PSE: Apprentissage Profond: Neural ODE pour Systèmes Dynamiques

Fractional Lorenz system

Hannah Plath et Thiago Maffei

Prof. Alexandre Allauzen

Importing libraries

```
#########################################################
# Useful imports, preliminary commands, etc

# Numpy and maths
import math
import numpy as np
from scipy.linalg import expm

# Matplotlib and figures-related commands
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
sns.color_palette("bright")
import matplotlib.cm as cm
```

Generating the data

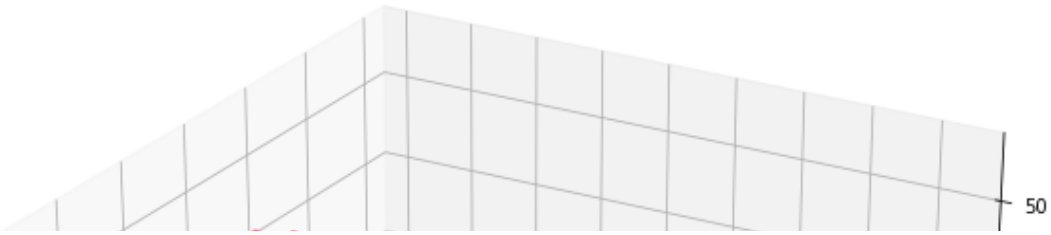
The equations describing the Lorenz system are represented below, with σ, ρ and β fixed parameters. So the problem boils down to three degrees of freedom.

$$\frac{dx}{dt} = \sigma(y - x)$$
$$\frac{dy}{dt} = x(\rho - z) - y$$
$$\frac{dz}{dt} = xy - \beta z$$

The class below generate the data from wich the model will try to learn.

```
class LorenzData():
```

```
  def __init__(self, rho = 28, sigma = 10, beta = 8/3, init_values = [0.01, 0, 0],
    self.time = np.linspace(0,tf, n_points)
    self.data = np.zeros((n_points, 3))
    self.data[0] = init_values
    x = init_values[0]
    y = init_values[1]
    z = init_values[2]

    for n in range(n_points - 1):
      dx = sigma*(y - x)*dt
      dy = (x*(rho - z) - y)*dt
      dz = (x*y - beta*z)*dt

      x += dx
      y += dy
      z += dz

      self.data[n+1] = [x,y,z]

  def getData(self):
    return self.data, self.time

  def scatterGraph(self, size = (12,10)):
      fig = plt.figure(figsize=size)
      ax = plt.axes(projection='3d')
      ax.plot3D(self.data[:,0], self.data[:,1], self.data[:,2], 'blue')
      ax.scatter3D(self.data[:,0], self.data[:,1], self.data[:,2], color = 'red');


lorenz = LorenzData()
data, times_np = lorenz.getData()
lorenz.scatterGraph()
```



ODE Solver

We determine both ODE solvers to "solve" the neural network (do the feedforward propagation) and also to apply the adaptative step size.


```
def ode_solve(x0, t0, t1, f, n_steps = 1):
    """
    Simplest Euler ODE initial value solver
    """
    h = (t1 - t0)/n_steps
    tn = t0
    yn = x0
    for i_step in range(n_steps):
        a1 = yn + h*f(yn)
        tn = tn + h
    return a1


def ode_solve_adaptive(x0, t0, t1, f, n_steps = 1):
  """
  Euler-2step ODE initial value solver
  """
  global hp
  hp = (t1 - t0)/n_steps
  tn = t0
  yn = x0
  for i_step in range(n_steps):
    a2 = yn + (hp/2)*f(yn) + (hp/2)*f(yn + hp/2)
    tn = tn + hp/2
  return a2
```

Neural ODE

We define the Neural ODE class (ODENet) and its methods needed for learning.

```
import torch as th
import torch.nn as nn
from torch.nn  import functional as F
```

```
class ODENet(nn.Module):
    def __init__(self, dim):
        super(ODENet, self).__init__()
        # modules with parameters
        self.nn = nn.Sequential(
            nn.Linear(dim,2*dim),
            nn.ReLU(),
            nn.Linear(2*dim, 2*dim),
            nn.ReLU(),
            nn.Linear(2*dim, dim),
        )

    def forward (self, z):
        return self.nn(z)

    def unfold_ode(self, z0,times, solver, nsteps=1):
        bs, *z_shape = z0.size()
        time_len = times.size(0)
        zs = th.zeros(time_len, bs, *z_shape).to(z0)
        zs[0] = z0
        for i_t in range(time_len - 1):
            ODENet.delta_t = times[i_t+1] - times[i_t]
            z0 = solver(z0, times[i_t], times[i_t+1], self.nn,nsteps)
            zs[i_t+1] = z0
        return zs

obs = th.Tensor(data) # data in torch
z0 = obs[0]
times = th.from_numpy(times_np).to(z0)
print(times.shape)
```

```
torch.Size([1000])
```

Training for a fixed step size

Here we have the training loop for our model using a fixed step size (5).

```
m = ODENet(3)

z0 = obs[0]

with th.no_grad():
    zs = m.unfold_ode(z0,times, ode_solve)

optimizer = th.optim.Adam(m.parameters(),lr=1e-3)
wsize = 1 # window size
indices = np.array(range(times.shape[0]-wsize),dtype=np.int)
print("nb ex.", indices.shape)
nepoch  = 500
nprint = 10
```

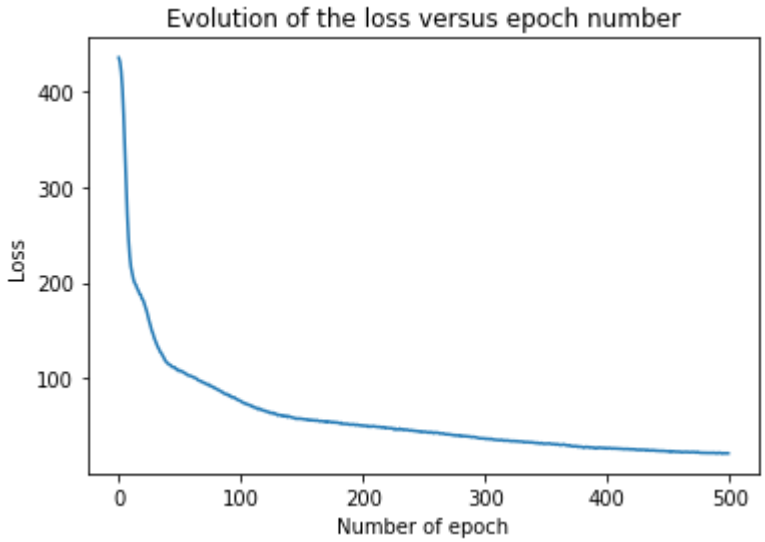
```
losses = th.zeros(nepoch)
ns = 5

for e in range(nepoch):
    np.random.shuffle(indices)
    ave_loss = th.zeros(1)
    for i in indices:
        x = obs[i]

        out = m.unfold_ode(x,times[i:i+wsize+1],ode_solve,ns)[-1]
        loss = F.mse_loss(out, obs[i+wsize].detach())

        ave_loss += loss
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

    if (e%nprint) == 0:
        print(e, ave_loss, ns)
    losses[e] = ave_loss.item()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:11: DeprecationW
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/de
  # This is added back by InteractiveShellApp.init_path()
nb ex. (999,)
0 tensor([434.9224], grad_fn=<AddBackward0>) 5
10 tensor([215.9942], grad_fn=<AddBackward0>) 5
20 tensor([181.7569], grad_fn=<AddBackward0>) 5
30 tensor([138.9993], grad_fn=<AddBackward0>) 5
40 tensor([115.6360], grad_fn=<AddBackward0>) 5
50 tensor([108.0654], grad_fn=<AddBackward0>) 5
60 tensor([101.9184], grad_fn=<AddBackward0>) 5
70 tensor([95.3735], grad_fn=<AddBackward0>) 5
80 tensor([88.8238], grad_fn=<AddBackward0>) 5
90 tensor([81.8790], grad_fn=<AddBackward0>) 5
100 tensor([76.1440], grad_fn=<AddBackward0>) 5
110 tensor([70.7425], grad_fn=<AddBackward0>) 5
120 tensor([66.0932], grad_fn=<AddBackward0>) 5
130 tensor([61.4111], grad_fn=<AddBackward0>) 5
140 tensor([59.8802], grad_fn=<AddBackward0>) 5
150 tensor([57.4181], grad_fn=<AddBackward0>) 5
160 tensor([56.1264], grad_fn=<AddBackward0>) 5
170 tensor([55.2526], grad_fn=<AddBackward0>) 5
180 tensor([53.4287], grad_fn=<AddBackward0>) 5
190 tensor([51.8216], grad_fn=<AddBackward0>) 5
200 tensor([50.7549], grad_fn=<AddBackward0>) 5
210 tensor([49.5343], grad_fn=<AddBackward0>) 5
220 tensor([48.1359], grad_fn=<AddBackward0>) 5
230 tensor([47.4940], grad_fn=<AddBackward0>) 5
240 tensor([45.6526], grad_fn=<AddBackward0>) 5
250 tensor([44.0251], grad_fn=<AddBackward0>) 5
260 tensor([43.3389], grad_fn=<AddBackward0>) 5
270 tensor([41.3356], grad_fn=<AddBackward0>) 5
280 tensor([39.3283], grad_fn=<AddBackward0>) 5
290 tensor([38.8860], grad_fn=<AddBackward0>) 5
300 tensor([37.1101], grad_fn=<AddBackward0>) 5
310 tensor([35.4236], grad_fn=<AddBackward0>) 5
320 tensor([34.5921], grad_fn=<AddBackward0>) 5
```

```
330 tensor([33.7666], grad_fn=<AddBackward0>) 5
340 tensor([32.9744], grad_fn=<AddBackward0>) 5
350 tensor([31.7705], grad_fn=<AddBackward0>) 5
360 tensor([30.9443], grad_fn=<AddBackward0>) 5
370 tensor([29.4879], grad_fn=<AddBackward0>) 5
380 tensor([27.9017], grad_fn=<AddBackward0>) 5
390 tensor([26.6805], grad_fn=<AddBackward0>) 5
400 tensor([26.7570], grad_fn=<AddBackward0>) 5
410 tensor([26.3586], grad_fn=<AddBackward0>) 5
420 tensor([25.8231], grad_fn=<AddBackward0>) 5
430 tensor([25.2546], grad_fn=<AddBackward0>) 5
440 tensor([24.2980], grad_fn=<AddBackward0>) 5
450 tensor([23.6754], grad_fn=<AddBackward0>) 5
460 tensor([23.3758], grad_fn=<AddBackward0>) 5
470 tensor([23.1048], grad_fn=<AddBackward0>) 5
480 tensor([22.1576], grad_fn=<AddBackward0>) 5
490 tensor([21.6490], grad_fn=<AddBackward0>) 5
```

```
#evolution of loss for the model
plt.plot(losses)
plt.title('Evolution of the loss versus epoch number')
plt.xlabel('Number of epoch')
plt.ylabel('Loss')
```

```
Text(0, 0.5, 'Loss')
```




```
#We·have·the·predictions
z0 = obs[0]
with th.no_grad():
  zpred = m.unfold_ode(z0,times,ode_solve, 5)
```

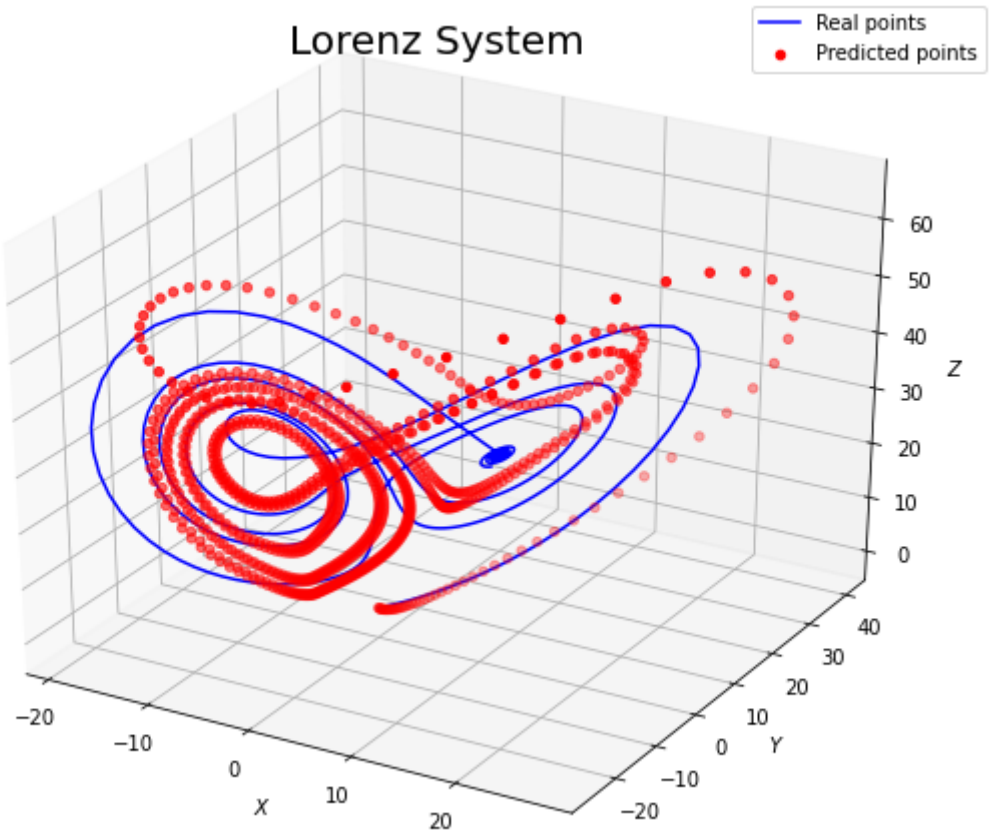
```
#Structure of our Neural net, can be used to analyse the weight for each layer
print(m)
```

```
ODENet(
  (nn): Sequential(
    (0): Linear(in_features=3, out_features=6, bias=True)
    (1): ReLU()
    (2): Linear(in_features=6, out_features=6, bias=True)
```

```
        (3): ReLU()
        (4): Linear(in_features=6, out_features=3, bias=True)
      )
    )
```

```
#Here·we·compare·the·prediction·of·our·model·with·the·real·points
fig = plt.figure(figsize=(10,8))
ax = plt.axes(projection='3d')
ax.scatter3D(zpred[:,0], zpred[:,1], zpred[:,2], color = 'red', label='Predicted pc
ax.plot3D(data[:,0], data[:,1], data[:,2], color = 'blue', label='Real points')
ax.set_xlabel('$X$')
ax.set_ylabel('$Y$')
ax.set_zlabel('$Z$')
ax.set_title('Lorenz System', fontsize=20)
plt.legend()
```

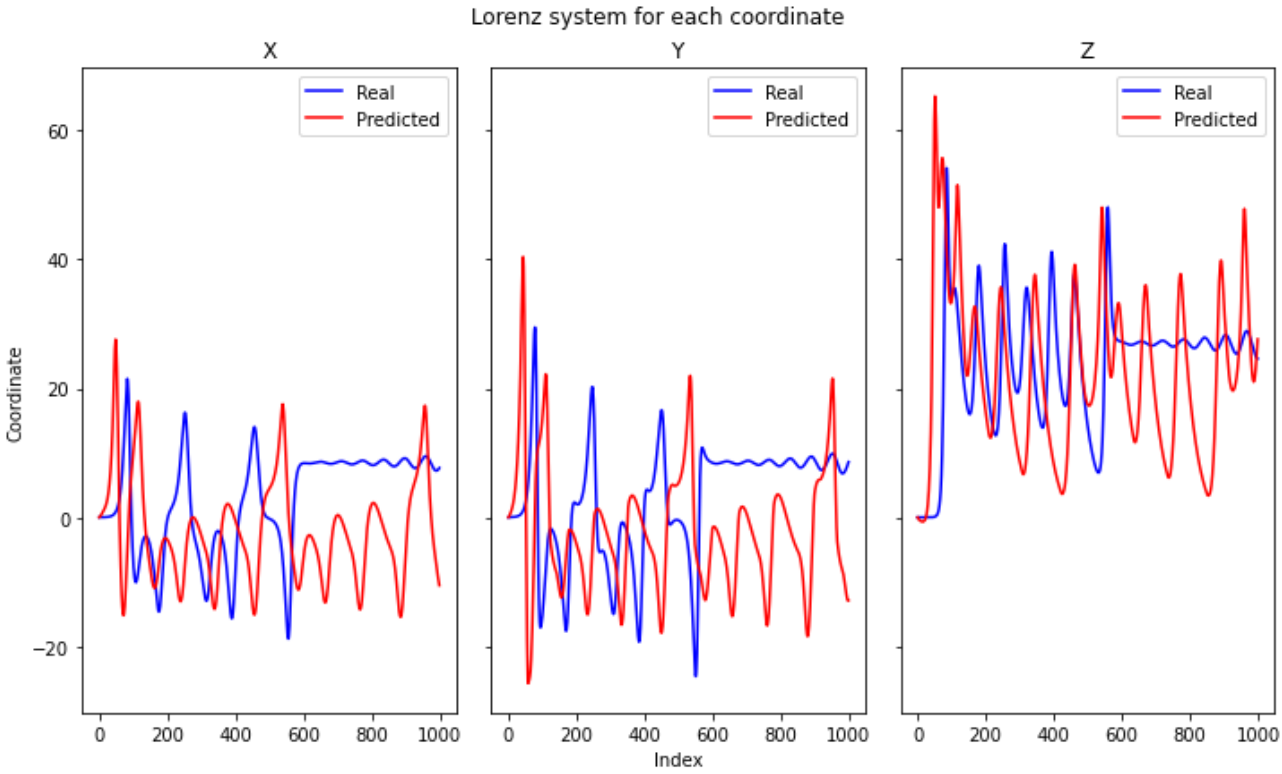
```
<matplotlib.legend.Legend at 0x7fc1fb1e9c10>
```




```
#Here we can plot the predicted and the real value for each direction of the space
fig, axs = plt.subplots(1,3, figsize=(10, 6), sharex=True, sharey=True)

axs[0].plot([i for i in range(0,1000)],data[:, 0], 'blue', label = 'Real')
axs[0].plot([i for i in range(0,1000)], zpred[:, 0], 'red', label = 'Predicted')
axs[0].set_title('X')
axs[0].legend()

axs[1].plot([i for i in range(0,1000)],data[:, 1], 'blue', label = 'Real')
axs[1].plot([i for i in range(0,1000)], zpred[:, 1], 'red', label = 'Predicted')
axs[1].set_title('Y')
```

```
axs[1].legend()

axs[2].plot([i for i in range(0,1000)],data[:, 2], 'blue', label = 'Real')
axs[2].plot([i for i in range(0,1000)], zpred[:, 2], 'red', label = 'Predicted')
axs[2].set_title('Z')
axs[2].legend()

fig.tight_layout()
fig.suptitle('Lorenz system for each coordinate')
plt.subplots_adjust(top=0.90)

axs[1].set_xlabel('Index')
axs[0].set_ylabel('Coordinate')
```

```
Text(58.0, 0.5, 'Coordinate')
```



Adaptative step size

Here we have the training loop for our model using an adaptative step size that starts after 250 epochs.

```
m = ODENet(3)

z0 = obs[0]

with th.no_grad():
```

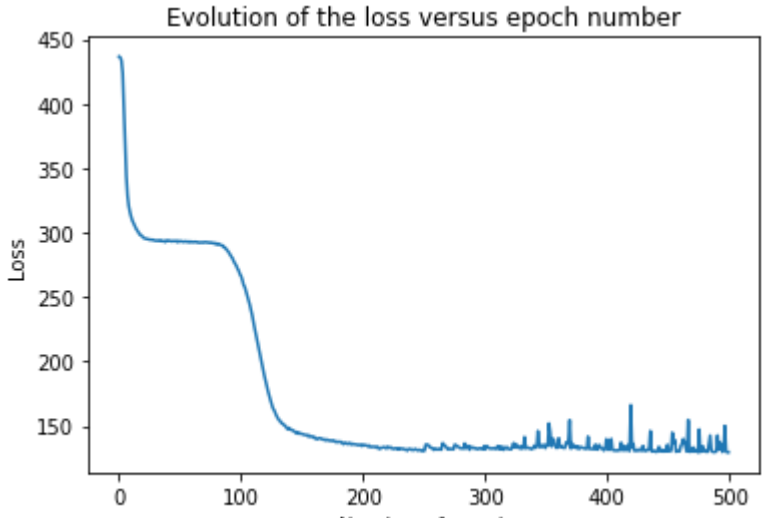
```
        zs = m.unfold_ode(z0,times, ode_solve)

    optimizer = th.optim.Adam(m.parameters(),lr=1e-3)
    wsize = 1 # window size
    indices = np.array(range(times.shape[0]-wsize),dtype=np.int)
    print("nb ex.", indices.shape)
    nepoch  = 500
    nprint = 10
    losses = th.zeros(nepoch)
    for e in range(nepoch):
        np.random.shuffle(indices)
        ave_loss = th.zeros(1)
        ns = 5
        for i in indices:
            x = obs[i]
            ns_adaptive = ns

            out = m.unfold_ode(x,times[i:i+wsize+1],ode_solve,ns)[-1]
            loss = F.mse_loss(out, obs[i+wsize].detach())

            """
            Here we use the method unfold_ode_adaptive to call the Euler 2-step
            method as ODE Solver.
            """

            out_adaptive = m.unfold_ode(x,times[i:i+wsize+1], ode_solve_adaptive, ns_ac
            loss_adaptive = F.mse_loss(out_adaptive, obs[i+wsize].detach())

            r = th.div(th.abs(th.sub(out, out_adaptive)), hp)[-1]

            erreur = 0.05
            if r.item() > erreur and e > 250:
              h_adap = 0.9*(erreur/r)*hp
              ns_adaptive = th.round(ODENet.delta_t / h_adap)
              ns = ns_adaptive.int()

            if ns > 20:
              ns = 20

            ave_loss += loss
            optimizer.zero_grad()
            loss.backward()
            optimizer.step()

        if (e%nprint) == 0:
            print(e, ave_loss, ns)
        losses[e] = ave_loss.item()
```

```
/usr/local/lib/python3.7/dist-packages/ipykernel_launcher.py:10: DeprecationW
Deprecated in NumPy 1.20; for more details and guidance: https://numpy.org/de
  # Remove the CWD from sys.path while we load stuff.
nb ex. (999,)
0 tensor([436.6155], grad_fn=<AddBackward0>) 5
10 tensor([312.0832], grad_fn=<AddBackward0>) 5
20 tensor([296.0985], grad_fn=<AddBackward0>) 5
```

```
30 tensor([293.7377], grad_fn=<AddBackward0>) 5
40 tensor([293.5625], grad_fn=<AddBackward0>) 5
50 tensor([293.2066], grad_fn=<AddBackward0>) 5
60 tensor([292.5550], grad_fn=<AddBackward0>) 5
70 tensor([292.5964], grad_fn=<AddBackward0>) 5
80 tensor([290.6629], grad_fn=<AddBackward0>) 5
90 tensor([284.4402], grad_fn=<AddBackward0>) 5
100 tensor([266.1534], grad_fn=<AddBackward0>) 5
110 tensor([230.5718], grad_fn=<AddBackward0>) 5
120 tensor([184.8501], grad_fn=<AddBackward0>) 5
130 tensor([156.0912], grad_fn=<AddBackward0>) 5
140 tensor([147.6205], grad_fn=<AddBackward0>) 5
150 tensor([143.8394], grad_fn=<AddBackward0>) 5
160 tensor([140.5389], grad_fn=<AddBackward0>) 5
170 tensor([139.3623], grad_fn=<AddBackward0>) 5
180 tensor([137.6761], grad_fn=<AddBackward0>) 5
190 tensor([135.5518], grad_fn=<AddBackward0>) 5
200 tensor([135.3111], grad_fn=<AddBackward0>) 5
210 tensor([133.6644], grad_fn=<AddBackward0>) 5
220 tensor([133.8221], grad_fn=<AddBackward0>) 5
230 tensor([131.8548], grad_fn=<AddBackward0>) 5
240 tensor([131.2513], grad_fn=<AddBackward0>) 5
250 tensor([130.6137], grad_fn=<AddBackward0>) 5
260 tensor([131.8203], grad_fn=<AddBackward0>) tensor(6, dtype=torch.int32)
270 tensor([131.8383], grad_fn=<AddBackward0>) tensor(7, dtype=torch.int32)
280 tensor([132.5658], grad_fn=<AddBackward0>) tensor(8, dtype=torch.int32)
290 tensor([133.1853], grad_fn=<AddBackward0>) tensor(9, dtype=torch.int32)
300 tensor([131.8257], grad_fn=<AddBackward0>) tensor(10, dtype=torch.int32)
310 tensor([134.3197], grad_fn=<AddBackward0>) tensor(11, dtype=torch.int32)
320 tensor([131.8555], grad_fn=<AddBackward0>) tensor(11, dtype=torch.int32)
330 tensor([134.1792], grad_fn=<AddBackward0>) tensor(13, dtype=torch.int32)
340 tensor([134.6575], grad_fn=<AddBackward0>) tensor(14, dtype=torch.int32)
350 tensor([135.3288], grad_fn=<AddBackward0>) tensor(15, dtype=torch.int32)
360 tensor([140.4067], grad_fn=<AddBackward0>) tensor(17, dtype=torch.int32)
370 tensor([132.3458], grad_fn=<AddBackward0>) tensor(18, dtype=torch.int32)
380 tensor([132.8960], grad_fn=<AddBackward0>) tensor(20, dtype=torch.int32)
390 tensor([135.5130], grad_fn=<AddBackward0>) 20
400 tensor([134.0237], grad_fn=<AddBackward0>) 20
410 tensor([131.1140], grad_fn=<AddBackward0>) 20
420 tensor([133.2559], grad_fn=<AddBackward0>) 20
430 tensor([130.8138], grad_fn=<AddBackward0>) 20
440 tensor([129.7639], grad_fn=<AddBackward0>) 20
450 tensor([130.2533], grad_fn=<AddBackward0>) 20
460 tensor([134.7279], grad_fn=<AddBackward0>) 20
470 tensor([133.2704], grad_fn=<AddBackward0>) 20
480 tensor([130.5467], grad_fn=<AddBackward0>) 20
490 tensor([142.5821], grad_fn=<AddBackward0>) 20
```

```
#Evolution of loss for the model
#It's possible to notice the instabilities after implementing the adaptative metho
#Also it's possible to notice that error is higher when comparing to a fixed small

plt.plot(losses)
plt.title('Evolution of the loss versus epoch number')
plt.xlabel('Number of epoch')
plt.ylabel('Loss')
```

```
Text(0, 0.5, 'Loss')
```



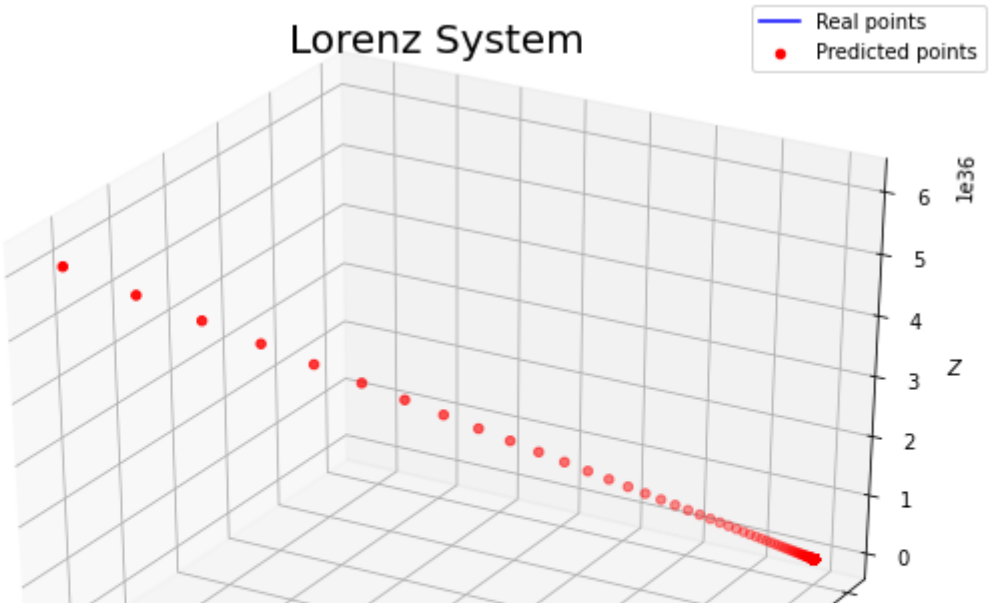

```
#We have the predictions
z0 = obs[0]
with th.no_grad():
  zpred = m.unfold_ode(z0,times,ode_solve_adaptive, 5)
```

```
print(m)
```

```
ODENet(
  (nn): Sequential(
    (0): Linear(in_features=3, out_features=6, bias=True)
    (1): ReLU()
    (2): Linear(in_features=6, out_features=6, bias=True)
    (3): ReLU()
    (4): Linear(in_features=6, out_features=3, bias=True)
  )
)
```

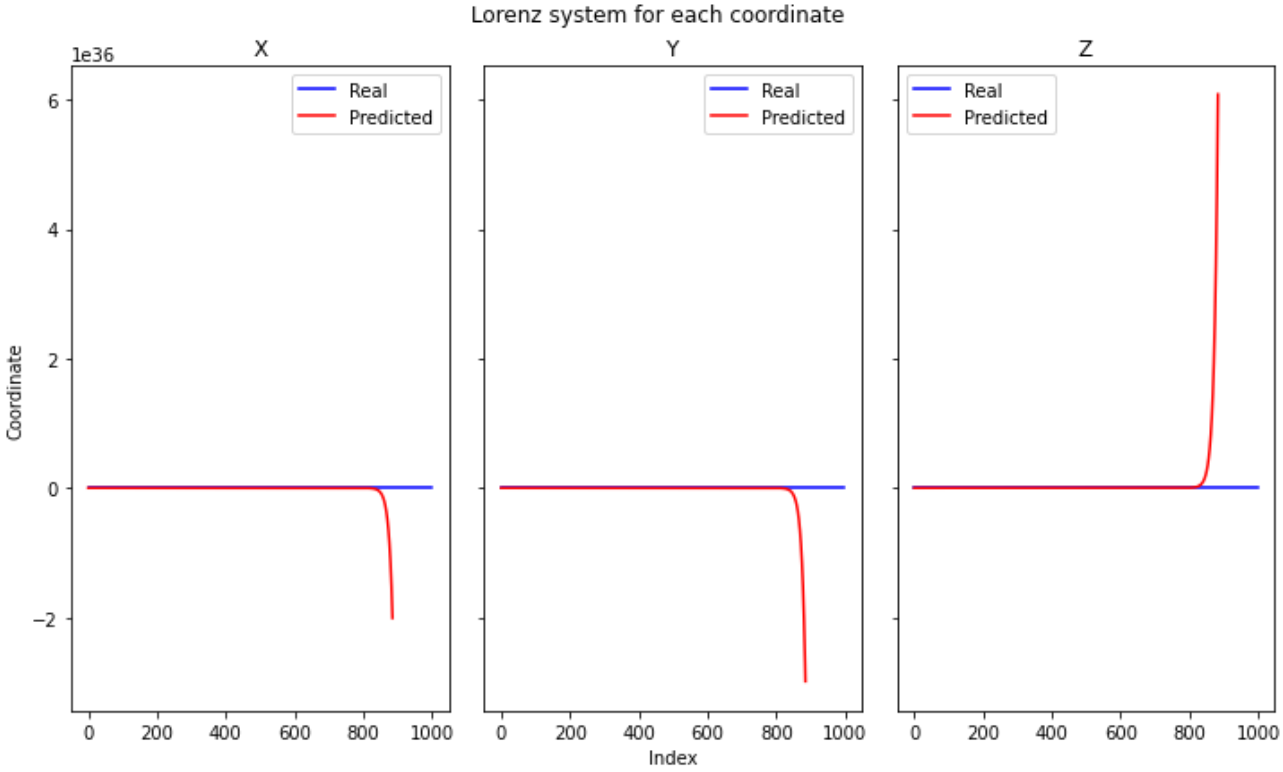
```
#Here we compare the prediction of our model with the real points

fig = plt.figure(figsize=(10,8))
ax = plt.axes(projection='3d')
ax.scatter3D(zpred[:,0], zpred[:,1], zpred[:,2], color = 'red', label='Predicted pc
ax.plot3D(data[:,0], data[:,1], data[:,2], color = 'blue', label='Real points')
ax.set_xlabel('$X$')
ax.set_ylabel('$Y$')
ax.set_zlabel('$Z$')
ax.set_title('Lorenz System', fontsize=20)
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc1fb6865d0>
```




```
#Here we can plot the predicted and the real value for each direction of the space
fig, axs = plt.subplots(1,3, figsize=(10, 6), sharex=True, sharey=True)

axs[0].plot([i for i in range(0,1000)],data[:, 0], 'blue', label = 'Real')
axs[0].plot([i for i in range(0,1000)], zpred[:, 0], 'red', label = 'Predicted')
axs[0].set_title('X')
axs[0].legend()

axs[1].plot([i for i in range(0,1000)],data[:, 1], 'blue', label = 'Real')
axs[1].plot([i for i in range(0,1000)], zpred[:, 1], 'red', label = 'Predicted')
axs[1].set_title('Y')
axs[1].legend()

axs[2].plot([i for i in range(0,1000)],data[:, 2], 'blue', label = 'Real')
axs[2].plot([i for i in range(0,1000)], zpred[:, 2], 'red', label = 'Predicted')
axs[2].set_title('Z')
axs[2].legend()

fig.tight_layout()
fig.suptitle('Lorenz system for each coordinate')
plt.subplots_adjust(top=0.90)

axs[1].set_xlabel('Index')
axs[0].set_ylabel('Coordinate')
```

```
Text(64.25, 0.5, 'Coordinate')
```



✓ 0s conclusão: 08:59