# Final project for HDEq Skills:

```
Due: Friday 3rd December at midnight
```

## 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. Use comments to make your scripts easier to follow and understand.

This assignment contributes 12% of the course mark. It will be marked out of 30, comprising the following:
**Part 1: 18 marks**
**Part 2: 12 marks**

You are welcome to use previous lab/assignment material.

For the presentation and coding style:

- The plots should be in a good size, properly labelled and markers/lines wisely chosen. It might not be a bad 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

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

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

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

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 $\lambda_n$ 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).

First, we let $v$ be the given Galerkin approximation of $u$.

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

first, we require that $v(0,t) = v(L,t) = 0$ for all t and functions $f_n$. This gives us our eigenvalues (we choose the given finitely many eigenvalues out of the infinite set of eigenvalues to maximise the accuracy of our approximation):

$$\lambda_n = \frac{n\pi}{L} \qquad \text{for} \qquad n = 1, 2 \ldots N$$

We can compute the derivatives of $v$:

$$v_{xx} = -\sum_{n=1}^{N} \lambda_n^2 f_n(t) \sin(\lambda_n x) \qquad v_t = \sum_{n=1}^{N} f_n'(t) \sin(\lambda_n x) \qquad v_{tt} = \sum_{n=1}^{N} f_n''(t) \sin(\lambda_n x)$$

So, requiring that $v$ solve the equation $v_{tt} + \nu v_t = c^2 v_{xx}$ gives:

$$0 = v_{tt} + \nu v_t - c^2 v_{xx} = \sum_{n=1}^{N} (f_n''(t) + \nu f_n'(t) + (c\lambda_n)^2 f_n(t)) \sin(\lambda_n x)$$

So, taking the inner product of both sides of the equation with $\sin(\lambda_n x)$ gives an ODE in $f_n$:

$$f_n''(t) + \nu f_n'(t) + (c\lambda_n)^2 f_n(t) = 0$$

Requiring that $v$ satisfy the initial conditions $v(x,0) = f(x)$ and $v_t(x,0) = 0$ gives:

$$\sum_{n=1}^{N} f_n(0)\sin(\lambda_n x) = f(x) \qquad \text{and} \qquad \sum_{n=1}^{N} f_n'(0)\sin(\lambda_n x) = 0$$

So, again taking the inner product with $\sin(\lambda_n x)$:

$$f_n(0) < \sin(\lambda_n x), \sin(\lambda_n x) > = < f(x), \sin(\lambda_n x) > \quad \text{and} \quad f_n'(0) = 0$$

So our initial conditions for our ODE are:

$$f_n(0) = \frac{1}{L} < f(x), \sin(\lambda_n x) > \qquad f_n'(0) = 0$$

note we continue $f$ for (x < 0) so that it is an odd function, so:

$$\frac{2}{L}\int^{L}$$

**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 [1]:
```
import sympy as sym
import numpy as np
from scipy.integrate import quad

L = 10

def f(x):
    """Helper function for the given initial condition on u(x, 0)"""
    return np.exp(np.sin(2*np.pi*x/L)**2)-1

def calc_fn_0(k):
    """Given k, returns the initial condition on f_k(0)."""
    # Compute the inner product of f and the sin term.
    integral, max_error = quad(
        lambda x: f(x)*np.sin(k*np.pi*x/L),
        0, L
    )
    # Use the forumla given above to compute f_k(0)
    return 2*integral/L

print(f"The numerical value of f_5(0) is {calc_fn_0(5)}")
```

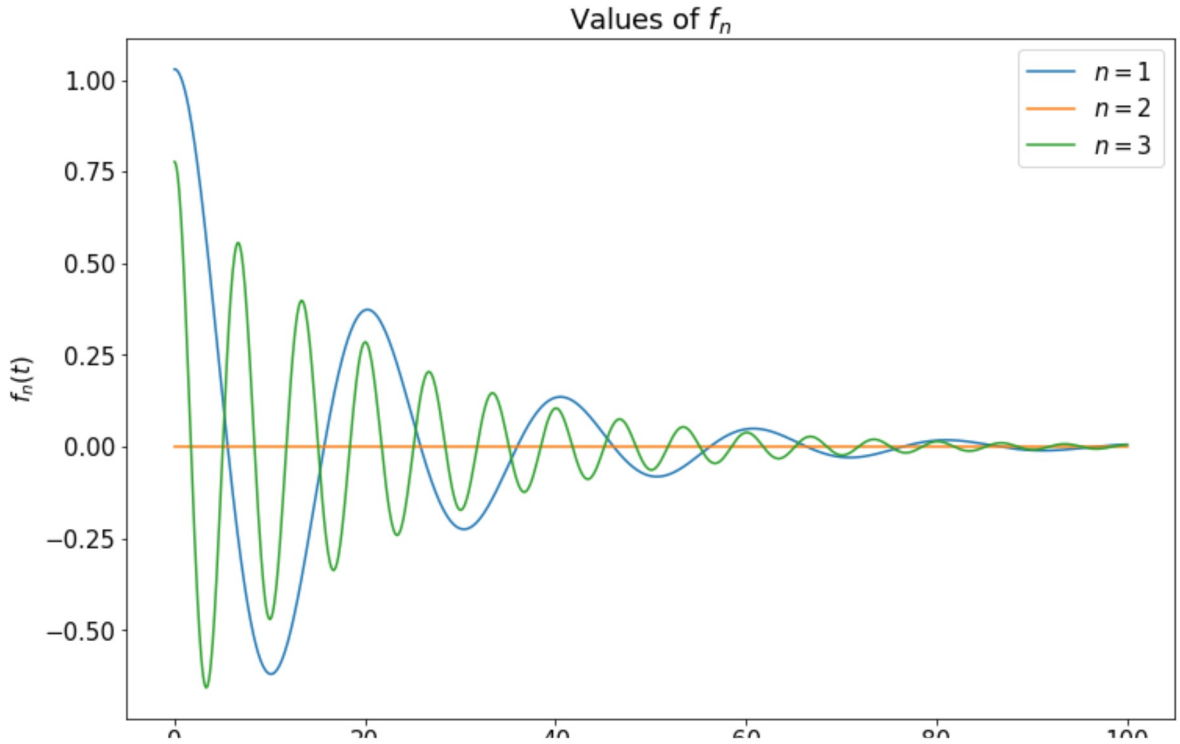
```
The numerical value of f_5(0) is -0.4264022028513228
```

**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 $\frac{df}{dt}$ 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.

In [2]:
```
import matplotlib.pyplot as plt
from scipy.integrate import odeint
# and this is the line in case the figure does not show up!
%matplotlib inline

# Set the global fontsize for matplotlib plots so that they are and more re
plt.rcParams.update({'font.size': 15})

# Define constants for ODE
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
    ----------------------------------------------------
    '''
    # Unpack f and f' from the given vector, use them to compute f''
    f, fp = F
    fpp = -fp*nu - f*(c * k*np.pi/L)**2

    # Return the time derivative of (f, f')
    return fp, fpp

def fn_solve(k, t_vec):
    '''
    This function derives the solution of f_k(t) for all the times in 't_ve
    'k' is given as an input and marks the index of the term in series solu
    '''
    # Compute the initial conditions and numerical solution to the ODE
    fn_0 = calc_fn_0(k)
    solution = odeint(dF_dt, (fn_0, 0), t_vec, args=(k,))

    # Unpack the values of f and f' from the solution and return f
    f_t, fp_t = solution.T
    return f_t


# Plot the solutions for f_n for n = 1, 2, 3

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

fig, ax = plt.subplots(figsize=(12, 8))

ax.set_title("Values of $f_n$")
ax.set_xlabel("$t$")
ax.set_ylabel("$f_n(t)$")

for n in range(1, 4):
    ax.plot(t_vec, fn_solve(n, t_vec), label=f"$n = {n}$")

ax.legend()
```

Out[2]: <matplotlib.legend.Legend at 0x21a9d149ca0>




**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

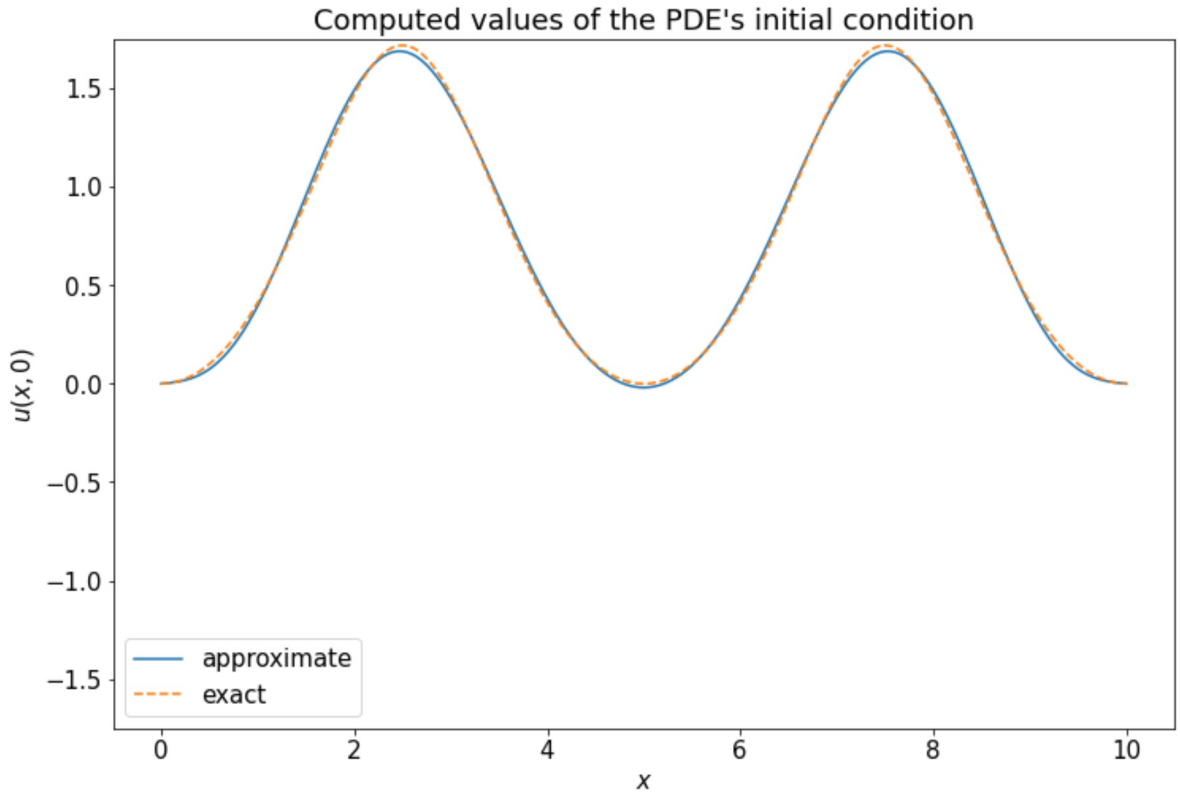
```
x_vec = np.linspace(0, L, 201)

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 [3]:
```
def waveq_dam_solve(N, t_vec, x_vec):
    """Returns a matrix u_x_t where u_x_t[i][j] = u(t_vec[i], x_vec[j]) ."
    # Construct a separate matrix for each n and then sum them elementwise.
    return sum(
        np.outer(fn_solve(n, t_vec), np.sin(n*np.pi*x_vec/L))  for n in rar
    )

# Numerically compute u(x, t) for the given values of x_vec and t_vec

x_vec = np.linspace(0, L, 201)
t_vec = np.linspace(0, 100, 1001)

u_x_t = waveq_dam_solve(7, t_vec, x_vec)
```

In [4]:
```
# Create and decorate axes

fig, ax = plt.subplots(figsize=(12, 8))

ax.set_xlabel('$x$')
ax.set_ylabel('$u(x, 0)$')
ax.set_ylim(-1.75, 1.75)
ax.set_title("Computed values of the PDE's initial condition")

# Assuming the PDE's solution is stored in u_x_t[:,:],
# compare the approx. solution at t=0 with the initial condition

line, = ax.plot(x_vec, u_x_t[0,:], label="approximate")
ax.plot(x_vec, np.exp((np.sin(2*np.pi*x_vec/L))**2)-1, "--", label="exact")

ax.legend()
```

Out[4]: <matplotlib.legend.Legend at 0x21a9e8d5f70>

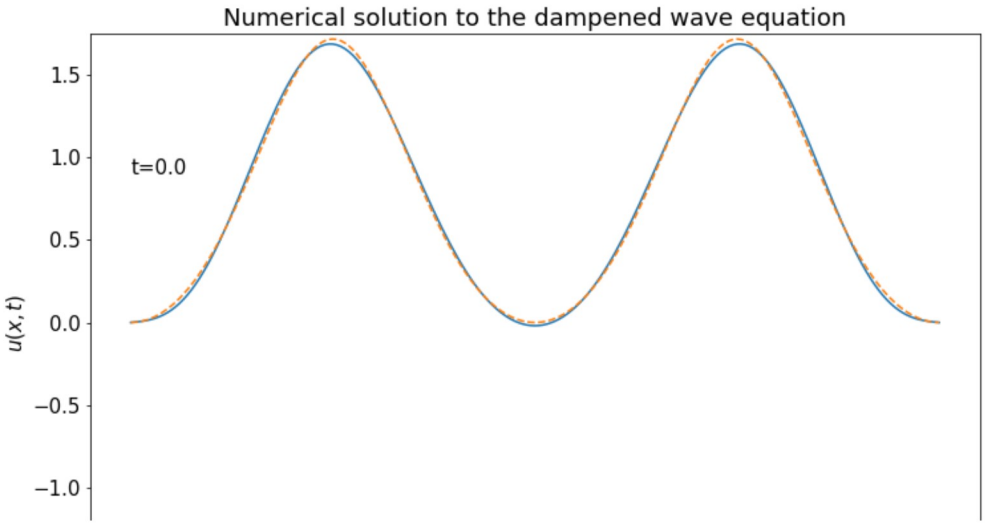


In [5]:
```
# imports for animation
import matplotlib.animation as animation

def animate_pde_solutions(lines, pde_solutions, txt, t_vec):
    """Animates the matplotlib artists in lines to follow the data in pde s
    updating txt with values from t_vec. Returns a matplotlib FuncAnimation
    def init():
        """Initialisation function for FuncAnimation."""
        # Initialise the curves for each value of N
        for line, sol in zip(lines, pde_solutions):
            line.set_ydata(sol[0,:])
        return lines

    def animate(i):
        """Function called to update the frame for each index i of the vect
        # Update the curve data for each value of N
        for line, sol in zip(lines, pde_solutions):
            line.set_ydata(sol[i,:])
        txt.set_text(f't={round(t_vec[i], 2)}') # update the annotation
        return [*lines, txt]

    return animation.FuncAnimation(fig, animate, len(t_vec)-1, init_func=ir
                                   interval=100, blit=True)
```

In [6]:
```
# import for displaying animation in jupyter notebook
from IPython.display import HTML

# Define constants for ODE
nu = .1
c = 1
L = 10

# Redecorate the axes
ax.set_xlabel('$x$')
ax.set_ylabel('$u(x, t)$')
ax.set_title("Numerical solution to the dampened wave equation")

# Change the legend labels
handles, _ = ax.get_legend_handles_labels()
ax.legend(handles, ["Solution over time", "Initial condition"], loc="lower

# Add timer text on the axes
txt = ax.text(0, 0.9, '$t=0$')

ani = animate_pde_solutions([line], [u_x_t], txt, t_vec)
HTML(ani.to_jshtml())
```

```
Animation size has reached 20993699 bytes, exceeding the limit of 20971520.
0. If you're sure you want a larger animation embedded, set the animation.e
mbed_limit rc parameter to a larger value (in MB). This and further frames
will be dropped.
```

Out[6]:



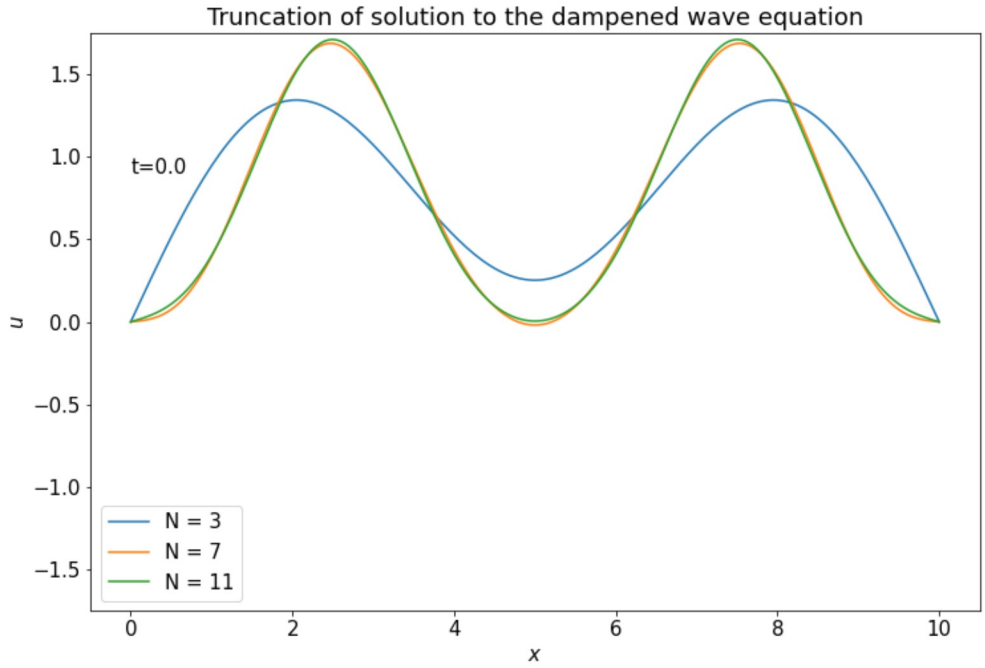

**Task 1.7 (2 marks)** Examine how the solution changes for different values of $N$. Consider $N = 3$, $N = 7$ and $N = 11$. Show the animation of the solution for all the three cases in one plot and compare them with each other. Discuss and analyse your results (for example, are three terms enough to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

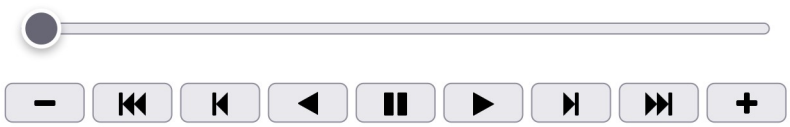
In [7]:

```
# Define constants for ODE
nu = .1
c = 1
L = 10

N_vals = [3, 7, 11]

# Generate a solution for each value of N
u_x_t_sols  = [
    waveq_dam_solve(N, t_vec, x_vec) for N in N_vals
]

# Create and decorate some axes
fig, ax = plt.subplots(figsize=(12, 8))
ax.set_xlabel('$x$')
ax.set_ylabel('$u$')
ax.set_ylim(-1.75, 1.75)
ax.set_title("Truncation of solution to the dampened wave equation")

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

# Compute and plot the initial conditions for each N
u_x_t_sols = []
lines = []

for N in N_vals:
    u_x_t = waveq_dam_solve(N, t_vec, x_vec)
    u_x_t_sols.append(u_x_t)
    line, = ax.plot(x_vec, u_x_t[0,:], label=f"N = {N}")
    lines.append(line)

plt.legend()
plt.close()


ani = animate_pde_solutions(lines, u_x_t_sols, txt, t_vec)
HTML(ani.to_jshtml())
```

```
Animation size has reached 20979348 bytes, exceeding the limit of 20971520.
0. If you're sure you want a larger animation embedded, set the animation.e
mbed_limit rc parameter to a larger value (in MB). This and further frames
will be dropped.
```

Out[7]:



Once Loop Reflect

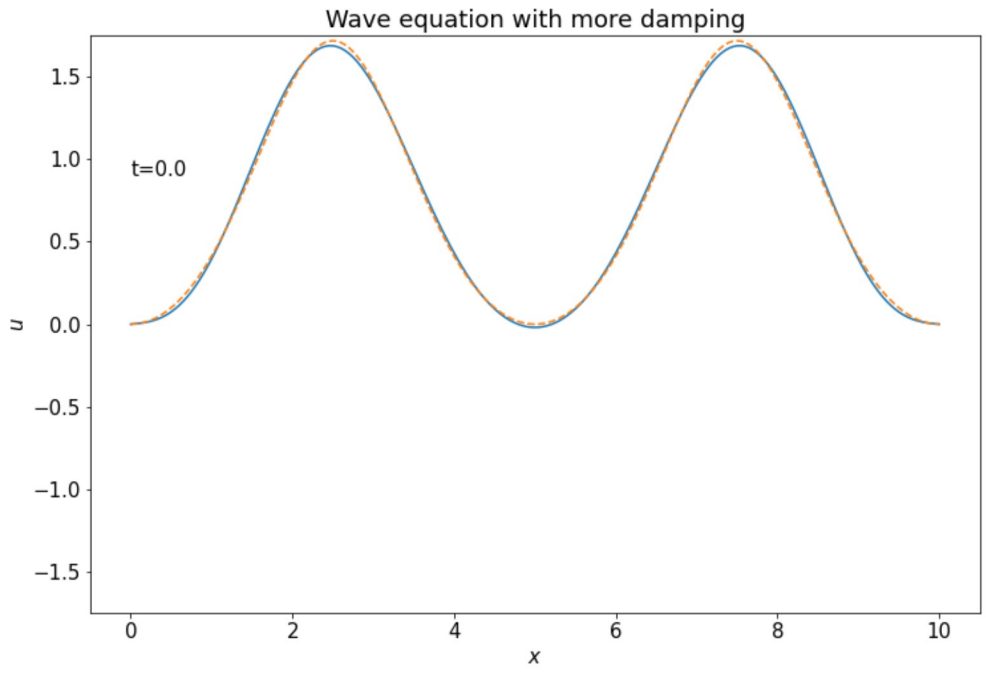
We see that truncation changes the initial state, however all of the solutions behave similarly. This is because all of these truncated solutions do still solve the dampened wave equation, it's just the initial state of the wave is approximated more poorly. We see that the difference in the initial state of the wave between three and five terms is pretty sizable, however the difference between 7 and 11 terms is pretty minimal, and 7 terms is probably sufficient if one just wants to get an idea of the shape of the solution.

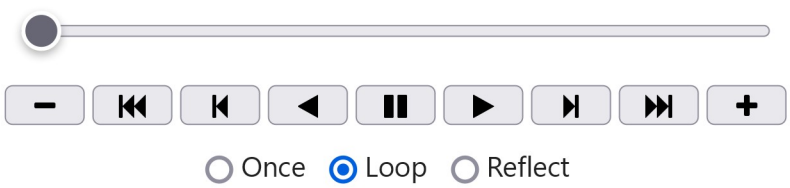
**Task 1.8 (1 marks)** Change the damping parameter to $\nu = 1$, and repeat the 'task 1.6' (for $N = 7$ and keep all other parameters the same). In few sentences, explain the changes caused by increasing the damping.

In [8]:
```
# Define constants for ODE
nu = 1
c = 1
L = 10

# Solve the ode for N = 7
x_vec = np.linspace(0, L, 201)
t_vec = np.linspace(0, 100, 1001)

u_x_t = waveq_dam_solve(7, t_vec, x_vec)

# Create and decorate some axes
fig, ax = plt.subplots(figsize=(12, 8))
ax.set_xlabel('$x$')
ax.set_ylabel('$u$')
ax.set_ylim(-1.75, 1.75)
ax.set_title("Wave equation with more damping")
txt = ax.text(0, 0.9, '$t=0$')

# Plot some curves for the solution and initial condition
line, = ax.plot(x_vec, u_x_t[0,:], label="solution")
ax.plot(x_vec, np.exp((np.sin(2*np.pi*x_vec/L))**2)-1, "--", label="initial
plt.close()

# Animate the solution curve
ani = animate_pde_solutions([line], [u_x_t], txt, t_vec)
HTML(ani.to_jshtml())
```

```
Animation size has reached 20995201 bytes, exceeding the limit of 20971520.
0. If you're sure you want a larger animation embedded, set the animation.e
mbed_limit rc parameter to a larger value (in MB). This and further frames
will be dropped.
```

Out[8]:



We can see increasing the damping drastically changed how our solution evolved over time, (but didn't change the initial condition, in contrast with the example from 1.7). This is of course expected as the parameters of the ode don't factor in to our fourier approximation of $f(x)$. Our more dampened solution did not "overshoot" the $u = 0$ line like the less dampened solution, this is because the damping term has the effect of reducing the acceleration of a point in the wave in the direction in which that point is travelling, so increasing this term's size should cause this effect. In fact, as our ODEs for $f_n$ are

$$f_n''(t) + \nu f_n'(t) + (c\lambda_n)^2 f_n(t) = 0$$

We can see these are simply dampened systems with damping constant $\zeta = \frac{\nu}{c\lambda_n}$. when $\nu = 0.1$, we have that $\zeta < 1$ for all our terms, so the system is underdamped which causes $f_n(t)$ to oscillate. In contrast, when $\nu = 1$ our $f_1$, $f_2$ and $f_3$ systems have $\zeta > 1$, so the system is overdamped and just exponentially decays (these terms are the most dominant in our solution for $u(x,t)$, so they drive this behaviour in the solution as a whole). As the $f_n$ terms are the only terms in our solution which vary over time, their behaviour over time governs the behaviour of our solution for $u$ over time.

# Part 2: Lorenz system

The Lorenz system is given by the coupled set of ODEs

$$\frac{\mathrm{d}x}{\mathrm{d}t} = -\sigma x + \sigma y, \qquad \frac{\mathrm{d}y}{\mathrm{d}t} = x(\rho - z) - y, \qquad \frac{\mathrm{d}z}{\mathrm{d}t} = xy - \beta z, \tag{3}$$

where $(\rho, \sigma, \beta)$ 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.

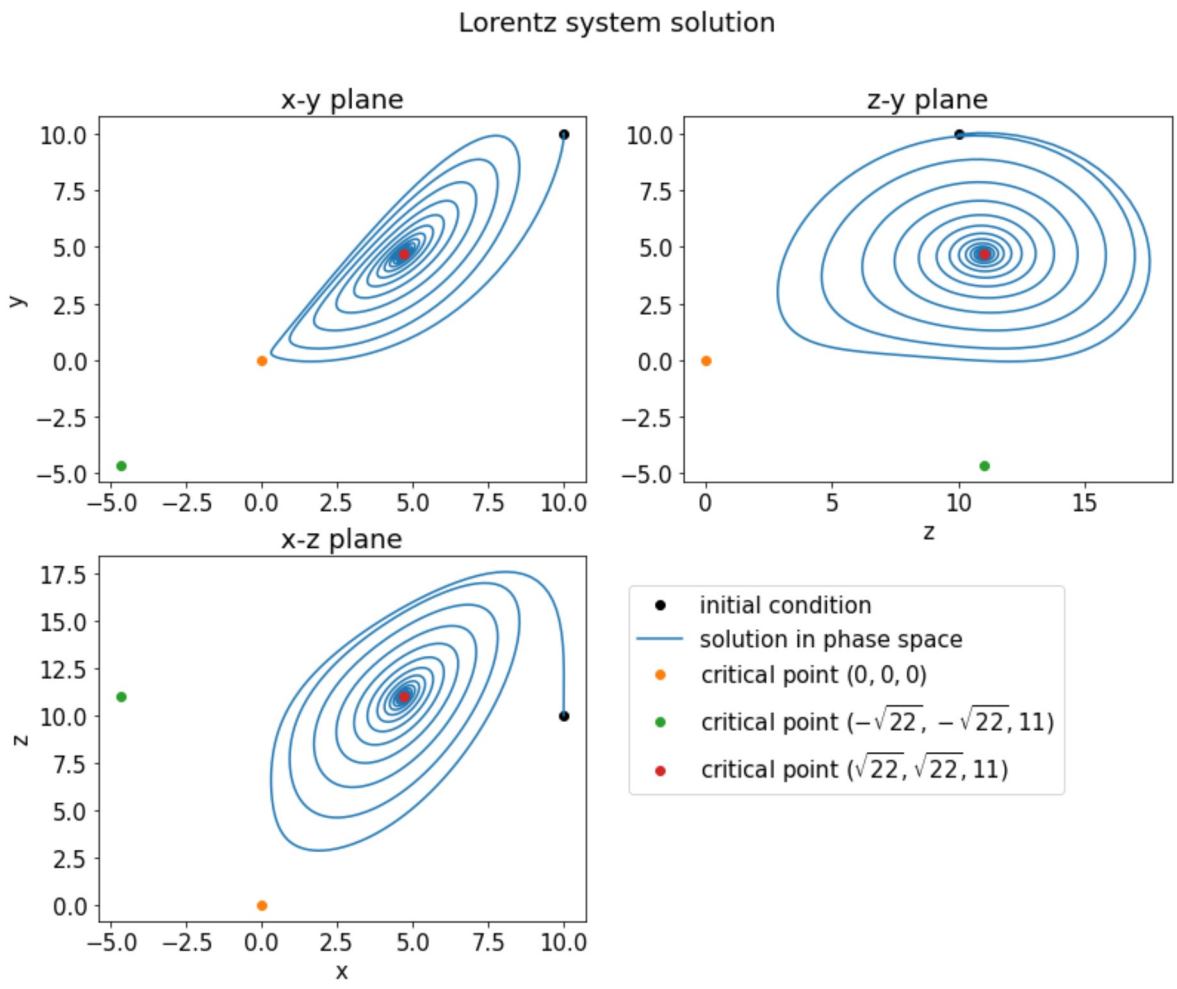
🚩 **Task 2.2 (2 marks)** Using a numerical method of your choice, find the solution starting from $x(t=0) = y(t=0) = z(t=0) = 10$ up to $t = 20$. Plot the solution in *x*-*y*, *y*-*z* and *x*-*z* planes (so you need three plots). Where does this initial condition end up in? Can you express the exact 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 [9]:
```
# Task 2.2
import numpy as np
import sympy as sym
from scipy.integrate import odeint
import matplotlib.pyplot as plt
from itertools import combinations
```

In [10]:
```
# Set the global fontsize for matplotlib plots so that they are and more re
plt.rcParams.update({'font.size': 15})


# Set parameters for ode
sigma = 8
beta = 2
rho = 12

# Use sympy to solve for the critical points
x, y, z = sym.symbols("x y z")

x_prime = -sigma*x + sigma*y
y_prime = x*(rho - z) - y
z_prime = x*y - beta*z

crit_point_dicts = sym.solve([x_prime, y_prime, z_prime])


def func(coords, t):
    """This function defines the RHS of the system of ODEs"""
    x_coord, y_coord, z_coord = coords
    return np.array([
        -sigma*x_coord + sigma*y_coord,
        x_coord*(rho - z_coord) - y_coord,
        x_coord*y_coord - beta*z_coord
    ])

# Numerically solve the ODE for some given initial conditions
initial_condition = (10, 10, 10)

solution = odeint(func, initial_condition, np.linspace(0, 20, 2000))

x_sol, y_sol, z_sol = solution.T

# Create and decorate the figure.
fig = plt.figure(figsize=(13, 10))
fig.suptitle("Lorentz system solution")

# Put the x against y axis in the top left of our subplot.
xy_ax = fig.add_subplot(2, 2, 1)
xy_ax.set_ylabel("y")
xy_ax.set_title("x-y plane")

# Put the y against z axis in the top right of our subplot, sharing a y axi
zy_ax = fig.add_subplot(2, 2, 2, sharey=xy_ax)
zy_ax.set_xlabel("z")
zy_ax.set_title("z-y plane")

# Put the x against z axis in the bottom right of our subplot, sharing an x
xz_ax = fig.add_subplot(2, 2, 3, sharex=xy_ax)
xz_ax.set_xlabel("x")
xz_ax.set_ylabel("z")
xz_ax.set_title("x-z plane")

# Create a lookup table for the data to be plotted
plotting_data = {
    var: (var_sol, var_ic) for var, var_sol, var_ic in zip((x, y, z), solut
}


for ax, var_1, var_2 in ((xy_ax, x, y), (zy_ax, z, y), (xz_ax, x, z)):
```

```
    for crit_point in crit_point_dicts:
        label = f"critical point $({sym.latex(crit_point[x])}, {sym.latex(c
        ax.plot([crit_point[var_1]], [crit_point[var_2]], "o", label=label)


# Add a legend to the empty corner of the figure, making sure to only add e
handles, labels = xy_ax.get_legend_handles_labels()

fig.legend(handles, labels, loc="upper left", bbox_to_anchor=(0.5, 0.0, 1,
```

Out[10]: <matplotlib.legend.Legend at 0x21aa2d75e80>




We can see that our solution converges to the stable critical point $(\sqrt{22}, \sqrt{22}, 11)$

**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 $6$.

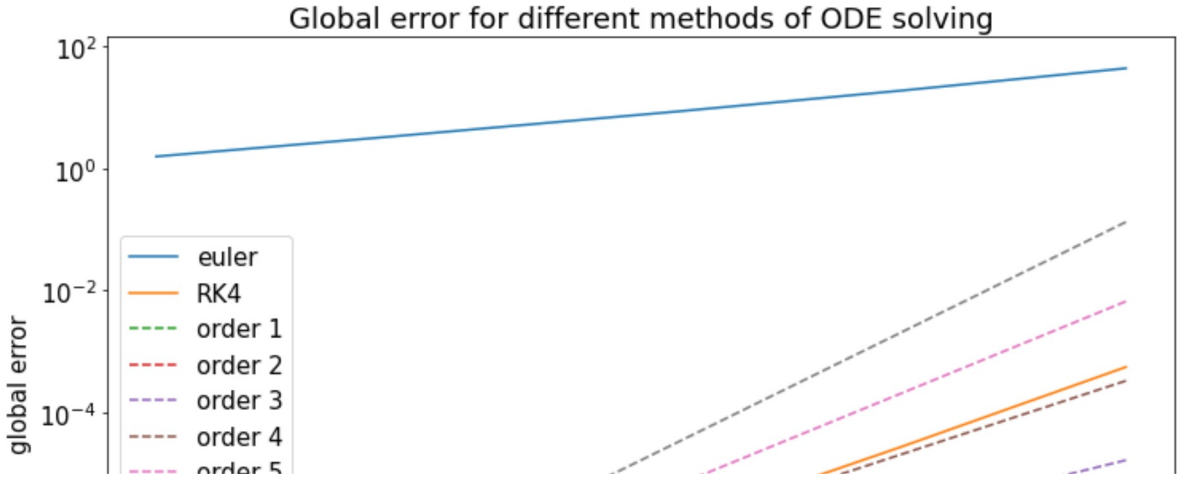
What is the slope of global error vs timestep for each method? Why do you get these slopes? You can estimate the slopes by drawing linear guidelines. For example, if you draw $h^6$ and your curve is close to this guideline, then you can conclude that the slope is close to $6$. You need to shift your 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

In [11]:
```
# forward 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 C
        y0: initial condition (make sure the dimension of y0 and func are t
    output:
        y: the solution of ODE.
        Each row in the solution array y corresponds to a value returned ir
    '''
    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) Runge-Kutta using 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 C
        y0: initial condition (make sure the dimension of y0 and func are t
    output:
        y: the solution of ODE.
        Each row in the solution array y corresponds to a value returned ir
    '''
    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 i in range(nT-1):
        dt = times[i+1]-times[i]
        t_i = times[i]
        y_i = y[i, :]
        # Calculate the k_1, k_2, k_3 and k_4
        k_1 = func(y_i, t_i)
        k_2 = func(y_i + 0.5*dt*k_1, t_i + 0.5*dt)
        k_3 = func(y_i + 0.5*dt*k_2, t_i + 0.5*dt)
        k_4 = func(y_i + dt*k_3, t_i + dt)
        # Calculate the next step
        y[i+1, :] = y_i + (1/6)*dt*(k_1 + 2*k_2 + 2*k_3 + k_4)
    return y


# Helper function: mitigates errors ascosiated with np.arange
def get_times(t_start, t_end, h):
```

```
initial_condition = (100, 100, 100)
t_start = 0
t_end = 0.3

h = 1e-6
# no_points = round((t_end - t_start)/h) + 1
# times = np.linspace(t_start, t_end, no_points)

exact_soln = ode_RK4(func, get_times(t_start, t_end, h) , initial_conditior
exact_final_x = exact_soln[-1][0]

# Calculate the global errors for each value of h and each method
h_vals = [0.002, 0.001, .0005, 0.0002, 0.0001]
RK4_x_errors = []
euler_x_errors = []

for h in h_vals:
    for method, error_list in ((ode_Euler, euler_x_errors),
                               (ode_RK4, RK4_x_errors)):

        # Subtract the final value of the solution from the exact value to
        soln = method(func, get_times(t_start, t_end, h), initial_conditior
        soln_final_x = soln[-1][0]
        error_list.append(abs(exact_final_x - soln_final_x))


# Plot the errors for each method on a log-log scale
fig, ax = plt.subplots(figsize=(12, 8))

ax.set_title("Global error for different methods of ODE solving")
ax.set_xlabel("step size")
ax.set_ylabel("global error")

ax.loglog(h_vals, euler_x_errors, label="euler")
ax.loglog(h_vals, RK4_x_errors, label="RK4")

# Plot guidelines for each possible value for the order of the global erro
for order in range(1, 7):
    h_eval = np.array(h_vals)
    shifted_y = RK4_x_errors[-1]*(h_eval**order)/(h_eval[-1]**order)
    ax.loglog(h_eval, shifted_y, "--", label=f"order {order}")

ax.legend()
```

Out[11]: <matplotlib.legend.Legend at 0x21a9e68faf0>




By comparing gradients, we can see the global error of the RK4 scheme has order 4 and the global error of the euler scheme has order 1. This is a consequence of the fact that the local truncation error of RK4 has order 5 and the local truncation error of euler has order 2, and the fact that the number of steps taken is proportional to $\frac{1}{h}$.

---

## Case of chaotic behaviour and strange attractor

Consider the set of parameters $\sigma = 8/3$, $\beta = 4$ and $\rho = 6$ 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 $\epsilon$ 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

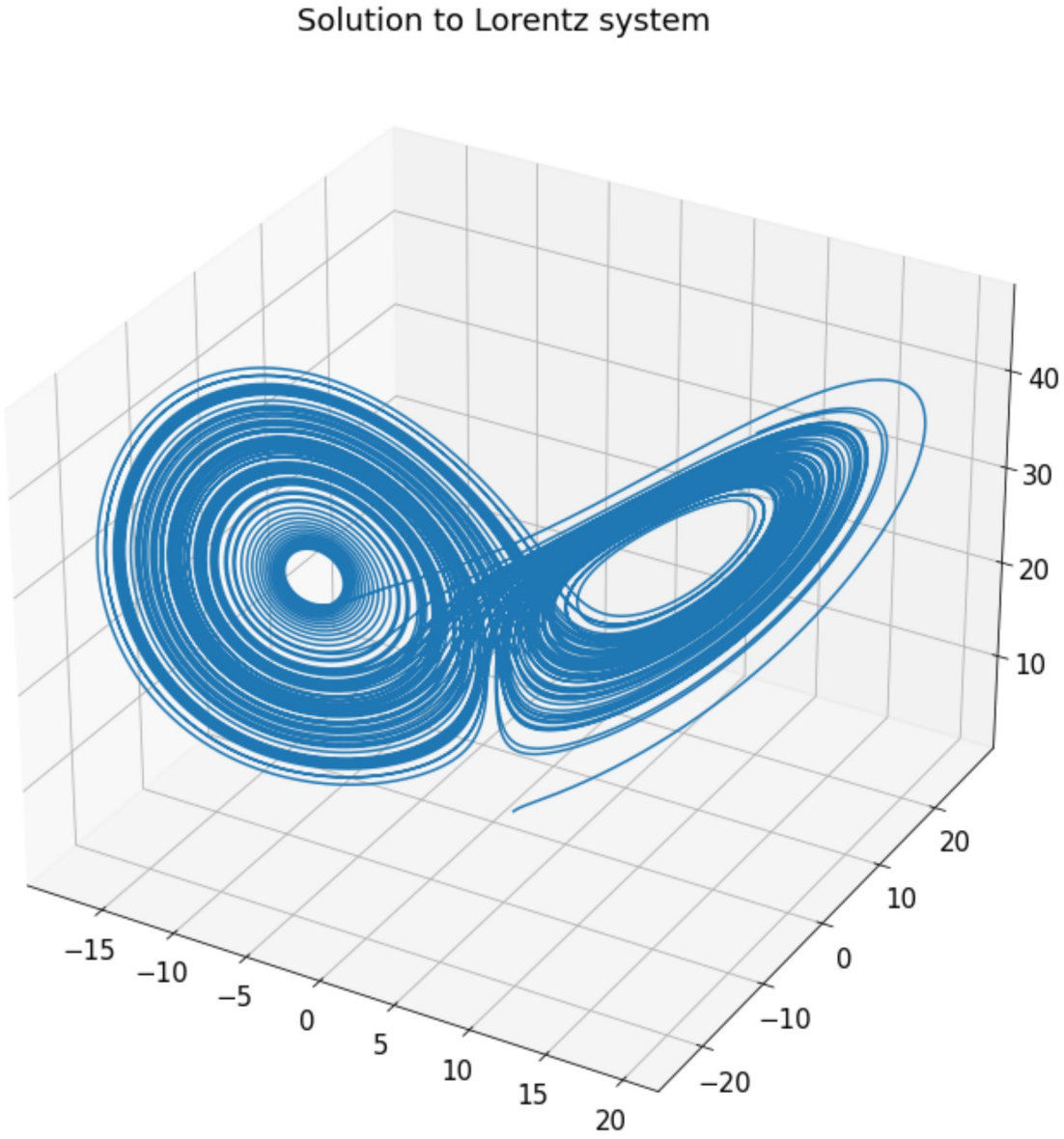
$$e(t) = \sqrt{(x^*(t) - x_1(t))^2 + (y^*(t) - y_1(t))^2 + (z^*(t) - z_1(t))^2}.$$

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

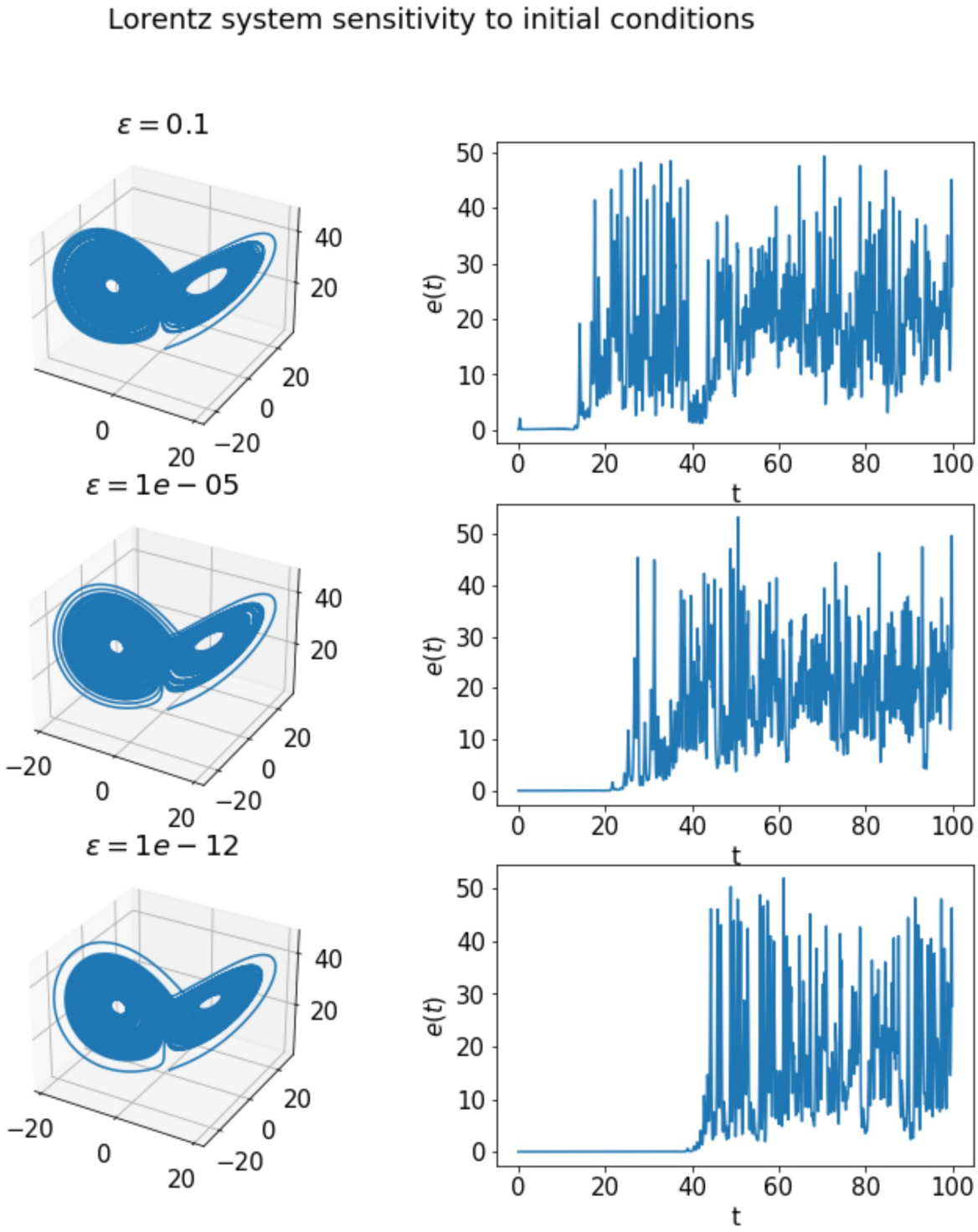
- If two initial conditions are very close, do their corresponding trajectories stay close to each other in time?
- Do you see an upper bound for $e(t)$? What does this imply?

*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 [12]:
```
from mpl_toolkits.mplot3d import Axes3D

# Set ODE params
rho = 28.
sigma = 10.
beta = 8.0/3.0

# Create and decorate axes
fig = plt.figure(figsize=(12, 12))
ax = fig.gca(projection="3d")
ax.set_title("Solution to Lorentz system")

# Plot the 3-D solution
initial_condition = (1, 1, 1)
t_vals = np.linspace(0, 100, 100*10000)
solution = ode_RK4(func, t_vals, initial_condition)

x_sol, y_sol, z_sol = solution.T

ax.plot(x_sol, y_sol, z_sol)

plt.draw()
plt.show()
```



In [13]:
```
fig = plt.figure(figsize=(12, 12))

fig.suptitle("Lorentz system sensitivity to initial conditions")

# We stack the plots for each epsilon below each other
for i, epsilon in enumerate([1e-1, 1e-5, 1e-12]):

    # plot the phase portrait on an axis on the left of the screen
    _3d_ax = fig.add_subplot(3, 2, 2*i+1, projection="3d")

    initial_condition = (1 - epsilon, 1, 1)

    new_solution = ode_RK4(func, t_vals, initial_condition)

    x_sol, y_sol, z_sol = new_solution.T
    _3d_ax.plot(x_sol, y_sol, z_sol)
    _3d_ax.set_title(f"$\epsilon = {epsilon}$")

    # Plot e(t) vs time on the right of the screen
    error_ax = fig.add_subplot(3, 2, 2*i+2)
    error_ax.set_xlabel("t")
    error_ax.set_ylabel("$e(t)$")
    euclidean_dist = np.linalg.norm(solution - new_solution, axis = 1)

    error_ax.plot(t_vals, euclidean_dist)


plt.draw()
plt.show()
```




We can see that although all of the trajectories follow the same figure-8 shape lorentz attractor, they end up significantly deviating from each other over time. This is a result of the fact that a small difference in the position of our solution compounds until the solutions are on different parts of the figure-8 loop; this compounding effect is due to the Lorentz system being chaotic for this set of parameters. Once the solutions have deviated from each other enough, the distance between the solutions becomes bounded by the fact that the solutions are still in the same loop, so the size of the loop bounds how far apart they can be.

In [ ]: