Week 6 Assignment

Question 1:

```
x0=0 # intial value of x
 y0=5 # initial value of y
 sol_x=[]
 sol_y=[]
 f=lambda x,y: -2*y # RHS of our ODE
#Euler Formula
 while x0<=10:
     yn=y0+h*f(x0,y0)
     sol_x.append(x0)
     sol_y.append(y0)
     x0=x0+h
     y0=yn
 return sol_x,sol_y
```

- The above code uses Euler's method to approximate the solution of the differential equation $\frac{dy}{dx} = -2y$ with initial condition $y(0) = 5$.
- We have used the Euler's forward formula to calculate it where h is the increment step.

```
•  f=lambda x: 5*np.exp(-2*x) # Original solution

   for i in range(len(h)):
           x = EulerForward(h[i])[0]
           y = EulerForward(h[i])[1]
           exact_y = [f(i) for i in x ] # exact values of y
           x_obs = np.linspace(min(x), max(x),150)
           y_obs = barycentric_interpolate(x, y, x_obs) # plotting
   the polynomial with barycentric_interpolate
           ax[i].plot(x_obs, y_obs,label="Predicted value")
           ax[i].plot(x,exact_y,label="exact value")
           ax[i].legend()
           ax[i].grid()
           ax[i].set_title("Forward Euler for step length
   "+str(h[i]),fontsize=10)
```

- The above code plots the values found via Euler's method against the exact solution.
- We have used **barycentric_interpolate** to interpolate the polynomial.
- Also the resultant plot contains 5 different subplots for5 different values of increment.

Question 2:

```
x0=0
y0=5
sol_x=[]
sol_y=[]
f=lambda x,y: -2*y
while x0<=10:
    yn=y0/(2*h+1) # Formula for yn
    sol_x.append(x0)
    sol_y.append(y0)
    x0=x0+h
    y0=yn
return sol_x,sol_y
```

- The above code approximates the same ode as question 1 but it uses Euler's backward method to approximate it.
- Notice the formula we have derived has follows

$$y_{n+1} = y_n + hf(x_{n+1}, y_{n+1})$$
$$y_{n+1} = y_n + h(-2y_{n+1})$$
$$y_{n+1} = \frac{y_n}{(2h+1)}$$

- The rest of the method used in this question is identical with the 1st one.

Question 3:

```
g = 9.8
L = 2 # length of the pendulum
t = 100 # time
# initial condition
theta_0 = np.pi/3
d_theta_0 = 0

# getting d^2(theta)/d(theta)^2
def get_d2_theta(theta):
    return  -(g/L)*np.sin(theta)

# using Euler's method to solve the system
def theta(t):
    theta = theta_0
    d_theta = d_theta_0
    delta_t = 1./60 # increment
    for time in np.arange(0,t,delta_t):
        d2_theta = get_d2_theta(theta)
        theta =theta + d_theta*delta_t
        d_theta =d_theta+ d2_theta*delta_t
    return theta
```

- The above code first calculates the value of the position of the pendulum with initial condition as shown in red

- We define a function **get_d2_theta** to estimate the value of the 2nd order derivative which we use the function **theta_functions** to calculate the required values .

```
x_data = [0,0]
y_data = [0,0]

fig, ax = plt.subplots()
ax.set_title("Animation emulating the motion of a
pendulum",fontsize=10)
ax.plot([-0.5,0.5],[0,0],"black",linewidth=3) # upper part of the
pendulum
ax.set_xlim(-2, 2)
ax.set_ylim(-2.5,1)
line, = ax.plot(0, 0)

def animation_frame(i):
    x = L*np.sin(theta(i))
    y = -L*np.cos(theta(i))

    x_data[1] = x
    y_data[1] = y

    line.set_xdata(x_data)
    line.set_ydata(y_data)

    return line,
# Animation for the pendulum
animation = FuncAnimation(fig, func=animation_frame,
frames=np.arange(0, 60, (1./60)),interval = 10)
```

- The above function animates the motion of a pendulum
- **animation_frame** is the function we have used in the **funcanimation**

Question 4:

```
def vanderpol(mu, x0, dx0):  # x0 and y0 are the intitial conditions
    if isinstance(mu, str) == True or mu <0:
        try:
            raise Exception("The parameter needs to be a positive real
number")
        except Exception as inst:
            print(type(inst))
            print(inst)
        return None
    # function returning van der pol equation
    def f(t, z):
        x, y = z
        return [y, mu * (1 - x ** 2) * y - x]

    # function for calculating time period
    def root(t, y):
        return y[0]

    a, b = 0, 10
    t = np.linspace(a, b, 500)
    fig, ax = plt.subplots(2)
    # solution for plotting using solve_ivp
```

```
    sol = solve_ivp(f, [a, b], [x0[1], dx0[1]], t_eval=t)
    ax[0].plot(sol.y[0], sol.y[1], "--", label="$\mu=$" + str(mu))   #
plotting phase protraits
    ax[1].plot(sol.t, sol.y[0], "r-", label="$\mu=$" + str(mu))   # plot
with respect to time
   # solution for time period
    sol1 = solve_ivp(f, [0, 40], [1, 0], events=root)
    # find the zeros of the solution over the interval [a, b]
    zeroes = sol1.t_events[0]
    # To estimate the period of the limit cycle we look at the spacing
between zeros
    l = len(zeroes)
    print("The time period is ", 2*(zeroes[l - 1] - zeroes[l - 2])) # e
find the half period in the previous step
                                                                    #
hence time period is twice of that
    ax[0].set_xlim([-3, 3])
    ax[0].set_title("Plot of  phase portrait",fontsize=10)
    ax[1].set_title("Plot with respect to time ",fontsize=10)
    ax[0].legend()
    ax[1].legend()
    fig.tight_layout(pad=3.0)
    plt.show()
```

- The function **vanderpol** takes as its argument the parameter μ and initial condition as tuples of the form (0,1) and (0,1) which in turn means the condition y (0) =1 and y’ (0) =0
- Function **f** is created to assist for the solution of the differential equation by making it 2 one order equations.
- **sol** is used to plot the solution of the differential equation
- **sol1 and root** function is used combinedly to access all the zeroes in the solution which we store in the list zeroes, which is the used to approximate the time period which is twice of that the value found out.
- A sample output of the above plot