Report for exercise 5 from group G

Tasks addressed:	5
Authors:	Angelos Kafounis Caner Karaoglu Joaquin Gomez Sanchez
Last compiled:	2022–08–31
Source code:	`https://github.com/joaquimgomez/ml-crowds-group-g`

The work on tasks was divided in the following way:

Angelos Kafounis	Task 1	30%
	Task 2	30%
	Task 3	40%
	Task 4	30%
	Task 5	35%
Caner Karaoglu	Task 1	30%
	Task 2	30%
	Task 3	30%
	Task 4	40%
	Task 5	30%
Joaquin Gomez Sanchez	Task 1	40%
	Task 2	40%
	Task 3	30%
	Task 4	30%
	Task 5	35%

Report on task 1, Approximating functions

For this first task we have to show our understanding about the approximation of functions, this using data from a linear function and from a non linear function.

First, let us approximate the function in the dataset with a linear function. Since we have a linear function, we have to find the parameters a (the slope) and b (the intercept) given the description of a linear function:

$$y = ax + b \tag{1}$$

In order to find the slope and the intercept we have used the well-known Least Squares Minimization (LSM), already implemented in `numpy.linalg.lstsq`. In our file `utils.py` is the method `leastSquaresMinimization`, which, given the data, unpacks the components of the data, i.e. x and y, obtains the coefficient matrix A and calls the LSM, returning the slope and the intercept.

The final result can be seen in Figure 1. We have that the obtained function perfectly approximates the data because the dataset follows a perfect line, which is easy to approximate.

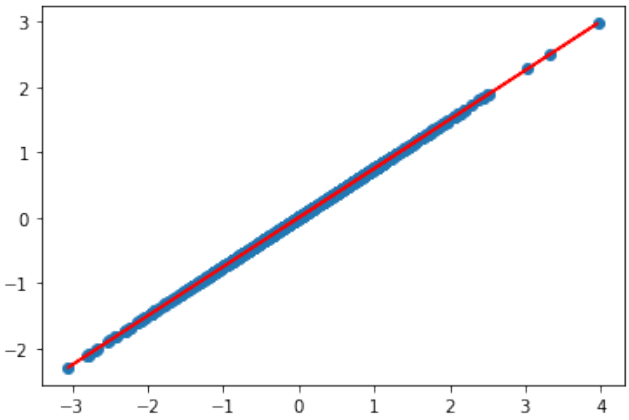


Figure 1: Original linear data (in blue) and approximated function (in red)

One could think about approximating the linear data using, for example, radial basis functions, but this is not a good idea since they are means to approximate multivariate functions by linear combinations of terms based on a single univariate function, and we only have a single univariate function.

On the contrary, if we try to use the LSM method to approximate non linear data as the given one, we have that this is not possible, as it can be appreciated in Figure 2. This impossibility is due to the limitations of the LSM because it only can find the two aforementioned parameters, i.e., it only can approximate linear functions.

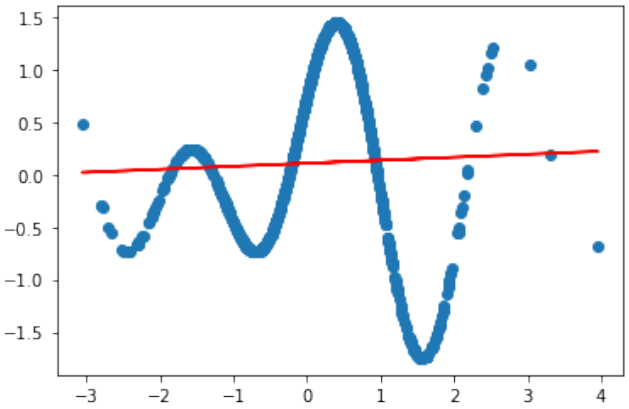


Figure 2: Original non linear data (in blue) and the approximated function (in red)

In order to approximate the non linear data we have to use some more complex methods, like the suggested one, i.e., combination of radial functions:

$$f(x) = \sum_{l=1}^{L} c_l \phi_l(x) \tag{2}$$

where $c_l \in \mathbb{R}^d$ and

$$\phi_l(x) = \exp(-\|x_l - x\|^2/\varepsilon^2) \tag{3}$$

i.e, the so-called Radial Basis Functions (RBF). The bandwidth term ϵ can be squared or not, for this task we have decided to keep the square.

In order to approximate the data, we have implemented the method `approximateNonLinearFunction`. This method, given the data, the number of radial basis functions functions to accumulate L and the bandwidth ϵ, it computes L function centers (x_l in Equation 3), computes L radial basis functions $\phi_l(x)$ (Equation 3) and then finish computing the accumulation of functions (Equation 2) after computing the coefficients c_l using `np.linalg.lstsq`.

To decide the centers, we tried firstly to chose random centers, which worked in terms of x because the function was correctly reproduced, but in terms of y the functions appeared moved in the axis above or below the real data. Then, we decided to compute the centers as follows: compute the "range" of values in terms of X, i.e., the domain of the data; dive the domain in L equal parts and then every center corresponds to one part of the data. In Python: `np.min(x) + i * ((np.max(x) - np.min(x)) / L)` where $i \in [0, L-1]$.

As we have to find the best L and ε, we have implemented and used the simple but useful Grid Search strategy with Mean Square Error (MSE). We have searched in a search space of $L \in [5, 15)$ with 1 unit steps and $\varepsilon \in [0.5, 1.5)$ with 0.1 unit steps. The combination with less MSE (concretely, 0.00011) is $L = 10$ and $\varepsilon = 1.0$.

In Figure 3 it can be seen the approximation (in red) for the best encountered parameters. It can be seen that we managed to approximate the function quite well.

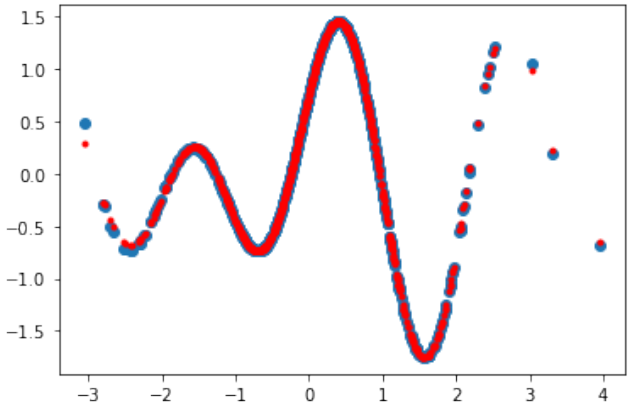


Figure 3: Original non linear data (in blue) and the approximated function (in red) using a combination of radial basis functions

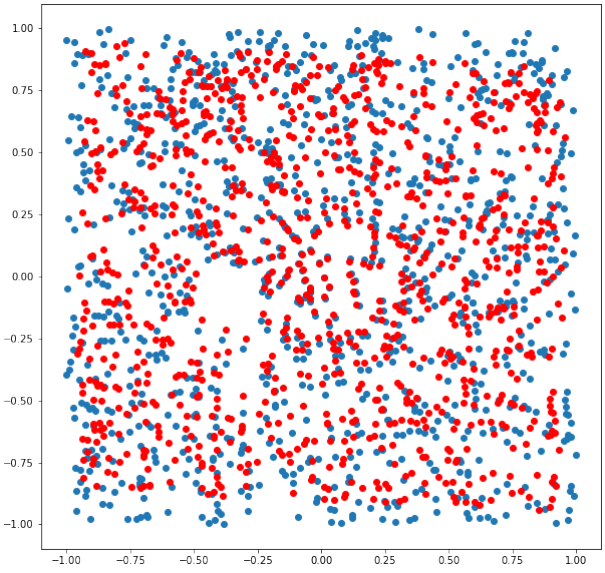
As mentioned before, all the implementations used for this task can be found in `utils.py` and `Task1.ipynb` files in the GitHub repository.

Report on task 2, Approximating linear vector fields

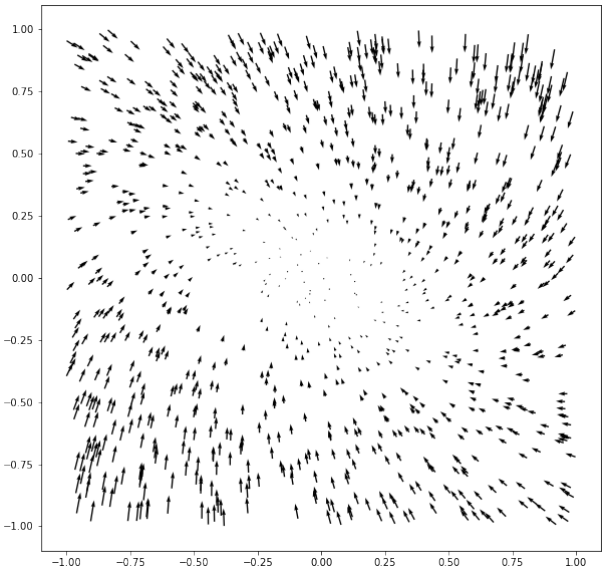
For the second task we are going to work with a linear vector field, in a similar way as in the previous task with the linear function.

Given the initial points x_0 and their evolution in the time, x_1, first, we are going to estimate the vectors $v^{(k)}$ at all points $x_0^{(k)}$ using Equation 4. With Figure 4a we can have an intuition of the evolution of the system with the x_0 points in blue and the x_1 points in red. In Figure 4b we have the computed vectors $v^{(k)}$.

$$\hat{v}^{(k)} = \frac{x_1^{(k)} - x_0^{(k)}}{\Delta t} \tag{4}$$



(a) Points x_0 (blue) and x_1 (red)



(b) $v^{(k)}$ computed using Equation 4

Figure 4: Evolution of the system

Then, we have approximated the matrix $A \in \mathbb{R}^{2\times 2}$ that describes the evolution of the system, i.e.,

$$\nu\left(x_0^{(k)}\right) = v^{(k)} = Ax_0^{(k)} \tag{5}$$

To approximate the matrix we have used the method `leastSquaresMinimization` implemented for the previous task. In this case, we computed the matrix parts for x and for y in a separated way, using the $v^{(k)}$ vector (see Listing 1).

```
ax, bx = leastSquaresMinimization(np.column_stack((x0Data[:,0], v[:,0])))

ay, by = leastSquaresMinimization(np.column_stack((x0Data[:,1], v[:,1])))
```

Listing 1: Implementation of our computation of A

Using the computed A matrix, we have solved the linear system $\dot{x}_1 = \hat{A}x_0^{(k)}$ for $T_{end} = 0.1$, using code similar to the previous task (i.e., the NumPy's method `polyval`), and again in a separate way for x and y. We have found that the MSE between the original x_1 and the approximated one is 0.0021. It is not perfect but we can consider this MSE absolutely good.

Finally, we simulated for the approximated system from the point $(10, 10)$ for 100 time steps. The final trajectory overlapped with the obtained phase portrait for the system can be seen in Figure 5.

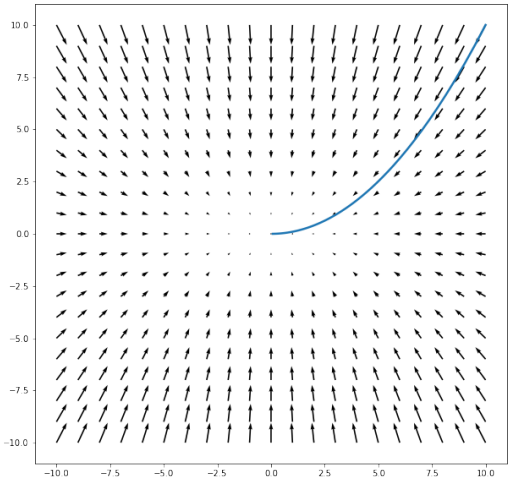


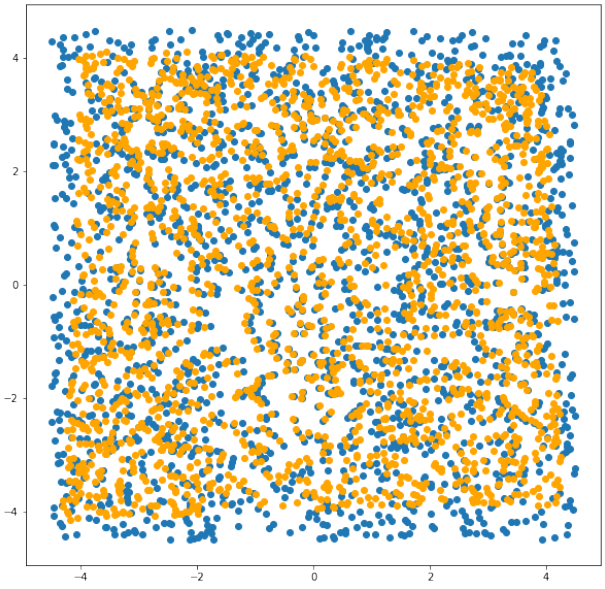
Figure 5: Phase portrait of the approximated system and trajectory for a simulation starting at $(10, 10)$ and for 100 time steps

Observing the phase portrait, and comparing it with the $v^{(k)}$ from Figure 4a, we can see that our approximation of the behavior is quite correct. The computed trajectory starting from $(10, 10)$ follows perfectly the phase portrait and it also seems to be the expected one (from the corner to the center with a small curve) if we follow the first $v^{(k)}$ vectors.

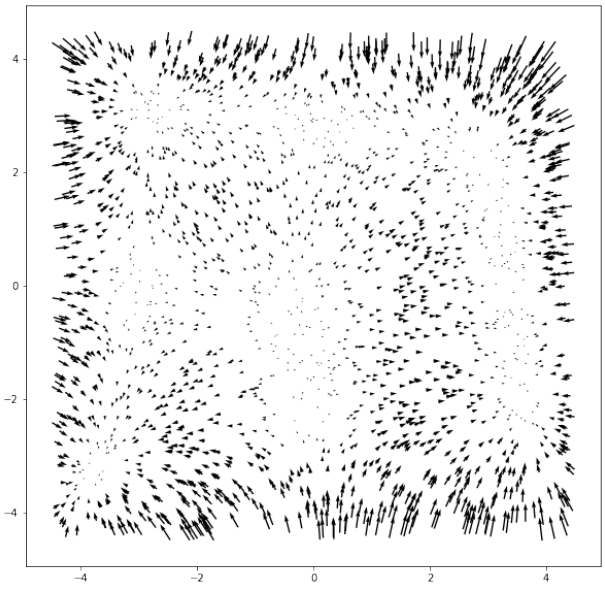
All the code used for the presented results is in the Jupyter Notebook `Task2.ipynb`. For this task we have not implemented extra functions from the ones already available in `utils.py` from the previous task.

Report on task 3, Approximating nonlinear vector fields

Now, let us try to approximate another vector field as in the previous task, but in this case a nonlinear vector field. For the field we also have two instances as it can be seen in Figure 6a (x_0 in blue and x_1 orange).



(a) Points x_0 (blue) and x_1 (orange)



(b) $v^{(k)}$ computed using Equation 4

Figure 6: Evolution of the system

First, we have to try to approximate the nonlinear vector field ψ with a linear operator $A \in \mathbb{R}^{2\times 2}$ such that

$$\frac{d}{ds}\psi(s, x(t))\Big|_{s=0} \approx \hat{f}_{linear}(x(t)) = Ax(t), \tag{6}$$

i.e., we have to do the same as in the previous Task 2.

After obtaining the two "components" of A, i.e., the approximation of the coordinates x and y, we have perform mean squared error setting $\Delta t = 0.1$. We obtain that the MSE is 0.0239, which is a really good value.

Next, we have tried to approximate the vector field using Radial Basis Functions (following Equation 7), which should be good for a nonlinear approximation. Following the same previous idea of approximating the

two components in a separated way, we have performed a Grid Search using MSE in order to find the best L and ϵ. For L we have search in the range $[100, 1000]$ with steps of 100 and for ϵ we have tried the values 0.05, 0.1, 0.15, 0.2, 0.5, 0.8 and 1.0.

$$\frac{d}{ds}\psi(s, x(t))\Big|_{s=0} \approx \hat{f}_{rbf}(x(t)) = C\phi(x(t)) \tag{7}$$

We have found that the hyper-parameters that better explain the evolution of the system in terms of MSE are $L = 100$ and $\epsilon = 0.005$, with $MSE = 6.442$.

As we can see, using the linear approximation we obtained a clearly smaller MSE, $0.0239 \lll 6.442$. From this, we should suppose that the vector field is linear, but if we take a look at Figure 6b we see that the field has four centers in the corners of the domain ($[-4.5, 4.5]^2$), which should not be possible to approximate using linear approximation. Then, we think that we need more data in order to properly obtain a nonlinear approximation. Also, we intuit that the good performance of the linear approximation may be due to the small difference between x_0 and x_1 in terms of space position. In Figure 8 we have plotted the original x_1 (in blue) and the approximated ones (in orange) using the obtained RBF.

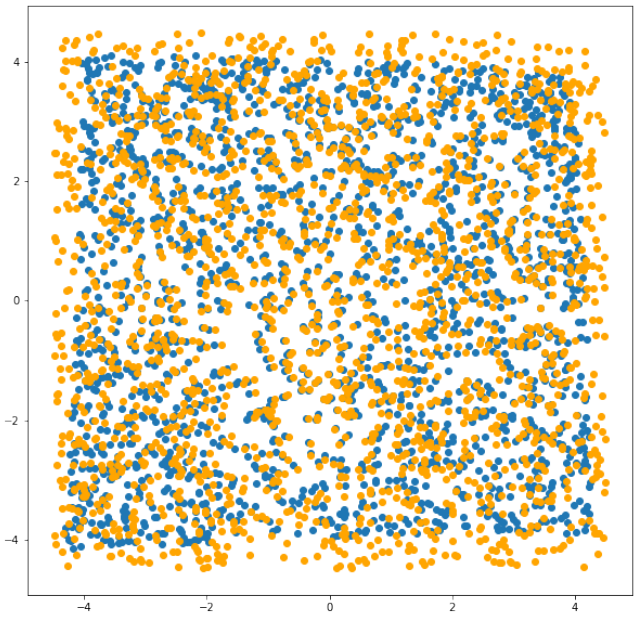


Figure 7: Original x_1 in blue and approximated x_1^{approx} in orange

Finally, we have simulated the system for time between $T_{init} = 0.1$ and $T_{end} = 10$ with 0.1 as step size. We can see in the resulting Figure 8 that we have 4 steady states in the corners of the domain, as we previously stated using the vectors from Figure 6b. We can also conclude that the system is not topologically equivalent to a linear system due to is behaviour.

All the code used for the presented results is in the Jupyter Notebook `Task3.ipynb`. For this task we have not implemented extra functions from the ones already available in `utils.py` from the previous tasks.

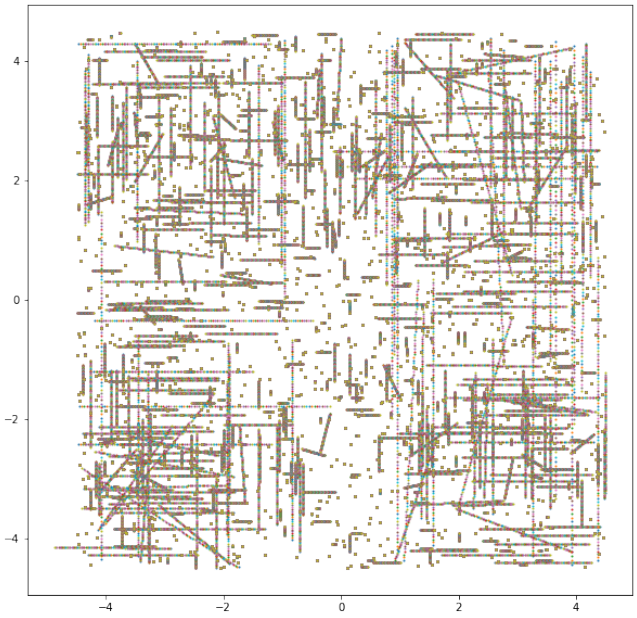


Figure 8: Trajectories for a large simulation time

Report on task 4, Time-delay embedding

For the first part of this 4th task, let us work with a periodic signal and the embedding of it into a state space where each point carries enough information to advance in time. This dataset consists of x and y pairs of 1000 different points.

Firstly, we have plotted the first coordinate against the time in Figure 9. We can see that the data clearly has a periodicity. We cannot embed the complete manifold in a one-dimensional space because of this periodicity. As a side note, time here is derived from the count of all points. In the figure we can easily see the small periodic motion when the first coordinate equals to -0.5.

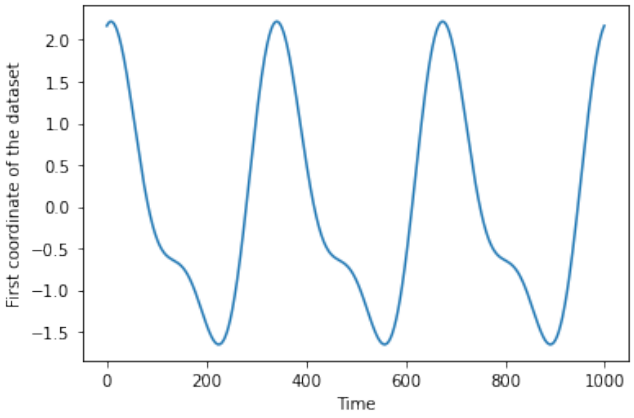
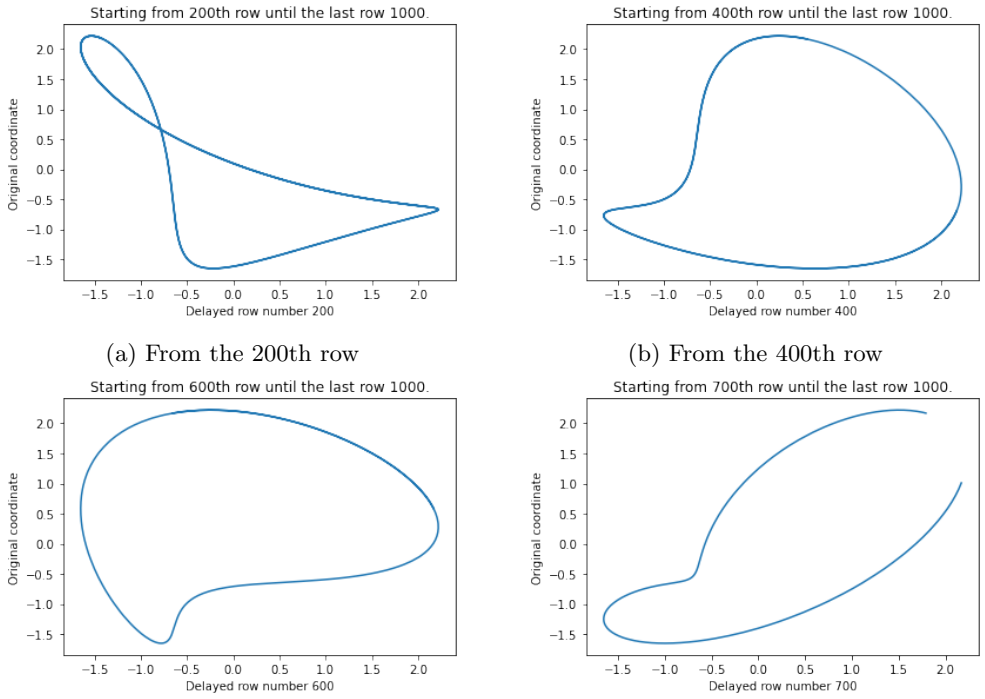


Figure 9: First coordinate of the dataset against time

Next we introduce a small time-delay Δn, which means, we skip Δn amount of rows and print their first coordinates w.r.t original first coordinates. This means, for example, if $\Delta n = 100$, from the 100th row we take the whole data up to the end, and plot this w.r.t from the 1st row until the `(len(dataset) -` Δn`)`. Plots can be seen in the Figure 10.



(a) From the 200th row

(b) From the 400th row

(c) From the 600th row

(d) From the 700th row

Figure 10: Plots under different tine-delays

Around $\Delta n = 600$ we can see that the shape of the embedding is visible around the point $(-0.5, -1.0)$. According to the Takens theorem we need $2d+1$ coordinates, i.e., in this case we need $2 \cdot 1 + 1 = 3$ coordinates.

For the second part of this task, we are trying to approximate the chaotic dynamics from a single time series. From Exercise 3 Task 4, we have copied the Lorenz Attractor code and simulated the system for the initial point $(10, 10, 10)$ with parameters $\sigma = 10, \rho = 28, \beta = 8/3$ (see Figure 11).

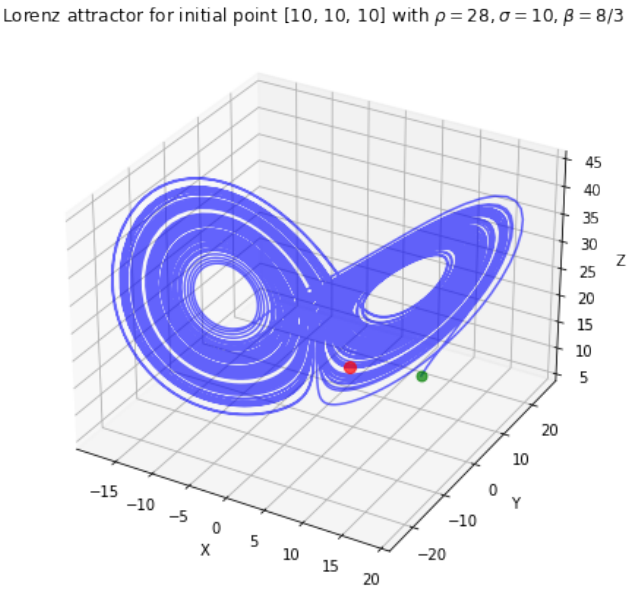
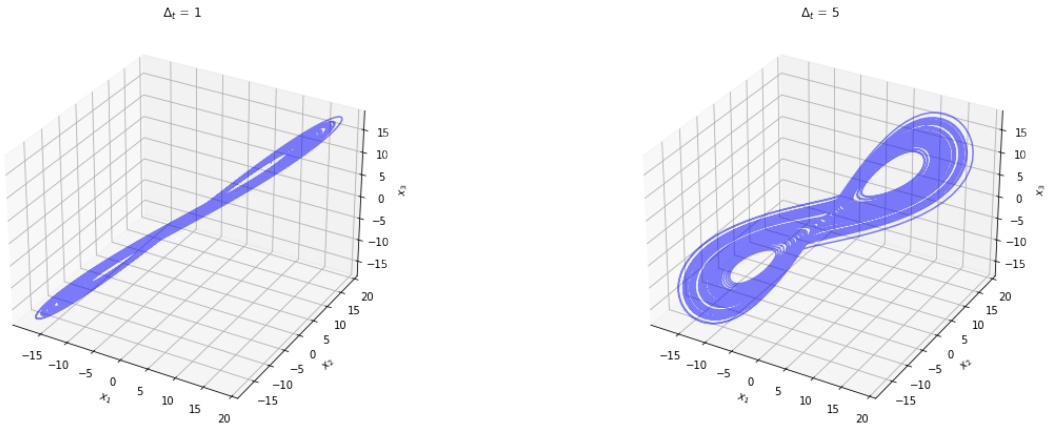


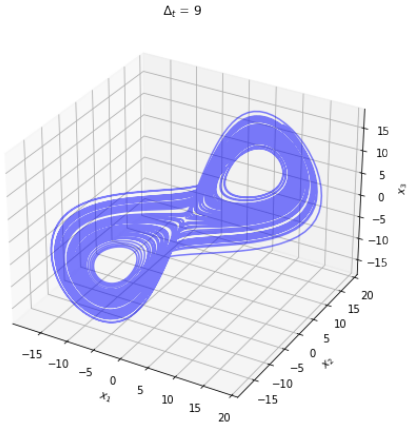
Figure 11: Lorenz Attractor

As the exercise sheet states, we are going to assume that we only have access to the x-coordinate and we do not know anything about the y and z-coordinates. This x coordinate will be our x_1. We are going to visualize $x_1 = x(t)$ against $x_2 = x(t + \Delta t)$ and $x_3 = x(t + 2\Delta t)$ for different Δt values (between 1 and 15). We have excluded $\Delta t = 0$, because for this case we have that $x_1 = x_2 = x_3$. See Figure 12 to see the some visualizations. Not all plots for values are shown here. All plots for all Δt values can be seen in the Jupyter notebook.

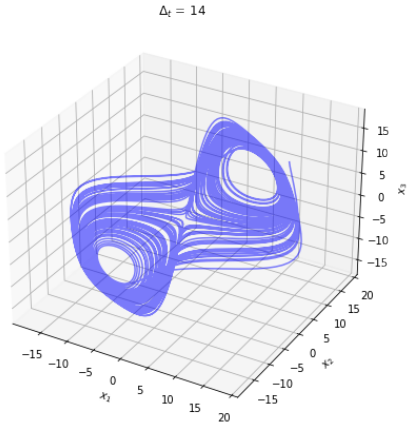


(a) For $\Delta t = 1$ (b) For $\Delta t = 5$

Figure 12: Visualization of Lorenz Attractor for different Δt values



(c) For $\Delta t = 9$

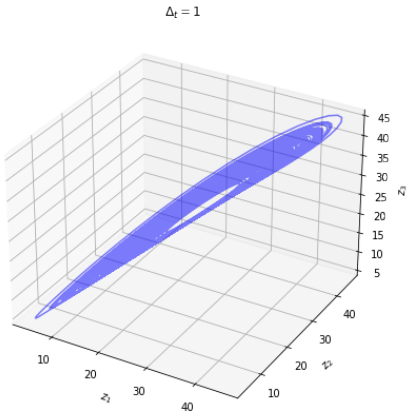


(d) For $\Delta t = 14$

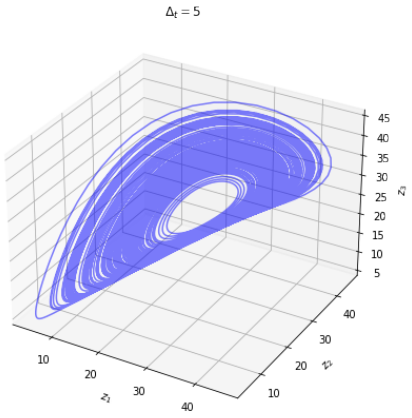
Figure 12: Visualization of Lorenz Attractor for different Δt values (cont.)

Here, by using just one dimension x_1, we can easily approximate the whole behavior of the attractor pretty closely. This specially happens for Δt around 9, because for small values the attractor appears squashed and for large values it appears stretched. As a side note, we do no have a distance function which measures the distances between states in this case. That is why the radius of the curves do not fit exactly.

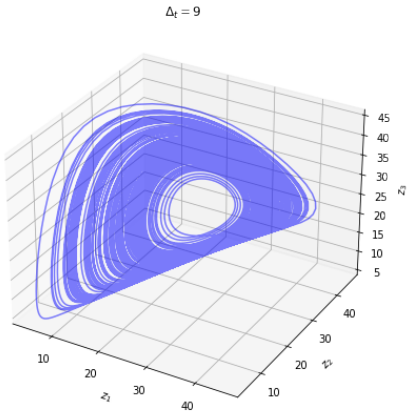
Following, we have done the same but now only using the z coordinate (see Figure 13).



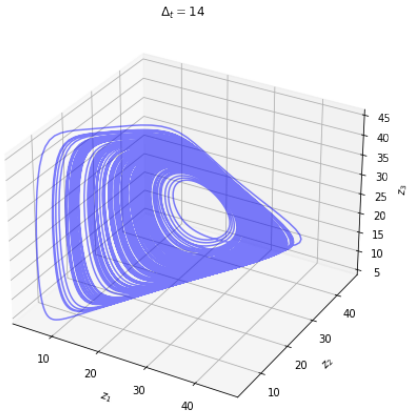
(a) For $\Delta t = 1$



(b) For $\Delta t = 5$



(c) For $\Delta t = 9$



(d) For $\Delta t = 14$

Figure 13: Visualization of Lorenz Attractor with different Δt values using z coordinates

It is not possible to capture the behaviour of the attractor only using the z coordinate. We think that this happens because most of the information happens in the $X - Y$ plane, e.g., the two centers of the attractor are mainly constructed over the $X - Y$ plane, so it makes sense that the z coordinate does not capture this information.

All the code used for the presented results is in the Jupyter Notebook `Task4.ipynb`. For this task we have not implemented extra functions.

Report on task 5, Learning crowd dynamics

For the 5th task of this Exercise 5 we have to learn a dynamical system that predicts the utilization of the MI building in Garching.

Firstly, from the 15000 time steps we have deleted the burn-in period of 1000 time steps at the beginning of the given file. Then, we have created the delay embedding with 350 delays for the first three measurement areas, i.e., the columns 2, 3 and 4 in the file.

After creating the embedding, we have performed PCA over the resulting data matrix. According to the Takens theorem we need $2d + 1$ dimensions in order to properly represent our dataset. As we have a one-dimensional dataset we need $2 \cdot 1 + 1 = 3$ dimensions. This is the value we have used to perform the PCA.

We have checked if this value corresponds with $\geq 90\%$ of the energy in the PCA components. In Figure 14 we can see that the number of dimensions holds the requirement.

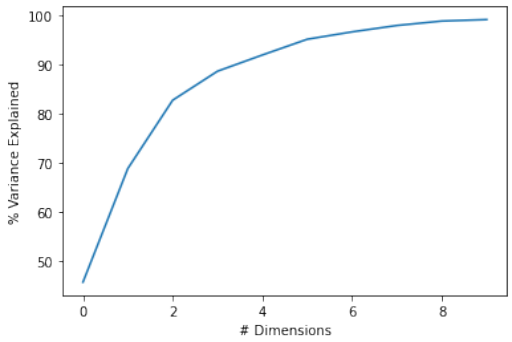


Figure 14: Energy per PCA component (plot cut up to the 8th component)

After the PCA we have colored the points by all measurements taken at the first time point of the delay, for all nine measurement areas (see Figure 15).

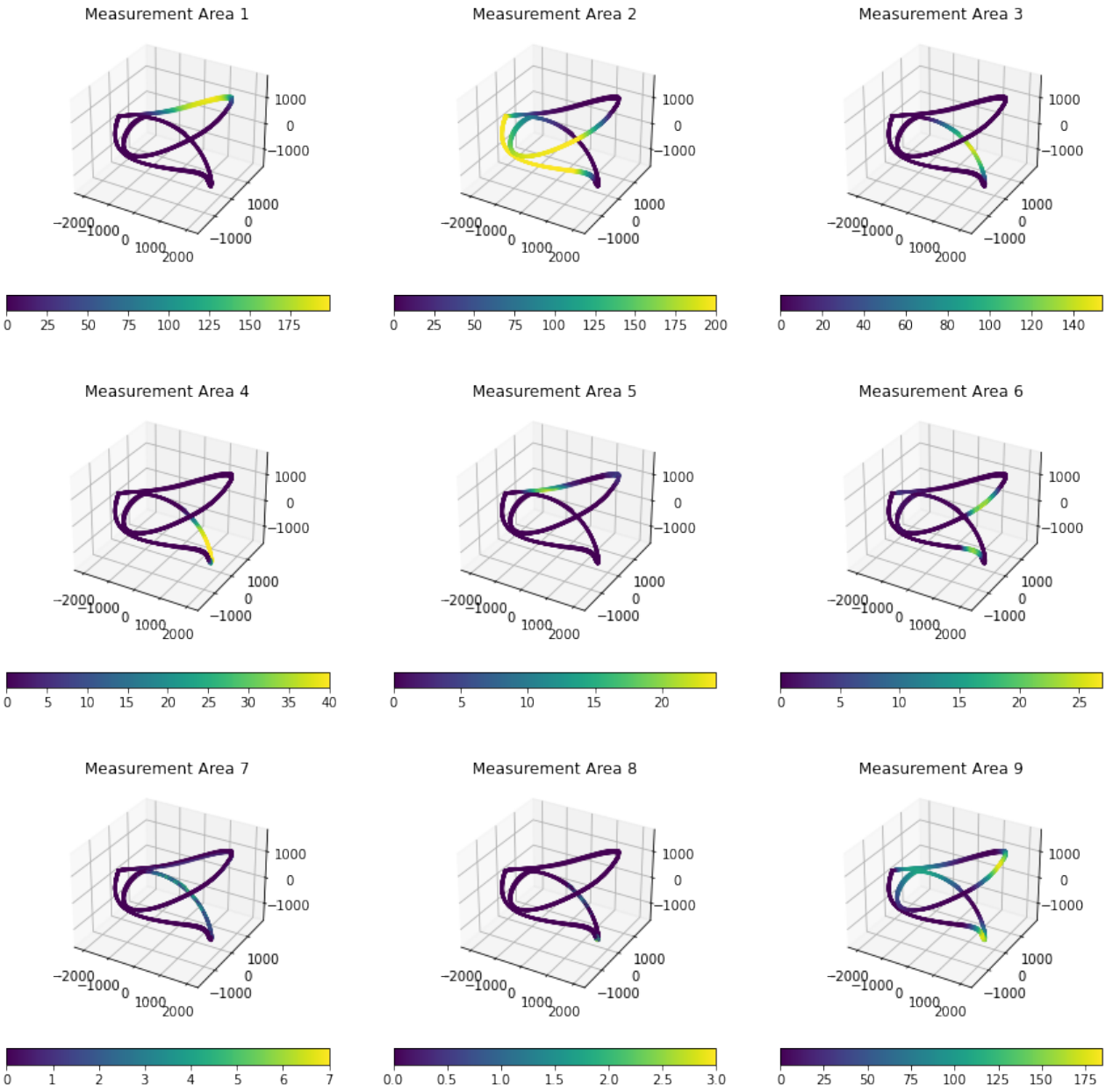


Figure 15: Coloring of the points by the first coordinate of the delay embedding

All the code used for the presented results is in the Jupyter Notebook `Task5.ipynb`. For this task we have not implemented extra functions.