הפקולטה להנדסת מכונות
Faculty of Mechanical Engineering

ה ט כ נ יון – מכון טכנולוגי לישראל
Technion – Israel Institute of Technology

Homework cover page

Analytic and machine learning-based modeling of dynamical systems - 036064

Dr. Maor Farid

☐ Homework no. _____

☒ Final project

Students details:

Chenhao Yang	941170706
Full name	ID no.
Jonathan Oh	941170383
Full name	ID no.

Before submitting your homework, you are requested to fill in the suitable survey on the course website. Submitting the survey is mandatory. Your feedback is highly important for the course's staff and serves us for optimizing the course contents for you and for maximizing the contribution of the HW and assimilation of the material learned. Of course, your answers do not influence your grade and evaluation in any way.

☒ I confirm that I have filled in and submitted the HW survey on the course website

Thanks for your collaboration and good luck!

Final project

In [1]:
```
import os
import re
import sys
import torch
import urllib
import shutil
import pathlib
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
from pprint import pprint

%load_ext autoreload
%autoreload 2
plt.rcParams.update({'font.size': 15,'axes.labelsize' : 15})
device = torch.device('cuda' if torch.cuda.is_available() else 'cpu')
print('Using device:', device)
# !nvidia-smi
```

```
Using device: cpu
```

Q1- Chaos in continuous dynamic systems – Lorenz Attractor (50 pt.)

The set of equations shown below are known as Lorenz equations. These equations were developed by Edward Lorenz (1963) as a toy model of Navier-Stokes equations- the meteorological equations that describe the flow, heat transfer, and pressure differences in the atmosphere. These equations form a simple-looking three-dimensional dynamic system, but they present chaotic dynamics across a wide range of parameters. All the problem's parameters (σ, ρ, β) are positive.

$$\dot{x} = \sigma(y - x) \tag{1}$$
$$\dot{y} = \rho x - y - xz \tag{2}$$
$$\dot{z} = xy - \beta z \tag{3}$$
$$\rho,\sigma,\beta > 0 \tag{4}$$

a. (8 pt.) Express the equilibrium points of the system as a function of the problem's parameters. For which parameter range does each one exist?

Answer:

$$\dot{x} = 0 = \sigma(y - x) \tag{5}$$
$$\Rightarrow y = x \tag{6}$$
$$\dot{y} = 0 = \rho x - y - xz = x(\rho - 1) - xz \tag{7}$$
$$x = 0 \; satisfies \; the \; condition \tag{8}$$
$$\Rightarrow z = \rho - 1 \tag{9}$$
$$\dot{z} = 0 = xy - \beta z = x^2 - \beta(\rho - 1) \tag{10}$$
$$\Rightarrow x^2 = \beta(\rho - 1) \tag{11}$$

Summary:

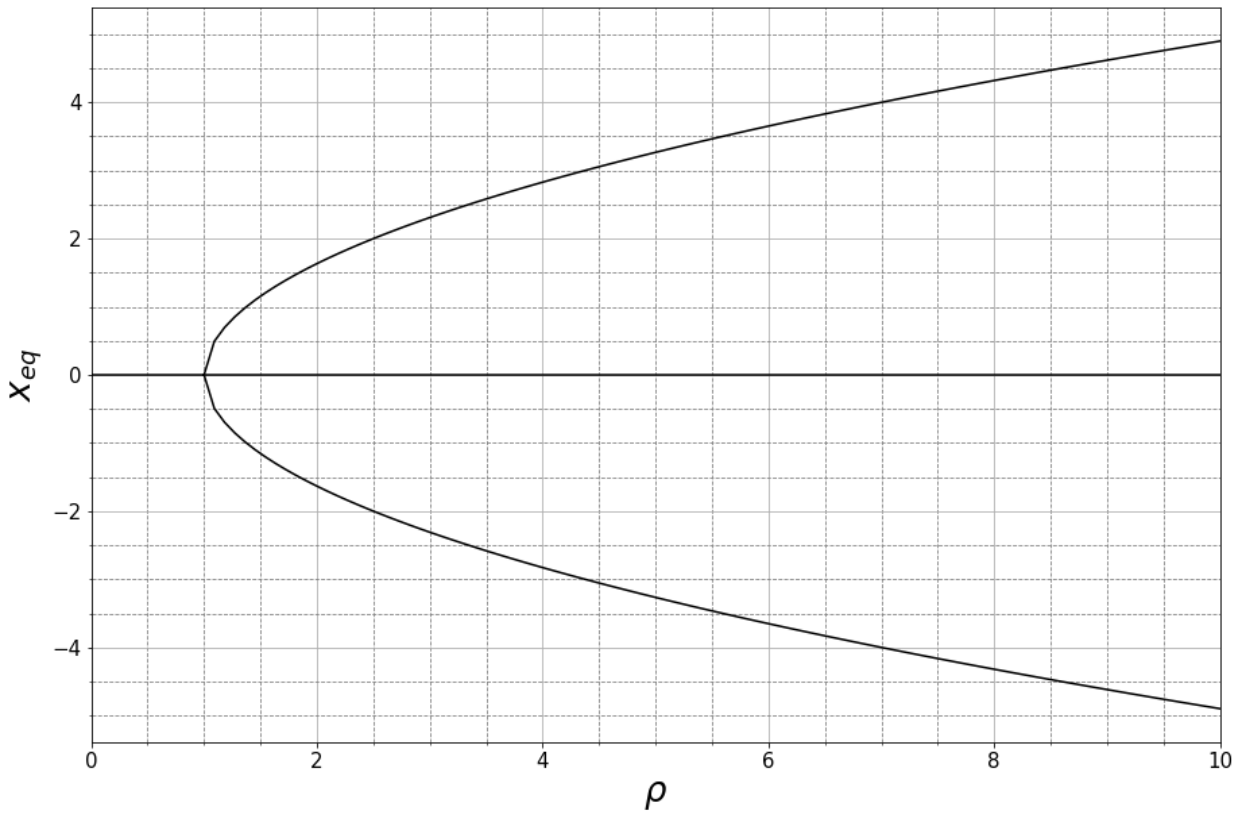
$$x_{eq} = y_{eq} = \pm\sqrt{\beta(\rho - 1)} \; or \; 0, \; z_{eq} = \rho - 1 \; or \; 0 \; (when \; x = 0)$$

Discuss solution of x_{eq}:

$\rho \leq 1$	$\rho > 1$
One solution	Three solutions
0	$0, \pm\sqrt{\beta(\rho-1)}$

b. (6 pt.) Draw a bifurcation diagram of the equilibrium values of coordinate x as a function of parameter ρ for $\beta = 8/3$ using Python. What bifurcation occurred? There is no need to classify the stability of the branches.

In [2]:
```
beta = 8/3

# rho range
rhos_1 = np.linspace(0, 1, 100)
rhos_2 = np.linspace(1, 10, 100)

# save solutions
sols_1 = np.zeros((rhos_1.size,1)) # one sol
sols_2 = np.zeros((rhos_2.size,3)) # three sols
sols_2[:,1] = np.sqrt(beta*(rhos_2-1))
sols_2[:,2] = - np.sqrt(beta*(rhos_2-1))

# plot
plt.figure(figsize=(15,10))
plt.plot(rhos_1, sols_1, '-', color='black')
plt.plot(rhos_2, sols_2, '-', color='black')
plt.xlabel(r'$\rho$',size='xx-large')
plt.ylabel(r'$x_{eq}$',size='xx-large')

plt.xlim([rhos_1[0],rhos_2[-1]])
plt.grid(True)
plt.minorticks_on()
plt.grid(b=True, which='minor', color=[0.5, 0.5, 0.5], linestyle='--')
# plt.ylim([-math.pi-0.2,math.pi+0.2])
```



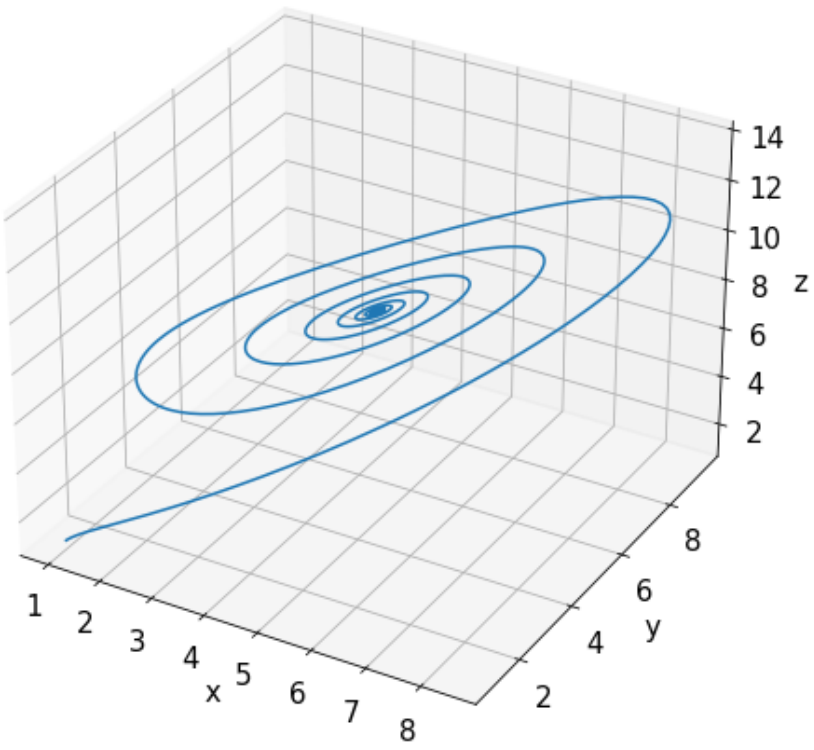
c. (6 pt.) Appendix A shows a numerical solution for the Lorenz equations. Write a code in Python that calls this function and shows the

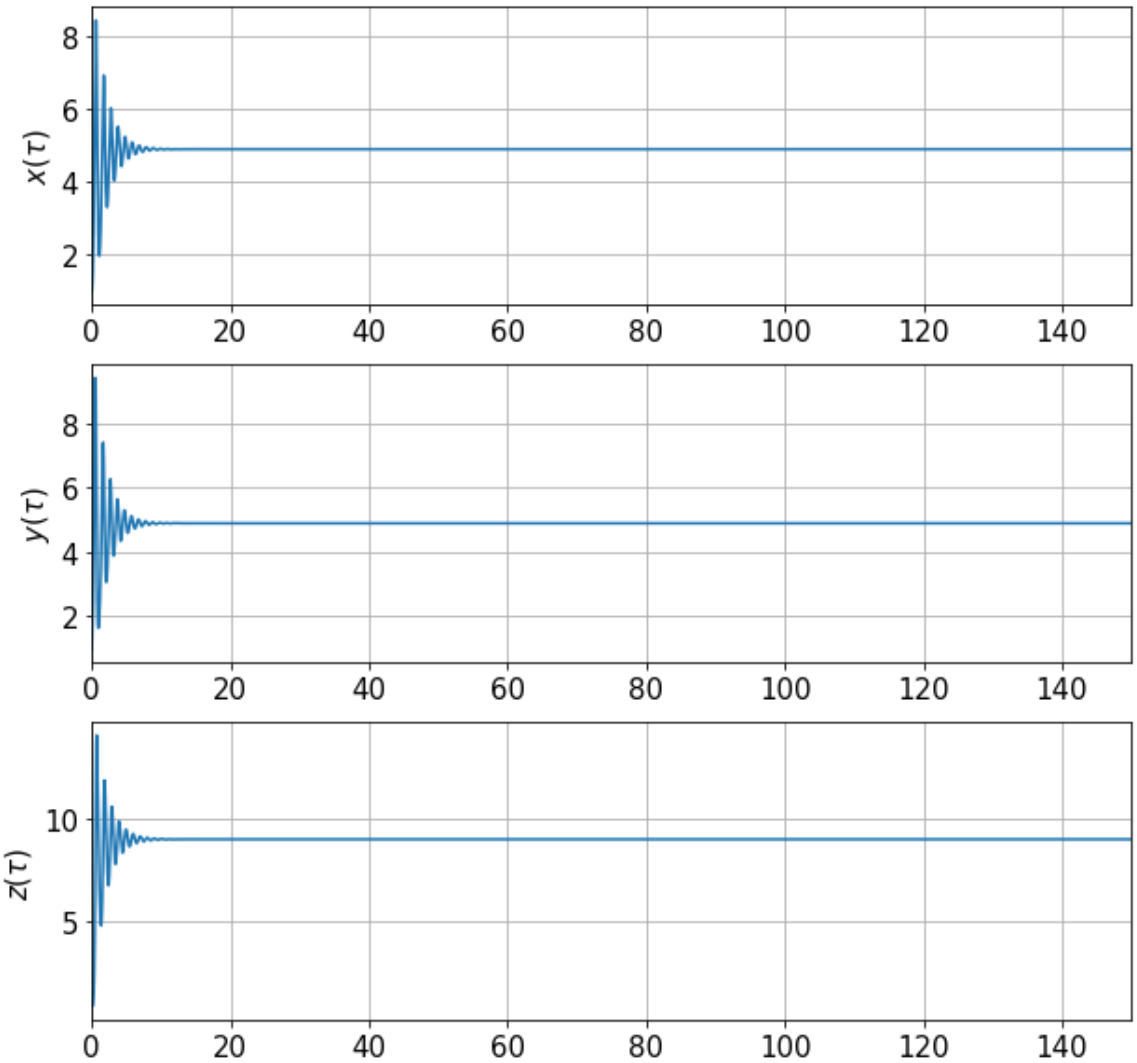
three-dimensional state space of the Lorenz equations on a single graph. On a second graph, show the time history for each degree of freedom, one above the other. Define the set of parameters for each section as follows:$\sigma = 10,\ \beta = 8/3$.

In [3]:
```
# define state equation function and solver function
from scipy.integrate import odeint
def Lorenz_EOM(x_vec, t, sigma, rho, beta):
    x, y, z = x_vec
    x_vec_dot = [sigma*(y - x), rho*x-y-x*z, x*y-beta*z]
    return x_vec_dot

def Lorenz_numerical_solver(IC_vec, sigma, rho, beta, tf):
  dt = 0.01
  t_vec = np.arange(0,tf,dt)
  sol_gt = odeint(Lorenz_EOM, IC_vec, t_vec, args=(sigma, rho, beta))
  x_vec, y_vec, z_vec = sol_gt[:,0], sol_gt[:,1], sol_gt[:,2]
  return t_vec, x_vec, y_vec, z_vec

# main code
sigma, rho, beta = 10, 10, 8/3
t_initial, t_final = 0, 150
IC_vec = [1, 1, 1]
[t_vec, x_vec, y_vec, z_vec] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta, t_f
```

In [4]:
```
# time history 3d
fig = plt.figure(figsize=(10,8))
ax = fig.gca(projection='3d')
ax.plot(x_vec, y_vec, z_vec)
plt.title(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,beta))
ax.set_xlabel('x')
ax.set_ylabel('y')
ax.set_zlabel('z')

# time history each
fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(10,10))
fig.suptitle(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,beta)
ax1.plot(t_vec, x_vec)
ax1.set_ylabel(r'$x(\tau)$')
ax1.set_xlim([0,150])
ax1.grid()
ax2.plot(t_vec, y_vec)
ax2.set_ylabel(r'$y(\tau)$')
ax2.set_xlim([t_initial, t_final])
ax2.grid()
ax3.plot(t_vec, z_vec)
ax3.set_ylabel(r'$z(\tau)$')
ax3.set_xlim([t_initial, t_final])
ax3.grid()
```

$\sigma = 10.00, \rho = 10.00, \beta = 2.67$





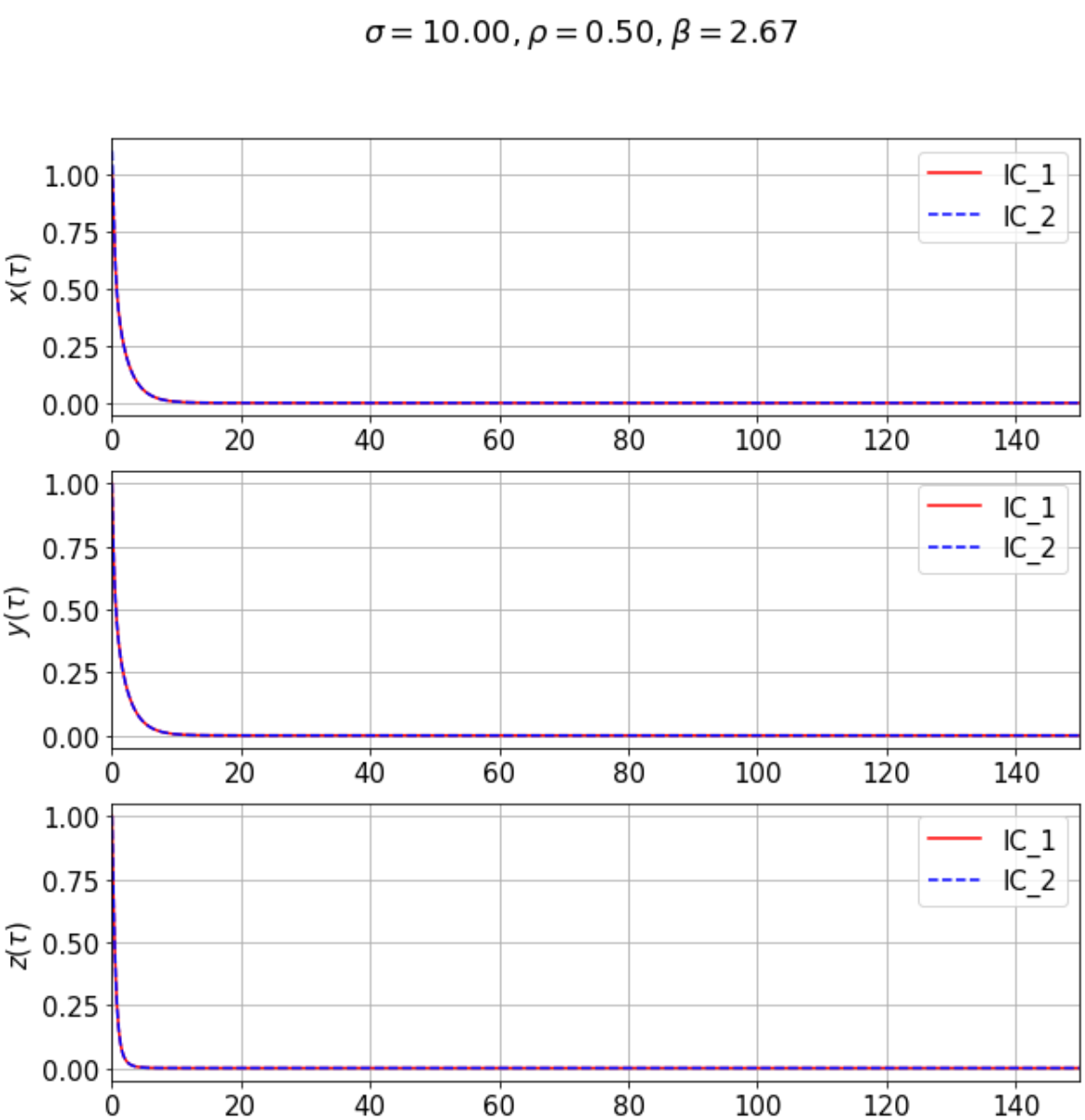
d. (6 pt.) For ρ = 0.5: perform two simulations, one for each of the following initial conditions (1,1,1), (1.1, 1, 1). For each simulation, plot the three-dimensional state space and the time histories of each degree of freedom, as was described in the previous section. In each graph, show the two simulations together using subsequent 'plot' commands. Color the first graph in red and the second in blue.

In [5]:

```
sigma, rho, beta = 10, 0.5, 8/3
t_initial, t_final = 0, 150
IC_vec = [1, 1, 1]
[t_vec, x_vec1, y_vec1, z_vec1] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [1.1, 1, 1]
[t_vec, x_vec2, y_vec2, z_vec2] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,

# time history each
fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(10,10))
fig.suptitle(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,beta)
ax1.plot(t_vec, x_vec1,'r')
ax1.plot(t_vec, x_vec2,'b--')
ax1.set_ylabel(r'$x(\tau)$')
ax1.set_xlim([0,150])
ax1.grid()
ax1.legend(['IC_1','IC_2'])
ax2.plot(t_vec, y_vec1,'r')
ax2.plot(t_vec, y_vec2,'b--')
ax2.set_ylabel(r'$y(\tau)$')
```

```
ax2.set_xlim([t_initial, t_final])
ax2.grid()
ax2.legend(['IC_1','IC_2'])
ax3.plot(t_vec, z_vec1,'r')
ax3.plot(t_vec, z_vec2,'b--')
ax3.set_ylabel(r'$z(\tau)$')
ax3.set_xlim([t_initial, t_final])
ax3.grid()
ax3.legend(['IC_1','IC_2'])
```

Out[5]: <matplotlib.legend.Legend at 0x7fe28dfd6130>



e. (6 pt.) Towards which equilibrium point the dynamics flow? What can be learned from the graphs obtained about the similarity between the two solutions obtained for each of the initial conditions? (the dynamics/trajectories are close or detached from one another?) and what it means about the nature of the dynamics (there is/there is not chaos)? Is it possible that for other initial conditions (and for the given value of the parameter ρ) the dynamics will flow to different equilibrium? Explain your answer.

Answer:

From previous bifurcation graph we can have conclusion that the dynamics flow towards equilibrium point `(0,0,0)` . The dynamics/trajectories are very close to each other, the 'red' line in above is

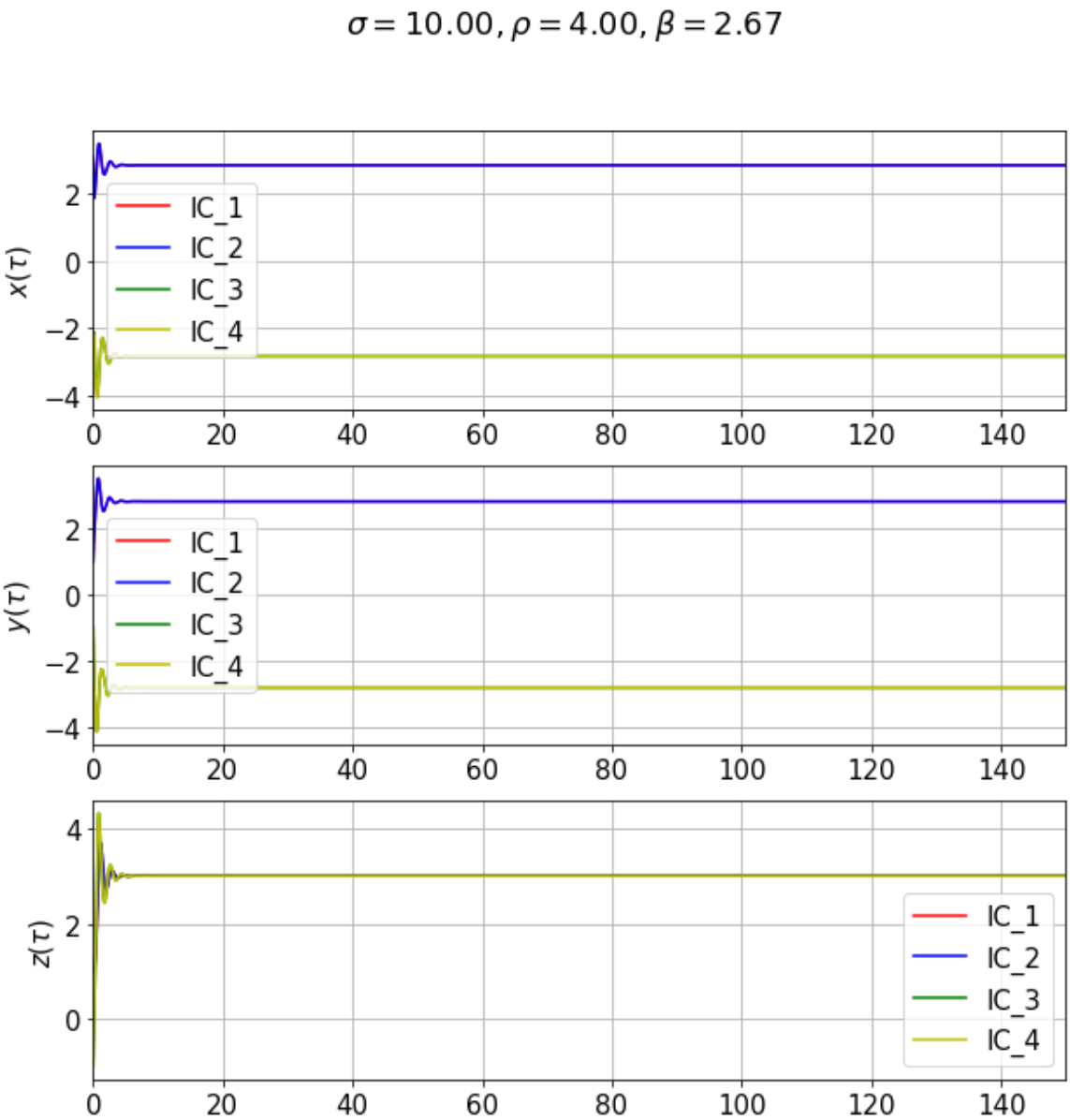
overlaping with the 'blue' line. There is no chaos because different initial conditions flows to the same equilibrium point.

When $\rho > 1$, it is possible that under different initial conditions, the dynamical system flows to different equilibrium points. It is observed from bifurcation graph.

f. (6 pt.) For $\rho = 4$: perform four simulations, one for each pair of initial conditions: first pair:(3,1,1), (3.1,1,1), second pair: (-3,-1,-1), (-3.1, -1,-1). Display all graphs on a single state-space plot and a single time histories graph, as was described above.

In [6]:
```
sigma, rho, beta = 10, 4, 8/3
t_initial, t_final = 0, 150
IC_vec = [3,1,1]
[t_vec, x_vec1, y_vec1, z_vec1] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [3.1, 1, 1]
[t_vec, x_vec2, y_vec2, z_vec2] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [-3, -1, -1]
[t_vec, x_vec3, y_vec3, z_vec3] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [-3.1, -1, -1]
[t_vec, x_vec4, y_vec4, z_vec4] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,

# time history each
fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(10,10))
fig.suptitle(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,beta)
x_vec = np.vstack
ax1.plot(t_vec, x_vec1, 'r-',t_vec, x_vec2, 'b-', t_vec, x_vec3, 'g-', t_vec, x_vec4
ax1.set_ylabel(r'$x(\tau)$')
ax1.set_xlim([0,150])
ax1.grid()
ax1.legend(['IC_1','IC_2','IC_3','IC_4'])

ax2.plot(t_vec, y_vec1, 'r-', t_vec, y_vec2, 'b-',t_vec, y_vec3, 'g-',t_vec, y_vec4,
ax2.set_ylabel(r'$y(\tau)$')
ax2.set_xlim([t_initial, t_final])
ax2.grid()
ax2.legend(['IC_1','IC_2','IC_3','IC_4'])

ax3.plot(t_vec, z_vec1, 'r-', t_vec, z_vec2, 'b-',t_vec, z_vec3, 'g-',t_vec, z_vec4,
ax3.set_ylabel(r'$z(\tau)$')
ax3.set_xlim([t_initial, t_final])
ax3.grid()
ax3.legend(['IC_1','IC_2','IC_3','IC_4'])
```

Out[6]:
```
<matplotlib.legend.Legend at 0x7fe28c4fc8b0>
```



Answer:

Like we talked before, when $\rho > 1$, it is possible that the dynamical system flows to different equilibrium points. In our example, $x_{eq} = y_{eq} = \pm\sqrt{\beta(\rho-1)} = \pm 2.8284$, $z_{eq} = \rho - 1 = 3$ as we solved before, it happens because of the bifurcation of equilibrium points.

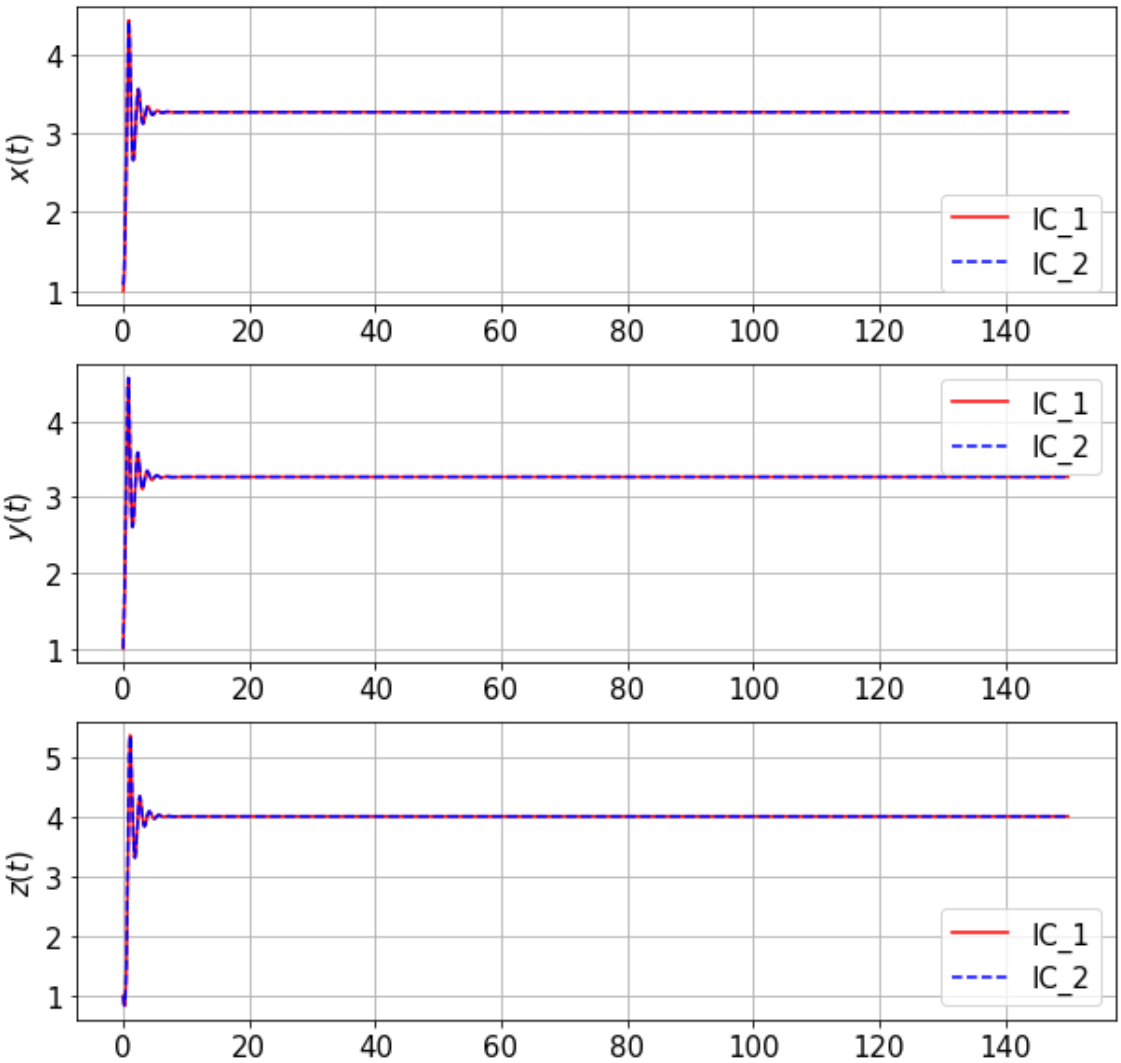
Therefore, there is chaos in the system.

g. (6 pt.) Perform two simulations for the following pair of initial conditions: (1,1,1), (1.1,1,1) for the following parameter values: ρ = 5,15,23. For each value of the parameter ρ, display the two simulations on a single state-space graph and a single time histories graph, as described above.

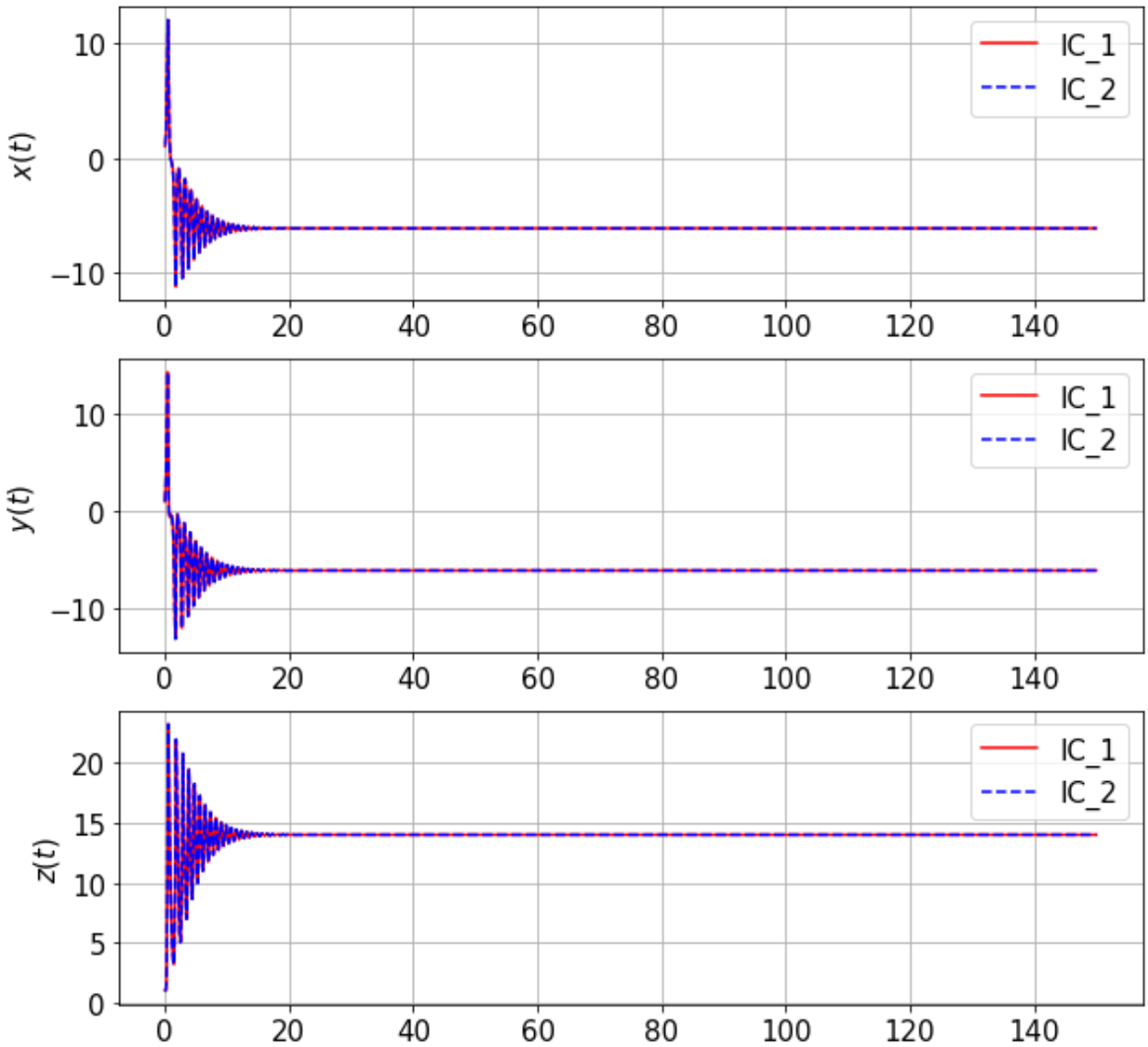
In [7]:
```
def draw_time_history(sigma,rho,beta, t_vec,x_vec1,x_vec2,y_vec1,y_vec2,z_vec1,z_vec
    fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(10,10))
    fig.suptitle(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,b
    x_vec = np.vstack
    ax1.plot(t_vec, x_vec1, 'r-',t_vec, x_vec2, 'b--')
    ax1.set_ylabel(r'$x(t)$')
    ax1.grid()
    ax1.legend(['IC_1','IC_2'])

    ax2.plot(t_vec, y_vec1, 'r-', t_vec, y_vec2, 'b--')
    ax2.set_ylabel(r'$y(t)$')
```

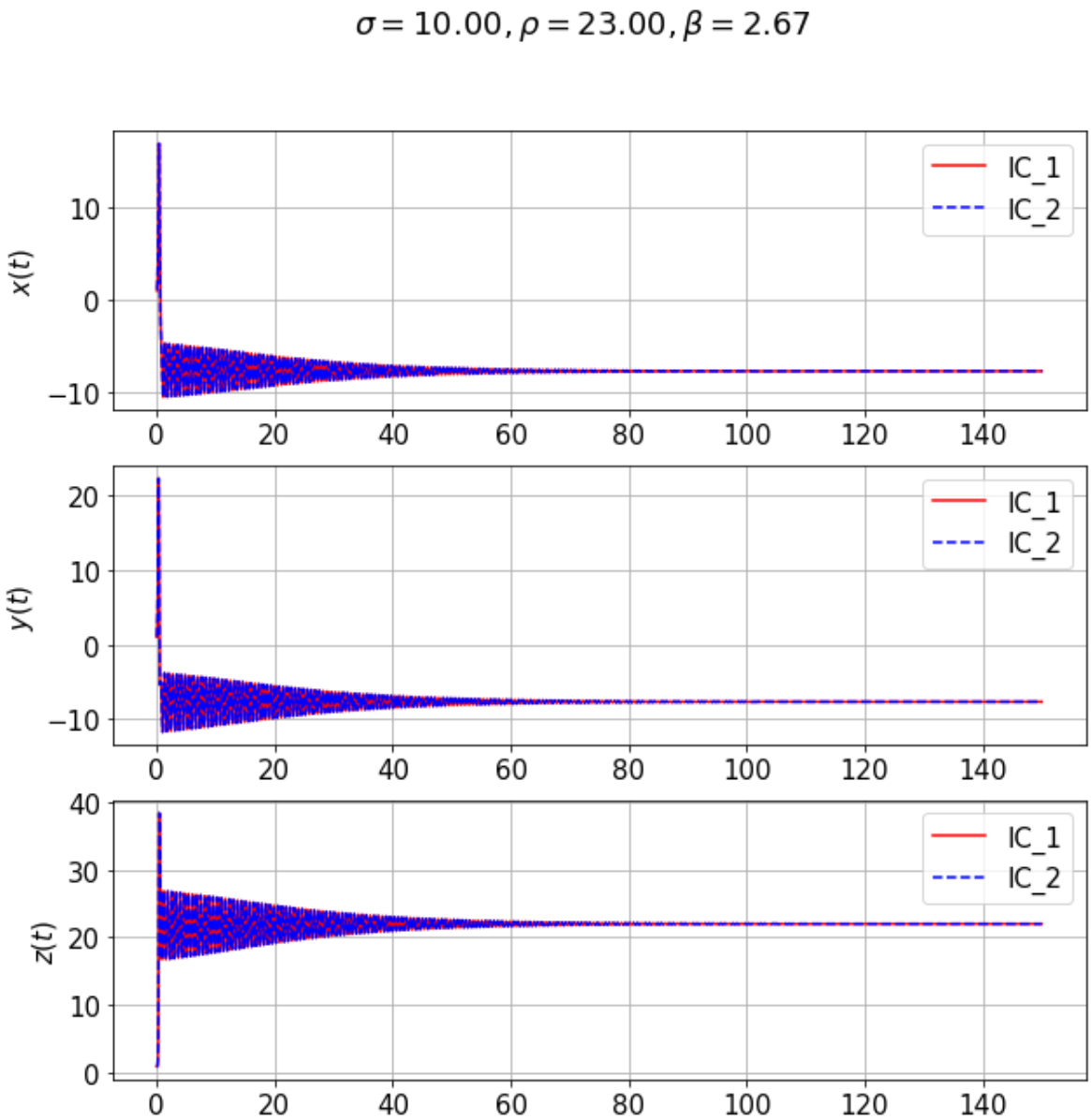
```
    ax2.grid()
    ax2.legend(['IC_1','IC_2'])

    ax3.plot(t_vec, z_vec1, 'r-', t_vec, z_vec2, 'b--')
    ax3.set_ylabel(r'$z(t)$')
    ax3.grid()
    ax3.legend(['IC_1','IC_2'])
    return

sigma, rho, beta = 10, 5, 8/3
IC_vec = [1,1,1]
[t_vec, x_vec1, y_vec1, z_vec1] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [1.1, 1, 1]
[t_vec, x_vec2, y_vec2, z_vec2] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
draw_time_history(sigma,rho,beta, t_vec,x_vec1,x_vec2,y_vec1,y_vec2,z_vec1,z_vec2)

sigma, rho, beta = 10, 15, 8/3
IC_vec = [1,1,1]
[t_vec, x_vec3, y_vec3, z_vec3] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [1.1, 1, 1]
[t_vec, x_vec4, y_vec4, z_vec4] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
draw_time_history(sigma,rho,beta, t_vec,x_vec3,x_vec4,y_vec3,y_vec4,z_vec3,z_vec4)

sigma, rho, beta = 10, 23, 8/3
IC_vec = [1,1,1]
[t_vec, x_vec5, y_vec5, z_vec5] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [1.1, 1, 1]
[t_vec, x_vec6, y_vec6, z_vec6] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
draw_time_history(sigma,rho,beta, t_vec,x_vec5,x_vec6,y_vec5,y_vec6,z_vec5,z_vec6)
```

$\sigma = 10.00, \rho = 5.00, \beta = 2.67$



$\sigma = 10.00, \rho = 15.00, \beta = 2.67$





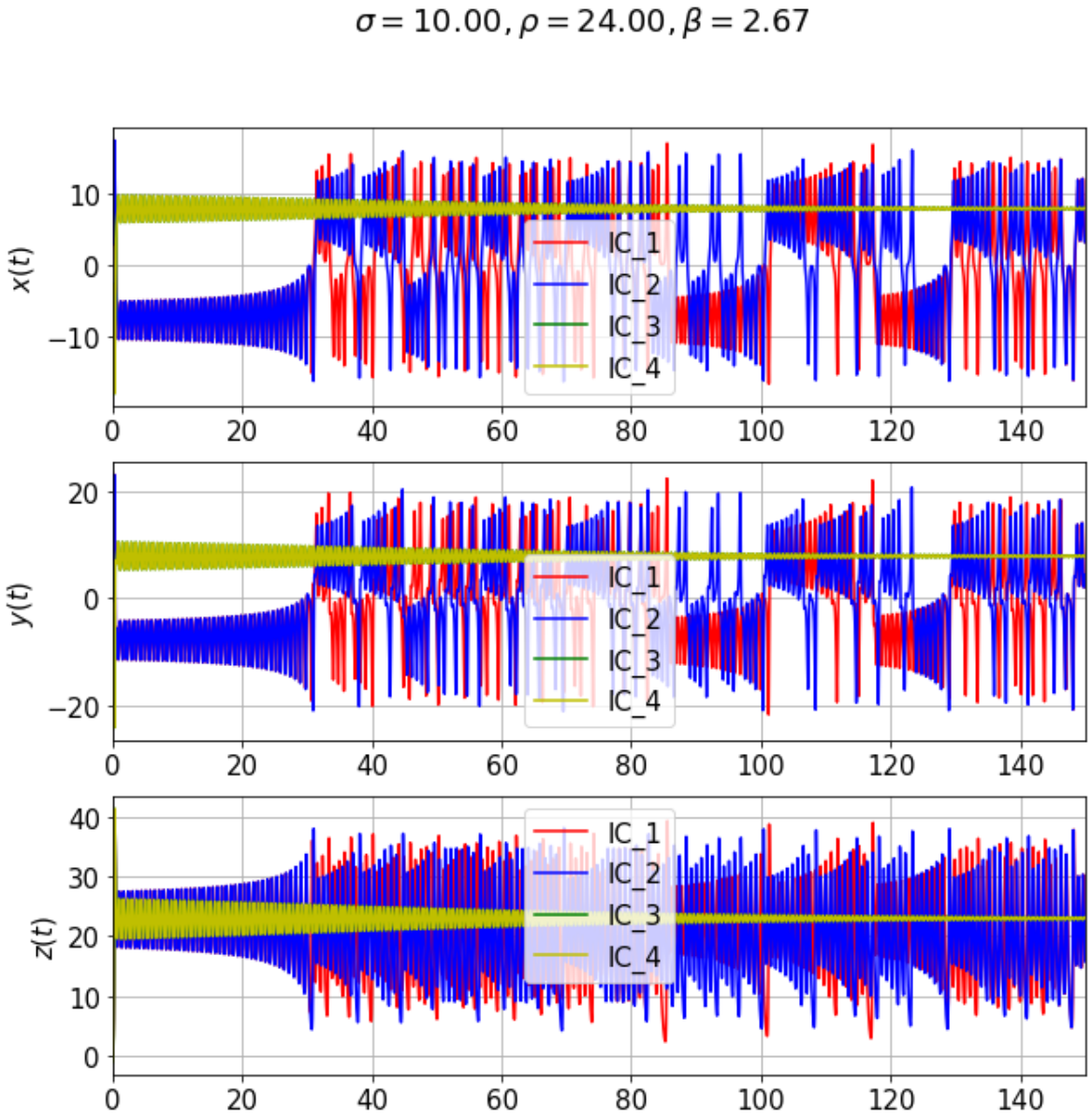
In this section, we tested different ρ but similar initial conditions. It is found that the systems behaves alike, and ended up to same equilibrium point.

h. (6 pt.) Repeat section (f) for ρ = 24 . What qualitative change occurred in comparison to the results obtained in section (f)? What dynamical regime was obtained?

In [8]:
```
sigma, rho, beta = 10, 24, 8/3
t_initial, t_final = 0, 150
IC_vec = [3,1,1]
[t_vec, x_vec1, y_vec1, z_vec1] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [3.1, 1, 1]
[t_vec, x_vec2, y_vec2, z_vec2] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [-3, -1, -1]
[t_vec, x_vec3, y_vec3, z_vec3] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,
IC_vec = [-3.1, -1, -1]
[t_vec, x_vec4, y_vec4, z_vec4] = Lorenz_numerical_solver(IC_vec, sigma, rho, beta,

# time history each
fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(10,10))
fig.suptitle(r'$\sigma={0:.2f}, \rho={1:.2f}, \beta={2:.2f}$'.format(sigma,rho,beta)
x_vec = np.vstack
ax1.plot(t_vec, x_vec1, 'r-',t_vec, x_vec2, 'b-', t_vec, x_vec3, 'g-', t_vec, x_vec4
ax1.set_ylabel(r'$x(t)$')
ax1.set_xlim([0,150])
ax1.grid()
```

```
ax1.legend(['IC_1','IC_2','IC_3','IC_4'])

ax2.plot(t_vec, y_vec1, 'r-', t_vec, y_vec2, 'b-',t_vec, y_vec3, 'g-',t_vec, y_vec4,
ax2.set_ylabel(r'$y(t)$')
ax2.set_xlim([t_initial, t_final])
ax2.grid()
ax2.legend(['IC_1','IC_2','IC_3','IC_4'])

ax3.plot(t_vec, z_vec1, 'r-', t_vec, z_vec2, 'b-',t_vec, z_vec3, 'g-',t_vec, z_vec4,
ax3.set_ylabel(r'$z(t)$')
ax3.set_xlim([t_initial, t_final])
ax3.grid()
ax3.legend(['IC_1','IC_2','IC_3','IC_4'])
```

Out[8]: <matplotlib.legend.Legend at 0x7fe290d8f490>



Compared with (f), with larger ρ, the system became:

- Divergence occurs
- No equilibrium state, chaotic
- Aperiodic responses
- Very sensitive to ICs

Q2- Chaos in discrete systems – The Logistic map (50 pt. + 5 bonus pt.)

Below an equation/logistic map is given, which describes the dynamics of the population growth of an animal community under limited resources. The growth is described discretely, i.e. it refers to the number of individuals in the community in each generation. The parameter that describes the growth rate of the population is denoted by r.

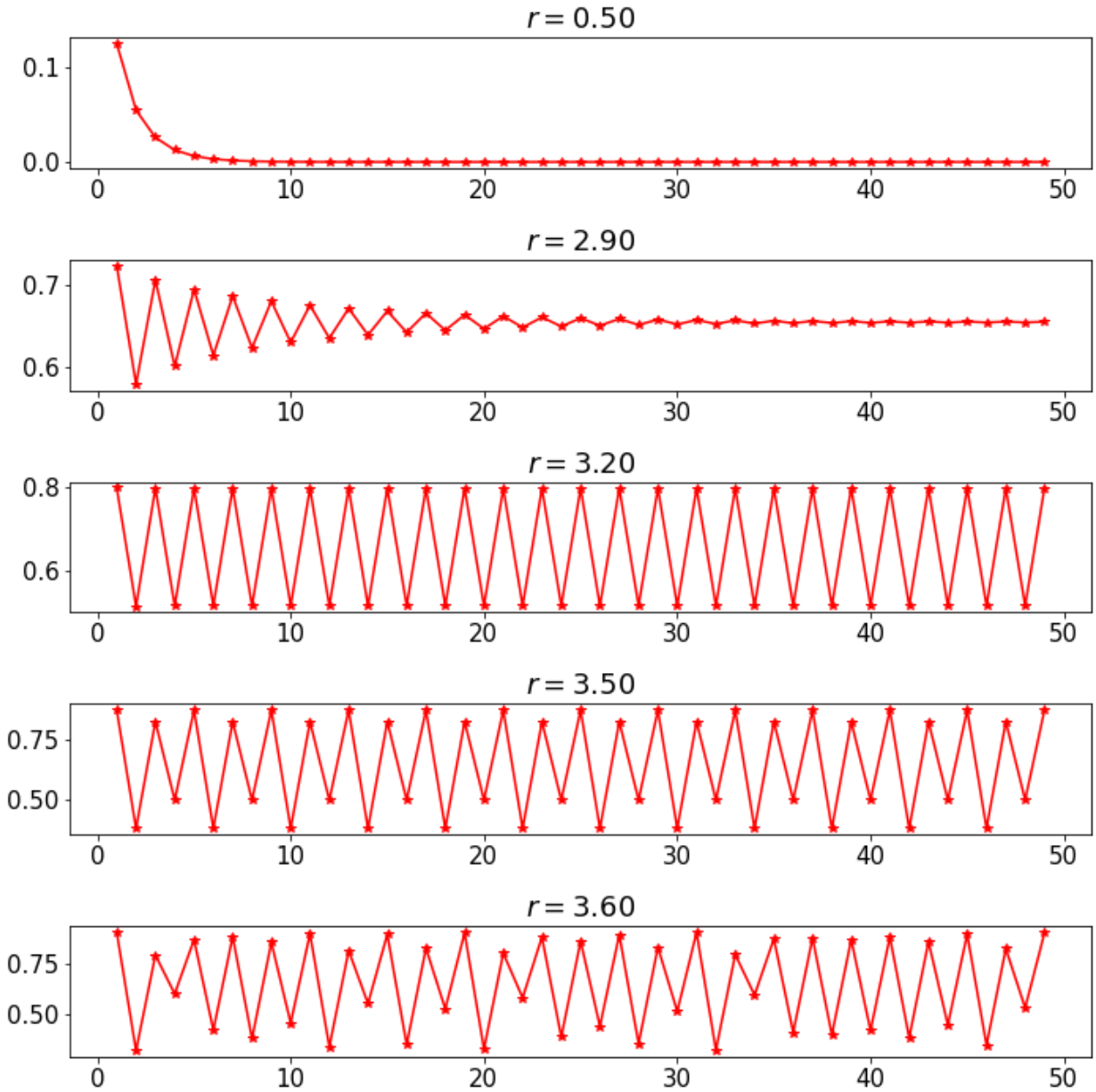
$$x_{n+1} = rx_n(1 - x_n)$$

In the current exercise, the dynamics/behavior of the logistic map will be analyzed for the range of: $0 \leq r \leq 4$

a. (14 pt.) Write a Python code that returns the series obtained for a given value of parameter r .

In [9]:
```
def iterator(r):
  # define system parameters
  iter_num = 50
  x0 = 0.5
  iter_vec = np.zeros(iter_num)
  x = np.zeros(iter_num)
  x[0]=x0

  for i in range(iter_num-1):
    x[i+1] = r*x[i]*(1-x[i])

  iter_vec = np.arange(1, iter_num) # the serial number of the iterations
  return x, iter_vec
```

In [10]:
```
rs = [0.5, 2.9, 3.2, 3.5, 3.6]
fig, axs = plt.subplots(5,figsize=(10,10))

for i, r in enumerate(rs):
  [x, iter_vec] = iterator(r)
  axs[i].plot(iter_vec, x[1:], 'r*-')
  axs[i].set_title(r'$r={0:.2f}$'.format(r))
fig.tight_layout()
```



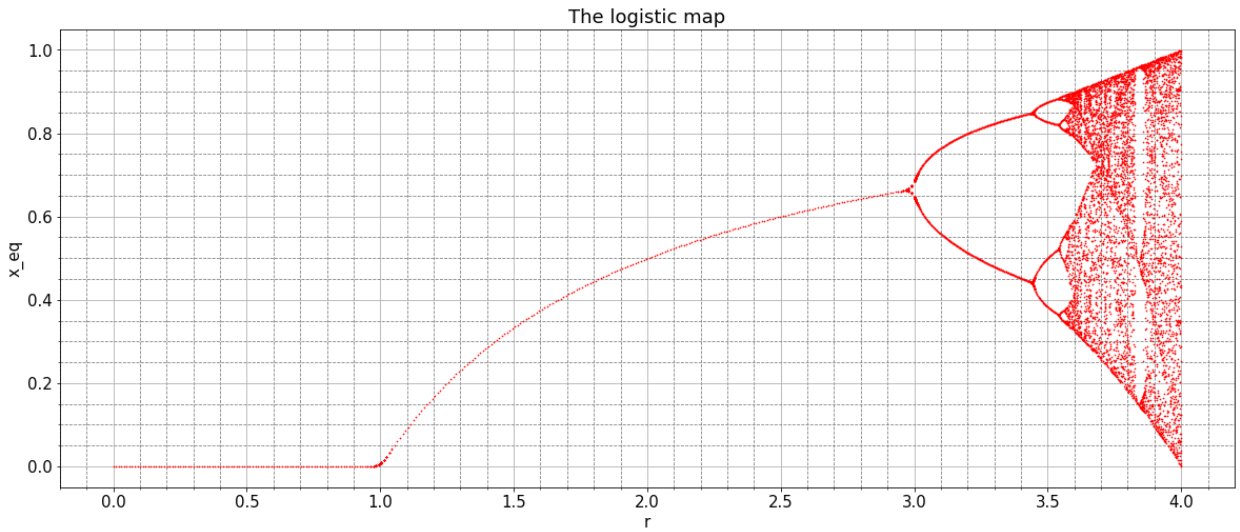
From the aboved graph, we noticed that `x` converges to a certain value for `r=0.5, 2.9` and `x` jumps to some certain values and stays oscillation in bounded region for `r=3.2, 3.5 3.6` .

b. (14 pt.) Write a Python code that plots the series obtained in region: $0 \leq r \leq 4$. The code should locate and display the equilibrium values of the population size as a function of the parameter `r` .

In [11]:

```
def iterator(r):
  # define system parameters
  iter_num, x0 = 150, 0.5
  iter_vec = np.zeros(iter_num)
  x = np.zeros(iter_num)
  x[0]=x0

  for i in range(iter_num-1):
    x[i+1] = r*x[i]*(1-x[i])

  iter_vec = np.arange(1, iter_num) # the serial number of the iterations
  x = np.round(x, 4)
  return x, iter_vec

rs = np.concatenate((np.linspace(0,3,300),np.linspace(3,4,500)), axis=0)
fig = plt.figure(figsize=(20,8))
m=0.8
Saved_Xs = [] # for saving solutions in next section
```

```
for i, r in enumerate(rs):
  [x_full,_] = iterator(r)
  x_steady_state = x_full[int(m*len(x_full)):]
  x_unique = np.unique(x_steady_state)
  temp = x_unique.tolist()
  Saved_Xs.append([r,len(temp),temp])
  plt.plot(np.ones_like(x_unique)*r,x_unique,'r.',markersize=1.3)

plt.title('The logistic map')
plt.xlabel('r')
plt.ylabel('x_eq')
plt.grid(True)
plt.minorticks_on()
plt.grid(b=True, which='minor', color=[0.5, 0.5, 0.5], linestyle='--')
```



c. (14 pt.) Find the values of the first six bifurcations from the graph obtained in the previous section manually and write down their values.

Answer First six bifurcations we found from graph:

$$r = [2.98492, 3.44289, 3.540977, 3.56196, 3.566485, 3.567448]$$

We rounded value of x_{eq} to 4 digits and found bifurcations by increasing of number of unique values.

d. (8 pt.) Calculate the following ratio: $\frac{r_n - r_{n-1}}{r_{n+1} - r_n}$ using $r_2, \ldots r_5$

In [12]:
```
r=[2.98492, 3.44289, 3.540977, 3.56196, 3.566485, 3.567448]
for i in range(4):
  ratio=(r[i+1]-r[i])/(r[i+2]-r[i+1])
  print('The ratio is', ratio)
```

```
The ratio is 4.669018320470601
The ratio is 4.67459371872464
The ratio is 4.637127071823133
The ratio is 4.6988577362408
```

Dynamic systems modeling using machine-learning methods

Q3 - Dynamic systems modeling using measured serial data (25 pt.)

a. (7 pt.) Load the serial data to dataframe using the pandas library.

In [13]:
```
# we upload the data file to github to
DATA_URL = 'https://raw.githubusercontent.com/afiretony/NLDML_final/main/proj_ex3.cs
DATA_DIR = pathlib.Path.home().joinpath('datasets')

def download_data(out_path=DATA_DIR, url=DATA_URL, force=False):
    pathlib.Path(out_path).mkdir(exist_ok=True)
    out_filename = os.path.join(out_path, os.path.basename(url))

    if os.path.isfile(out_filename) and not force:
        print(f'DATA {out_filename} exists, skipping download.')
    else:
        print(f'Downloading {url}...')
        with urllib.request.urlopen(url) as response, open(out_filename, 'wb') as ou
            shutil.copyfileobj(response, out_file)
        print(f'Saved to {out_filename}.')
    return out_filename

DATA_path = download_data(DATA_DIR, DATA_URL, False)
```

```
DATA /Users/chenhao/datasets/proj_ex3.csv exists, skipping download.
```

In [14]:
```
# read data
df = pd.read_csv(DATA_path)
x = df[['x']].values
x_dot = df[['x_dot']].values
m = x.size
print("First five entires of the table")
df.head()
```

```
First five entires of the table
```

Out[14]:

	x	x_dot
0	1.070	0.891
1	0.939	0.983
2	1.070	0.893
3	0.976	1.030
4	0.842	0.987

b. (7 pt.) Run the pySINDy library and find the system's equations of motion.

In [18]:
```
import pysindy as ps
differentiation_method = ps.FiniteDifference(order=2)
optimizer = ps.STLSQ(threshold=0.2)
feature_library = ps.PolynomialLibrary(degree=3)

model = ps.SINDy(
    # differentiation_method=differentiation_method,
    feature_library=feature_library,
    # optimizer=optimizer,
    feature_names=["x", "x_dot"]
)
x_train = np.stack((x.squeeze(), x_dot.squeeze()), axis=-1)
model.fit(x_train, t=0.002)
model.print()
```

```
x' = 0.993 x_dot
x_dot' = -0.991 x + 4.795 x_dot + -4.718 x^2 x_dot
```

Show the resulting equations of motion.

$$\dot{x} = 0.993 x_{dot}$$

$$\dot{x}_{dot} = -0.991 x + 4.795 x_{dot} - 4.718 x^2 x_{dot}$$

The program did not aware $\dot{x} = x_{dot}$, our system is:

$$\ddot{x} = -0.991x + 4.795\dot{x} - 4.718x^2\dot{x}$$

c. (7 pt.) Write a numerical solver that uses the EOM you received from SINDy, and compare the numerical simulation obtained from this EOM with the measured data. Compare those results both using both time histories and in state space plot.

Answer:

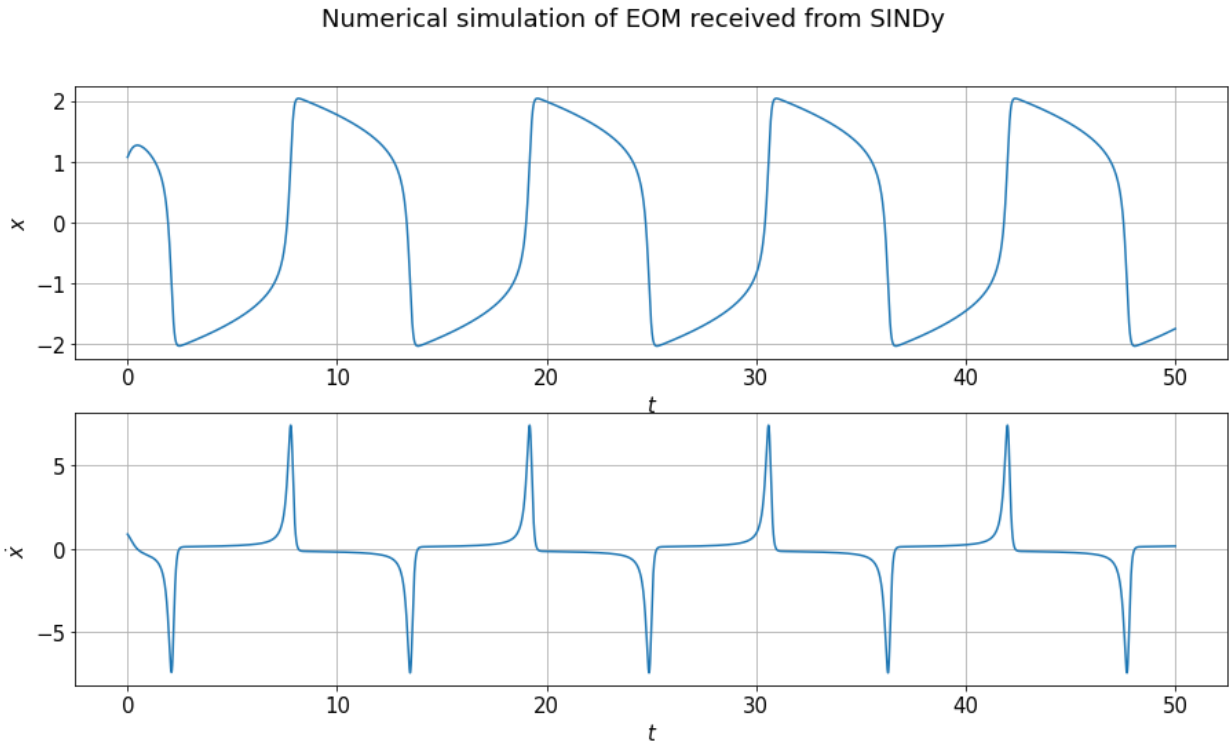
$$X = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}, \dot{X} = \begin{pmatrix} \dot{x} \\ \ddot{x} \end{pmatrix}$$

Our system is:

$$\dot{X} = \begin{pmatrix} \dot{x} \\ -0.991x + 4.795\dot{x} - 4.718x^2\dot{x} \end{pmatrix}$$

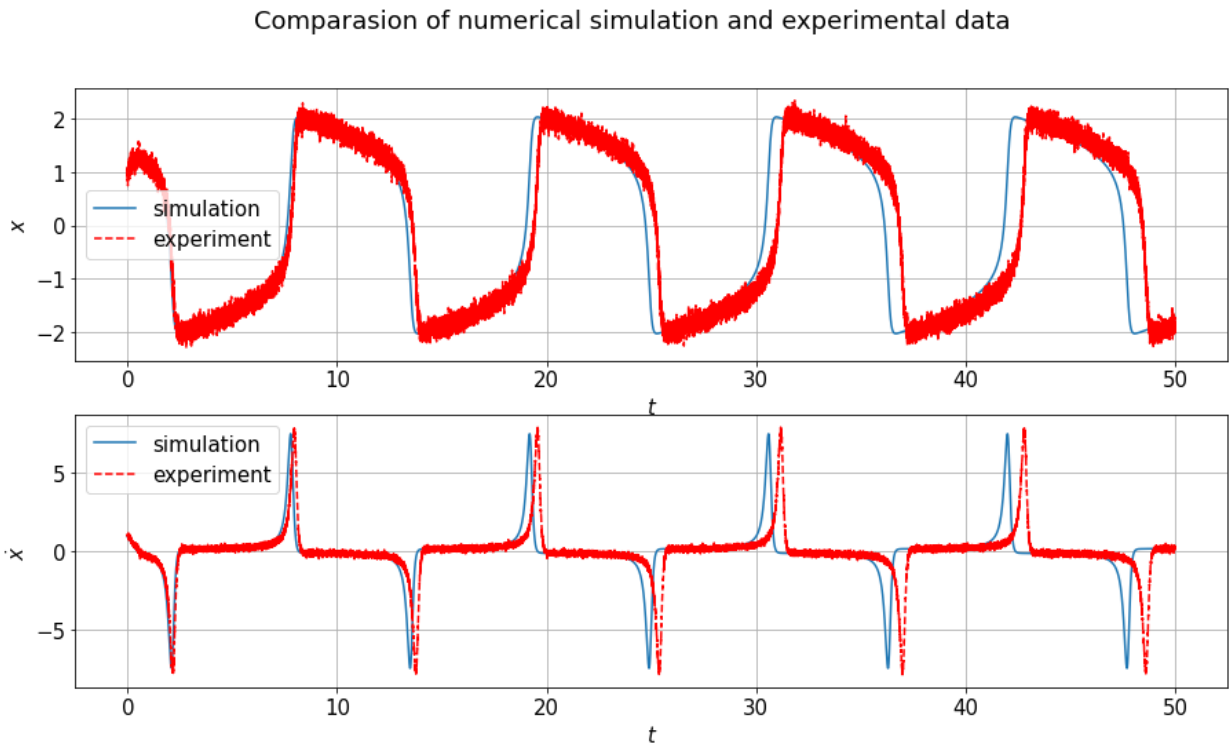
In [19]:

```
from scipy.integrate import odeint

# Defining time vector, vector of initial conditions, and system parameter
tf, dt = 50, 0.002
t_vec = np.arange(0, tf, dt)

# DEfining the numerical solver for linear oscillator
def NL_EOM(x_vec, t):
    x, x_dot = x_vec
    x_vec_dot = [x_dot,-0.991*x+4.795*x_dot - 4.718*x**2*x_dot]
    return x_vec_dot

x0 = 1.070
v0 = 0.891

# ode run
IC_vec = [x0, v0]
sol = odeint(NL_EOM, IC_vec, t_vec)
x, x_dot = sol[:, 0], sol[:, 1]

# Plot
fig, (ax1, ax2) = plt.subplots(2,figsize=(15,8))
fig.suptitle('Numerical simulation of EOM received from SINDy')
ax1.plot(t_vec, x)
ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.grid(True)
ax2.plot(t_vec, x_dot)
ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$\dot x$')
ax2.grid(True)
```



In [20]:
```
# Plot
fig, (ax1, ax2) = plt.subplots(2,figsize=(15,8))
fig.suptitle('Comparasion of numerical simulation and experimental data')
ax1.plot(t_vec, x)
ax1.plot(t_vec, x_train[:,0],'r--')
ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.grid(True)
ax1.legend(['simulation','experiment'])
ax2.plot(t_vec, x_dot)
ax2.plot(t_vec, x_train[:,1],'r--')
ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$\dot x$')
ax2.grid(True)
ax2.legend(['simulation','experiment'])
```

Out[20]: <matplotlib.legend.Legend at 0x7fe275f1ee80>



d. (4 pt.) Calculate the error between the two responses using the RMSE metric. Do you think the approximation is good enough? What causes the error and how can it be reduced in reality?

In [21]:
```
RMSE_x = np.sqrt(np.mean((x_train[:,0]-x)**2))
RMSE_xdot = np.sqrt(np.mean((x_train[:,1]-x_dot)**2))
print('RMSE of x is: ', np.round(RMSE_x,5))
print('RMSE of x_dot is: ', np.round(RMSE_xdot,5))
```

```
RMSE of x is:  0.71757
RMSE of x_dot is:  1.80972
```

The smaller RMSE is, the less the error. From the graph and calculation, our approximation is not bad. There are several ways to reduce the error:

- Adjust the initial conditions when simulating. As dipicted from the graph, there are some shift in data that resulted mis-alignment and error.
- More training data. It is more of an emperial conclusion than theoretical: adding more examples to the training set monotonically increases the accuracy of the model.
- Higher degree of system. Though high degree system can result problem of overfitting, it can help underfitted model. In our case the degree seems to be sufficient choice.

Q4 - Dynamic systems modeling using serial data – selection of basic functions (25 pt.)

The EOM of a free-damped pendulum is given as follows:

$$\ddot{x} + 2\zeta\omega\dot{x} + \omega^2 \sin x = 0$$

a. (7 pt.) Perform an order reduction and write a numerical solver that returns the response of the system.

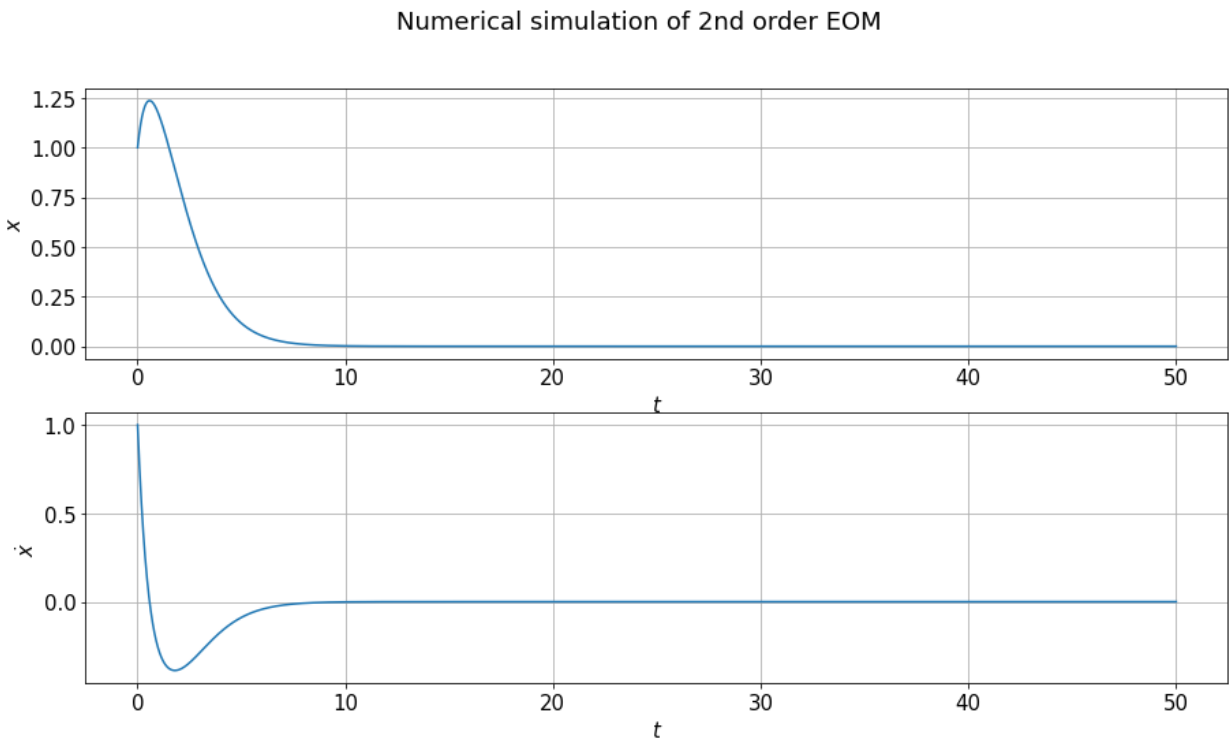
Answer:

$$X = \begin{pmatrix} x \\ \dot{x} \end{pmatrix}$$

$$\dot{X} = \begin{pmatrix} \dot{x} \\ \ddot{x} \end{pmatrix} = \begin{pmatrix} \dot{x} \\ -2\zeta\omega\dot{x} - \omega^2 \sin x \end{pmatrix}$$

In [22]:
```
from scipy.integrate import odeint

# Defining time vector, vector of initial conditions, and system parameter
tf, dt = 50, 0.002
t_vec = np.arange(0, tf, dt)

# DEfining the numerical solver for linear oscillator
def NL_EOM(x_vec, t, zeta, omega):
    x, x_dot = x_vec
    x_vec_dot = [x_dot,-2*zeta*omega*x_dot - omega**2*np.sin(x)]
    return x_vec_dot

x0 = 1
v0 = 1
zeta = 1
omega = 1

# ode run
IC_vec = [x0, v0]
sol = odeint(NL_EOM, IC_vec, t_vec, args=(zeta, omega))
x, x_dot = sol[:, 0], sol[:, 1]
```

```
# Plot
fig, (ax1, ax2) = plt.subplots(2,figsize=(15,8))
fig.suptitle('Numerical simulation of 2nd order EOM')
ax1.plot(t_vec, x)
ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.grid(True)
ax2.plot(t_vec, x_dot)
ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$\dot x$')
ax2.grid(True)
```



In [23]:
```
differentiation_method = ps.FiniteDifference(order=2)
optimizer = ps.STLSQ(threshold=0.2)
feature_library = ps.PolynomialLibrary(degree=3)

model = ps.SINDy(
    differentiation_method=differentiation_method,
    feature_library=feature_library,
    optimizer=optimizer,
    feature_names=["x", "x_dot"]
)
x_train = np.stack((x.squeeze(), x_dot.squeeze()), axis=-1)
model.fit(x_train, t=0.002)
model.print()
```

```
x' = 1.000 x_dot
x_dot' = -0.796 x_dot + -0.618 x^2 + -0.254 x x_dot + -0.413 x_dot^2 + -0.583 x^2 x_
dot + -0.179 x x_dot^2
```

Resulting EOM:

$$\ddot{x} = -0.796\dot{x} + -0.618x^2 + -0.254x\dot{x} + -0.413\dot{x}^2 + -0.583x^2\dot{x} + -0.179x\dot{x}^2$$

It is not identical to the original EOM:

$$\ddot{x} + 2\dot{x} + \sin x = 0$$

Why?

First of all, we used polynomial library which doesn't contain sinusoidal functions, so the system is unware of existence of sinusoidal non-linear term and trying to approximate the system using

polynomial terms. Secondly, the system is a damped 2nd order system, meaning it doesn't have much oscillating behaviour, so the system can be easily approximated using non-oscillating function.

c. (7 pt.) Choose basic functions that are appropriate for the current dynamical system so that the equations obtained from the characterization process will be as close as possible to the original equation. You can use the basic functions given in the pySINDY library or define custom basic functions. Explain why you chose these basic functions.

Answer:

Since there are sinusoidal term in the system, besides polynomial library, we would like to add a sinusoidal function into the custom library as $f_0(x) = sin(x)$.

In [24]:
```
from pysindy.feature_library import PolynomialLibrary, CustomLibrary
from pysindy.feature_library import ConcatLibrary
functions = [lambda x : np.sin(x)]
lib_Polynomial = PolynomialLibrary()
lib_custom = CustomLibrary(library_functions=functions)
lib_concat = ConcatLibrary([lib_custom, lib_Polynomial])

differentiation_method = ps.FiniteDifference(order=2)
optimizer = ps.STLSQ(threshold=0.2)

model = ps.SINDy(
    # differentiation_method=differentiation_method,
    feature_library=lib_concat,
    # optimizer=optimizer,
    feature_names=["x", "x_dot"]
)
x_train = np.stack((x.squeeze(), x_dot.squeeze()), axis=-1)
model.fit(x_train, t=0.002)
model.print()
```

```
x' = 1.000 x_dot
x_dot' = -1.000 f0(x) + -2.000 x_dot
```

Result from pySINDY:

$$\ddot{x} = -1.000 \sin x - 2.000\dot{x}$$

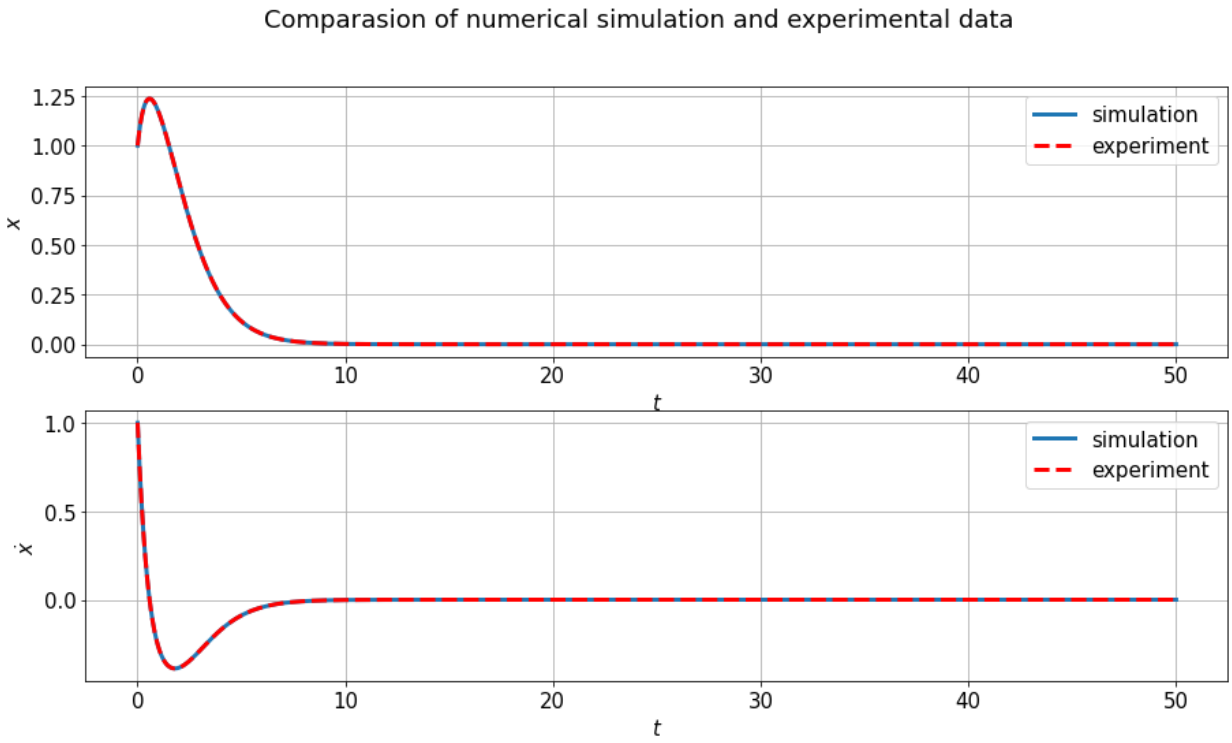
Perfectly identical to the original system! It testified our assumption that adding a sinusoidal library would greatly improve the results.

There should be no error since the approximated EOM is identical to original one, however, we still perform a simulation for the sake of instructions.

In [25]:
```
# Defining time vector, vector of initial conditions, and system parameter
tf, dt = 50, 0.002
t_vec = np.arange(0, tf, dt)

# DEfining the numerical solver for linear oscillator
def NL_EOM(x_vec, t):
    x, x_dot = x_vec
    x_vec_dot = [x_dot,-1.000*np.sin(x)-2.000*x_dot]
    return x_vec_dot

x0 = 1
v0 = 1

# ode run
IC_vec = [x0, v0]
sol = odeint(NL_EOM, IC_vec, t_vec)
SINDYx, SINDYx_dot = sol[:, 0], sol[:, 1]
```

In [26]:
```
# Plot
fig, (ax1, ax2) = plt.subplots(2,figsize=(15,8))
fig.suptitle('Comparasion of numerical simulation and experimental data')
ax1.plot(t_vec, SINDYx, linewidth=3)
ax1.plot(t_vec, x_train[:,0],'r--', linewidth=3)
ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.grid(True)
ax1.legend(['simulation','experiment'])
ax2.plot(t_vec, SINDYx_dot, linewidth=3)
ax2.plot(t_vec, x_train[:,1],'r--', linewidth=3)
ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$\dot x$')
ax2.grid(True)
ax2.legend(['simulation','experiment'])
```

Out[26]: <matplotlib.legend.Legend at 0x7fe2777ab130>



In [27]:
```
# RMSE error
RMSE_x = np.sqrt(np.mean((x_train[:,0]-SINDYx)**2))
RMSE_xdot = np.sqrt(np.mean((x_train[:,1]-SINDYx_dot)**2))
print('RMSE of x is: ', np.round(RMSE_x,5))
print('RMSE of x_dot is: ', np.round(RMSE_xdot,5))
```

```
RMSE of x is:  0.0
RMSE of x_dot is:  0.0
```

We compared simulation results and original results, they are the same and the RMSE error for both $x, \dot{x}$ are zero.

Q5 - Modeling of dynamical systems using noisy serial data – examination of the effect of measurement noise (10 bonus pt.)

a. (2 pt.) Observe the Lorenz EOM (Equation 1). Perform numerical simulation. Take the signals you received for the three degrees of freedom,and enter them as input to the SINDy algorithm for receiving the system's EOM. If necessary, choose the base functions accordingly.

Answer: Remind of Lorenz EOM:

$$\dot{x} = \sigma(y - x) \tag{12}$$
$$\dot{y} = \rho x - y - xz \tag{13}$$
$$\dot{z} = xy - \beta z \tag{14}$$
$$\rho, \sigma, \beta = 10, 10, 8/3 \tag{15}$$

Based on the EOM, 2nd order polynomial library is sufficient to approximate the system.

In [28]:
```
# numerical simulation - DATA-ACQUISITION
def Lorenz_EOM(x_vec, t, sigma, rho, beta):
    x, y, z = x_vec
    x_vec_dot = [sigma*(y - x), rho*x-y-x*z, x*y-beta*z]
    return x_vec_dot

sigma, rho, beta = 10, 10, 8/3
tf = 15
dt = 0.001
t_vec = np.arange(0,tf,dt)
x0, y0, z0 = -8, 8, 27
IC_vec = [x0, y0, z0]
sol_gt = odeint(Lorenz_EOM, IC_vec, t_vec, args=(sigma, rho, beta))
x_vec, y_vec, z_vec = sol_gt[:,0], sol_gt[:,1], sol_gt[:,2]

x_train = np.stack((x_vec, y_vec, z_vec), axis=-1)
```

In [29]:
```
# Fit using sindy
def SINDY_fit(x_train):
  np.random.seed(42)
  differentiation_method = ps.FiniteDifference(order=2)
  optimizer = ps.STLSQ(threshold=0.4)
  feature_library = ps.PolynomialLibrary(degree=2)

  model = ps.SINDy(
      differentiation_method=differentiation_method,
      feature_library=feature_library,
      optimizer=optimizer,
      feature_names=['x','y','z']
  )
  model.fit(x_train, t=dt)
  x0, y0, z0 = -8, 8, 27
  sim = model.simulate([x0, y0, z0], t=t_vec)
  SINDYx, SINDYy, SINDYz = sim[:,0], sim[:,1], sim[:,2]
  model.print()
  return SINDYx, SINDYy, SINDYz

[SINDYx, SINDYy, SINDYz] = SINDY_fit(x_train)
```

```
x' = -10.000 x + 10.000 y
y' = 9.999 x + -1.000 y + -1.000 x z
z' = -2.667 z + 1.000 x y
```

Lorenz EOM obtained using SINDy:

$$\dot{x} = -10.000x + 10.000y \tag{16}$$
$$\dot{y} = 9.999x - 1.000y - 1.000xz \tag{17}$$
$$\dot{z} = -2.667z + 1.000xy \tag{18}$$

In [30]:
```
def cal_err(x_train, SINDYx, SINDYy, SINDYz):
  # RMSE error
  RMSE_x = np.sqrt(np.mean((x_train[:,0]-SINDYx)**2))
  RMSE_y = np.sqrt(np.mean((x_train[:,1]-SINDYy)**2))
  RMSE_z = np.sqrt(np.mean((x_train[:,2]-SINDYz)**2))
  return RMSE_x, RMSE_y, RMSE_z

[RMSE_x, RMSE_y, RMSE_z] = cal_err(x_train, SINDYx, SINDYy, SINDYz)
print('RMSE of x is: ', np.round(RMSE_x,5))
print('RMSE of y is: ', np.round(RMSE_y,5))
print('RMSE of z is: ', np.round(RMSE_z,5))
```

```
RMSE of x is:  0.00089
RMSE of y is:  0.00107
RMSE of z is:  0.00138
```

The fit is good enough.

b. (2 pt.) Additive noise - Define three vectors that have identical lengths as the response vectors, and contain zero-mean normally-distribute values with a standard deviation of one.

In [31]:
```
np.random.seed(42) # set random seed
mu, sigma = 0, 1 # mean and standard deviation
epsilon = 1
noisy_x_train = x_train + epsilon*np.random.normal(mu, sigma, x_train.shape)
```

c. (2 pt.) Repeat section (a) for noisy data for increasing noise level ϵ, until the solution obtained using pySINDY loses its validity. View the equations of motion obtained from the SINDy algorithm for each case. Plot on a single graph the responses obtained from the equations of motion for increasing noise intensities (including zero noise).

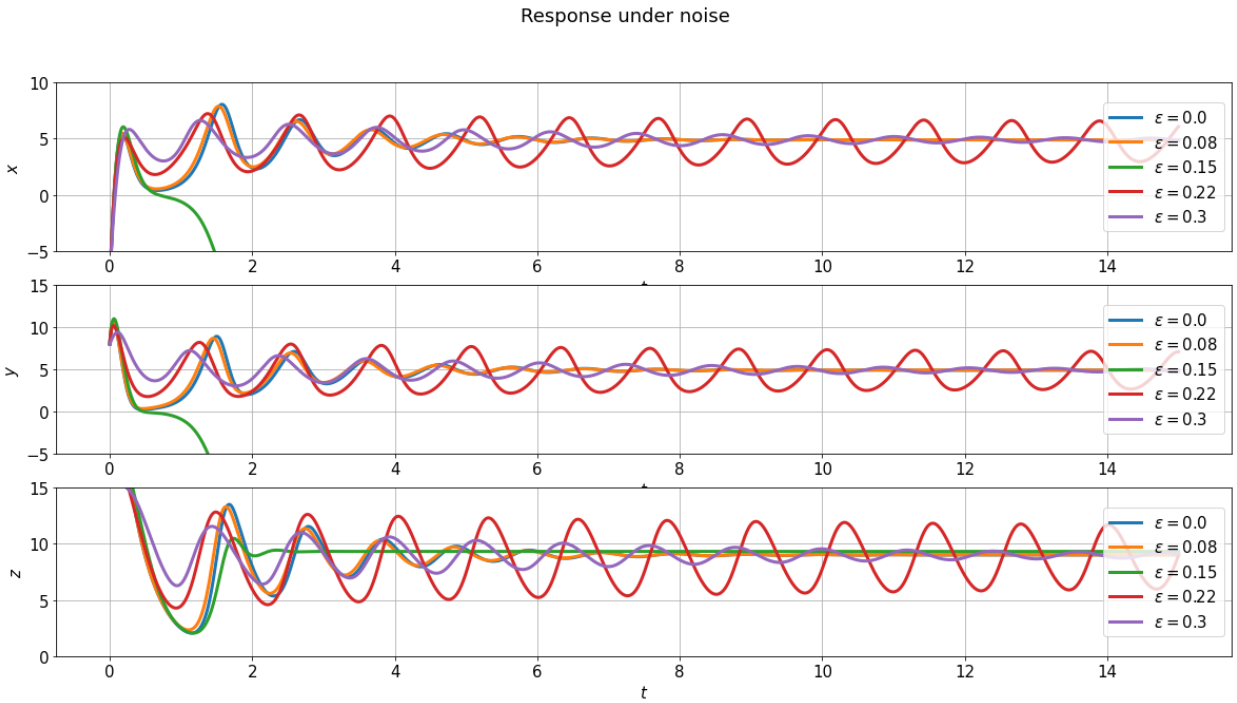
In [42]:
```
epsilons = np.linspace(0,0.3,5)
SINDYs = np.zeros((len(epsilons),3,len(t_vec)))
ERRORs = np.zeros((len(epsilons),3))
legend_list=[]

for i, epsilon in enumerate(epsilons):
  noisy_x_train = x_train + epsilon*np.random.normal(mu, sigma, x_train.shape)
  [SINDYx, SINDYy, SINDYz] = SINDY_fit(noisy_x_train)
  [RMSE_x, RMSE_y, RMSE_z] = cal_err(x_train, SINDYx, SINDYy, SINDYz)
  SINDYs[i] = [SINDYx, SINDYy, SINDYz]
  ERRORs[i] = [RMSE_x, RMSE_y, RMSE_z]
  legend_list.append(r'$\epsilon=$' + str(np.round(epsilon,2)))
```

```
x' = -10.000 x + 10.000 y
y' = 9.999 x + -1.000 y + -1.000 x z
z' = -2.667 z + 1.000 x y
x' = -9.939 x + 9.938 y
y' = 9.824 x + -1.017 y + -0.978 x z
z' = -2.682 z + 1.005 x y
x' = -9.674 x + 9.675 y
y' = -0.832 1 + 8.138 x + -0.886 x z
z' = 2.563 1 + 8.767 x + -6.625 y + -3.683 z + 0.835 y^2
x' = 8.515 1 + 0.618 x + -1.618 z + -0.473 x z + 0.550 y z
y' = 3.268 1 + 7.138 x + -0.437 z + -0.780 x z
z' = 2.635 1 + 9.145 x + -7.123 y + -3.619 z + 0.832 y^2
x' = 14.238 1 + 7.806 x + -7.179 y + -2.585 z + -0.836 x y + -0.461 x z + 0.944 y^2
+ 0.536 y z
y' = 12.104 1 + 2.055 x + 5.064 y + -2.748 z + -0.930 x y
z' = 2.551 1 + 9.538 x + -7.586 y + -3.520 z + 0.812 y^2
```

In [43]:
```
# Plot
fig, (ax1, ax2, ax3) = plt.subplots(3,figsize=(20,10))
fig.suptitle('Response under noise')
for i in range(len(epsilons)):
  ax1.plot(t_vec, SINDYs[i,0,:], linewidth=3)
ax1.set_xlabel(r'$t$')
ax1.set_ylabel(r'$x$')
ax1.grid(True)
ax1.set_ylim(-5,10)
ax1.legend(legend_list,loc='center right')

for i in range(len(epsilons)):
  ax2.plot(t_vec, SINDYs[i,1,:], linewidth=3)
ax2.set_xlabel(r'$t$')
ax2.set_ylabel(r'$y$')
ax2.grid(True)
ax2.set_ylim(-5,15)
```

```
ax2.legend(legend_list,loc='center right')

for i in range(len(epsilons)):
  ax3.plot(t_vec, SINDYs[i,2,:], linewidth=3)
ax3.set_xlabel(r'$t$')
ax3.set_ylabel(r'$z$')
ax3.grid(True)
ax3.set_ylim(0,15)
ax3.legend(legend_list,loc='center right')
```

Out[43]: <matplotlib.legend.Legend at 0x7fe28deda460>



d. (4 pt.) From which noise amplitude the solution obtained by pySINDY loses its validity?

Answer:

We can dipict from graph that from $\epsilon = 0.15$, the noise plagues results the solution obtained by pySINDY heavily and it loses validity because the prediction is useless. The simulation results are very different from those trained with small noise. (unexpected osicillations when $t > 10s$ or value explodes somewhere)

Besides graphical evidence, we can compute RMSE error for each ϵ we performed simulation:

In [44]:
```
ERRORs.mean(axis=1).tolist()
data = {
    'eps': epsilons.tolist(),
    'RMSE': ERRORs.mean(axis=1).tolist()
}
df = pd.DataFrame(data)
df
```

Out[44]:

	eps	RMSE
0	0.000	0.001113
1	0.075	0.267769
2	0.150	8.267904
3	0.225	1.957434
4	0.300	1.363706

From the table, when ϵ larger or equal to 0.15, RMSE is huge, the solution losses validity.