Final project for HDEq Skills:

`Due: Thursday 7th December 2023 @ 12:00 (noon)`

Name: Aristide Launois, s2276967

Instruction and marking scheme

Your submission (like your assignments) will be a Jupyter Notebook (.ipynb file) and a pdf file. For your analysis and discussion use Markdown cells instead of comments in the code.

For the presentation and coding style:

- The plots should be in a good size, properly labelled and markers/lines wisely chosen. It might be a good idea to combine some plots and put several curves together for easier comparison and conciseness.
- Your code should be well commented with meaningful variable names and good structures. You don't need to over comment it though!
- The writing should be clear and concise with minimal language or typing mistakes.

It is important that you think about good academic practice when you construct your answers. The work you submit must reflect **your own understanding**, in line with the School's policy on academic misconduct: https://edin.ac/2LtVQMw. Any form of plagiarism that is detected by the software or markers will be reported to the school for the due process.

Part 1: Wave equation with linear damping

Consider the following wave equation that includes a damping term

$$u_{tt} + \nu u_t = c^2 u_{xx} \tag{1}$$

with the boundary conditions

$$u\left(x=0,t\right)=0, \quad u\left(x=L,t\right)=0, \tag{2}$$

and the initial conditions

$$u\left(x,t=0\right)=f(x)=\exp\left(\sin^2\left(\frac{2\pi x}{L}\right)\right)-1, \quad u_t\left(x,t=0\right)=0, \tag{3}$$

For this problem, let's set $L=10$, $\nu=0.1$ and $c=1$.

We use a method to solve this PDE that is known as "Galerkin". In so doing, we approximate the solution with

$$u(x,t) \approx \sum_{n=1}^{N} f_n(t)\sin(\lambda_n x) \tag{4}$$

and then derive an appropriate ODE for $f_n(t)$ to solve numerically. This is a form of separation of variables in which the equations for the functions of time are solved numerically rather than by hand. It can be shown that when $N \to \infty$ the above series converges to the exact solution.

🚩 **Task 1.1 (1 mark)** Considering the above form of solution, find λ_n in terms of n and L.

🚩 **Task 1.2 (1 mark)** Using the assumed form of solution and PDE, find an ODE that describes the evolution of $f_n(t)$ in time, (where $n = 1, 2, \ldots, N$).

🚩 **Task 1.3 (2 marks)** Using the initial conditions of the PDE and your knowledge of Fourier series, find two initial conditions for f_n and $\frac{df_n}{dt}$ at $t = 0$. Your answer in this part might in the form of an integral, which you should not evaluate at this point (we will calculate this integral later numerically).

🚩 **Task 1.1 (1 mark) Answer:** From the above form of the solution and the boundary conditions we find,

$$u(L,t) = 0 = \sum_{n=1}^{N} f_n(t) sin(L\lambda_n) \Rightarrow sin(L\lambda_n) = 0 \Rightarrow \lambda_n = \frac{n\pi}{L} \qquad n = 1, 2, \ldots \tag{5}$$

Task 1.2 Answer: We substitute our assumed form of the solution into the PDE. The partial derivatives are easy to find and we get,

$$\sum_{n=1}^{N} \ddot{f}_n(t) sin(\lambda_n x) + \nu \sum_{n=1}^{N} \dot{f}_n(t) sin(\lambda_n x) = -c^2 \sum_{n=1}^{N} \lambda_n^2 f_n(t) sin(\lambda_n x). \tag{6}$$

Rearranging,

$$\sum_{n=1}^{N} \left(\ddot{f}_n(t) + \nu \ddot{f}_n(t) + c^2 \lambda_n^2 f_n(t) \right) sin(\lambda_n x) = 0. \tag{7}$$

So our ODE for $f_n(t)$ for $n = 1, 2, \ldots, N$ is,

$$\ddot{f}_n(t) + \nu \dot{f}_n(t) + c^2 \lambda_n^2 f_n(t) = 0. \tag{8}$$

Now setting $L = 10$, $\nu = 0.1, c = 1$ and $\lambda_n = \frac{n\pi}{L}$ yields,

$$\ddot{f}_n(t) + \frac{1}{10}\dot{f}_n(t) + \frac{n\pi}{10}^2 f_n(t) = 0. \tag{9}$$

Task 1.3 Answer: Using the initial condition $u(x,0) = f(x)$ and the assumed form of the solution we find

$$f(x) = \sum_{n=1}^{N} f_n(0) sin(\frac{n\pi x}{L}) \Rightarrow f_n(0) = \frac{2}{L}\int_0^L f(x) sin(\frac{n\pi x}{L}) \mathrm{d}x. \tag{10}$$

Using the second initial condition $u_t(x,0) = 0$ and the assumed form of the solution we find

$$0 = \sum_{n=1}^{N} \dot{f}_n(0) sin(\frac{n\pi x}{L}) \Rightarrow \dot{f}_n(0) = 0. \tag{11}$$

🚩 **Task 1.4 (3 marks)** Write a function that takes n (the index of the terms in the series solution, for example $n = 3$ corresponds to $f_3(0)$) as an input and returns $f_n(0)$. You might solve this part using different tools: 1) `sympy` module or 2) direct numerical calculation using `scipy.integrate` (see the documentation here). Make sure your final result is a numerical real value (not symbolic expressions). Remember from your labs that you can use `.N()` method to evaluate a sympy expression (if you decide to use `sympy`, which is not the only way to complete this task).

Once you have written your function, use it to calculate $f_n(0)$ for $L = 10, n = 5$ and the given initial condition $u(x, t = 0) = \exp\left(\sin^2\left(\frac{2\pi x}{L}\right)\right) - 1$.

In [ ]:
```
#Task 1.4

import sympy as sym
import numpy as np
import scipy.integrate as integrate

#Set up
L = 10

def f(x):
   '''Function that returns the value of our initial condition f(x)'''

   return np.exp((np.sin(((2*np.pi*x)/L)))**2)-1

def calc_fn_0(k):
   '''Function that returns f_k(0) which is defined the same as f_n(0) for n=k abov

   return (2/L)*integrate.quad(lambda x: f(x)*np.sin(((k*np.pi*x)/L)),0,L)[0]

#Display result
print("f_5(0) = "+str(calc_fn_0(5)))
```

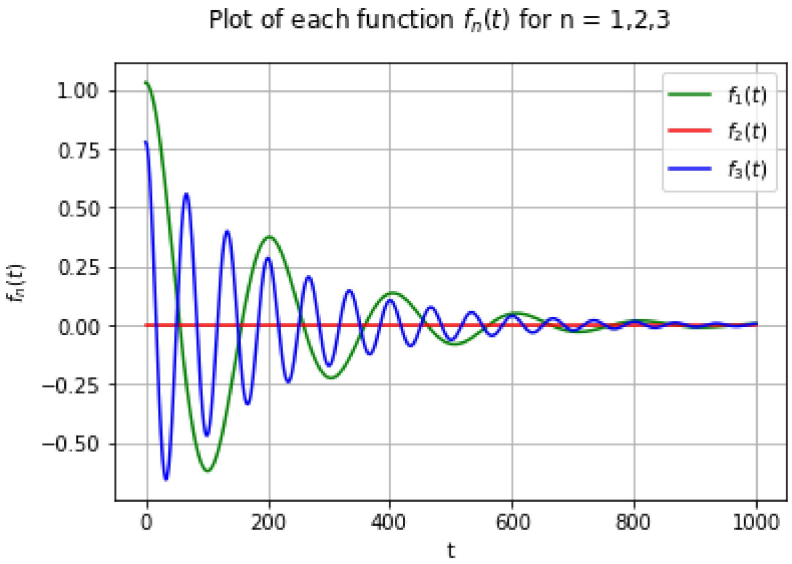
```
f_5(0) = -0.4264022028513228
```

🚩 **Task 1.5 (4 marks)** Write a function to solve the ODEs that you derived in 'Task 1.2' for each $f_n(t)$. Your function takes n (the index of terms in the solution series) and `t_vec` (the vector of times at which you want to have the solution). Then, it calculates the corresponding initial conditions (f_n and $\frac{df_n}{dt}$ at $t = 0$), solves the associated ODE and returns the solution at each point in `t_vec`. Use `odeint` to solve the ODE numerically. You may also use the function you wrote in the previous task.

Use the function to plot $f_1(t)$, $f_2(t)$ and $f_3(t)$ for `t_vec = np.linspace(0, 100, 1001)`.

In [ ]:
```
#Task 1.5

import matplotlib.pyplot as plt
from scipy.integrate import odeint, quad
# and this is the line in case the figure does not show up!
%matplotlib inline

#Set up
nu = .1
c = 1
L = 10

def dF_dt(F, t, k):
    '''
    this function defines the RHS of the system of ODEs for f_n's
    ----------------------------------------------------
    '''
    u, v = F
    return [v, -(((c*k*np.pi)/L)**2)*u-nu*v]


def fn_solve(k, t_vec):
    '''
    This function derives the solution of f_k(t) for all the times in 't_vec'
    'k' is given as an input and marks the index of the term in series solution
    '''
    #Find initial conditions
    F0 = [calc_fn_0(k), 0]
    #Solve the ODE system using odeint
    Fsol = odeint(dF_dt, F0, t_vec,(k,))
    #return only the u=f_n(t) solutions as we do not need the v=f'_n(t) solutions
    return [Fsol[i][0] for i in range(len(Fsol))]

# Plot the u=f_n(t) coordinates of the solution
t_vec = np.linspace(0,100,1001)
col = ['b','g','r']
for i in range(1,4):
    #Plot each f_n(t) for n = 1,2,3
    Fsol = fn_solve(i, t_vec)
    plt.plot(Fsol, col[i%len(col)], label='$f_'+str(i)+'(t)$')
plt.legend(loc='best')
plt.suptitle('Plot of each function $f_n(t)$ for n = 1,2,3')
```

```
plt.xlabel('t')
plt.ylabel('$f_n(t)$')
plt.grid()
```



Task 1.6 (4 marks) Write a function that takes N (the total number of terms in the sine series of the solution), `t_vec` and `x_vec` (which is the vector of points in space) as inputs and returns the solution $u(x,t)$ of the PDE at the times `t_vec` and points `x_vec`. Your function output should be a $N_T \times N_X$ numpy array, where N_T is the number of points in time (i.e. the size of `t_vec`) and N_X is the number of points in space (i.e. the size of `x_vec`). Make good use of the functions you already wrote in the previous tasks.

With this function, find the solution for

`x_vec = np.linspace(0, L, 201)`

`t_vec = np.linspace(0, 100, 1001)`

`N =7`

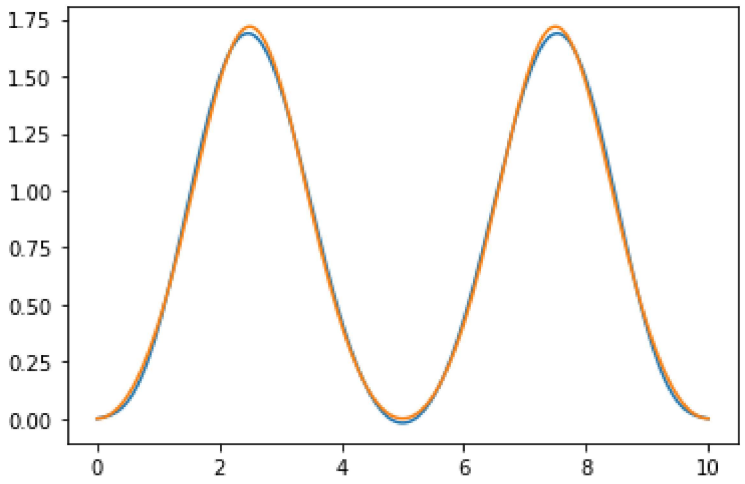
Make an animation to show how $u(x,t)$ changes in time (each frame of animation is the solution at all spatial points but for one specific point in time)

In [ ]:
```
#Task 1.6

from IPython.display import display_latex
import matplotlib.animation as animation
import matplotlib.pyplot as plt
import numpy as np

def waveq_dam_solve(N, t_vec, x_vec):
    '''Function that outputs a 2d array of solutions for (x,t) points in t_vec and

    #Set up empty array
    sols = np.zeros((np.size(t_vec),np.size(x_vec)))
    #Find our f_n(t)s
```

```
    fn_sols = [fn_solve(k, t_vec) for k in range(1,N+1)]
    #Find the sin function values
    sin_ln = [np.sin((k*np.pi*x_vec)/L) for k in range(1,N+1)]
    #Using the assumed form of the solution sum and find each u(x,t) add to array
    for t in range(np.size(t_vec)):
        for x in range(np.size(x_vec)):
            for n in range(N):
                sols[t,x] += fn_sols[n][t]*sin_ln[n][x]

    return sols

#Set up
N = 7
t_vec = np.linspace(0, 100, 1001)
x_vec = np.linspace(0, L, 201)

#Find solutions
u_x_t = waveq_dam_solve(N,t_vec,x_vec)
```

In [ ]:
```
# Assuming the PDE's solution is stored in u_x_t[:,:],
# compare the approx. solution at t=0 with the initial condition
plt.plot(x_vec, u_x_t[0,:])
plt.plot(x_vec,np.exp((np.sin(2*np.pi*x_vec/L))**2)-1)
```

Out[ ]:
```
[<matplotlib.lines.Line2D at 0x1e373b6e3d0>]
```

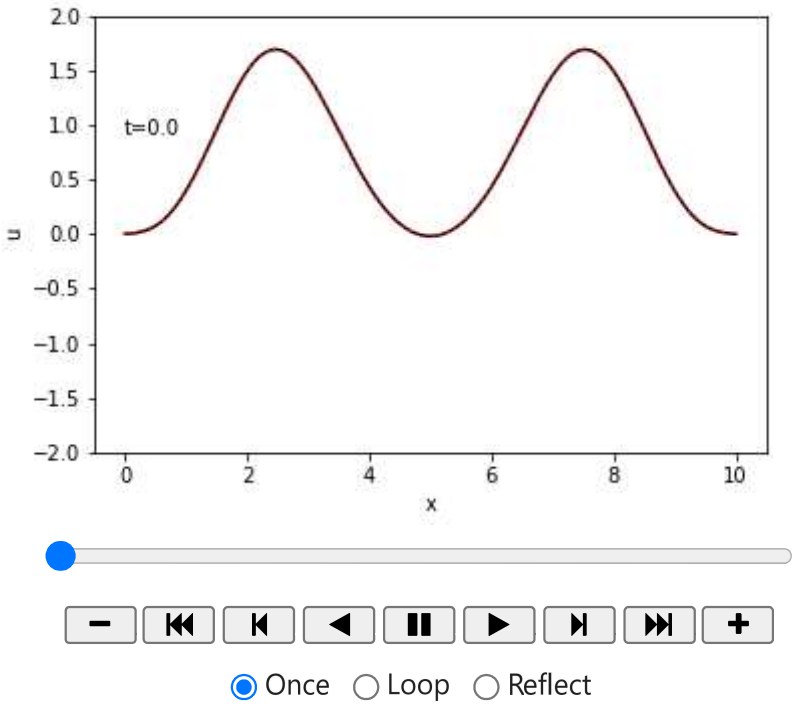


In [ ]:
```
import matplotlib.animation as animation

'''Plotting and animation as done in Lab 4'''

fig, ax = plt.subplots()

# set up the initial frame
line, = ax.plot(x_vec, u_x_t[0,:], 'k-')
plt.plot(x_vec,u_x_t[0,:],'r:')
plt.xlabel('x')
plt.ylabel('u')
plt.ylim(-2,2)
plt.close()

# add an annotation showing the time (this will be updated in each frame)
```

```
txt = ax.text(0, 0.9, 't=0')

def init():
    line.set_ydata(u_x_t[0,:])
    return line,

def animate(i):
    line.set_ydata(u_x_t[i,:])  # update the data
    txt.set_text('t='+str((t_vec[1]-t_vec[0])*i)) # update the annotation
    return line, txt

#Produce an animation for 0<=t<=100.
ani = animation.FuncAnimation(fig, animate, np.arange(0, np.size(t_vec)), init_func
                              interval=10, blit=True, repeat=False)
```

In [ ]:
```
#Create animation
from IPython.display import HTML
HTML(ani.to_jshtml())
```

Out[ ]:



Once Loop Reflect

In [ ]:
```
#Save animation
ani.save('animation1.6.gif', writer='Pillow ', fps=20)
```

```
MovieWriter Pillow  unavailable; using Pillow instead.
```

Task 1.7 (2 marks) Examine how the solution changes for different values of N. Consider $N = 3$, $N = 7$ and $N = 11$. Show the animation of the solution for all the three cases in one plot and compare them with each other. Discuss and analyse your results (for example, are three terms enough to get an acceptable solution? What about seven?)

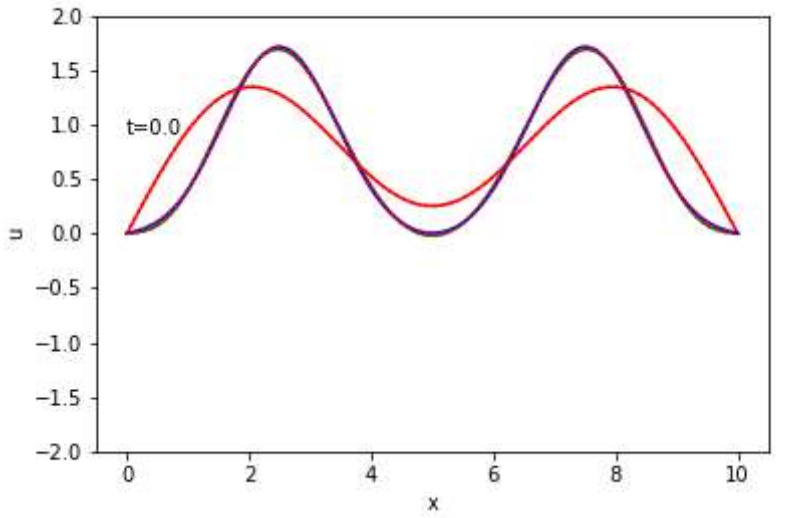
In [ ]:
```
#Task 1.7

#Finding our solutions for different N
```

```
t_vec = np.linspace(0, 100, 1001)
x_vec = np.linspace(0, L, 201)
u_x_t3 = waveq_dam_solve(3,t_vec,x_vec)
u_x_t7 = waveq_dam_solve(7,t_vec,x_vec)
u_x_t11 = waveq_dam_solve(11,t_vec,x_vec)

'''Plotting and animation as done in Lab 4'''

plt.rcParams['animation.embed_limit'] = 2**128 #Increase the max size of animation
fig, ax = plt.subplots()

# set up the initial frame
line3, = ax.plot(x_vec, u_x_t3[0,:], 'r')
line7, = ax.plot(x_vec, u_x_t7[0,:], 'g')
line11, = ax.plot(x_vec, u_x_t11[0,:], 'b')
plt.plot(x_vec,u_x_t3[0,:],'r:')
plt.plot(x_vec,u_x_t7[0,:],'r:')
plt.plot(x_vec,u_x_t11[0,:],'r:')
plt.xlabel('x')
plt.ylabel('u')
plt.ylim(-2,2)
plt.close()

# add an annotation showing the time (this will be updated in each frame)
txt = ax.text(0, 0.9, 't=0')

def init():
    line3.set_ydata(u_x_t3[0,:])
    line7.set_ydata(u_x_t7[0,:])
    line11.set_ydata(u_x_t11[0,:])
    return line,

def animate(i):
    line3.set_ydata(u_x_t3[i,:])
    line7.set_ydata(u_x_t7[i,:])
    line11.set_ydata(u_x_t11[i,:])  # update the data
    txt.set_text('t='+str((t_vec[1]-t_vec[0])*i)) # update the annotation
    return line, txt

#Produce an animation for 0<=t<=100.
aniN = animation.FuncAnimation(fig, animate, np.arange(0, np.size(t_vec)), init_fun
                               interval=10, blit=True, repeat=False)
```

In [ ]:

```
#Create animation
from IPython.display import HTML
HTML(aniN.to_jshtml())
```

Out[ ]:



Once Loop Reflect

In [ ]:
```
#Save animation
aniN.save('animation1.7.gif', writer='Pillow ', fps=20)
```

```
MovieWriter Pillow  unavailable; using Pillow instead.
```

We see in the animation above that for $N = 7$ and $N = 11$ that they overlap throughout the whole animation. So we can argue that for our desired animations the solution obtained from $N = 7$ is acceptable since for larger N the animation does not change much. However, for $N = 3$ we can see that even the inital condition itself is far from the exact initial condition. This tells us that $N = 3$ does not give us what we would consider an acceptable solution.

Task 1.8 (1 marks) Change the damping parameter to $\nu = 1$, and repeat the 'task 1.6' (for $N = 7$ and keep all other parameters the same). In few sentences, explain the changes caused by increasing the damping.

Task 1.8 Answer: When the damping parameter is increased in this way we see that our wave moves much less and settles to the line $u(x, t) = 0$ much quicker reaching this around t=50 rather than around t=100 when the damping is at 0.1.

Part 2: Lorenz system

The Lorenz system is given by the coupled set of ODEs

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\sigma x + \sigma y, \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y, \qquad \frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z, \tag{12}$$

where (ρ, σ, β) are system parameters. Lorenz derived these equation as a simplified mathematical model for atmospheric convection, which is an aspect of weather. While the Lorenz system is deterministic, you will find in this problem that in certain sets of parameters there is sensitive dependence of the solution on the initial condition. Assuming this system is a representation of weather, small errors invariably arise (e.g. numerical discretisation, measurements inaccuracy), grow and affect long term solutions (cf. loss of predictability in weather forecasting). However, there are statistical properties (cf. the climatology) that may be robust and may be described accordingly.

Case of stable solution

Consider the parameters $\sigma = 8$, $\beta = 2$ and $\rho = 12$.

🚩 **Task 2.1 (1 mark)** Find all the critical points of this system.

Task 2.1 Answer: To find the critical points we simply need to set $\frac{dx}{dt} = \frac{dy}{dt} = \frac{dz}{dt} = 0$ and solve for points (x, y, z). Setting our derivatives to zero and setting our parameters as required we find,

$$\frac{dx}{dt} = \sigma(y - x) \Rightarrow 0 = 8(x - y) \Rightarrow x = y \tag{13}$$

Then, the next equation gives,

$$\frac{dy}{dt} = x(\rho - z) - y \Rightarrow x(12 - z) - y = 0 \Rightarrow x(11 - z) = 0 \tag{14}$$

Where we used the first equation which gives $x = y$. From this we know that either $x = 0$ or $z = 11$. We use the next equation to find our final critical points for these two cases.

$$\frac{dz}{dt} = xy - \beta z \Rightarrow xy = 2z \Rightarrow x^2 = 2z \tag{15}$$

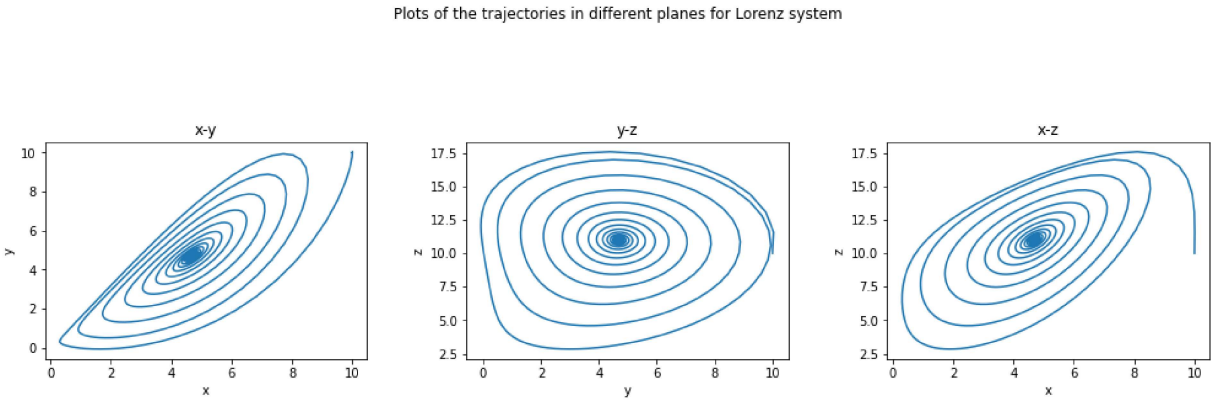
If $x = 0$ then $y = 0$ and by the last equation $z = 0$. If $z = 11$ then $x = y = \pm\sqrt{22}$. So our critical points are:

$(0, 0, 0)$
, $(\sqrt{22}, \sqrt{22}, 11)$, $(-\sqrt{22}, -\sqrt{22}, 11)$.

🚩 **Task 2.2 (2 marks)** Using a numerical method of your choice, find the solution starting from $x(t = 0) = y(t = 0) = z(t = 0) = 10$ up to $t = 20$. Plot the solution in *x-y*, *y-z* and *x-z* planes (so you need three plots). Where does this initial condition end up in? Can you express the exact location of this solution as $t \to \infty$? Make sure your timestep is small enough to have robust results. You can check this by running your code for several different timesteps.

In [ ]:
```
# Task 2.2

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

#Set up
rho = 12.0
sigma = 8.0
beta = 2.0

init = [10,10,10]

t= np.linspace(0,20,1001)

def dV_dt(V, t):
    '''
    this function defines the RHS of the system where V' = (dx/dt,dy/dt,dz/dt)
    ----------------------------------------------------
    '''
    x,y,z=V
    return [sigma*(y-x), x*(rho-z)-y,x*y-beta*z]

#Solve the ODE system using odeint
Vsol = odeint(dV_dt,init,t)

#Plots
fig, ax = plt.subplots(1,3, figsize = (15,5))
fig.suptitle('Plots of the trajectories in different planes for Lorenz system')
fig.tight_layout(pad=5.0)
ax[0].plot(Vsol[:,0],Vsol[:,1])
ax[0].set_title("x-y")
ax[0].set_xlabel("x")
ax[0].set_ylabel("y")
ax[1].plot(Vsol[:,1],Vsol[:,2])
ax[1].set_title("y-z")
ax[1].set_xlabel("y")
ax[1].set_ylabel("z")
ax[2].plot(Vsol[:,0],Vsol[:,2])
ax[2].set_title("x-z")
ax[2].set_xlabel("x")
ax[2].set_ylabel("z")
```

Out[ ]:
```
Text(698.9485294117648, 0.5, 'z')
```



Since these initial conditions start close enough to the critical point $(\sqrt{22}, \sqrt{22}, 11)$, the exact location of this solution as $t \to \infty$ will be this critical point $(\sqrt{22}, \sqrt{22}, 11)$. We can see this as in each of the three plots for the *x-y*, *y-z* and *x-z*plane we see these solutions approach the critical point in each plane.

🚩 **Task 2.3 (4 marks)** Consider two different numerical methods: forward Euler and Ruge-kutta (4th order) methods. Set the initial condition to $x = y = z = 100$ and the final time to 0.3. The Ruge-kutta method with timestep h=10^{-6} is accurate enough that **we can regard it as an exact solution**. Using this *estimate numerically* the **global error** for variable x and plot it in log-scale as a function of timestep. In so doing, consider the values `[0.002, 0.001, .0005, 0.0002, 0.0001]` for timesteps.

Note that the order of error can be estimated from the slope of plots in log-scales. If a function is proportional to h^6, then in log-scale it looks linear and have the slope of 6.

What is the slope of global error vs timestep for each method? Why do you get these slopes? You can estimate the slopes by drawing linear guidelines. For example, if you draw h^6 and your curve is close to this guideline, then you can conclude that the slope is close to 6. You need to shift your guidelines up and down to bring them closer to the results of numerical solutions. Are the results different than the slopes you expect for the local error as a function of timestep? Can you reason why?

(Remember that you have already implemented forward Euler and Ruge-kutta in your lab notebooks. There is also something called "copy-paste".)

In [ ]:
```
#Task 2.3

from pandas import DataFrame
import math

'''Code copied from labs'''

def timesteps(start, stop, h):
    num_steps = math.ceil((stop - start)/h)
    return np.linspace(start, start+num_steps*h, num_steps+1)

def Euler_step(func, start, stop, h, ics):
```

```
    times = timesteps(start, stop, h)
    values = ode_Euler(func, times, ics)
    return values, times

def RK4_step(func, start, stop, h, ics):
    times = timesteps(start, stop, h)
    values = ode_RK4(func, times, ics)
    return values, times

# Euler scheme
def ode_Euler(func, times, y0):
    '''
    integrates the system of y' = func(y, t) using forward Euler method
    for the time steps in times and given initial condition y0
    ----------------------------------------------------------
    inputs:
        func: the RHS function in the system of ODE
        times: the points in time (or the span of independent variable in ODE)
        y0: initial condition (make sure the dimension of y0 and func are the same)
    output:
        y: the solution of ODE.
        Each row in the solution array y corresponds to a value returned in column
    '''
    # guess why I put these two lines here?
    times = np.array(times)
    y0 = np.array(y0)
    n = y0.size       # the dimension of ODE
    nT = times.size   # the number of time steps
    y = np.zeros([nT,n])
    y[0, :] = y0
    # loop for timesteps
    for k in range(nT-1):
        y[k+1, :] = y[k, :] + (times[k+1]-times[k])*func(y[k, :], times[k])
    return y

# Runge-Kutta 4 scheme

def ode_RK4(func,times,y0):
    '''
    integrates the system of y' = func(y, t) using Runge-Kutta 4th order
    for the time steps in times and given initial condition y0
    ----------------------------------------------------------
    inputs:
        func: the RHS function in the system of ODE
        times: the points in time (or the span of independent variable in ODE)
        y0: initial condition (make sure the dimension of y0 and func are the same)
    output:
        y: the solution of ODE.
        Each row in the solution array y corresponds to a value returned in column
    '''
    times = np.array(times)
    y0 = np.array(y0)
    n = y0.size       # the dimension of ODE
    nT = times.size   # the number of time steps
    y = np.zeros([nT,n])
    y[0, :] = y0
```

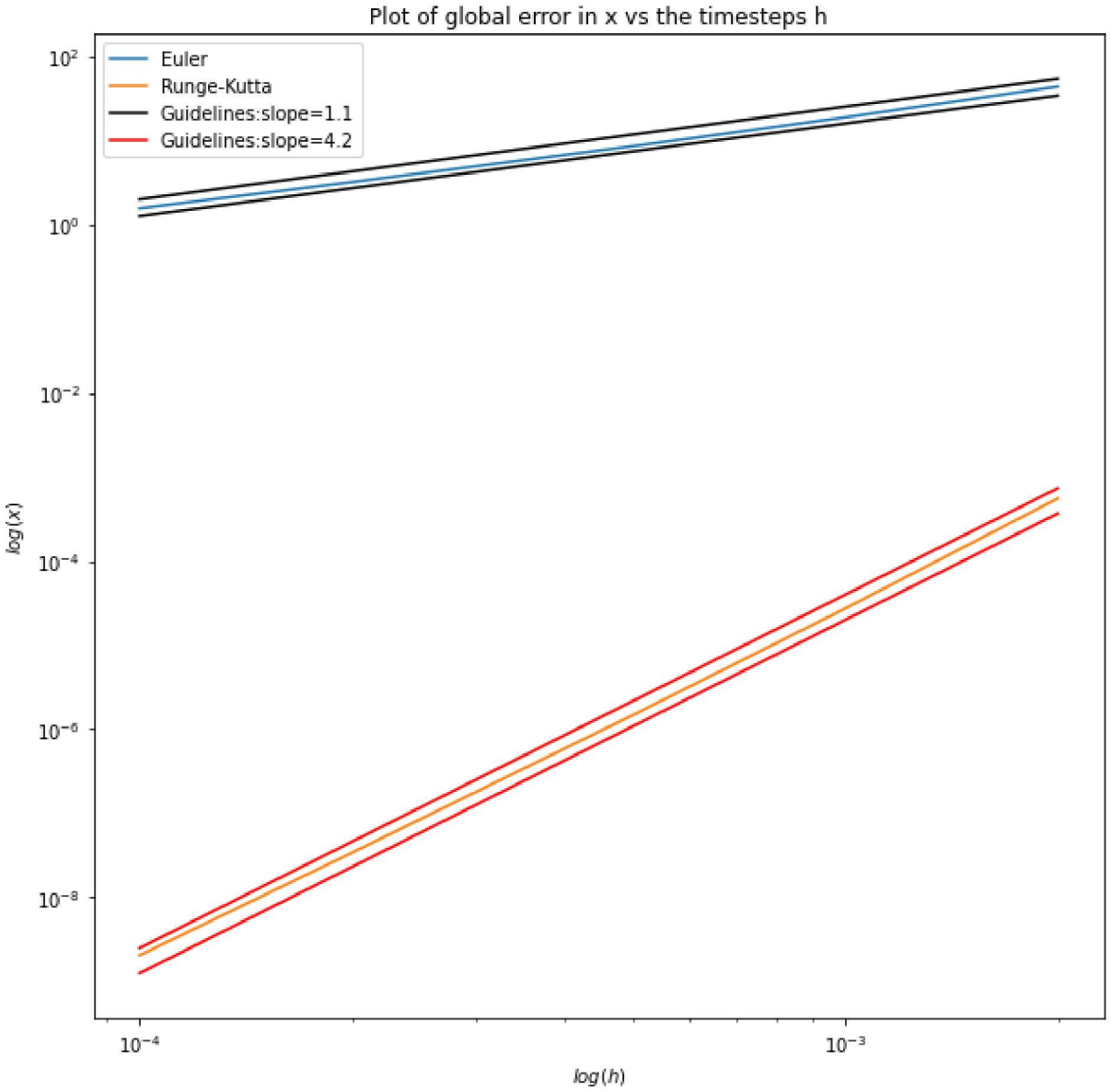
```
    for k in range(nT-1):
        dt = times[k+1] - times[k]
        f1 = func(y[k,:], times[k])
        f2 = func(y[k,:] + 0.5*dt*f1, times[k] + 0.5*dt)
        f3 = func(y[k,:] + 0.5*dt*f2, times[k] + 0.5*dt)
        f4 = func(y[k,:] + dt*f3, times[k] + dt)
        y[k+1,:] = y[k,:]+dt/6*(f1+2*f2+2*f3+f4)
    return y

'''End'''
```

Out[ ]: 'End'

In [ ]:
```
#Set up
init = [100,100,100]
h = 0.000001
test_h = [0.002,0.001,0.0005,0.0002,0.0001]

def dy_dt(y, t):
    '''Function that defines the RHS of the Lorenz system'''
    return np.array([sigma*(y[1]-y[0]), y[0]*(rho-y[2])-y[1],y[0]*y[1]-beta*y[2]])

#Finding the exact solutions to this ODE using the Runge-Kutta 4th order method wit
df_exact = RK4_step(dy_dt, 0, 0.3, h, init)
df_exactxyz = DataFrame(data = df_exact[0], index = np.round(df_exact[1],6), column

'''Find errors'''
#Lists that hold errors for each method for each timestep
err_Euler = []
err_RK4 =[]
for i in range(len(test_h)):
    df_Euler = Euler_step(dy_dt, 0, 0.3, test_h[i], init) #Solve using euler
    df_eulerxyz = DataFrame(data = df_Euler[0], index = np.round(df_Euler[1],6), co
    err_Euler.append(df_eulerxyz.subtract(df_exactxyz.filter(items=np.round(timeste
    df_RK4 = RK4_step(dy_dt, 0, 0.3, test_h[i], init) #Next lines same as the three
    df_rkxyz = DataFrame(data = df_RK4[0], index = np.round(df_RK4[1],6), columns=[
    err_RK4.append(df_rkxyz.subtract(df_exactxyz.filter(items=np.round(timesteps(0,

'''plot global error of x vals as a function of the different timesteps used'''
fig = plt.figure(figsize = (10,10))
ax = fig.add_subplot()

#Timesteps to plot global error against
times_h = np.array([0.002,0.001,0.0005,0.0002,0.0001])

#Setting log scale
ax.set_yscale('log')
ax.set_xscale('log')

#Labels
ax.set_title("Plot of global error in x vs the timesteps h")
ax.set_xlabel('$log(h)$')
ax.set_ylabel('$log(x)$')

#Plotting of the global error from our error lists we found
ax.plot(times_h, np.array([err_Euler[i]['x'].filter(items=[0.3], axis=0) for i in r
```

```
ax.plot(times_h, np.array([err_RK4[i]['x'].filter(items=[0.3], axis=0) for i in ran

#Rough guidelines to estimate slope
ax.plot(times_h, (10**4.5)*times_h**1.1, color = 'k',label="Guidelines:slope=1.1")
ax.plot(times_h, (10**4.7)*times_h**1.1, color = 'k')
ax.plot(times_h, (10**8.2)*times_h**4.2, color = 'r',label="Guidelines:slope=4.2")
ax.plot(times_h, (10**7.9)*times_h**4.2, color = 'r')

ax.legend()
```

Out[ ]: <matplotlib.legend.Legend at 0x1e3242f2f70>



The slope in log scale of the global error vs timestep for the Euler method is close 1.1 whereas the error for the Runge-Kutta method is close to 4.2. These slopes tell us that the error for the Euler method changes less when we change the timestep compared to that of the Runge-Kutta method. This being said we still see that for these timesteps the error of the Runge-Kutta method is smaller than that of the euler method.

Case of chaotic behaviour and strange attractor

Consider the set of parameters $\sigma = 10$, $\beta = 8/3$ and $\rho = 28$ and stick to Runge-Kutta scheme with timestep of $h = 10^{-4}$ for this part.

Task 2.4 (2 marks) Draw the three-dimensional phase portraits using $x = y = z = 1$ as initial condition. You can use the script below to plot 3D curves.

Task 2.5 (3 marks) Change the initial condition as $x = 1 + \epsilon, y = 1, z = 1$, where ϵ is 10^{-1}, 10^{-5} and 10^{-12}. Draw the phase portrait for each initial condition and compare it to what you got for Task 2.4. Do they look different? Representing the outputs of these new solutions (with new initial conditions) with x_1, y_1 and z_1 and the solution starting from $x = y = z = 1$ with x^*, y^* and z^*, compute a measure of difference between these solutions given by

$$e(t) = \sqrt{(x^*(t) - x_1(t))^2 + (y^*(t) - y_1(t))^2 + (z^*(t) - z_1(t))^2}.$$

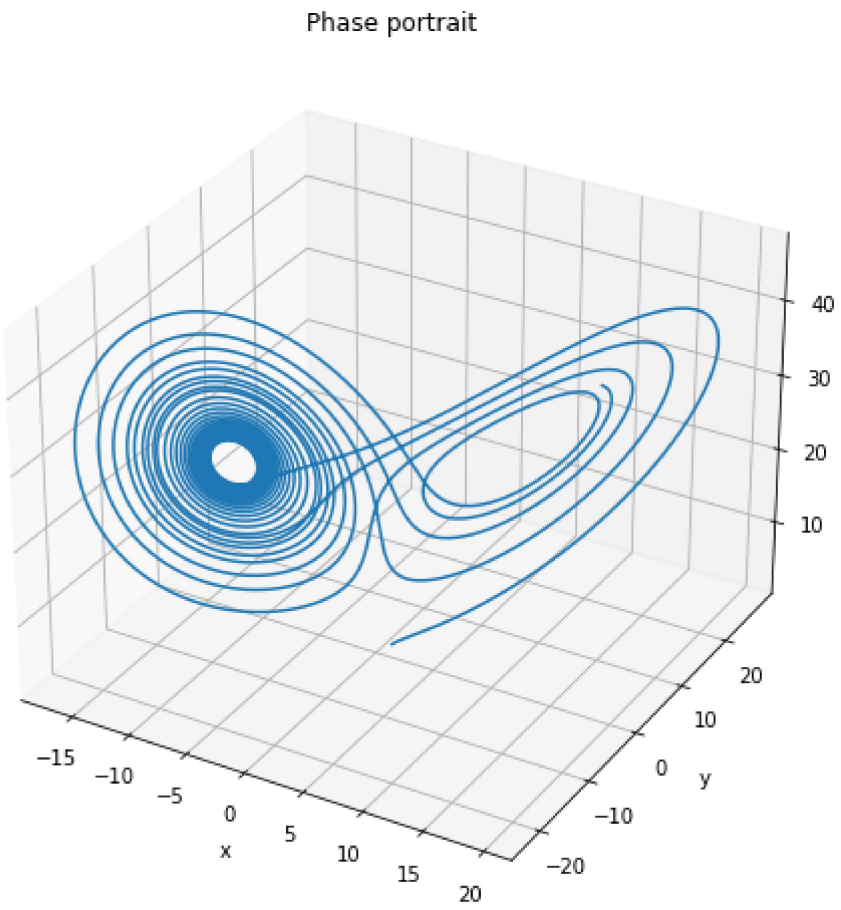
Plot the time-series diagrams of $e(t)$ for three different values of ϵ. From the behaviour of $e(t)$ answer the following questions:

- If two initial conditions are very close, do their corresponding trajectories stay close to each other in time?
- Do you see an upper bound for $e(t)$? What does this imply?

The moral of the story is that the exact realisations (the "weather") can have sensitive dependence on initial conditions, and in reality errors always exist, so loss of accuracy in weather prediction is fairly rapid. However there are gross features that remain roughly invariant and can be described, for example the butterfly attractor that you see in your results (the "climate").

In [ ]:
```
#Task 2.4

from mpl_toolkits.mplot3d import Axes3D


#Set up
sigma = 10
beta = 8/3
rho = 28
init = [1,1,1]
h = 0.0001

#Find solutions using the Runge-Kutta scheme with timstep h=10^-4
df_RK4 = RK4_step(dy_dt, 0, 20, h, init)
df_rkxyz = DataFrame(data = df_RK4[0], index = np.round(df_RK4[1],3), columns=['x',


#Plot 3d
def plot3d(x,y,z,title,size):
```

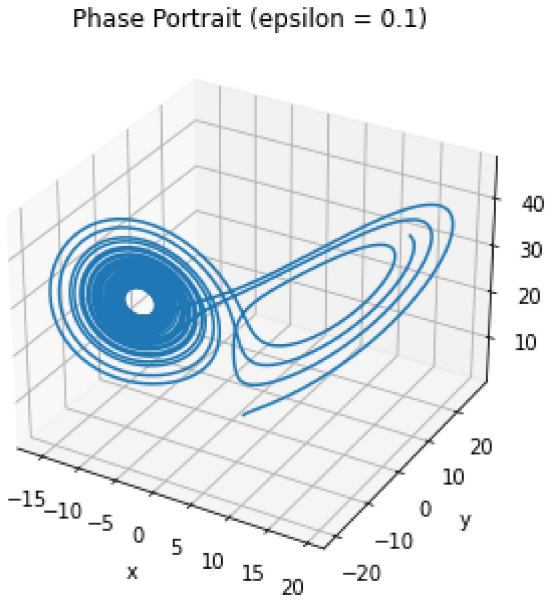
```
    '''Function that plots a 3d curve given the x,y,z coords for each t value. Can
    fig = plt.figure(figsize = size)
    ax = plt.subplot(projection = '3d')
    ax.plot3D(x, y, z) # x and y and z are the time series of the solution

    #set title
    ax.set_title(title)

    # Set axes label
    ax.set_xlabel('x', labelpad=5)
    ax.set_ylabel('y', labelpad=5)
    ax.set_zlabel('z', labelpad=5)
    plt.draw()
    plt.show()

#Use plot3d function to plot phase portrait
plot3d(df_rkxyz['x'],df_rkxyz['y'],df_rkxyz['z'],"Phase portrait",(8,8))
```

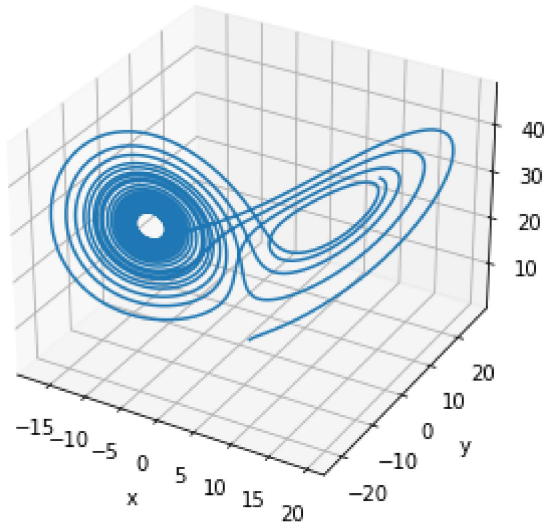


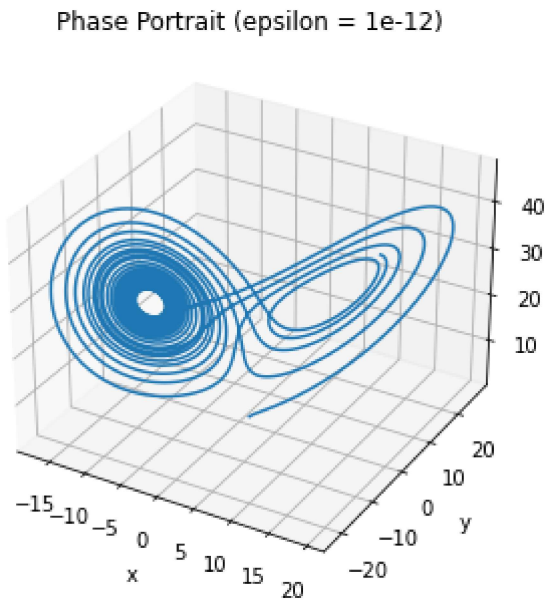
In [ ]:

```
#Task 2.5

#Set up
eps = [10**-1,10**-5,10**-12]
init = [1,1,1]

#Draw phase portrait for each epsilon
for e in eps:

    #Solve with new initial condition
```

```
    df_eps = RK4_step(dy_dt, 0, 20, h, [j+e for j in init])
    df_epsxyz = DataFrame(data = df_eps[0], index = np.round(df_eps[1],3), columns=

    #Plot
    plot3d(df_epsxyz['x'],df_epsxyz['y'],df_epsxyz['z'], "Phase Portrait (epsilon =
```

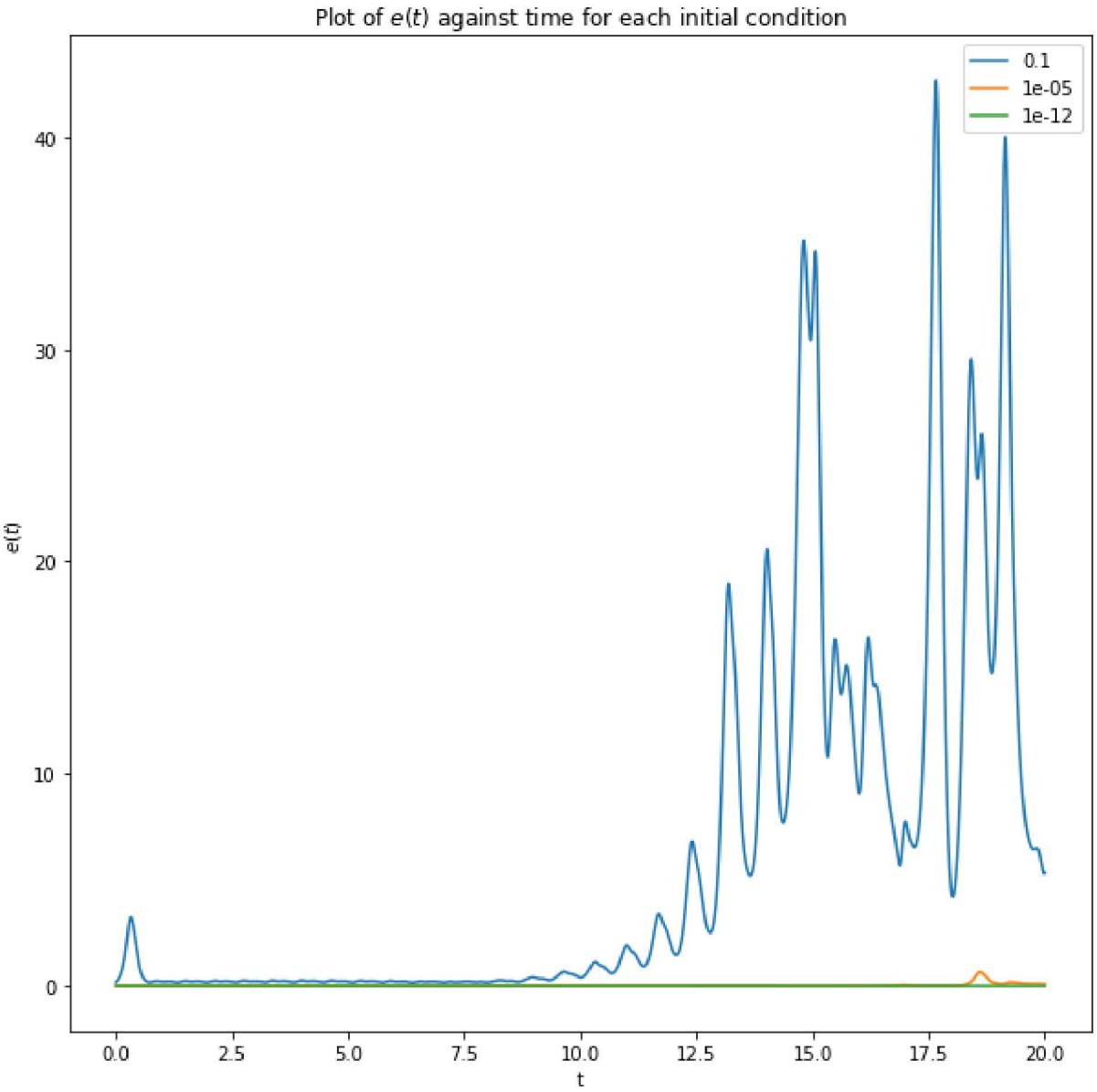






We see here that the first phase portrait for $\epsilon = 10^{-1}$ looks different to the other two. Whereas the other two look very similar to that of the phase portrait for the original initial conditions without adding the ϵ. This tells us that closer initial conditions will give closer trajectories.

In [ ]:
```
#Original
df_RK4 = RK4_step(dy_dt, 0, 20, h, init)
df_rkxyz = DataFrame(data = df_RK4[0], index = np.round(df_RK4[1],3), columns=['x',

#Set up plots
fig = plt.figure(figsize = (10,10))
ax = plt.subplot()

#Find our e(t) for each epsilon
for e in eps:

    #Solve with new initial conditions
    df_eps = RK4_step(dy_dt, 0, 20, h, [j+e for j in init])
    df_epsxyz = DataFrame(data = df_eps[0], index = np.round(df_eps[1],3), columns=

    #Find our e(t)
    et = np.sqrt((df_rkxyz['x']-df_epsxyz['x'])**2+(df_rkxyz['y']-df_epsxyz['y'])**

    #Plot e(t) for each epsilon
    ax.set_xlabel('t')
    ax.set_ylabel('$e(t)$')
    ax.set_title("Plot of $e(t)$ against time for each initial condition")
    ax.plot(timesteps(0,20,h),et, label = str(e))
    ax.legend()
```



Clearly we can see that when initial conditions are very close together their trajectories also stay close to each other in time. However, we also see that for the initial conditions with difference of $\epsilon = 10^{-1}$ the $e(t)$ after $t = 10$ is very high meaning that even this difference in initial conditions is too large for the trajectories to stay close to each other in time. We cannot clearly see an upper bound for $e(t)$ which tells that trajectories can depend heavily on the initial conditions.