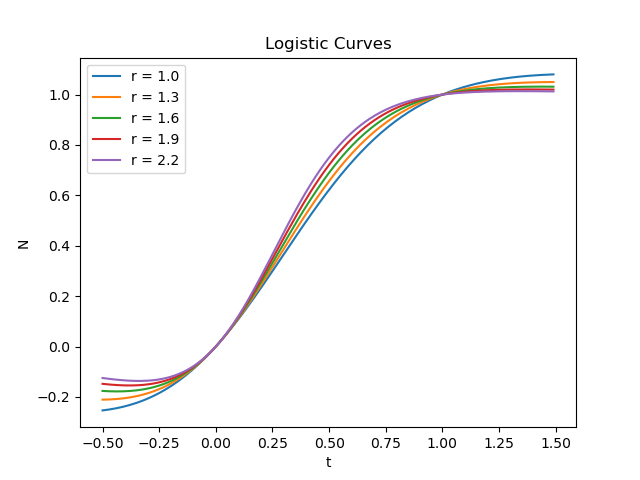
In Python, the Logistic Curve:

def logistic\_curve(r,t):

y = 1 / (1 + ((1 / t) - 1)\*np.exp(-r \* t))

return y

For ranges *r <1.5* in a stable system



*Testing the formula for stability:*

Python function:

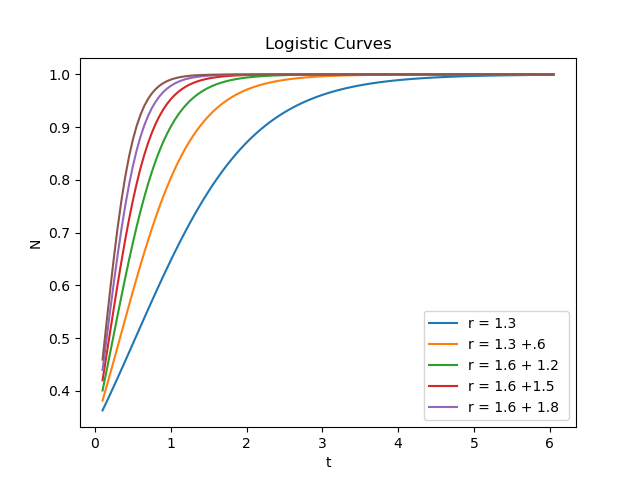
def logistic\_curve(l,k,r,t):

n = l/(l+((k-l)\*(math.exp(-r\*t))))

return n

Where population *N0 = l = 1 million people*

for ranges *2.0 < r < 3.4*



# PENDULUMS & CHAOS

A simple pendulum, consists of a point mass on a massless string as shown in……………the equation of motion for this pendulum is..

to keep the angle *θ* small, in which case *sin θ ≈ θ* For this approximation,

which is simple harmonic motion, with a known solution:

The reason that the small-angle equation is used the above equation is that otherwise we can't solve the equation analytically - numeric solutions what we are looking for.

In *Python:*

theta\_0 = np.pi/180\*10 # 10 degrees in radians

g = 9.8

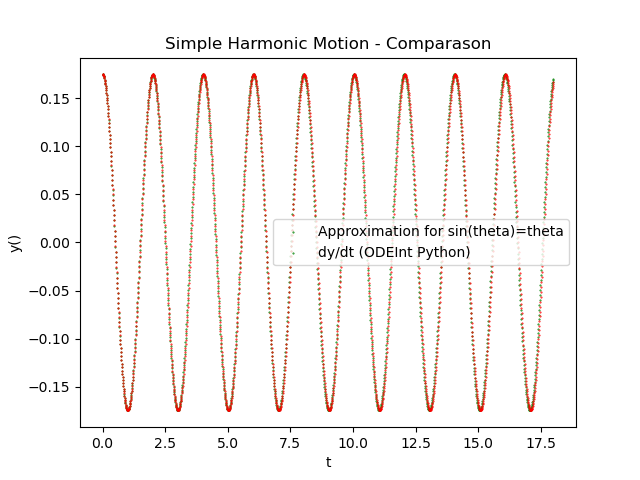
L = 1

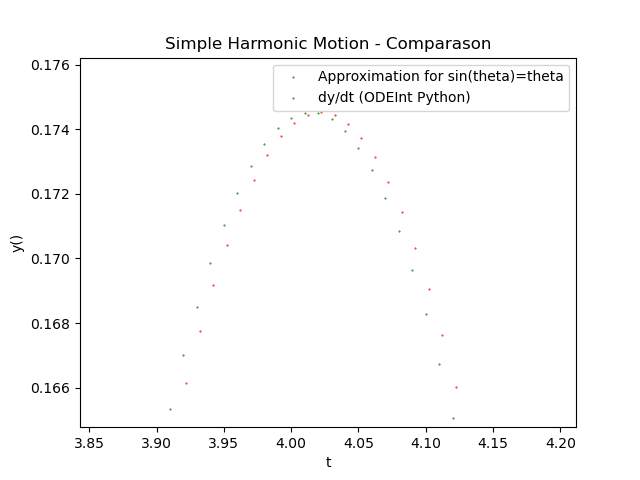
def harmonic(theta):

y = -theta\_0\*np.cos(np.sqrt(g/L)\*theta)

return y

Results:





Angular position vs time for a simple pendulum.

1. The red curve is obtained using *scipy.integrate.odeint()*
2. The green curve is the result of the small-angle approximation.

Comparing Small Approximation to Integration using *ODEInt() in Python:*

import numpy as np; from scipy.integrate import odeint; import matplotlib.pyplot as plt

theta\_0 = np.pi/180\*10 # 10 Degrees;g = 9.8; L = 1

def harmonicApprox(t):

y = theta\_0\*np.cos(np.sqrt(g/L)\*t)

return y

def harmonic(x,t):

y = x[0]

dy = x[1]

xdot = [[], []]

xdot[0] = dy

xdot[1] = -g\*np.sin(y)/L

return xdot

ilimit = 1800; delta\_t = 0.01; t0=0; t=t0

# Iterations for harmonicApprox function

for i in range(ilimit):

plt.scatter(t,harmonicApprox(t), s = 0.2, c = 'green')

t+= delta\_t

# Harmonic Function

t = np.linspace(0,ilimit/100,ilimit)

z2 = odeint(harmonic,[theta\_0,0],t)

plt.scatter(t,z2[:,0], s=0.2, c='red')

## PHASE SPACE

SciPy library has a derivative function also.

It expects a function, though, rather than a list of data points. It should be called with the

function name, the point at which one wants the derivative, and (optionally) the value of dx, the order n of the derivative, optional arguments to the function, and the number of points to use. from By default, dx = 1:0 and the order is the three-point approximation. To get better results, use a smaller value of dx and at least a 5-point approximation:

import numpy as np

from scipy.misc import derivative

theta\_0 = np.pi/180\*10 # 10 Degrees

g = 9.8

L = 1

def harmonic(t):

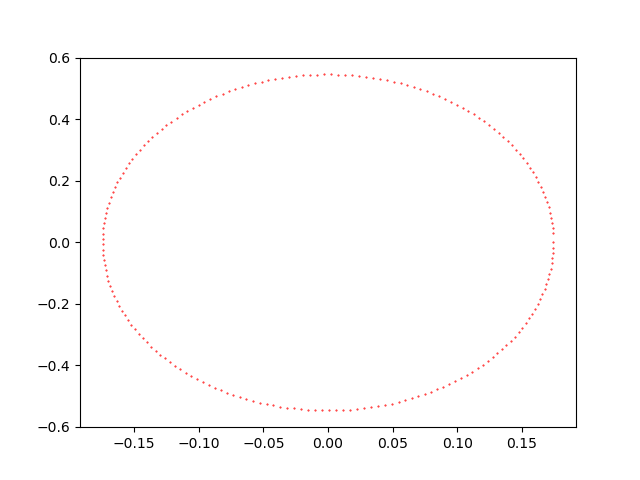
y = theta\_0\*np.cos(np.sqrt(g/L)\*t)

**return y**

**def velocity(t):**

**y = derivative(harmonic, t, dx=0.01, order=5)**

**return y**



To make our pendulum more realistic, let's add a damping term to equation

This damping depends on the angular velocity, which would be consistent with viscous damping or magnetic braking. Damped simple pendulum, with g/L = 1 and damping parameter β = 0:5.

Python:

You are given a system of linear equations as follows:

g/L = 1 and

z =

In Python:

import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

def model(x,t):

y = x[0]

dy=x[1]

K=.05

xdot=[[],[]]

xdot[0] = dy

xdot[1] = -(K\*dy) - np.sin(y)

return xdot

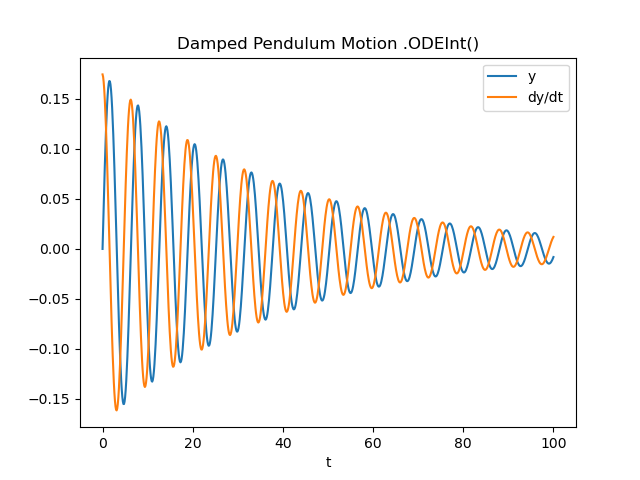
y0= 10\*np.pi/180

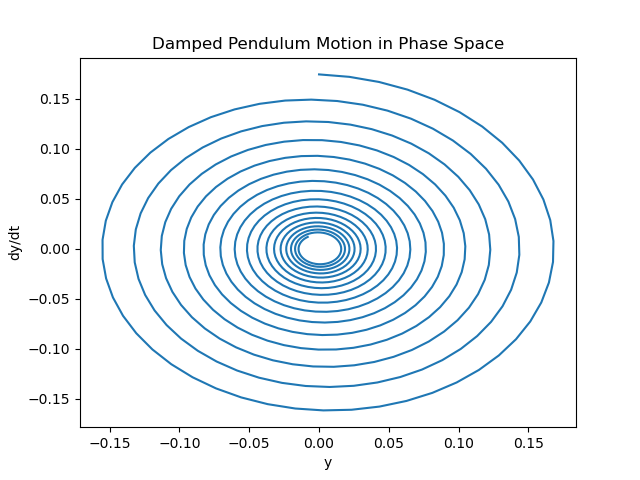
t = np.linspace(0, 50, 200)

z2 = odeint(model, [0,y0], t)

plt.plot(t,z2[:,0])

plt.plot(t,z2[:,1])





This motion is more realistic, but eventually everything just stops, Adding a driving term to keep things going. This equation now represents a pendulum with a time-varying torque with strength A and driving frequency ⱷ.

)

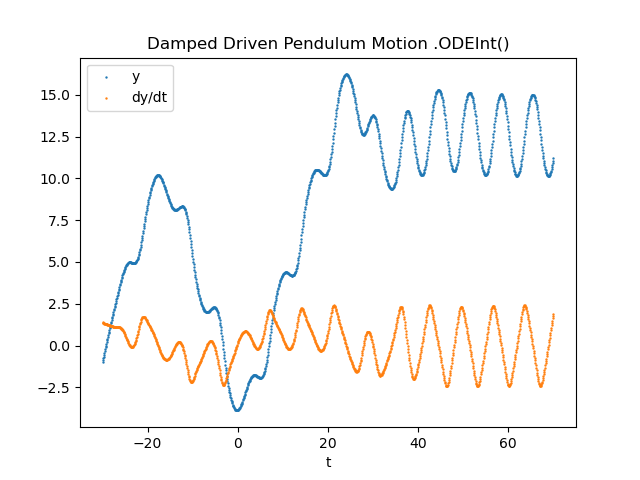
A damped, driven pendulum. *A = 1.5, ⱷ = 0.9,* and *ß = 0.5.*

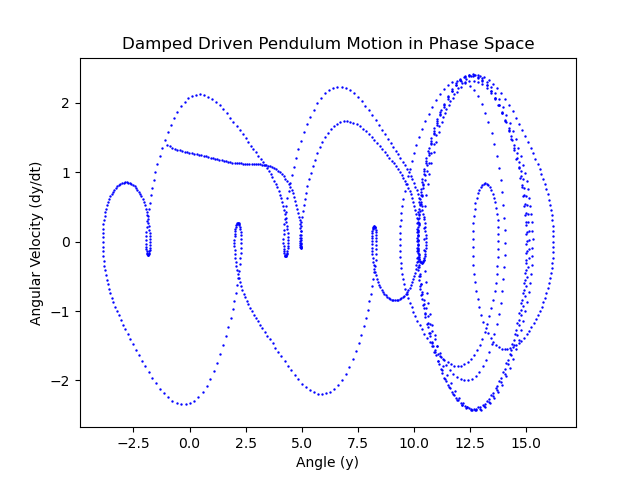
Python:

y0 = 80\*np.pi/180 # start at 80 degrees

xdot[1] = -np.sin(y) - ( 0.5\*dy) + 1.5\*np.cos(0.9\*t)

t = np.linspace(-30, 70,1200)





The motion shown is not random.is not random. It repeats itself, but the period of that repetition may belonger than the age of the universe! The motion is still deterministic: if the pendulum is re-started in the exact same conditions, it will undergo the exact same motion. But if the pendulum is started in even slightly different conditions, the motion will be different although the attractor will be the same. This sensitive dependence on initial conditions is the hallmark of chaotic behaviour.

# MATRICES

find the values of w, x, y, and z:

w + 3x - 5y + 2z = 0

4x - 2y + z = 6

2w - x + 3y - z = 5

w + x + y + z = 10

M and b and solve using the *linalg.solve()* function of *numpy:*

M = array([[1,3,-5,2 ],[0,4,-2,1 ],[2,-1,3,-1],[1,1,1,1]])

b = array ([0,6,5,10])

print(linalg.solve(M, b))

Results:  
[1.2.3.4]

# PYTHON LIBRARIES

## NUMPY

*Numpy* uses the *LINPACK* linear algebra package, which is written in *FORTRAN* and is extremely fast. The fact that arrays are multiplied element-wise by default is a great advantage. It means that if you have large lists of numbers that need to be multiplied together, you can define each array of numbers as a *numpy* array and then just multiply the arrays. Doing the multiplications this way is much faster than using a for loop, and generally easier as well.

Example:

Calculate *sine* of *{π/6, π/5, π/4, π/3, π/2}.*

x = array ([pi/6, pi/5, pi/4, pi/3, pi/2])

print sin(x)

[0.5 0.5878525 0.70710678 0.86 0.8660254]

**arange()** Creates an ‘’Array Range’’. This is just like the range() function built into python, except it returns a numpy array and the step and doesn't have to be an integer.

print arange(0, 1, 0.25)

[0. 0.25 0.50 .75]

**linspace()** Creates a \linearly-spaced" array. This is similar to *arange()* except instead of the step value you tell it the number of points you want. The resulting array will contain both endpoints and evenly spaced values between those points.

print linspace(0,1,5)

[0 0.25 0.5 0.75 1]

**logspace()** Just like *linspace ()* but the values are spaced so that they are evenly distributed on a logarithmic scale. The stop and start values should be given as the powers of 10 that are desired.

print logspace (1,3,4)

[1 10 100 1000]

**zeros()** Produces a *numpy* array led with zeros of the specified type. This is useful for setting up an empty array that your program can then add with calculated values. The type can be *float, int or double* (others) and specify what format should be used for storing values. The *default* is float.

print zeros ( [2 ,3] , int )

[[0 0 0]

[0 0 0]]

**ones()** Just like zeros() but in ones instead

## SCIPY

Where *numpy* provides basic numeric tools such as the ability to do nice thingswith matrices, *Scipy* provides higher-level tools such as numeric integration, differential equation solvers, and so on.

*Scipy* provides the ability to integrate functions (rather than lists of y values) using Gaussian Quadrature.

from scipy.integrate import quad

print quad(sin,0,pi)

(2.0, 2.2204460492503131 e14)

The quad() function takes any function (sine, in this case) and integrates over the range given (0, pi). It returns a tuple containing the value of the integral (2.0) and the uncertainty in that value.

If you need to integrate to infinity, *scipy* provides for that also:

from scipy.integrate import inf

print quad (lambda x : exp(-x) , 0, inf)

(1.0000000000000002, 5.8426067429060041 e-11)

## SECOND ORDER DIFFERENTIAL EQUATIONS

Answer:

(1)

(2)

In Python:

import numpy as np

from scipy.integrate import odeint

import matplotlib.pyplot as plt

def model(x,t):

y = x[0]

dy=x[1]

K=30

xdot=[[],[]]

xdot[0] = dy

xdot[1] = -(0.9 +0.7\*t) \*dy - K \* y

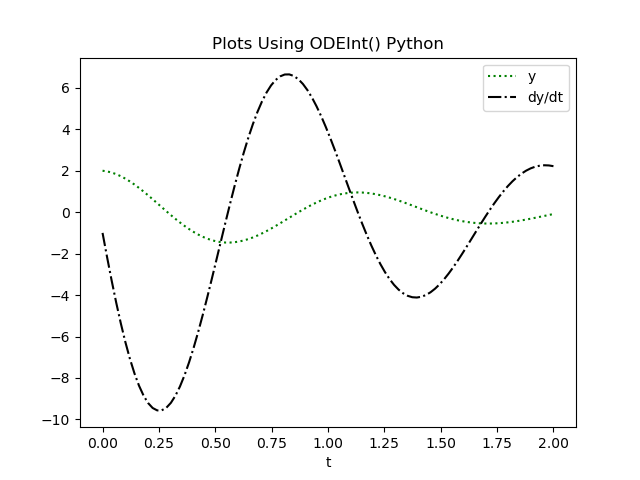
return xdot

t = np.linspace(0,2,100)

z2 = odeint(model,[2,-1],t)

plt.plot(t,z2[:,0], 'g:')

plt.plot(t,z2[:,1], 'k-.')



# MONTECARLO TECHNIQUES

Psuedo-random generators

1. They must be uniformly distributed.
2. They must be uncorrelated.
3. We need a lot of points.

*xn+1 = (axn + b) mod c*

Let *a = 5, b = 3,* and *c = 8.* For a starting *x0 = 1,* this generates the sequence 0, 3, 2, 5, 4, 7, 6, 1. At this point, the sequence repeats itself. So c has to be as large as possible to reduce repition.One set of parameters that was used heavily in the 70's and early 80's is *Randu*: *a = 65539, b = 0,* and *c = 2 31*

*However, Randu* sequence is highly correlated! If you use *Randu* to plot random points in three dimensions, every point falls on one of 15 parallel planes.

Python comes with a pseudo-random number generator called the *Merseinne Twister*

random.random( ) # generates a random float between 0 and 1

random.uniform(a,b) # generates a random float on the range ( a , b ) .

random.choice(list) # returns a random element from the l i s t

random.gauss(mu,sigma) # returns point s with gaussian distribution # centered on mu, with width sigma

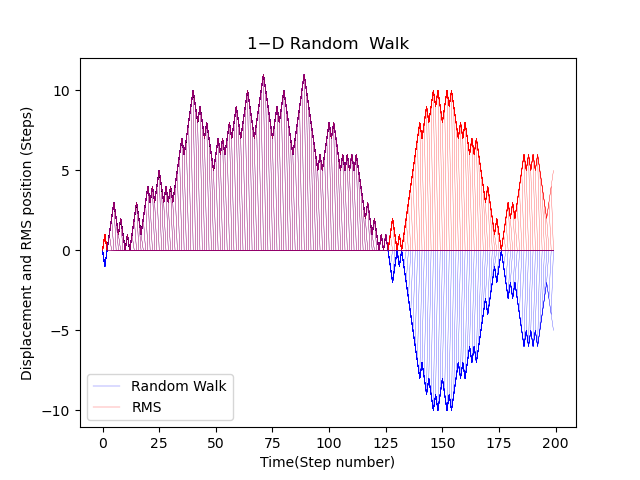
random.randint(a,b) # random integer on range [ a , b ]

# STOCHASTIC METHDODS

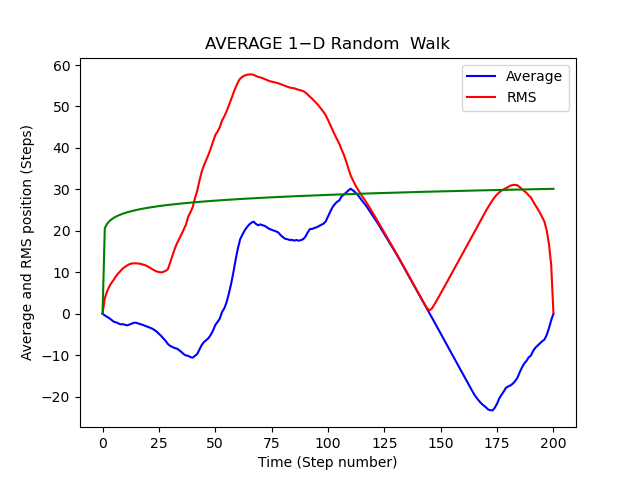
The fundamental idea behind these methods is that large ensembles act in an average way even if individual elements are random. We lose the details about each molecule, and only learn the behaviour of the ensemble.

## RANDOM WALK

We'll start with the simplest method, the random walk in one dimension. Start with a drunken frat boy standing on the sidewalk outside of the Bar at about 2am on a Saturday morning. He can take a step east, or a step west. For the sake of this model, we'll assume that the probability of either direction is 0.5, and the step lengths are all the same. The question we'd like to answer is, how fast, on average, does this random walk move the frat boy away from the Bar?"



This program only tells us one walk, though. We need to know the aver- age behavior of a drunk frat boy. To calculate this average behavior, we take several thousand non-interacting drunk frat boys and run the simulation several thousand times to determine the average RMS displacement.



Power it : y ( t ) = A\*t^B:

A = 20.615567 +/- 4.799836 .

B = 0.071642 +/- 0.051597 .

Average RMS Displacement for a one-dimensional random walk with a uniform stepsize of 1. This graph shows the average for 2000 walks. A least-squares t is also shown for x(t) = AtB, where A = 0:994 0:002

and B = 0:4991 0:0005.

Diffusion & Entropy

Random walk model of diffusion in a 2D environment. Starts with 400 particles in a square grid centred at (100 ,100) . At each step, the program picks each particle and moves it (or not). one integ e r s t e p in the x and y d i r e c t i o n s . I f the move would

ta k e the p a r t i c l e beyond the boundary of space (200 x200 ) , then

the p a r t i c l e bounces o f f the wa l l and moves the ot he r d i r e c t i o n .

The program p l o t s the p o s i t i o n s of a l l p a r t i c l e s a f t e r each s t e p .

# PARTIAL DIFFERENTIAL EQUATIONS

## INTEREST RATES

At a growth rate of 5% per annum, assume amount, M as a function of time t:

## THE HEAT EQUATION:

Implies rate of change of Temperature with time is proportion to change in location x, both have, t nad location as inputs.

The equation must satisfy three constraints:

1. The PDE
2. Boundary conditions when t = 0 and x = 0 ; x = 0 and x = L, total length of rod
3. Initial conditions

We first breakdown the equation into it’s wave functions to make things easier because, sine ways have clean derivatives

Lets assume the solution to the equation is a sine wave in the T-x plane

if the derivative of a function is itself then

However, at boundaries :

So let as use the cosine instead

But we adjust frequency with constant ⱷ

If *f(x) = cos(ⱷx)*

*Then f’(x) = - ⱷsin(ⱷx)*

*And f’’(x) = ⱷ2cos(ⱷx)*

*Therefore T = cos(nπ/L.x) where ⱷ= nπ/L*

*When n= 0,1,2 etc… we get the correct curve*

*In Python:*

import numpy as np

from matplotlib import pyplot as plt

from mpl\_toolkits import mplot3d

def T(x,t):

alfa = 750; L = 1000

return np.cos(3 \* x \* (np.pi / L)) \* np.exp(-alfa \* t \* (3 \* np.pi / L) \*\* 2)

x = np.linspace(0, 1000, 50)

t = np.linspace(0, 100, 50)

X, Y = np.meshgrid(x, t)

Z = T(X, Y)

fig = plt.figure()

ax = plt.axes(projection="3d")

ax.plot\_wireframe(X, Y, Z, color='green')

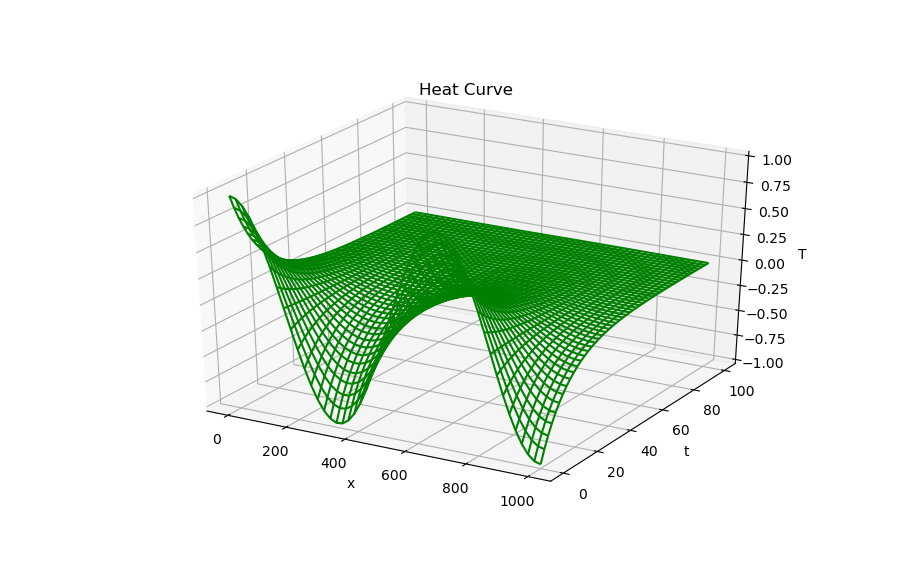
plt.title('Heat Curve')

ax.set\_xlabel('x')

ax.set\_ylabel('t')

ax.set\_zlabel('T')

plt.show()



## SCHRODINGER'S EQUATION

Left side of the equation describes the quantum state of a system as it change over time *t.* Which is linked to the total energy of the system

is a constant, *i* is an imaginary number

The right side is the Hamilton Operator ie Kinetic + Potential Energy

For a Helium atom,

Kinetic energy of both elections + electrostatic attraction between both electrons and the nucleus… etc

Another version is:

Energy level of the quantum system x Wave-function = Kinetic Energy + Potential Energy

Solutions for the wave-function:

Probability Distribution = ψ(x)2

Energy Level:

Where m = mass

Note: As centre of L is 0, Boundary conditions for –L/2 and L/2, ψ(x)2 and ψ(x)2 =0

In Python:

import numpy as np

from matplotlib import pyplot as plt

import seaborn as sns

sns.set()

L=25

def wave(x,n):

return np.sqrt(2/L)\*np.cos(np.pi\*n\*x/L)

x = np.linspace(-L, L, L\*8)

plt.xlim(-L/2, L/2)

plt.plot(x,wave(x,1))

plt.plot(x,wave(x,3))

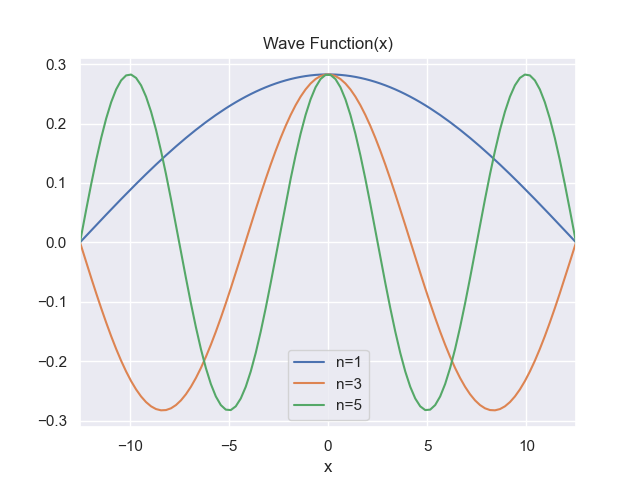
plt.plot(x,wave(x,5))

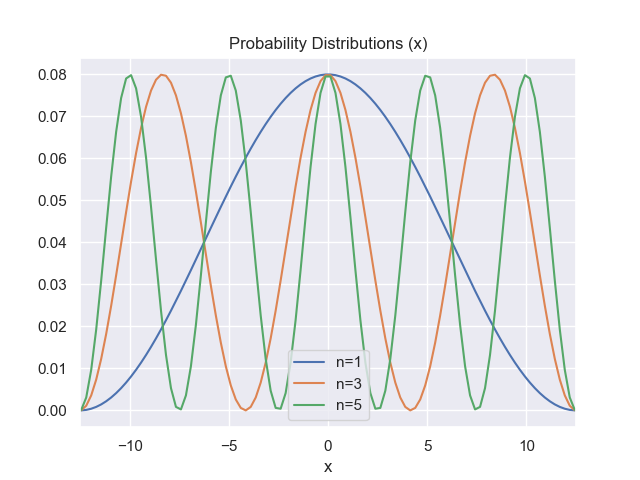
plt.title('Wave Function(x)')

plt.xlabel('x')

plt.legend(['n=1', 'n=3', 'n=5'])

plt.show()





## THE WAVE EQUATION

Describes the behaviour of a wave in time *t* and space *x* with the displacement of oscillation *u*

import numpy as np

from matplotlib import pyplot as plt

x = np.linspace(0, 5, 50)

plt.plot(x,np.sin(x))

plt.plot(x,-np.cos(x))

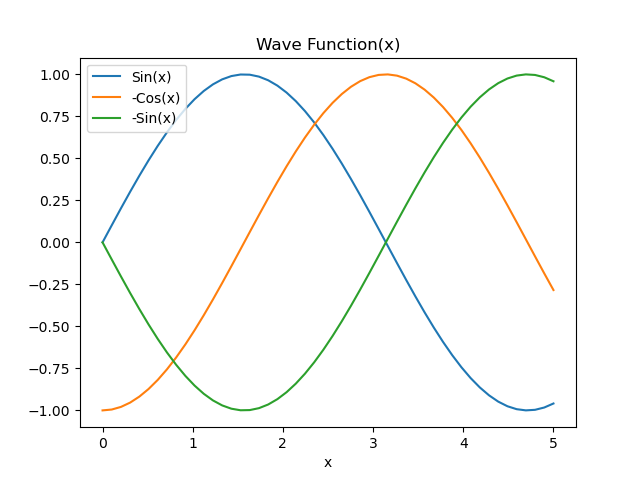
plt.plot(x,-np.sin(x))

plt.title('Wave Function(x)')

plt.xlabel('x')

plt.legend(['Sin(x)', '-Cos(x)', '-Sin(x)'])

plt.show()



*u* therefore depends on values *x* and *t* so

*w* = angular frequency

*k* = wave number

# TAYLOR & MaCLAURiN SERIES applications

Series apply numerical methods for

1. Approximating functions:
   1. Taylor’s series approximate a function with a polynomial
   2. Most accurate around point *a*
   3. Become a Maclaurin Series when *a = 0*
2. Solving differential equations with difficult or impossible to derive solutions

## appoximating FUNCTIONS

McLaren Series for the function:

In Python:

import numpy as np

import matplotlib.pylab as plt

import seaborn as sns

sns.set()

x = np.linspace(-2, 2, 100)

y = np.exp(-x\*\*2)

y1 = 1 - x\*\*2

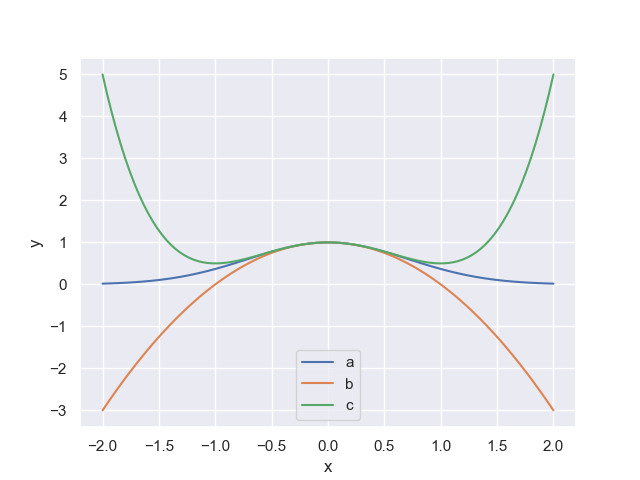
y2 = y1 + (0.5\*x\*\*4)

plt.figure()

plt.plot(x, y); plt.plot(x, y1); plt.plot(x, y2)

plt.xlabel('x'); plt.ylabel('y')

plt.legend(['a','b', 'c'])



For values +0.5 < x <0.5

McLaren Series for the function:

In Python:

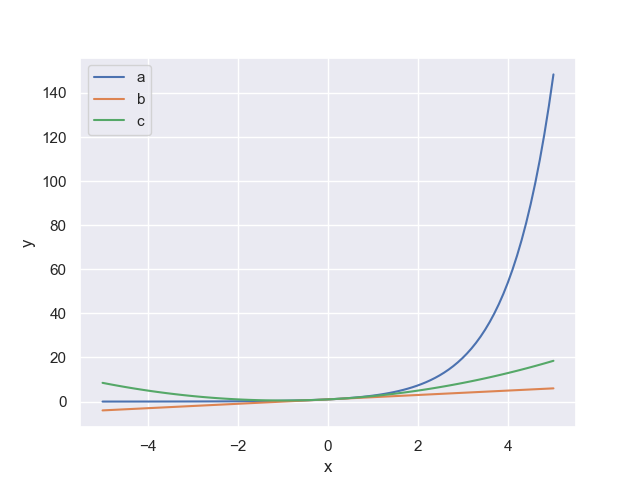
x = np.linspace(-5, 5, 100)

y = np.exp(x)

y1 = 1 + x

y2 = y1 + (x\*\*2)/2

y3 = y2 + (x\*\*3)/6



McLaren Series for the function:

+…

McLaren Series for the function:

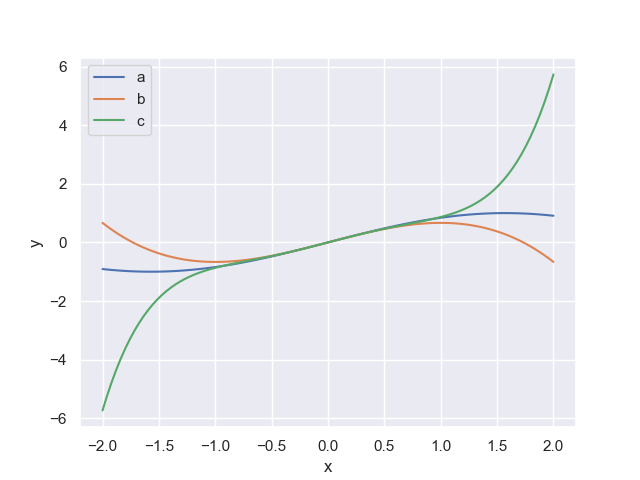
In Python:

x = np.linspace(-2, 2, 100)

y = np.sin(x)

y1 = x - (x\*\*3)/3

y2 = y1 + (x\*\*5)/5



When x is close to 0. i.e*. x < 10 degrees* or *π/18 radians.*

*sin(x) ≈ x*

## solving DIFFERENTIAL EQUATIONS

Differential equations describe change rather than absolute values

### Pendulum Motion

This equation is difficult to solve:

for small angles *sin(x) ≈ x*

Solution to this equation is therefore:

### potential energy of an object

Even still objects have intrinsic energy,

where *m0*  is the objects mass when not moving.

The total energy is:

Which is an approximation

Actual Energy is:

We can treat this as a Maclaurin Series:

We can ignore these elements since as v/c ≈ 0 at low speeds

# FOURIER SERIES

Fourier analysis is a method of defining periodic waveform s in terms of trigonometric function s. Fourier analysis is used in electronics, acoustics, and communications. Many waveforms consist of energy at a fundamental frequency and also at harmonic frequencies (multiples of the fundamental).

A Fourier series is an expansion of a periodic function f(x) in terms of an infinite sum of sines and cosines. The computation and study of Fourier series is known as harmonic analysis and is extremely useful as a way to break up an arbitrary periodic function into a set of simple terms that can be plugged in, solved individually, and then recombined to obtain the solution to the original problem or an approximation to it to whatever accuracy is desired or practical

The Fourier series is named in honour of Jean-Baptiste Joseph Fourier (1768–1830), who introduced the series for the purpose of solving the heat equation in a metal plate. The heat equation is a partial differential equation. Prior to Fourier's work, no solution to the heat equation was known in the general case, although particular solutions were known if the heat source behaved in a simple way, in particular, if the heat source was a sine or cosine wave. These simple solutions are now sometimes called eigen solutions. Fourier's idea was to model a complicated heat source as a superposition (or linear combination) of simple sine and cosine waves, and to write the solution as a superposition of the corresponding eigen solutions. This superposition or linear combination is called the Fourier series.

Although the original motivation was to solve the heat equation, it later became obvious that the same techniques could be applied to a wide array of mathematical and physical problems, and especially those involving linear differential equations with constant coefficients, for which the eigensolutions are sinusoids.

The Fourier series has many such applications in electrical engineering, vibration analysis, acoustics, optics, signal processing, image processing, quantum mechanics, econometrics, etc.

The Fourier Pylynomial is expressed by the function:

Imagine two metal rods of temp -1̊ Cand 1̊ c respectively touching each other at one end, each 0.5m long.

The resulting temperature distribution would be *discontinuous* **step function** as follows:

How can we express this distribution or any other distribution as a sign wave?

To do this, we get the sum of sine waves restricted by boundary conditions as a set of cosine waves as a multiple of a base frequency say , *π/Lx* where L is the length of the two rods.

The heat distribution can be expressed by the infinite series:

With frequency ω at time *t*

Let

If the end points remain fixed then we use the infinite series:

For the step function we use:

Infinite Fourier series for the step function is:

In Python:

import numpy as np

import matplotlib.pylab as plt

import seaborn as sns

sns.set()

x = np.linspace(-1, 1, 500)

y = (4\*(np.sin(np.pi\*x) + np.sin(3\*np.pi\*x)/3 + np.sin(5\*np.pi\*x)/5 + np.sin(7\*np.pi\*x)/7 + np.sin(9\*np.pi\*x)/9))/np.pi

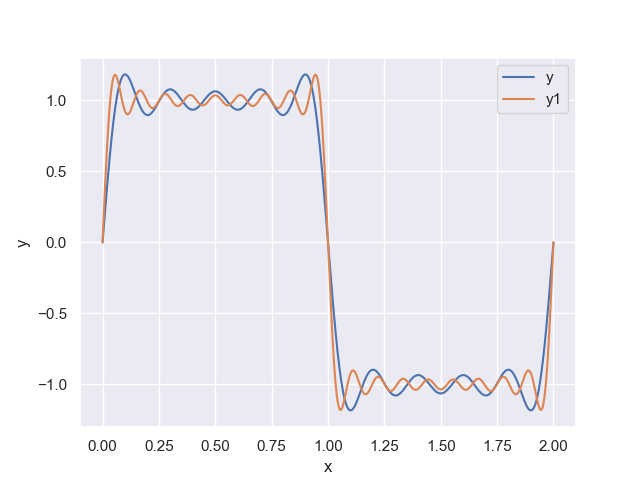
y1 = y + 4\*(np.sin(11\*np.pi\*x)/11 + np.sin(13\*np.pi\*x)/13 + np.sin(15\*np.pi\*x)/15 + np.sin(17\*np.pi\*x)/17)/np.pi

plt.plot(x, y);plt.plot(x, y1)

plt.xlabel('x'); plt.ylabel('y')

plt.legend(['y','y1'])

plt.show()



Conclusion:

When *x* < 1, *f(x)* ≈ 1  
When *x* = 0, *f(x)* = 0

When *x* >1, *f(x)* ≈ -1

Ie the sine wave approximates the step function.

# FOURIER analysis

The Fourier Transform is an important image processing tool which is used to decompose an image into its sine and cosine components.

Advantages. The main advantage of Fourier analysis is that very little information is lost from the signal during the transformation. The Fourier transform maintains information on amplitude, harmonics, and phase and uses all parts of the waveform to translate the signal into the frequency domain

The Fourier series is used to represent a periodic function by a discrete sum of complex exponentials, while the Fourier transform is then used to represent a general, nonperiodic function by a continuous superposition or integral of complex exponentials.

## cONTINOUS FOURIER TRANSFORM

From Euler’s Constant”:

Consider the counter-clockwise, circular, complex function with respect to time t:

-clockwise, and add frequency (cycles per secont)

: is the *analysing function*

Add an oscillating function *g(t)*

**The Continuous Fourier Transform is:**

but, =

Function f(t) is said to be in the ***time*** domain while is said to be in the ***frequency*** domain

For example, consider the signal function:

The Python *Numpy* library includes a Fast-Fourier Transformation (FFT) algorithm to calculate the function.

In Python:

import numpy as np; import matplotlib.pylab as plt

import seaborn as sns; from numpy.fft import fft, fftfreq, ifft

sns.set()

n =1000 # number of points

Lx = 100 # length in metres

omega = 2.0\*np.pi/Lx # angular frequency / time in seconds

# Create the composite signal y

x = np.linspace(0, Lx, n)

y1 = 1.0\*np.cos(5.0\*omega\*x)

y2 = 2.0\*np.sin(10.0\*omega\*x)

y3 = 0.5\*np.sin(20.0\*omega\*x)

y = y1 + y2 + y3

freq = fftfreq(n) # Creates all frequencies

mask = freq > 0 # ignore 1/2 of values they are complex conjuctures of the others

fft\_val = fft(y) # fft values

fft\_theo = 2.0\*np.abs(fft\_val/n) # true theoretical fft

plt.figure(1)

plt.title('Original Signal')

plt.plot(x, y, color = 'red', label = 'Original')

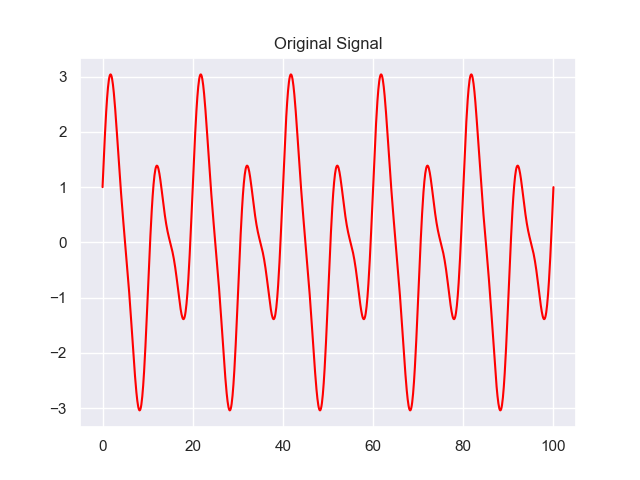
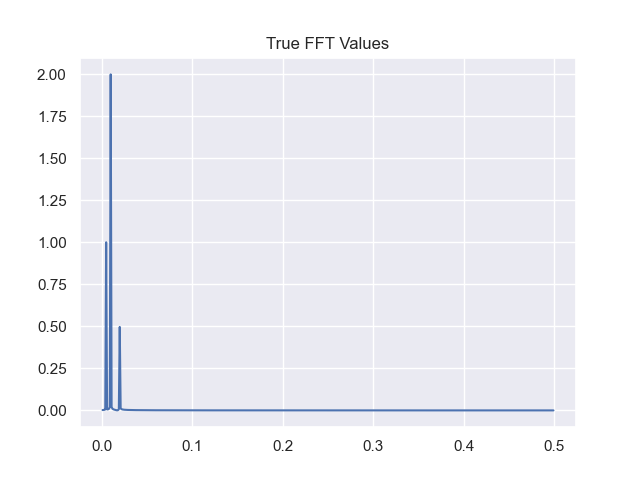
plt.figure(2)

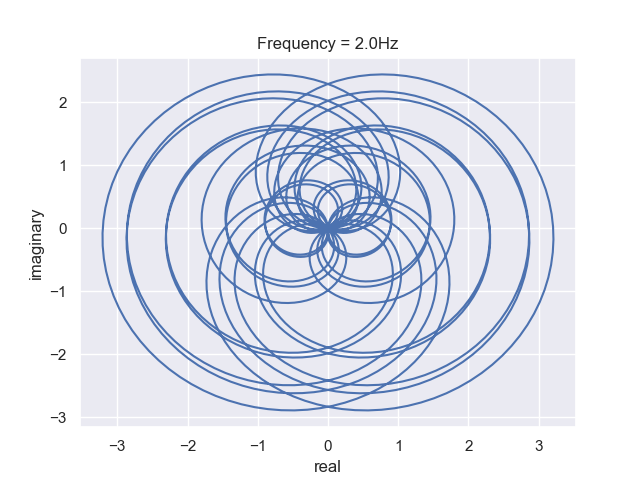
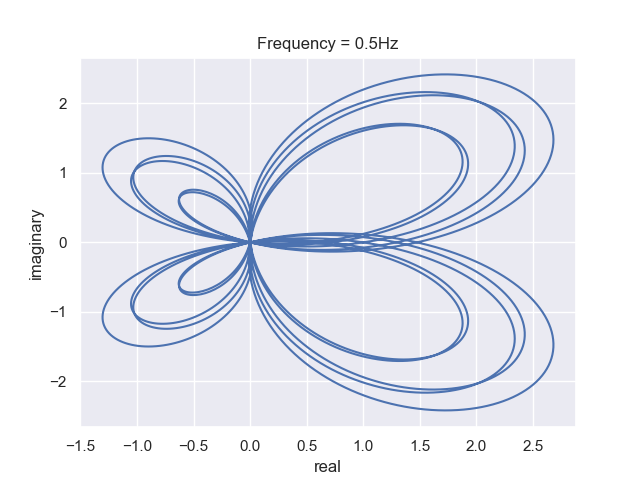
plt.title('True FFT Values')

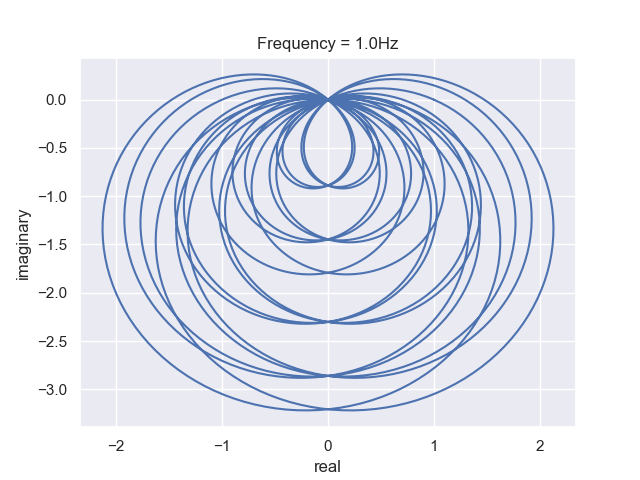
plt.plot(freq[mask], fft\_theo[mask], label = 'FFT Values')

plt.show()

Results:





Continuous Fourier Transformation is (CFT) defined by the function :

Discreet Fourier Transformation is described by the function: what is the k

If I graph the DFT for the above function in Python:

, let N = 4

Add a complex constant *cn*

+…

*c0* becomes the centre of mass of the function

In python:

# WAVELET ANALYSIS

The function Ψ *(t)* is a wavelet if:

1. *Ψ (t)* is decaying sufficiently fast
2. = 0 i.e. net area under the graph = zero

The wavelet transform described by:

Where *Ψ\_ab (t)* is the analysis function , *x(t)* is the signal and *a* is the scale & ***b*** is the translation