Numerical Software Lab
Spring 2020

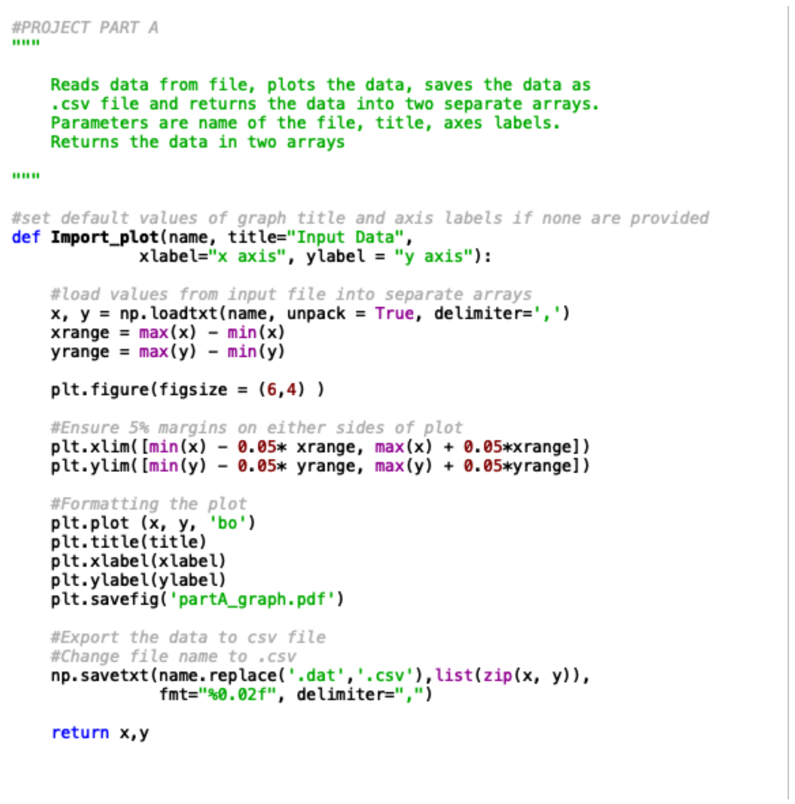
Lalit Chaudhary

May 15, 2020

GUIDED PROJECT

Contents

1 Project Part A



```python
#PROJECT PART A
"""

    Reads data from file, plots the data, saves the data as
    .csv file and returns the data into two separate arrays.
    Parameters are name of the file, title, axes labels.
    Returns the data in two arrays

"""

#set default values of graph title and axis labels if none are provided
def Import_plot(name, title="Input Data",
             xlabel="x axis", ylabel = "y axis"):

    #load values from input file into separate arrays
    x, y = np.loadtxt(name, unpack = True, delimiter=',')
    xrange = max(x) - min(x)
    yrange = max(y) - min(y)

    plt.figure(figsize = (6,4) )

    #Ensure 5% margins on either sides of plot
    plt.xlim([min(x) - 0.05* xrange, max(x) + 0.05*xrange])
    plt.ylim([min(y) - 0.05* yrange, max(y) + 0.05*yrange])

    #Formatting the plot
    plt.plot (x, y, 'bo')
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig('partA_graph.pdf')

    #Export the data to csv file
    #Change file name to .csv
    np.savetxt(name.replace('.dat','.csv'),list(zip(x, y)),
               fmt="%0.02f", delimiter=",")

    return x,y
```

Figure 1: Code Snippet for Project Part A

The function was called with the following parameters:

```
xdata, ydata = Import_plot('edata5.dat')
```

Since the source of data was unknown, the axes labels and the title were not provided as the arguments. Thus, defaults values were taken. Moreover, the data were returned in two arrays, `xdata` and `ydata`.

The following graph was obtained:

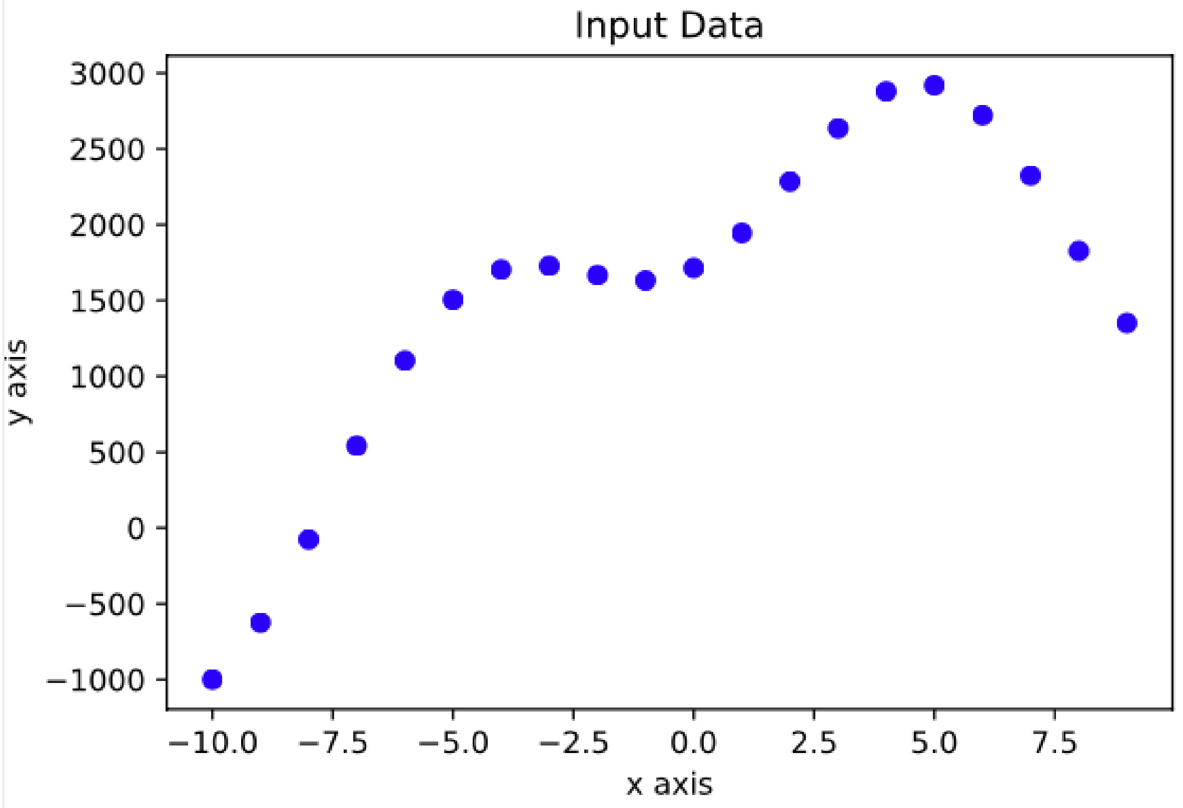
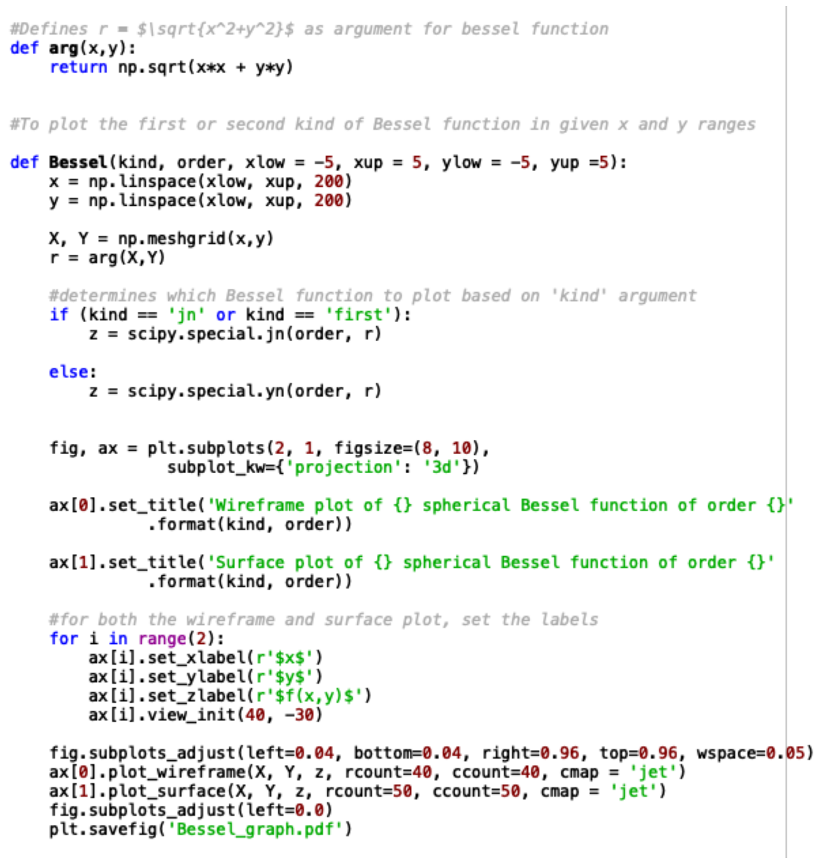


Figure 2: Plot for Part A of the project

2 Project Part B



```python
#Defines r = $\sqrt{x^2+y^2}$ as argument for bessel function
def arg(x,y):
    return np.sqrt(x*x + y*y)


#To plot the first or second kind of Bessel function in given x and y ranges

def Bessel(kind, order, xlow = -5, xup = 5, ylow = -5, yup =5):
    x = np.linspace(xlow, xup, 200)
    y = np.linspace(xlow, xup, 200)

    X, Y = np.meshgrid(x,y)
    r = arg(X,Y)

    #determines which Bessel function to plot based on 'kind' argument
    if (kind == 'jn' or kind == 'first'):
        z = scipy.special.jn(order, r)

    else:
        z = scipy.special.yn(order, r)


    fig, ax = plt.subplots(2, 1, figsize=(8, 10),
                subplot_kw={'projection': '3d'})

    ax[0].set_title('Wireframe plot of {} spherical Bessel function of order {}'
            .format(kind, order))

    ax[1].set_title('Surface plot of {} spherical Bessel function of order {}'
            .format(kind, order))

    #for both the wireframe and surface plot, set the labels
    for i in range(2):
        ax[i].set_xlabel(r'$x$')
        ax[i].set_ylabel(r'$y$')
        ax[i].set_zlabel(r'$f(x,y)$')
        ax[i].view_init(40, -30)

    fig.subplots_adjust(left=0.04, bottom=0.04, right=0.96, top=0.96, wspace=0.05)
    ax[0].plot_wireframe(X, Y, z, rcount=40, ccount=40, cmap = 'jet')
    ax[1].plot_surface(X, Y, z, rcount=50, ccount=50, cmap = 'jet')
    fig.subplots_adjust(left=0.0)
    plt.savefig('Bessel_graph.pdf')
```

Figure 3: Code Snippet for Project Part B

First, the function was called with the following parameters:

```
Bessel('first', 0)
```

Here, the values of ranges for x and y axes were set to default as none were provided. The following plot was obtained:

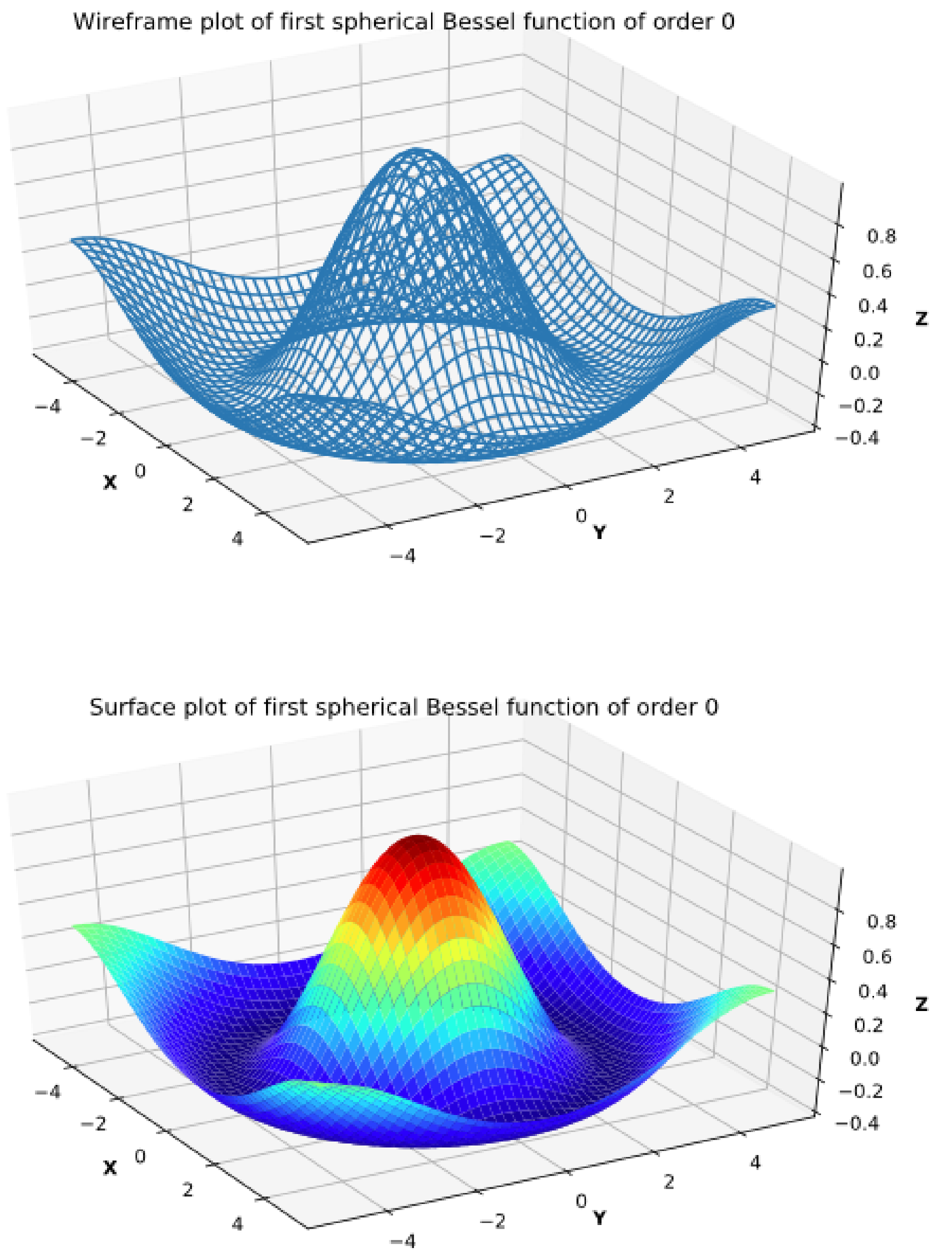
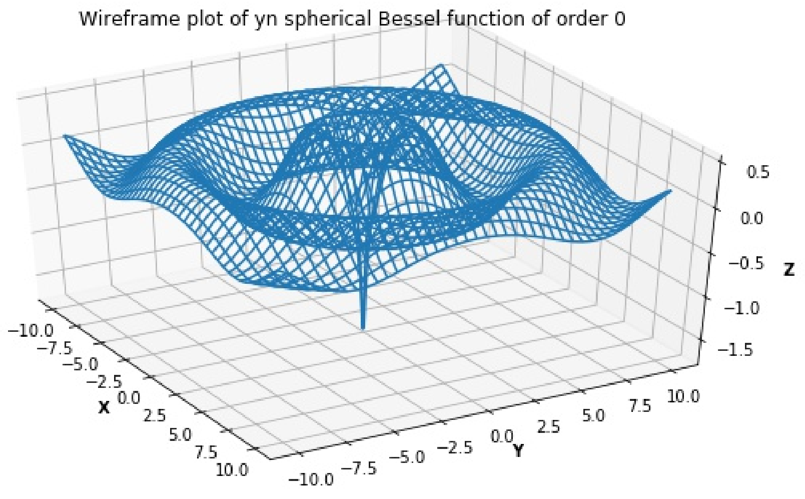


Figure 4: 3D plot of Spherical Bessel Function of first kind

The function was once again called with different parameters:

```
Bessel('yn', 0, 10, -10, 10, 10)
```

The following plot was obtained:



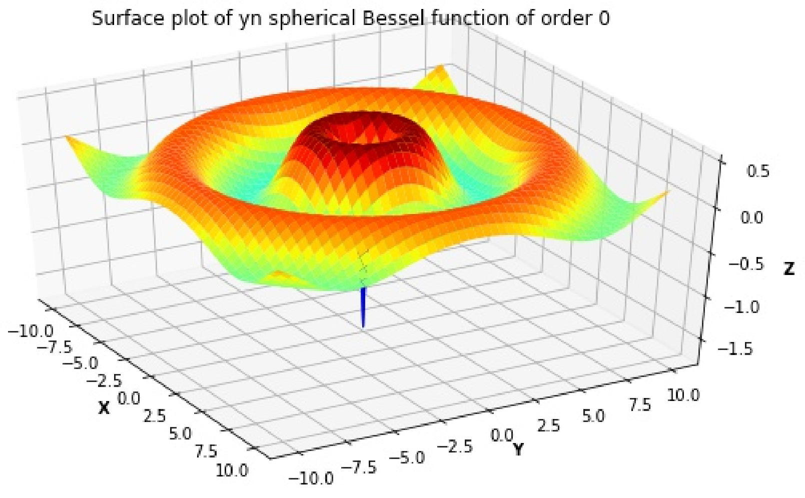
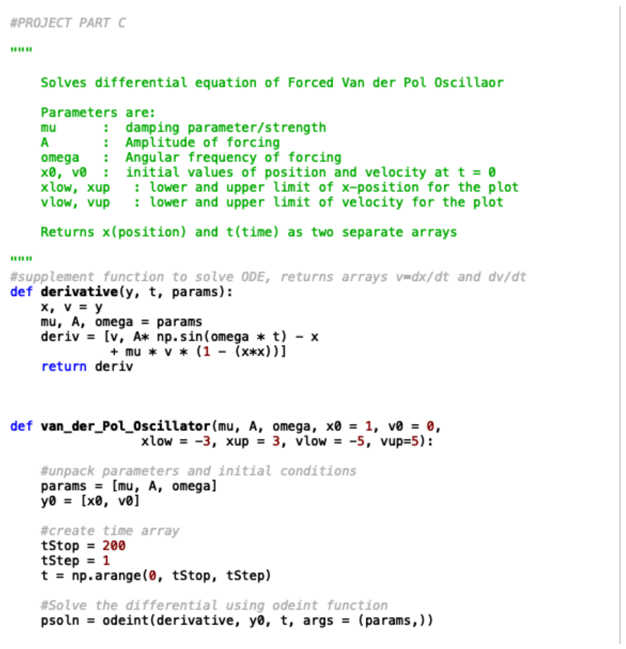
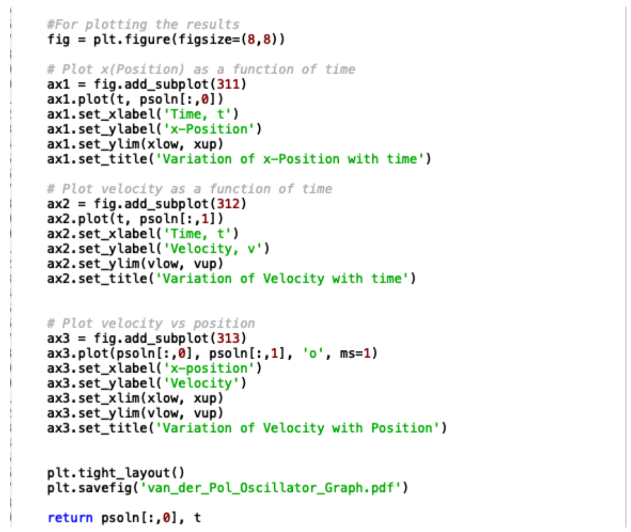


Figure 5: 3D plot of Spherical Bessel function of second kind

3 Project Part C



```
#PROJECT PART C

"""

    Solves differential equation of Forced Van der Pol Oscillaor

    Parameters are:
    mu       :  damping parameter/strength
    A        :  Amplitude of forcing
    omega    :  Angular frequency of forcing
    x0, v0   :  initial values of position and velocity at t = 0
    xlow, xup   : lower and upper limit of x-position for the plot
    vlow, vup   : lower and upper limit of velocity for the plot

    Returns x(position) and t(time) as two separate arrays

"""
#supplement function to solve ODE, returns arrays v=dx/dt and dv/dt
def derivative(y, t, params):
    x, v = y
    mu, A, omega = params
    deriv = [v, A* np.sin(omega * t) - x
             + mu * v * (1 - (x*x))]
    return deriv


def van_der_Pol_Oscillator(mu, A, omega, x0 = 1, v0 = 0,
                 xlow = -3, xup = 3, vlow = -5, vup=5):

    #unpack parameters and initial conditions
    params = [mu, A, omega]
    y0 = [x0, v0]

    #create time array
    tStop = 200
    tStep = 1
    t = np.arange(0, tStop, tStep)

    #Solve the differential using odeint function
    psoln = odeint(derivative, y0, t, args = (params,))
```



```
    #For plotting the results
    fig = plt.figure(figsize=(8,8))

    # Plot x(Position) as a function of time
    ax1 = fig.add_subplot(311)
    ax1.plot(t, psoln[:,0])
    ax1.set_xlabel('Time, t')
    ax1.set_ylabel('x-Position')
    ax1.set_ylim(xlow, xup)
    ax1.set_title('Variation of x-Position with time')

    # Plot velocity as a function of time
    ax2 = fig.add_subplot(312)
    ax2.plot(t, psoln[:,1])
    ax2.set_xlabel('Time, t')
    ax2.set_ylabel('Velocity, v')
    ax2.set_ylim(vlow, vup)
    ax2.set_title('Variation of Velocity with time')


    # Plot velocity vs position
    ax3 = fig.add_subplot(313)
    ax3.plot(psoln[:,0], psoln[:,1], 'o', ms=1)
    ax3.set_xlabel('x-position')
    ax3.set_ylabel('Velocity')
    ax3.set_xlim(xlow, xup)
    ax3.set_ylim(vlow, vup)
    ax3.set_title('Variation of Velocity with Position')


    plt.tight_layout()
    plt.savefig('van_der_Pol_Oscillator_Graph.pdf')

    return psoln[:,0], t
```

Figure 6: Code Snippet for Project Part C

The function was called with the following parameters:

```
xpos, time = van_der_Pol_Oscillator(1.5, 2.5, 1.6*np.pi)
```

Here, the starting and end values `x` and `v` were set to default values as none were provided. Moreover, the `x` and `t` arrays were returned and stored in `xpos` and `time`.
The following plot was obtained:

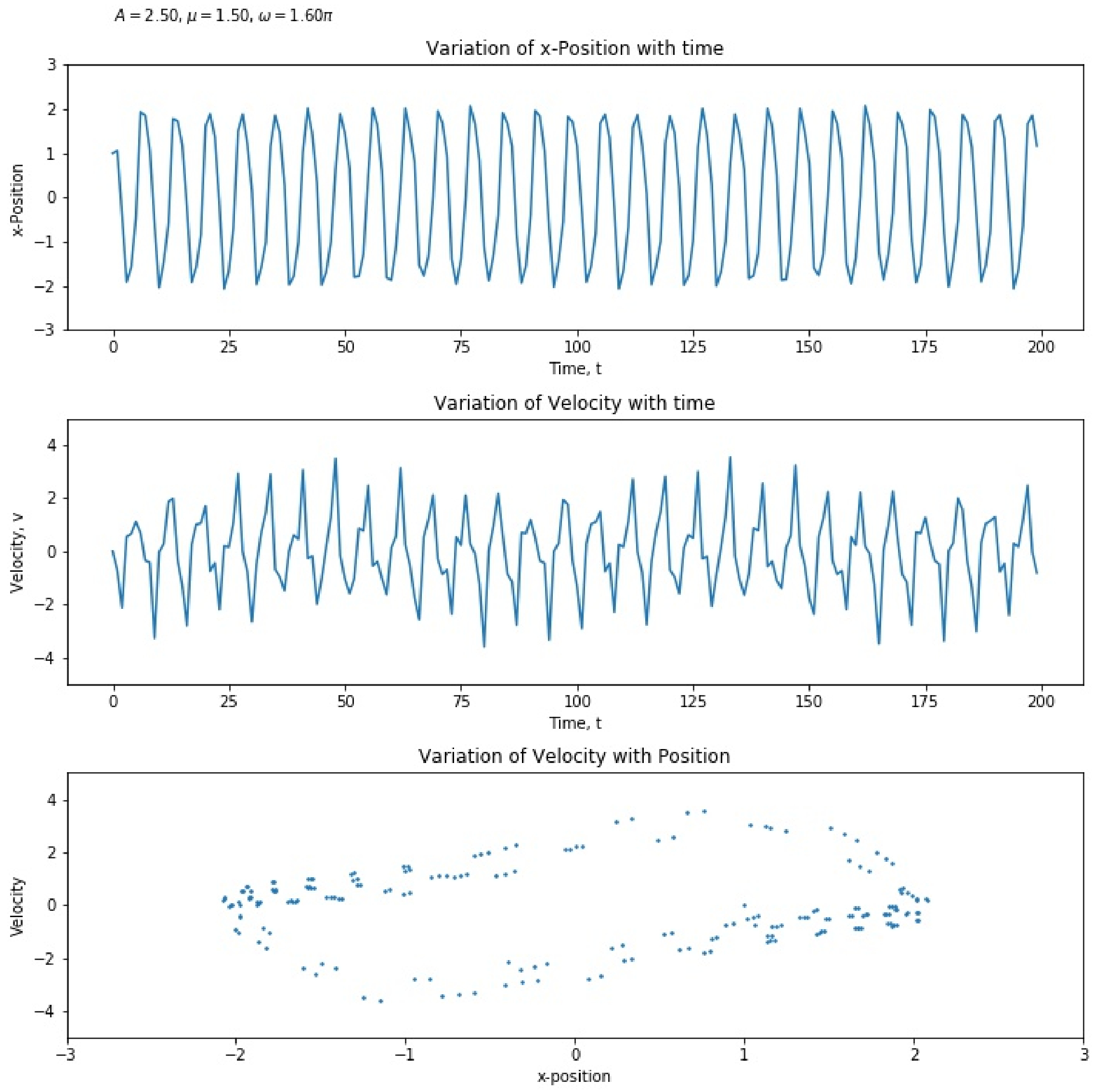
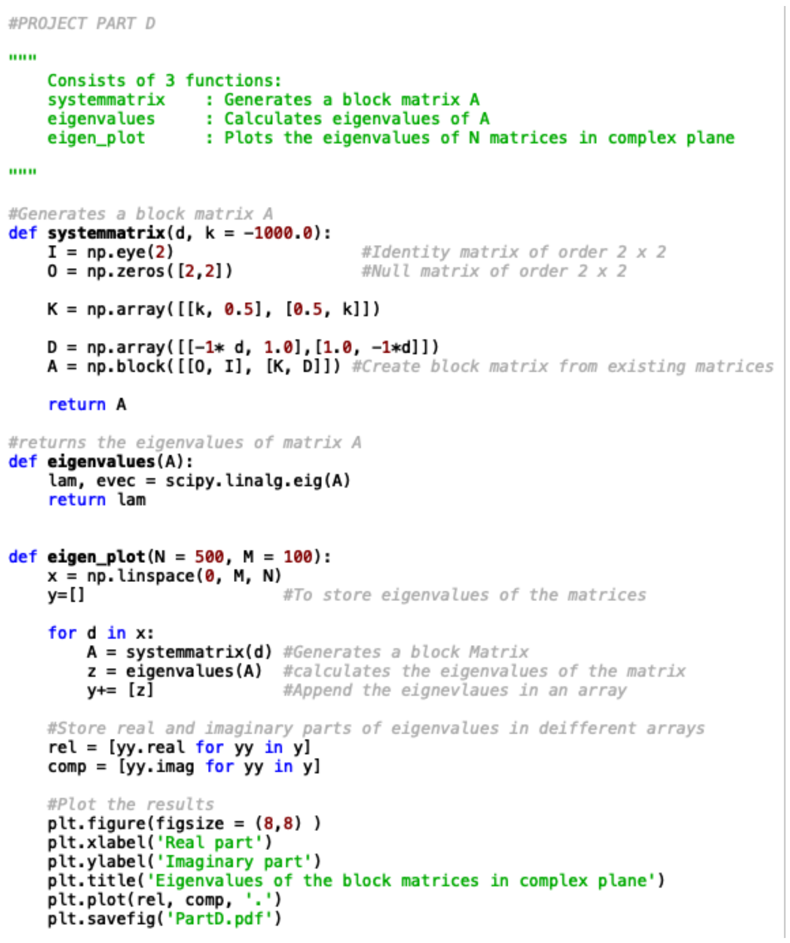


Figure 7: Solution of Van der Pol Oscillator differential equation

4 Project Part D



```python
#PROJECT PART D

"""
    Consists of 3 functions:
    systemmatrix    : Generates a block matrix A
    eigenvalues     : Calculates eigenvalues of A
    eigen_plot      : Plots the eigenvalues of N matrices in complex plane

"""

#Generates a block matrix A
def systemmatrix(d, k = -1000.0):
    I = np.eye(2)                       #Identity matrix of order 2 x 2
    O = np.zeros([2,2])                 #Null matrix of order 2 x 2

    K = np.array([[k, 0.5], [0.5, k]])

    D = np.array([[-1* d, 1.0],[1.0, -1*d]])
    A = np.block([[O, I], [K, D]]) #Create block matrix from existing matrices

    return A

#returns the eigenvalues of matrix A
def eigenvalues(A):
    lam, evec = scipy.linalg.eig(A)
    return lam


def eigen_plot(N = 500, M = 100):
    x = np.linspace(0, M, N)
    y=[]                        #To store eigenvalues of the matrices

    for d in x:
        A = systemmatrix(d) #Generates a block Matrix
        z = eigenvalues(A)  #calculates the eigenvalues of the matrix
        y+= [z]             #Append the eignevlaues in an array

    #Store real and imaginary parts of eigenvalues in deifferent arrays
    rel = [yy.real for yy in y]
    comp = [yy.imag for yy in y]

    #Plot the results
    plt.figure(figsize = (8,8) )
    plt.xlabel('Real part')
    plt.ylabel('Imaginary part')
    plt.title('Eigenvalues of the block matrices in complex plane')
    plt.plot(rel, comp, '.')
    plt.savefig('PartD.pdf')
```

Figure 8: Code Snippet for Project Part D

The function was called with the following parameters:

`eigen_plot(856, 80)`

The following plot was obtained:

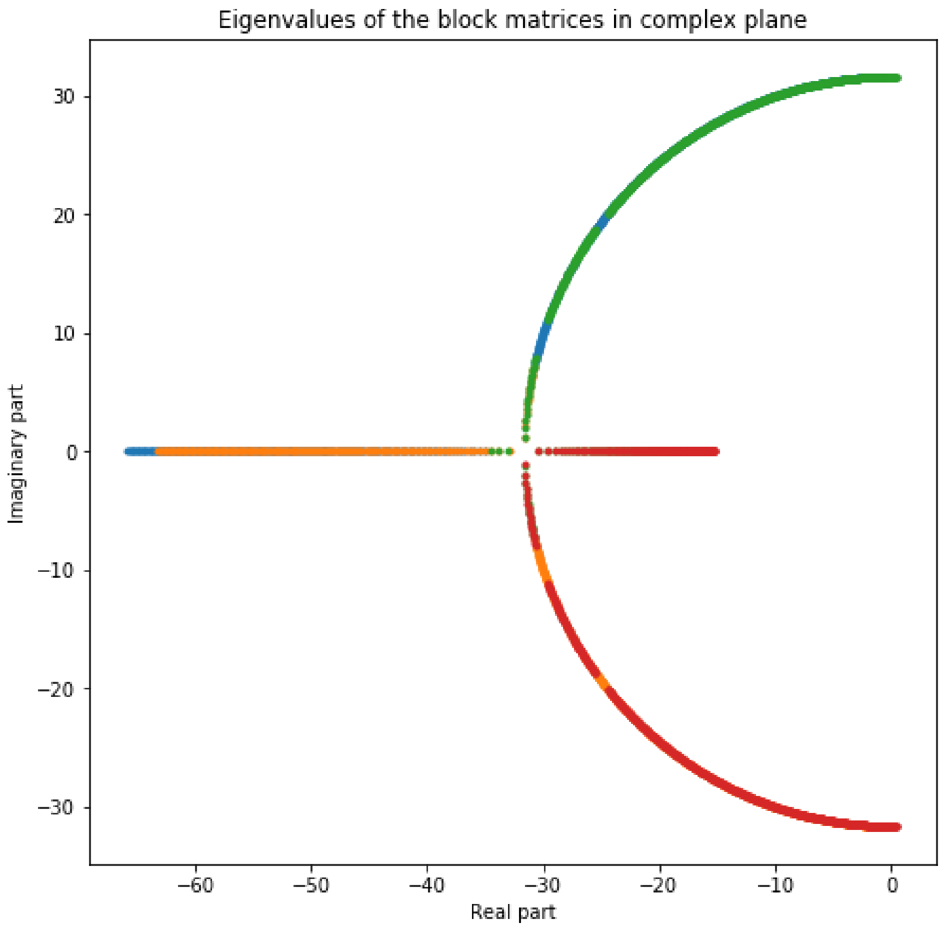
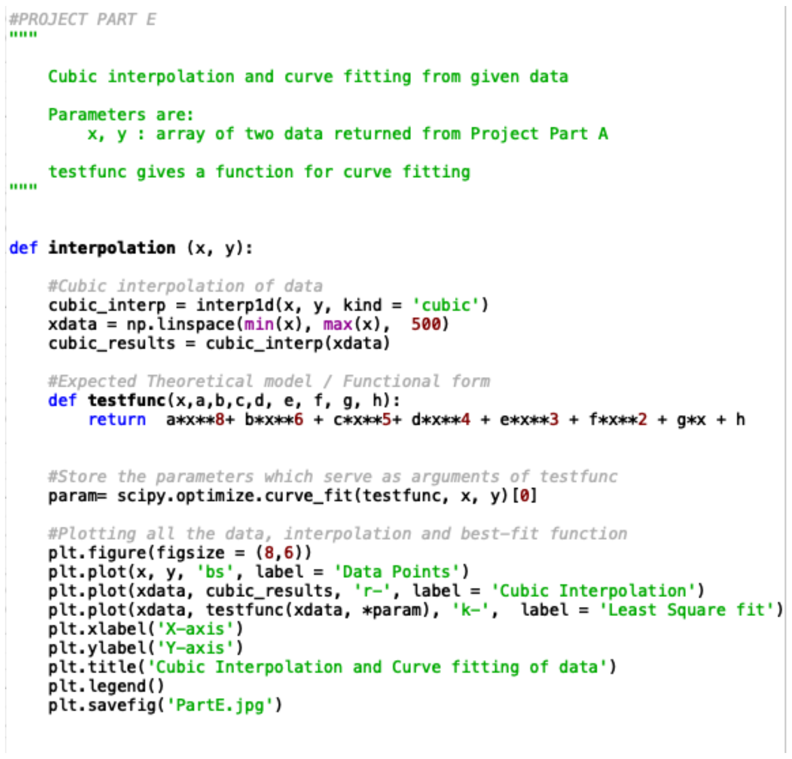


Figure 9: Eigenvalues of block matrices in complex plane

The plot is symmetrical about the x-axis, which signifies that the complex eigenvalues always come in pairs i.e. if λ is a complex eigenvalue of a matrix A, then $\overline{\lambda}$ is also an eigenvalue of A.

5 Project Part E



```
#PROJECT PART E
"""

    Cubic interpolation and curve fitting from given data

    Parameters are:
        x, y : array of two data returned from Project Part A

    testfunc gives a function for curve fitting
"""


def interpolation (x, y):

    #Cubic interpolation of data
    cubic_interp = interp1d(x, y, kind = 'cubic')
    xdata = np.linspace(min(x), max(x),  500)
    cubic_results = cubic_interp(xdata)

    #Expected Theoretical model / Functional form
    def testfunc(x,a,b,c,d, e, f, g, h):
        return  a*x**8+ b*x**6 + c*x**5+ d*x**4 + e*x**3 + f*x**2 + g*x + h


    #Store the parameters which serve as arguments of testfunc
    param= scipy.optimize.curve_fit(testfunc, x, y)[0]

    #Plotting all the data, interpolation and best-fit function
    plt.figure(figsize = (8,6))
    plt.plot(x, y, 'bs', label = 'Data Points')
    plt.plot(xdata, cubic_results, 'r-', label = 'Cubic Interpolation')
    plt.plot(xdata, testfunc(xdata, *param), 'k-',  label = 'Least Square fit')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.title('Cubic Interpolation and Curve fitting of data')
    plt.legend()
    plt.savefig('PartE.jpg')
```

Figure 10: Cubic Interpolation and Curve Fitting of Data

For the data arrays returned from Part A of the project, i.e.,

```
xdata, ydata = Import_plot("edata5.dat")
```

cubic interpolation of the data was performed. Moreover, the functional form of the graph was guessed to be the following polynomial form of degree 8:

$$ax^8 + bx^6 + cx^5 + dx^4 + ex^3 + fx^2 + gx + h$$

The function was called with the following syntax

```
interpolation(xdata, ydata)
```

The following plot was obtained:

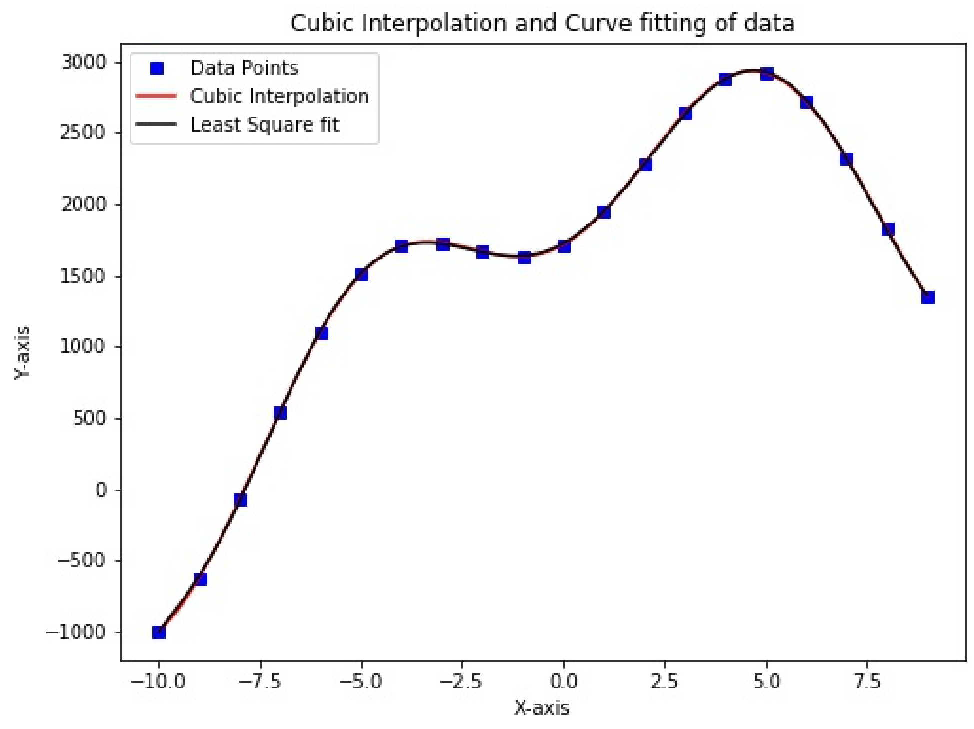
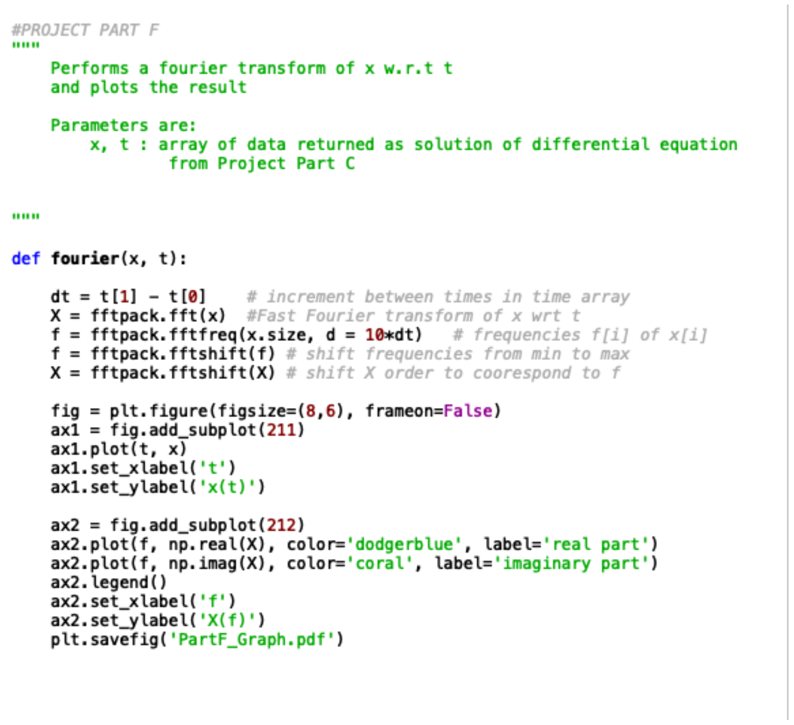


Figure 11: Cubic Interpolation and Curve Fitting

6 Project Part F



```
#PROJECT PART F
"""
    Performs a fourier transform of x w.r.t t
    and plots the result

    Parameters are:
        x, t : array of data returned as solution of differential equation
                from Project Part C


"""

def fourier(x, t):

    dt = t[1] - t[0]    # increment between times in time array
    X = fftpack.fft(x)  #Fast Fourier transform of x wrt t
    f = fftpack.fftfreq(x.size, d = 10*dt)  # frequencies f[i] of x[i]
    f = fftpack.fftshift(f) # shift frequencies from min to max
    X = fftpack.fftshift(X) # shift X order to coorespond to f

    fig = plt.figure(figsize=(8,6), frameon=False)
    ax1 = fig.add_subplot(211)
    ax1.plot(t, x)
    ax1.set_xlabel('t')
    ax1.set_ylabel('x(t)')

    ax2 = fig.add_subplot(212)
    ax2.plot(f, np.real(X), color='dodgerblue', label='real part')
    ax2.plot(f, np.imag(X), color='coral', label='imaginary part')
    ax2.legend()
    ax2.set_xlabel('f')
    ax2.set_ylabel('X(f)')
    plt.savefig('PartF_Graph.pdf')
```

Figure 12: Code Snippet for Project Part F

For the data arrays returned from Part C of the project, i.e.,

```
xpos, time = van_der_Pol_Oscillator(1.5, 2.5, 1.6*np.pi)
```

the fourier transform of `xpos` eith respect to `time` was performed by following function call:

```
fourier(xpos, time)
```

The result is shown in plot below:

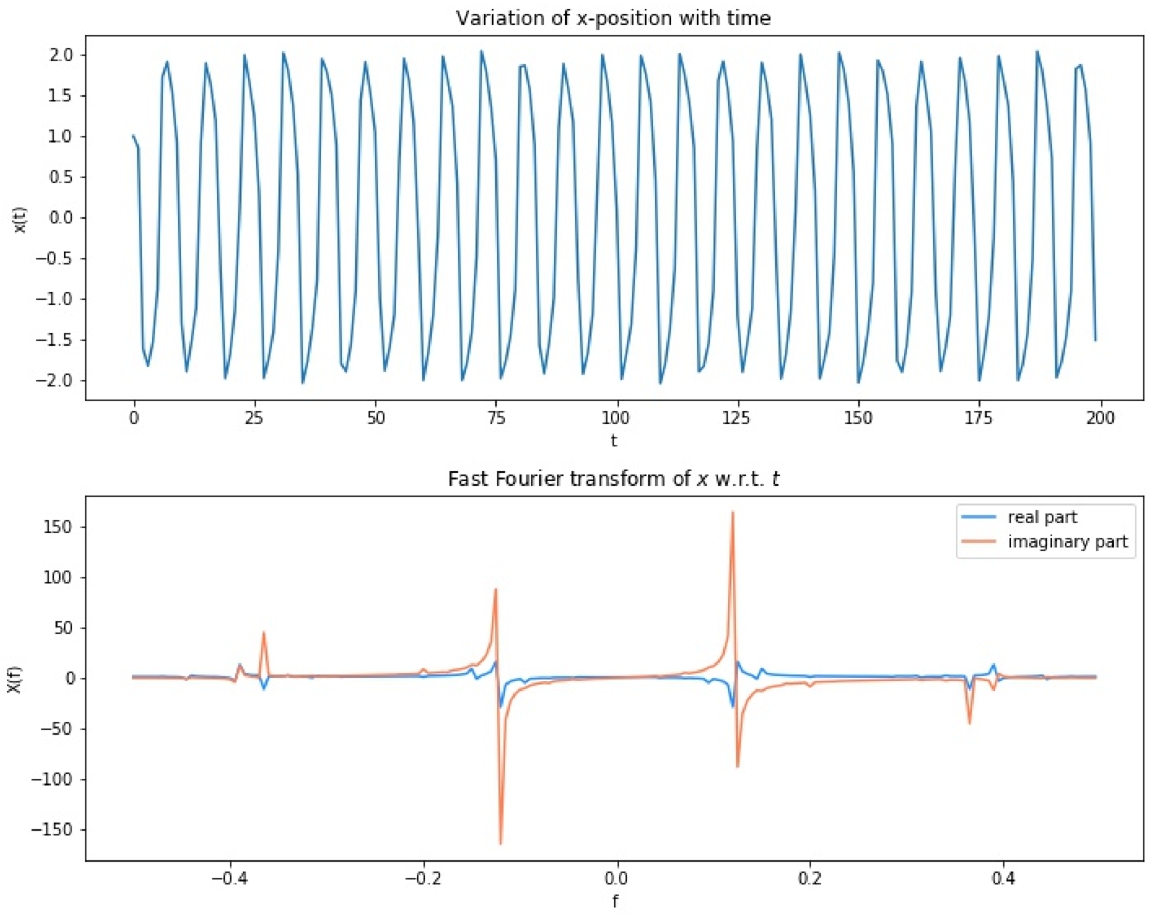
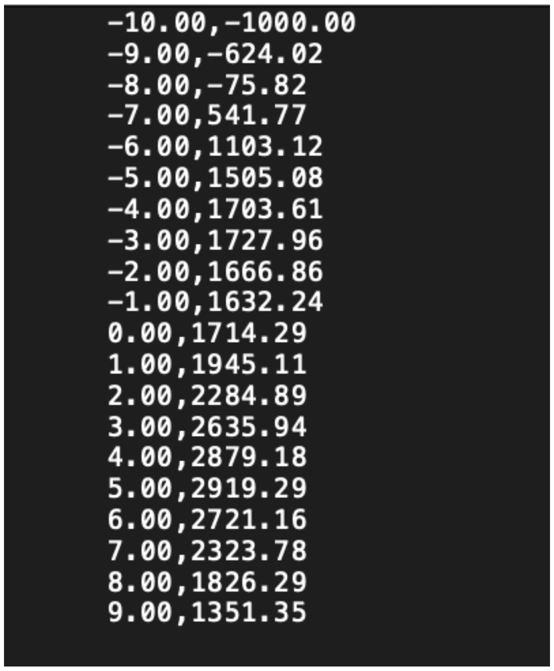


Figure 13: [top]Plot of x vs t; [below] result of Fourier Transform

A 'edata5.dat' (Source input for project part A)



```
-10.00,-1000.00
-9.00,-624.02
-8.00,-75.82
-7.00,541.77
-6.00,1103.12
-5.00,1505.08
-4.00,1703.61
-3.00,1727.96
-2.00,1666.86
-1.00,1632.24
0.00,1714.29
1.00,1945.11
2.00,2284.89
3.00,2635.94
4.00,2879.18
5.00,2919.29
6.00,2721.16
7.00,2323.78
8.00,1826.29
9.00,1351.35
```

Figure 14: Data input for project part A

B Full Code

```python
#!/usr/bin/env python3
# -*- coding: utf-8 -*-
"""
Created on Thu May 14 21:24:26 2020

@author: Lalit Chaudhary
"""

import numpy as np
import matplotlib.pyplot as plt
from scipy.integrate import odeint
from scipy import fftpack
from scipy.interpolate import interp1d
```

```
import scipy.linalg
from mpl_toolkits.mplot3d import Axes3D



#PROJECT PART A
"""

    Reads data from file, plots the data, saves the data as
    .csv file and returns the data into two separate arrays.
    Parameters are name of the file, title, axes labels.
    Returns the data in two arrays

"""

#set default values of graph title and axis labels if none are provided
def Import_plot(name, title="Input Data",
             xlabel="x axis", ylabel = "y axis"):

    #load values from input file into separate arrays
    x, y = np.loadtxt(name, unpack = True, delimiter=',')
    xrange = max(x) - min(x)
    yrange = max(y) - min(y)

    plt.figure(figsize = (6,4) )

    #Ensure 5% margins on either sides of plot
    plt.xlim([min(x) - 0.05* xrange, max(x) + 0.05*xrange])
    plt.ylim([min(y) - 0.05* yrange, max(y) + 0.05*yrange])

    #Formatting the plot
    plt.plot (x, y, 'bo')
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.savefig('partA.pdf')

    #Export the data to csv file
    #Change file name to .csv
    np.savetxt(name.replace('.dat','.csv'),list(zip(x, y)),
               fmt="%0.02f", delimiter=",")

    return x,y
```

```
#PROJECT PART B
"""

    3D plot of spherical bessel function of first or second kind.

    Parameters are:
    kind        : 'jn' or 'first' for first kind
                  'yn' or 'second' for second kind
    order       : order of the Bessel function
    xlow, xup   : lower and upper limit of x-axis for the 3D plot
    ylow, yup   : lower and upper limit of y-axis for the 3D plot

"""


#Defines r = $\sqrt{x^2+y^2}$ as argument for bessel function
def arg(x,y):
    return np.sqrt(x*x + y*y)


#To plot the first or second kind of Bessel function in given x and y ranges
def Bessel(kind, order, xlow = -5, xup = 5, ylow = -5, yup =5):
    x = np.linspace(xlow, xup, 200)
    y = np.linspace(xlow, xup, 200)

    X, Y = np.meshgrid(x,y)
    r = arg(X,Y)

    #determines which Bessel function to plot based on 'kind' argument
    if (kind == 'jn' or kind == 'first'):
        z = scipy.special.jn(order, r)

    else:
        z = scipy.special.yn(order, r)

    #Plotting the results
    fig, ax = plt.subplots(2, 1, figsize=(8, 10),
                subplot_kw={'projection': '3d'})

    ax[0].set_title('Wireframe plot of {} spherical Bessel function of order {}'
              .format(kind, order))

    ax[1].set_title('Surface plot of {} spherical Bessel function of order {}'
              .format(kind, order))
    #for both the wireframe and surface plot, set the labels
    for i in range(2):
        ax[i].set_xlabel('X', fontweight='bold')
        ax[i].set_ylabel('Y', fontweight='bold')
        ax[i].set_zlabel('Z', fontweight='bold')
        ax[i].view_init(40, -30)

    fig.subplots_adjust(left=0.04, bottom=0.04, right=0.96, top=0.96, wspace=0.05)
    ax[0].plot_wireframe(X, Y, z, rcount=40, ccount=40, cmap = 'jet')
    ax[1].plot_surface(X, Y, z, rcount=50, ccount=50, cmap = 'jet')
    fig.subplots_adjust(left=0.0)
    plt.savefig('PartB.pdf')



#PROJECT PART C

"""

    Solves differential equation of Forced Van der Pol Oscillaor

    Parameters are:
    mu       :  damping parameter/strength
    A        :  Amplitude of forcing
    omega    :  Angular frequency of forcing
    x0, v0   :  initial values of position and velocity at t = 0
    xlow, xup    : lower and upper limit of x-position for the plot
    vlow, vup    : lower and upper limit of velocity for the plot

    Returns x(position) and t(time) as two separate arrays

"""
#supplement function to solve ODE, returns arrays v=dx/dt and dv/dt
def derivative(y, t, params):
    x, v = y
    mu, A, omega = params
    deriv = [v, A* np.sin(omega * t) - x
             + mu * v * (1 - (x*x))]
    return deriv



def van_der_Pol_Oscillator(mu, A, omega, x0 = 1, v0 = 0,
                 xlow = -3, xup = 3, vlow = -5, vup=5):

    #unpack parameters and initial conditions
```

```python
params = [mu, A, omega]
y0 = [x0, v0]

#create time array
tStop = 200
tStep = 1
t = np.arange(0, tStop, tStep)

#Solve the differential using odeint function
psoln = odeint(derivative, y0, t, args = (params,))

#For plotting the results
fig = plt.figure(figsize=(10,10))

# Plot x(Position) as a function of time
ax1 = fig.add_subplot(311)
ax1.plot(t, psoln[:,0])
ax1.set_xlabel('Time, t')
ax1.set_ylabel('x-Position')
ax1.set_ylim(xlow, xup)
ax1.set_title('Variation of x-Position with time')
ax1.text(0, xup+1, r'$A ={0:.2f}, \mu = {1:.2f}, \omega = {2:.2f}\pi$'
         .format(A, mu, omega/(np.pi)))


# Plot velocity as a function of time
ax2 = fig.add_subplot(312)
ax2.plot(t, psoln[:,1])
ax2.set_xlabel('Time, t')
ax2.set_ylabel('Velocity, v')
ax2.set_ylim(vlow, vup)
ax2.set_title('Variation of Velocity with time')


# Plot velocity vs position
ax3 = fig.add_subplot(313)
ax3.plot(psoln[:,0], psoln[:,1], '.', ms = 3)
ax3.set_xlabel('x-position')
ax3.set_ylabel('Velocity, v')
ax3.set_xlim(xlow, xup)
ax3.set_ylim(vlow, vup)
ax3.set_title('Variation of Velocity with Position')


plt.tight_layout()
plt.savefig('PartC.pdf')
```

```
    return psoln[:,0], t


#PROJECT PART D

"""
    Consists of 3 functions:
    systemmatrix    : Generates a block matrix A
    eigenvalues     : Calculates eigenvalues of A
    eigen_plot      : Plots the eigenvalues of N matrices in complex plane

"""

#Generates a block matrix A
def systemmatrix(d, k = -1000.0):
    I = np.eye(2)                    #Identity matrix of orden 2 x 2
    O = np.array([[0,0],[0,0]])      #Null matrix of order 2 x 2

    K = np.array([[k, 0.5], [0.5, k]])

    D = np.array([[-1* d, 1.0],[1.0, -1*d]])
    A = np.array([[O, I], [K, D]])

    return A

#returns the eigenvalues of matrix A
def eigenvalues(A):
    lam, evec = scipy.linalg.eig(A.reshape(4,4))
    return lam


def eigen_plot(N = 500, M = 100):
    x = np.linspace(0, M, N)
    y=[]                        #To store eigenvalues of the matrices

    for d in x:
        A = systemmatrix(d) #Generates a block Matrix
        z = eigenvalues(A)  #calculates the eigenvalues of the matrix
        y+= [z]             #Append the eignevlaues in an array

    #Store real and imaginary parts of eigenvalues in deifferent arrays
    rel = [yy.real for yy in y]
    comp = [yy.imag for yy in y]

    #Plot the results
    plt.figure(figsize = (8,8) )
    plt.xlabel('Real part')
    plt.ylabel('Imaginary part')
    plt.title('Eigenvalues of the block matrices in complex plane')
    plt.plot(rel, comp, '.')
    plt.savefig('PartD.pdf')


#PROJECT PART E
"""

    Cubic interpolation and curve fitting from given data

    Parameters are:
        x, y : array of two data returned from Project Part A

    testfunc gives a function for curve fitting
"""


def interpolation (x, y):

    #Cubic interpolation of data
    cubic_interp = interp1d(x, y, kind = 'cubic')
    xdata = np.linspace(min(x), max(x),  500)
    cubic_results = cubic_interp(xdata)

    #Expected Theoretical model / Functional form
    def testfunc(x,a,b,c,d, e, f, g, h):
        return  a*x**8+ b*x**6 + c*x**5+ d*x**4 + e*x**3 + f*x**2 + g*x + h


    #Store the parameters which serve as arguments of testfunc
    param= scipy.optimize.curve_fit(testfunc, x, y)[0]

    #Plotting all the data, interpolation and best-fit function
    plt.figure(figsize = (8,6))
    plt.plot(x, y, 'bs', label = 'Data Points')
    plt.plot(xdata, cubic_results, 'r-', label = 'Cubic Interpolation')
    plt.plot(xdata, testfunc(xdata, *param), 'k-',  label = 'Least Square fit')
    plt.xlabel('X-axis')
    plt.ylabel('Y-axis')
    plt.title('Cubic Interpolation and Curve fitting of data')
    plt.legend()
    plt.savefig('PartE.pdf')
```

```
#PROJECT PART F
"""
    Performs a fourier transform of x w.r.t t
    and plots the result

    Parameters are:
        x, t : array of data returned as solution of differential equation
               from Project Part C


"""

def fourier(x, t):

    dt = t[1] - t[0]     # increment between times in time array
    X = fftpack.fft(x)   #Fast Fourier transform of x wrt t
    f = fftpack.fftfreq(x.size, d = dt)    # frequencies f[i] of x[i]
    f = fftpack.fftshift(f) # shift frequencies from min to max
    X = fftpack.fftshift(X) # shift X order to coorespond to f

    fig = plt.figure(figsize=(10,8), frameon=False)

    #Plotting original function, x vs t
    ax1 = fig.add_subplot(211)
    ax1.plot(t, x)
    ax1.set_xlabel('t')
    ax1.set_ylabel('x(t)')
    ax1.set_title('Variation of x-position with time')

    #Plotting Fourier transform of the original function
    ax2 = fig.add_subplot(212)
    ax2.plot(f, np.real(X), color='dodgerblue', label='real part')
    ax2.plot(f, np.imag(X), color='coral', label='imaginary part')
    ax2.legend()
    ax2.set_xlabel('f')
    ax2.set_ylabel('X(f)')
    ax2.set_title(r'Fast Fourier transform of $x$ w.r.t. $t$')
    plt.tight_layout()
    plt.savefig('PartF.pdf')




#Main part of the file calling all the functions
```

```
xdata, ydata = Import_plot("edata5.dat")

Bessel('first', 0)
#Bessel('yn', 0, 10, -10, 10, 10)

xpos, time = van_der_Pol_Oscillator(2.5, 1.5, 1.6*np.pi)
eigen_plot(856, 80)
interpolation(xdata, ydata)
fourier(xpos, time)
```