Linn Boldt-Christmas

Scientific Computing Assignment 3

Candidate no. 117418

Due in December 7th, 2015

University of Sussex

Physics & Astronomy

QUESTION 1

The driven pendulum:

A simple pendulum can be driven by, for example, exerting a small oscillating force horizontally on the mass. Then the equation of motion for the pendulum becomes:

$$\frac{d^2\theta}{dt^2} = -\frac{g}{l}\sin\theta + C\cos\theta\sin\Omega t$$

Where C and Ω are constants. In calculation below set l = 10 cm, C = 2 s^{-2} and Ω = s^{-1}.

(a) Make this equation dimensionless by setting $\omega^2 = g/l$, $\beta = \Omega/\omega$ and $\gamma = C/\omega^2$, and changing the variable to $x = \omega t$. Then, convert this equation to a system of 2 first order differential equations. Show your work.

The first section of this question, regarding the task of making the equation dimensionless, is purely mathematical and does not need to be done in code. Instead, it will be done through commenting within the code for the purpose of ensuring that the coded script will be all-inclusive and can be understood without this write-up if so required.

```
# part a

# say T = theta, O = capital omega, w = lower case omega, B =
beta, j = gamma
# d^2T/dt^2 = -(g/L)*sin(T)+C*cos(T)sin(Ot)
# = -w^2 * sin(T) + j*w*cos(T)sin(Bwt)
# = -w^2 * sin(T) + j*w*cos(T)sin(Bx)
# d^2T/dx^2 = -sin(T) + j*cos(T)sin(Bx)
```

This equation is dimensionless, and it will be used in the next part of this question.

(b) Use the 4th order explicit Runge-Kutta method to solve this non-dimensional equation for θ as a function of time and make a plot of θ as a function of time from t = 0 to t = 40s. Start the pendulum at rest with θ = 0 and dθ/dt = 0.

First, we need to import the necessary modules that we will be using throughout the question. Next, it is important for us to define the constants. Note that the constants that are labelled with a 1 after them are labelled as such because a later step of the question requires us to use a different value for the same term.

```
import math as m

from run_kut4 import *

from printSoln import *

import numpy as np

from pylab import *
```

```
g = 9.81#m/s^2

L = 0.10#cm

C = 2.0#s^-2

O1 = 5.0#s^-1

w = m.sqrt(g/L)

j = C/(w**2)

B1 = O1/w
```

Now, we want to define our function (again labelled with a 1 as we are going to repeat it for a secon value later). We do this using the def/return function, and first creating an empty array. Then, we set our 0th order function to be 1st order θ (labelled T), and the 1st order function to equal the dimensionless equation found in part a.

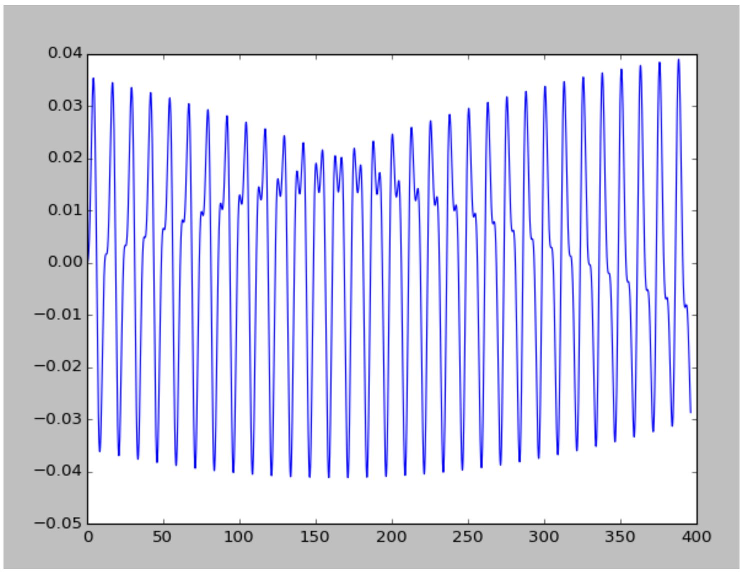
```
def f1(x,T):

    f1 = np.array([0.0,0.0])

    f1[0] = T[1]

    f1[1] = -m.sin(T[0])+j*m.cos(T[0])*m.sin(B1*x)

    return f1
```

Next, we want our initial conditions (as given to us in the question). These, combined with our defined function and constants, we use the Runge-Kutta method to do our integration. The printSoln command is included but commented out for the sake of simplicity.

```
# initial conditions

initial = np.array([0.0,0.0])

tStop = 40*w

h = 0.1

freq = 1

tInitial=0.0


X1,Y1 = integrate(f1,tInitial,initial,tStop,h)

#printSoln(X1,Y1,freq)


figure(1)

plot(X1,Y1[:,0]) # first term of array
```

```
show()
```

Running this code, we get:



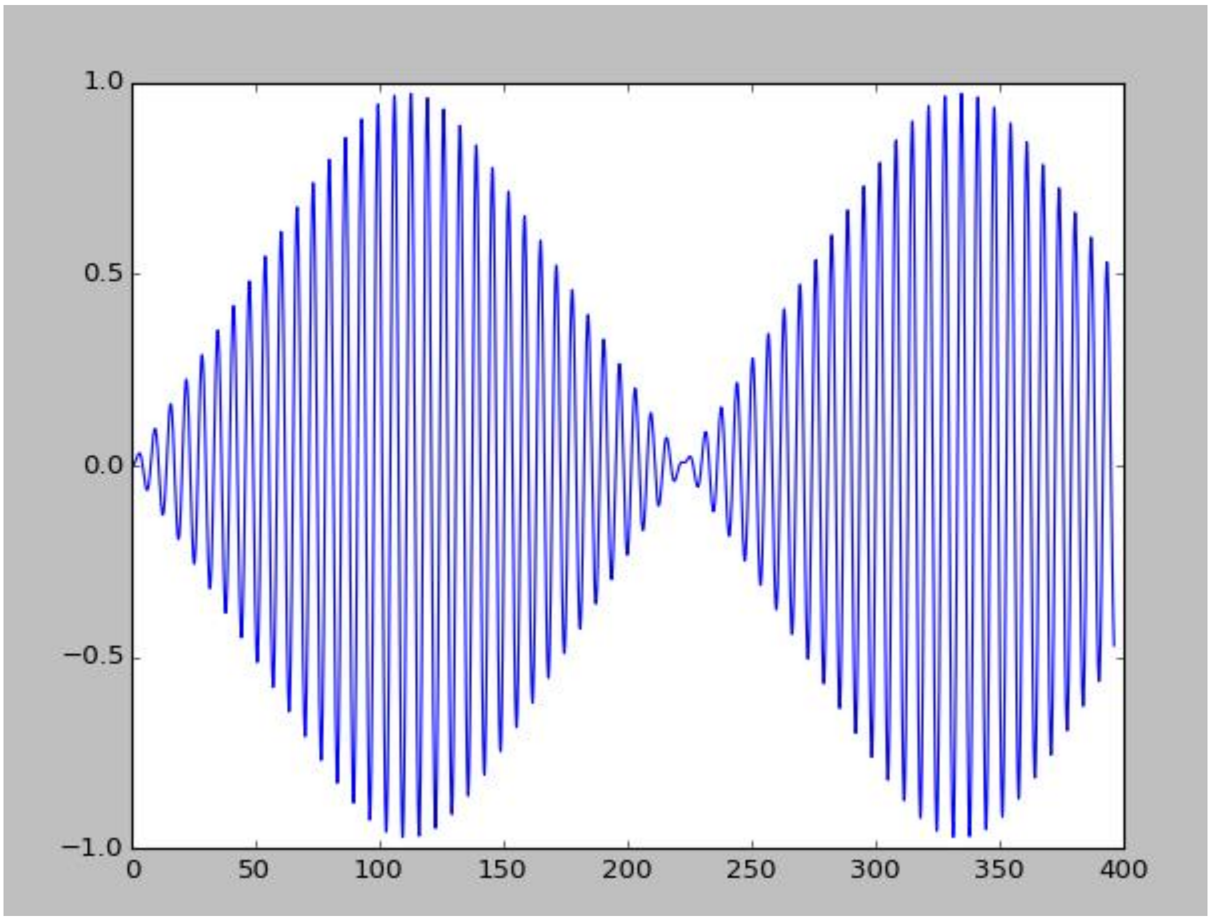
(c) *Now change the value of Ω, while keeping C the same, to find a value for which the pendulum resonates with the driving force and swings widely from side to side. Make a plot for this case also.*

The code behind this part is very much the same as that for part a. Since we are comparing the same maths and the same plot but only changing Ω, it is intuitive that the code will be the same. In this case, Ω = 9.81, i.e. the same as gravity. Resonance occurs when the system oscillates at its natural frequency, and as the system is working under gravity, this will be the natural frequency. Using the same plot and code but changing Ω to this, we get the code:

```
#part c


O2 = 9.81

B2 = O2/w


def f2(x,T):

    f2 = np.array([0.0,0.0])

    f2[0] = T[1]

    f2[1] = -m.sin(T[0])+j*m.cos(T[0])*m.sin(B2*x)

    return f2
```

```
X2,Y2 = integrate(f2,tInitial,initial,tStop,h)
#printSoln(X2,Y2,freq)


figure(2)
plot(X2,Y2[:,0]) # first term of array
show()
```

Running this code, we get the plot:



Question 2

Chaotic system: The equations

$$\frac{du}{dt} = -a(u - v)$$

$$\frac{dv}{dt} = cu - v - uw$$

$$\frac{dw}{dt} = -bw + uv$$

Known as the Lorenz equations, are encountered in the theory of fluid dynamics (atmospheric convection) and other areas of physics – lasers, electric circuits and chemical reactions. Letting a = *5.0,* b = 0.9, *and* c = 8.2, *use* scipy.integrate.odeint *to solve these equations from* t = 0 *to* 10 *with the initial conditions* u(0) = 0, v(0)=1.0, w(0)=2.0 *and plot* u(t). *Repeat the solution with* c = 8.3 *and same* a *and* b *as above. What conclusions can you draw from the results? Again for* c = 8.3 *plot the spatial trajectory* u *vs* v. *The result you find is called Lorentz attractor and is related to the discovery of chaotic systems.*

First, we have to define a function that gathers all of our chaotic system equations together. We do this by doing a "def" and "return" function, including an empty array at the start. After importing the right modules and definining our constants, we then can create this fuction:

```
import scipy

from scipy import integrate

import numpy as np

import pylab as pl


a = 5.0

b = 0.9

c1 = 8.2 #calling this c1 because we will use a different c
value later and call that c2


def F(y,t):

    F=np.zeros(3)

    F[0]=-a*(y[0]-y[1])

    F[1]=c1*y[0]-y[1]-y[0]*y[2]

    F[2]=-b*y[2]+y[0]*y[1]

    return F
```

Then we tell Python what we want our integration interval (range) be, i.e. where we want our intigration to start and stop. These are usually donated by a and b in maths, so we name them x_a and x_b. We also create an array of the initial conditions given to us in the question.

```
x_a = 0.0 # beginning of integration

x_b = 100.0 # end ofintegration

y_initial = np.array([0.0,1.0,2.0])
```

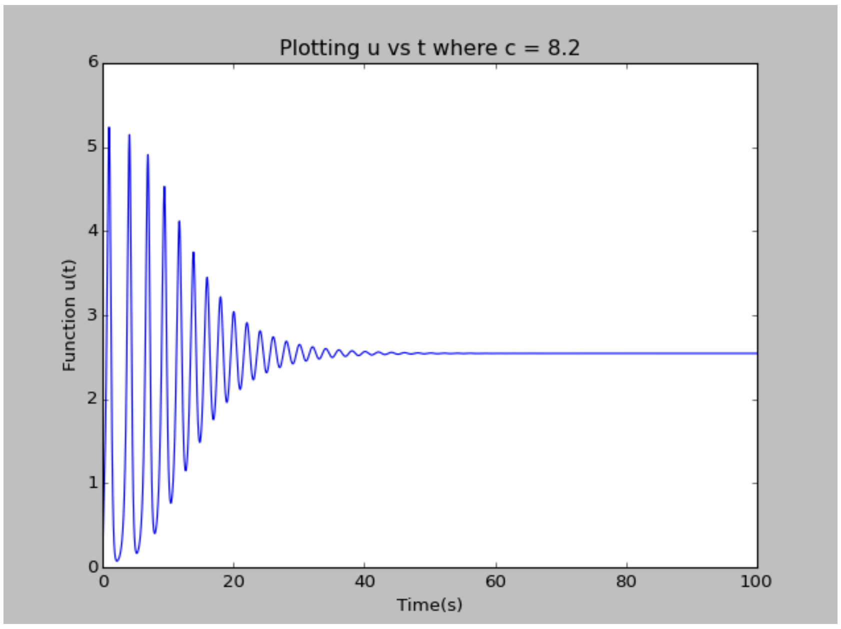
These are going to be necessary for our scipy.integrate.odeint command that we can define next. We use the np.linspace command to give our range of x, and then use that along with our function and initial conditions to finally create our integral.

```
x=np.linspace(0.0,x_b,50000)
I_1=integrate.odeint(F,y_initial,x)
```

Next, we take the first term of our array and plot this with our x linspace from the code above. This will give us the u(t) plot that we are looking for.

```
pl.figure(1)
pl.plot(x,Y_1)
pl.title('Plotting u vs t where c = 8.2')
pl.xlabel('Time(s)')
pl.ylabel('Function u(t)')
pl.show(1)
```

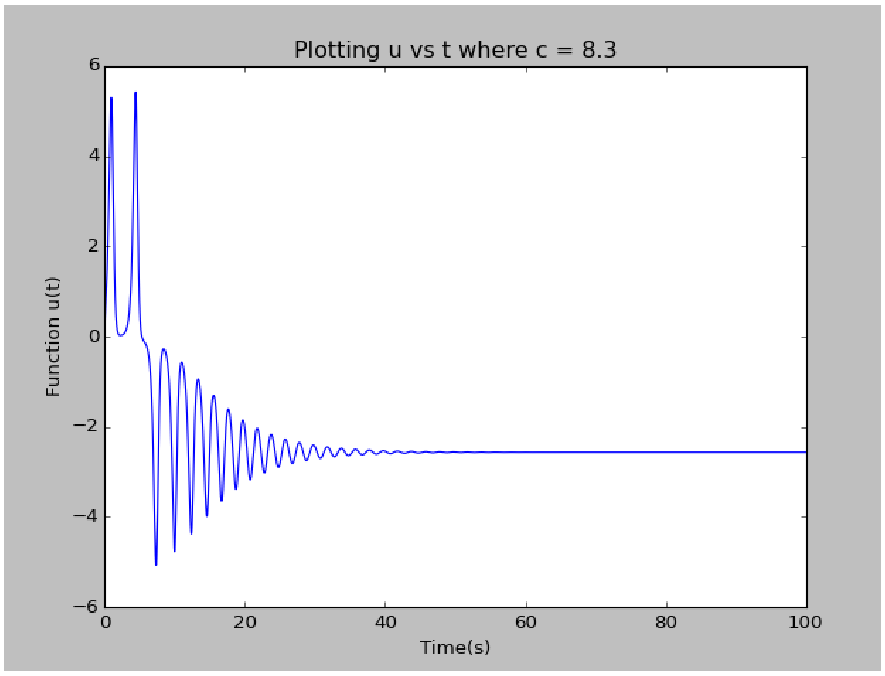
Running this code, we get this plot:



Now let’s try changing our c value to 8.3 instead of 8.2. This is a fairly small change, so in a stable system, this should not make a particularly large difference. For this part, the code and logic behind it is more or less the same, as all we want to do is change this one constant and see how that affects the system. In order to do that, everything else should be kept the same (so that we know that any differences that may or may not arise come from changing c and not from anything else).

```
c2 = 8.3
def G(y,t):
    G=np.zeros(3)
    G[0]=-a*(y[0]-y[1])
    G[1]=c2*y[0]-y[1]-y[0]*y[2]
    G[2]=-b*y[2]+y[0]*y[1]
    return G

I_2=integrate.odeint(G,y_initial,x)

Y_2=I_2[:,0] # using first term of array

pl.figure(2)
pl.plot(x,Y_2)
pl.title('Plotting u vs t where c = 8.3')
pl.xlabel('Time(s)')
pl.ylabel('Function u(t)')
pl.show(2)
```

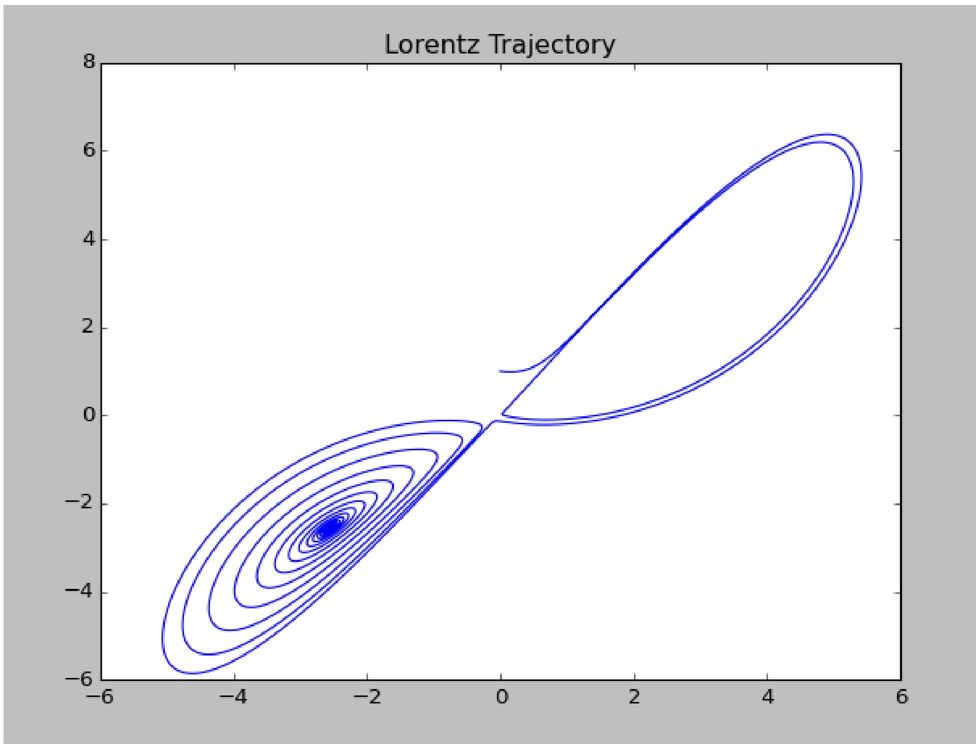
Running this code, we get the following plot:



Clearly, this small change in c made a significant shift in the plot for our system. This indicates that our system is chaotic, meaning it is not going to remain stable under changes, minute or not. If the system is indeed chaotic, the Lorentz attractor of it should resemble a butterfly shape (like a skewed figure 8). We therefore need to plot it in order to confirm this.We do this by taking the next term (term 1) in the array from our second integral, and plotting that against the first term (term 0).

```
Y_3=I_2[:,1]

pl.figure(3)

pl.plot(Y_2,Y_3)

pl.title('Lorentz Trajectory')

pl.show(3)
```

Running this code, we see our Lorentz attractor looking like this:



As suspected, our Lorentz trajectory is butterfly shaped and so our system is chaotic. Since our first two plots deviated so much from such a small change in one of the constants, this is not surprising and the code works as anticipated.

QUESTION 3

Fourier analysis and smoothing:

The function f(t) *represents a square wave with amplitude 1 and frequency 1 Hz.*

f(t) = +1 if 2t is even

f(t) = -1 if 2t is odd

where 2t is rounded down to the next lowest intger. Let us attempt to smooth this funcion using a Fourier transform.

(a) *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 (ie for 0 < x < 1).*

A square wave is easily identifiable by its very “sudden” spikes in the wave form; unlike the sine and cosine waves that have smooth ascents and descents, the square wave is very ‘binary’ in the sense that it suddenly goes from a maximum to a minimum without a slow curve to connect them.

So, the first thing to find out is: where does this happen? There is no code required to work that out, as it is quite intuitive, considering that our 2t terms are rounded down to the next lowest integer. So, as soon as $1 \leq 2t < 2$, 2t will be rounded down to 1 an be odd. As soon as $0 < 2t < 1$, it will be even (considering 0 to be even). 2t = 1 when t = 0.5, so, that’s where that cross-over will happen. This means that exactly half of our terms will be = 1 and the second half will = -1.

For our N array, we therefore need to create an array with the first 500 terms equal to positive one, and the next 500 terms equal to negative one. We do this by creating two separate arrays, one with 500 positive ones and one with 500 negative ones and combining them with the concatenate command from numpy.

```
import numpy as np

from numpy import fft

import matplotlib.pyplot as plt

 # part a


A = np.ones(500) # first 500 ones are even, so f(t)=1

B = np.array(-A) # next 500 ones are odd, so f(t)=-1


N = np.concatenate((A,B)) # combines A and B to describe the
function
```

(b) *Calculate the coefficients of the discrete Fourier transform of this function using the function* rfft *from* numpy.fft *which produces an array of* (1/2)*N + 1 *complex numbers.*

Here, it is a very straightforward case of appropriately importing rfft from numpy.fft (done in the code attached in part a) and using this on our N array.

```
# part b

DFT = fft.rfft(N)

print '---DIRECT FOURIER TRANSFORM ---'

print DFT
```

Running this code, we get a very long transcript of our Direct Fourier Transform. For the sake of being concise, the full transcript will not be featured here but the first few terms look like this:

```
---DIRECT FOURIER TRANSFORM ---

[  0.00000000e+00 +0.00000000e+00j   2.00000000e+00 -
6.36617678e+02j

   3.04649581e-14 +7.78156258e-14j   2.00000000e+00 -
2.12200308e+02j

   7.24645476e-14 -1.88149777e-13j   2.00000000e+00 -
1.27313482e+02j

  -3.31761749e-14 -8.73042333e-14j   2.00000000e+00 -
9.09310205e+01j
```

(c) *Now set all but the first five, ten or 50 Fourier coefficients to zero (make separate copies of the Fourier coefficients array for this purpose).*

We need to now take our N array now and change certain terms of the array. We can do this by using the "for i in range" command for each DFT, and changing the range for that of the first five, ten or 50.

```
DFT_5 = fft.rfft(N)

for i in range (5,501):

    DFT_5[i]=0


DFT_10 = fft.rfft(N)

for i in range (10,501):

    DFT_10[i]=0
```

```
DFT_50 = fft.rfft(N)

for i in range (50,501):

    DFT_50[i]=0
```

We needed to redefine the DFT function each time with its own unique name because if not, Python gets confused by these conflicting statements. In the first one, DFT_5, we made any term that lay in the range 5-501 equal to zero, as this would leave the first five (i.e. 0,1,2,3,4, which is why our range goes up to 501) to be non-zero. The same logic is applied to DFT_10 and DFT_50, leaving the first 10 and 50 terms respectively to be non-zero.

Printing this gives very long transcripts, so for that reason, the printed/running code for this part is not included here.

(d) *Calculate the inverse Fourier transform of the resulting arrays, zeroes and all, using the function* irfft *to recover the smoothed signal.*

Now we take the inverse Fourier transform of all of our DFTs. We will then use these in the next part in order to see what happens when the DFT deviates more and more from the N array, and how the approximation gets further and further away from the true value accordingly. This part is again fairly straightforward, as it is merely a case of plugging in DFT, DFT_5, DFT_10, and DFT_50 into the irfft command from fft.

```
# part d

IFT =fft.irfft(DFT)

IFT_5 = fft.irfft(DFT_5)

IFT_10 = fft.irfft(DFT_10)

IFT_50 = fft.irfft(DFT_50)
```

(e) *Make a plot of all three results and on the same axes show the original square-wave as well. You should find that the the signal is not simply smoothed - there are artifacts, wiggles, in the results. Explain briefly where these come from.*

We are now about to see the deviations from the original N array's DFT and IFT compared to the DFT/IFTs of the array when three different number of terms have been made zero.

The first DFT, which in part d gave us, IFT, will be the original square-wave. This is because that DFT and IFT were a very simple case of taking the array N, doing a transform on it, then doing that inverse transform. Much like if you differentiate x and then immediately integrate it back, you'll end up with x again (if the integration constant is zero!), IFT will give us back our original wave again.

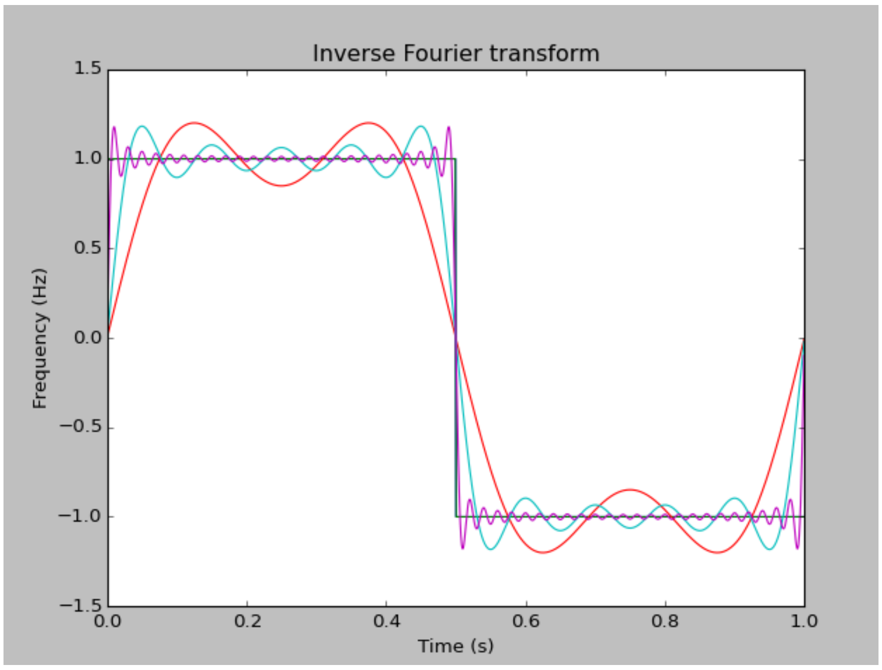
The other three DFT/IFTs, the ones with the various different number of non-zero terms, will not be totally like the original square wave. The more terms of N that were set to zero (instead of the original non-zero values), the further away the approximation becomes to the original. That is why we see three separate plots superimposed on the original with varying degrees of similarity; the most similar plot, the one that lies closest to the original, will be the one with 50 non-zero terms. That is the one that has kept the largest number of its original terms, so it will be the one closest to the original.

Following that logic, the second-most similar plot will be that of the 10 non-zero terms, and the third-most (ie least) similar plot is the DFT/IFT that only kept a mere 5 of its original terms. The less non-zero terms, and therefore the less accurate, the more artifacts/wiggles arise as the plot becomes a less and less precise approximation.

In our code, we need to give a t linspace to have something to plot these IFTs against. After that, we use pyplot to create our plots.

```
t = np.linspace(0,1,1000) # 1000 time values between 0-1

plt.plot(t,N)
plt.plot(t,IFT)
plt.plot(t,IFT_5)
plt.plot(t,IFT_10)
plt.plot(t,IFT_50)
plt.title('Inverse Fourier transform')
plt.xlabel('Time (s)')
plt.ylabel('Frequency (Hz)')
plt.show()
```

Running this code, we finally get our figure that shows our original wave along with the other IFTs and their artifacts:


Inverse Fourier transform
Frequency (Hz)
Time (s)