Report for exercise 5 from group H

Tasks addressed:	5
Authors:	Taiba Basit (03734212)
	Zeenat Farheen (03734213)
	Fabian Nhan (03687620)
Last compiled:	2021–06–23
Source code:	https://github.com/Combo1/MLCMS/tree/main/exercise5

The work on tasks was divided in the following way:

Taiba Basit (03734212)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
Zeenat Farheen (03734213)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%
Fabian Nhan (03687620)	Task 1	33%
	Task 2	33%
	Task 3	33%
	Task 4	33%
	Task 5	33%

Report on task 1/5, Approximating functions

Task 1.1

For this first subtask, we downloaded the data **linear_function_data.txt** from moodle. Then we used np.loadtxt to load the data. After loading the data we plotted the data using matplotlib.pyplot. Figure 1 shows the plot of the data.

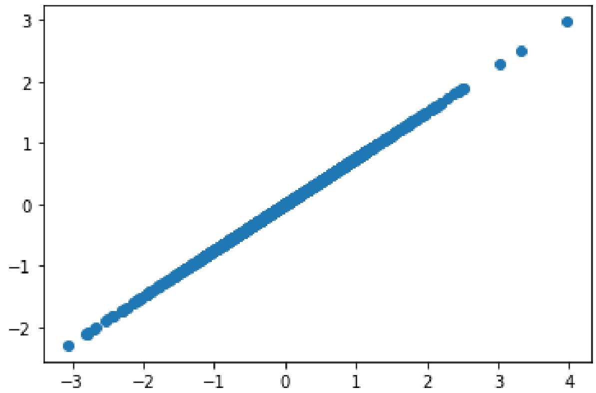


Figure 1: Plot of data in linear_function_data.txt

Our task was to approximate a linear function between two Euclidean spaces R^n, R^d with $n, d \in N$ is a map $f_{linear} : R^n \to R^d$, such that for $x \in R^n$,

$$f_{linear}(x) = Ax \in R^d$$

for some matrix $A \in R^{dxn}$. To approximate the function f_{linear} i.e. to find A, we used the least square minimization method. We created a function least_square_minimization which takes in the input parameters for x and f_{linear} and finds the value of A. For our purpose we used the linalg.lstsq method from numpy which returns the value of A. We have set the value of parameter rcond to None as setting it to a large value as suggested in the exercise sheet did not result in any significant changes in the value of A obtained. Figure 2 shows the original function and also the the approximated function. As we can observe from the figure, the approximated function gives a very good fit on the original data.

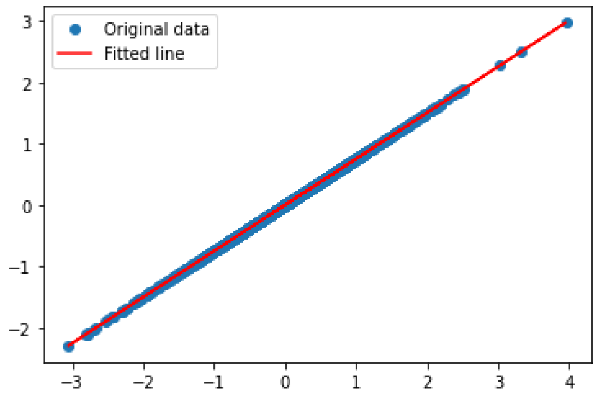


Figure 2: Plot of linear dataset (A) and the approximated function

Task 1.2

For the second subtask, we downloaded the data **nonlinear_function_data.txt** from moodle. Then we used np.loadtxt to load the data. After loading the data we plotted the data using matplotlib.pyplot. Figure 3 shows the plot of the data.

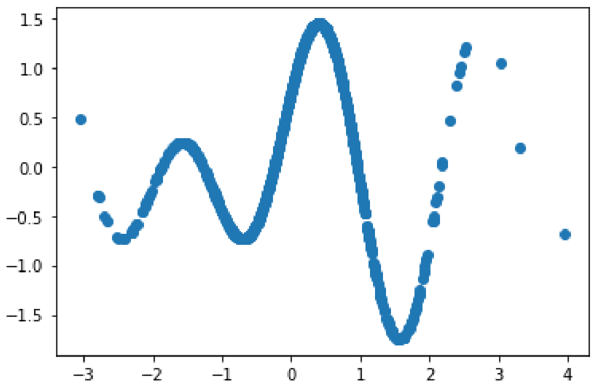


Figure 3: Plot of data in nonlinear_function_data.txt

For this task we had to approximate the non-linear function with a linear function. We followed the same steps in task 1.1. We called the function least_square_minimization and found the value of A. We then used this value of A to find the approximated linear function using $f_{linear} = Ax$. Figure 4 shows the original function in blue and also the the approximated function in red. As it can be clearly seen from the figure 4, the linear function is unable to approximate the underlying non-linear data and we need much higher dimensions for the approximated function.

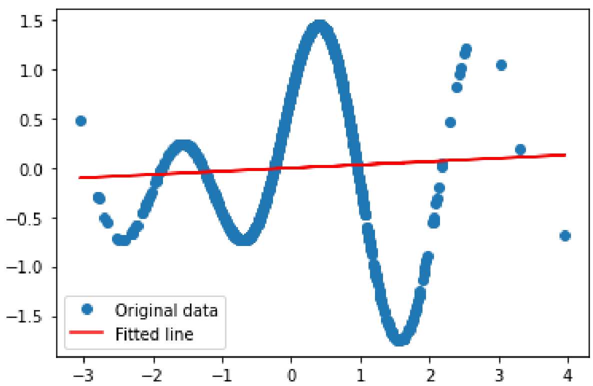


Figure 4: Plot of non-linear dataset (B) and the approximated linear function

Task 1.3

The third subtask was to approximate the **nonlinear_function_data.txt** using a combination of radial functions.

We can approximate the non linear function by writing it as the combination of radial basis function ϕ_l, as:

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

The radial basis function that we used for our task is defined by:

$$\phi_l(x) = exp(-\|x_l - x\|^2 / \epsilon^2)$$

Here x_l is the center of the function and is selected randomly. The parameter ϵ is the bandwidth.

In our code we defined a function radial_basis_function which takes the input data, number of radial basis functions to calculate i.e. the value of L and the value of epsilon. It returns the calculated ϕ_l and also the value of the centers i.e x_l. After this, we approximate the function by solving for C using our previously defined function of least_square_minimization with paramater rcond set to None.

We first started by choosing a value for L by varying it between 3 and 15 with a constant of epsilon at 1. The results for different values of L are shown in figure below.

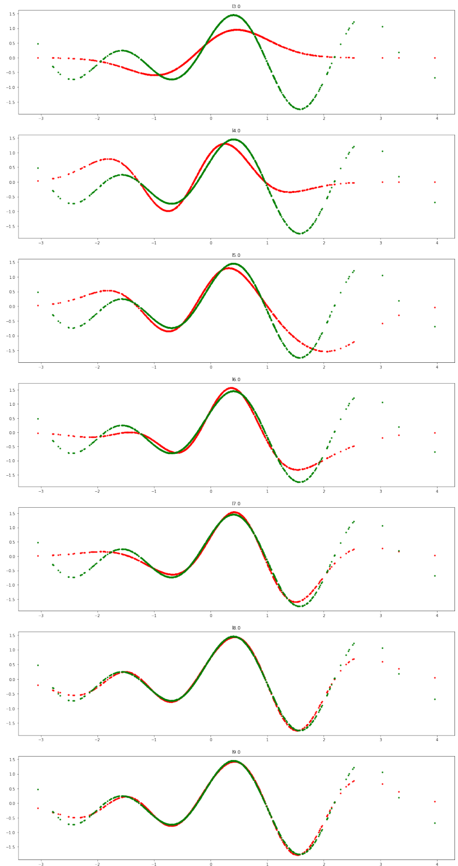


Figure 5: Plot of approximated function and original data with different values of L $\in$ [3,15] and $\epsilon = 1$

We can see from that the curve starts overfitting for values of $L > 8$ and the curve underfits for smaller values of L. Thus we choose L = 8 to get a good fit.

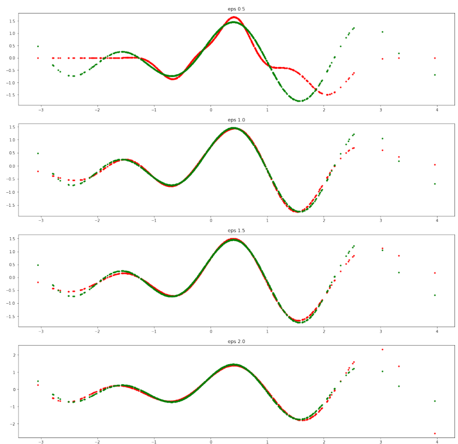


Figure 6: Plot of the original function and the approximated function for different values of ϵ with L = 8

Next we choose a value for ϵ by varying it between 0.5 and 2 with a step size of 0.5. We keep the value of L constant at 8. The results obtained are shown in figure 6. We chose the value of ϵ to be 1 as the underlying non-linear data is fairly smooth and this value gave us a good fit without overfitting the data.

The resulting plot with parameter value of L = 8 and $\epsilon = 1$ is shown in figure 7.

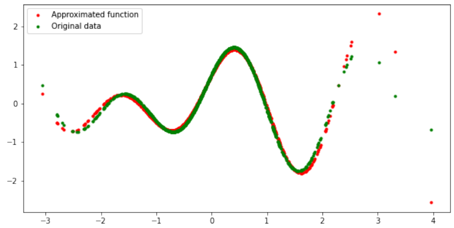


Figure 7: Plot of the original function and the approximated function

We also plotted the constituting radial basis functions of the non-linear data as shown in figure 8 Figure 8 shows the plot of radial basis functions.

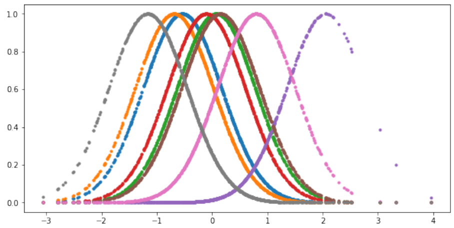


Figure 8: Plot of calculated radial basis functions for L=8 and $\epsilon = 1$

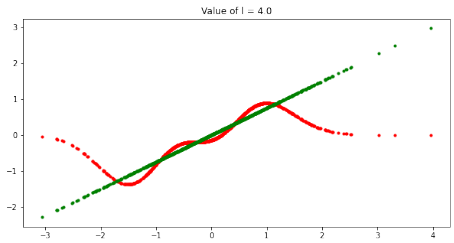


Figure 9: Approximating linear function with radial basis functions for L=3 and $\epsilon = 1$

We also approximated the linear function for dataset (A) using a combination of radial basis functions. However, we found that we obtain very poor approximations with lower values of L and ϵ as shown in figure 9. The curve fits the middle part if the data fairly well at L = 8 and eps = 1, as shown in figure below, but we can see that the approximated functions follows a non-linear pattern at the end points of the data which will give large errors while predicting the data. Thus it is not reasonable to approximate linear functions using radial basis functions

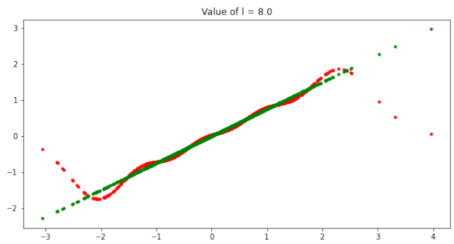


Figure 10: Approximating linear function with radial basis functions for L=8 and $\epsilon = 1$

Report on task 2/5, Approximating linear vector fields

Task 2.1

We downloaded the datasets linear_vectorfield_data_x0.txt and linear_vectorfield_data_x1.txt from moodle. We read the data using np.loadtxt and plotted the data using matplotlib.pyplot. Figure 11 shows the plot of data.

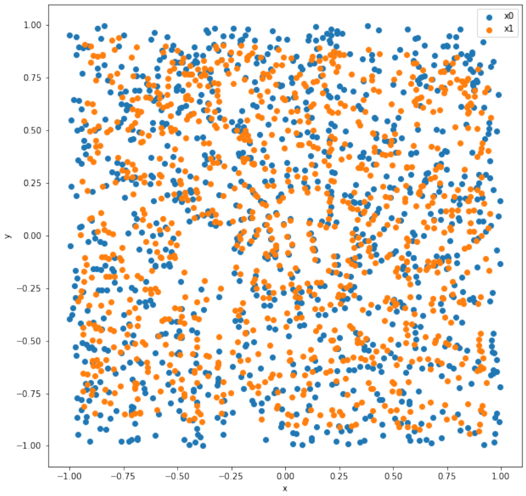


Figure 11: Plot of data in linear_vectorfield_data_x0.txt and linear_vectorfield_data_x1.txt

For this part we had to estimate the linear vector field which was used to generate point x_1 form point x_0. A vector field is a section of the tangent bundle: $v : M \to TM$, such that$v(x) \in T_x M$.
Firstly, we calculated the vector field v_k, by using the below formula:

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

In our code we defined a function **estimate_vector_field** which takes x_0, x_1 and Δt as parameters and returns the calculated vector field. We chose the value of Δt as0.5 because this gave us minimum error in the next part of the task.

Next, using the previously estimated values of vector field, we solved for $A \in R^{2x2}$, by using the least squares minimization method. The vector field is linear so for all k:

$$v^{(k)} = Ax_0^{(k)}$$

Task 2.2

Now, after calculating the matrix A, we can solve the linear system $\dot{x} = \hat{A}x$. To solve this we used the function *solve_ivp* from scipy.integrate. First, we defined a function $vectorField()$ which is a function for calculating the next state of the system from the current state and the calculated value of A. The function generateVectorField() is used to solve the linear system for a specified time t_end. We pass the parameters x_0 and t_{end} to the function. The function then returns the estimated value of the next state of system i.e. $x_1^{(k)}$.

We then defined a function to calculate the MSE loss. The function $mse()$ takes the original x_1 and the predicted x_1 and then returns the mse loss using the following formula:

$$\frac{1}{N} \sum_{k=1}^{N} \left\| \hat{x}_1^{(k)} - x_1^{(k)} \right\|^2$$

.

To get an estimate of our system, we first solved the system till time t_end = 0.1. The results obtained are shown in figure 12. As we can see the blue dots are lagging behind the orange dots. This means that the time delay between x0and x1 is greater than 0.1. The mean square error between the predicted and real values of x1 was 0.003.

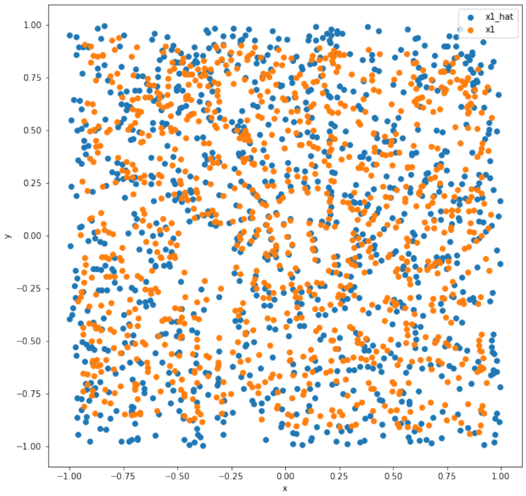


Figure 12: Plot of x_0 and x_1 for $T_{end} = 0.1$

To predict values closer to the true data of x1, we solved the system for t_end = 1. The results obtained are shown in figure 13. As we can see, there is a complete overlap between the predicted and the given values. This suggests that the true time delay between x0 and x1 is approximately 1. The mean square error obtained was $9.959 * 10^{-6}$

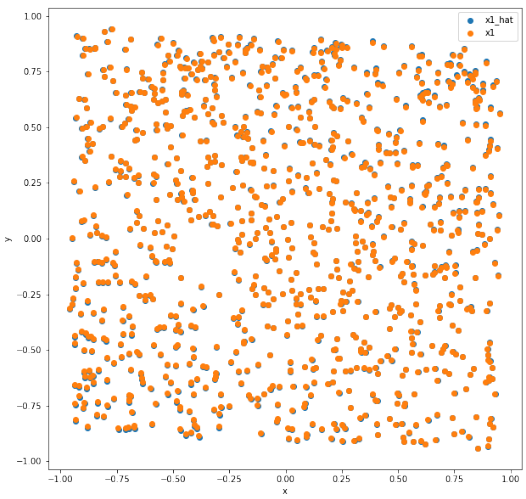


Figure 13: Plot of x_0 and x_1 for $T_{end} = 1$

To find the stable point of the linear vector field, we solved the system for t_end = 100. The results obtained are shown in figure 14. As we can see, all points converge to the origin and the origin is the stable point of the system.

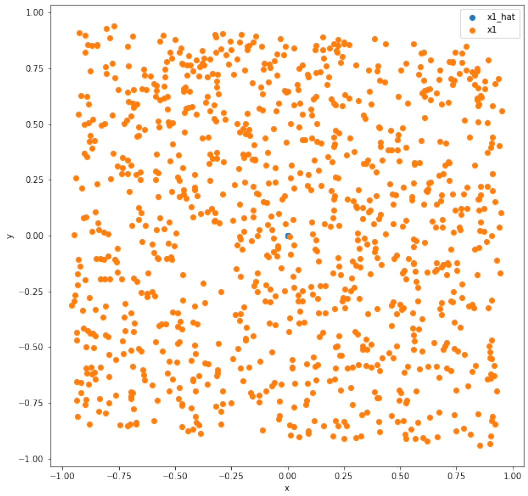


Figure 14: Plot of x_0 and x_1 for $T_{end} = 100$

Task 2.3

For this subpart of task we took the intital point (10,10) and solved the linear system for $t_end = 100$. To generate the trajectory, we used the *solve_ivp* function of scipy.integrate and passed inital value as (10,10) and the t_end as 100. Then to generate the phase portraits, we first created the meshgrid in the domain $[-10, 10]^2$. We then passed this to our $vectorField$ function which calculated the next state of the system based on previously calculated A. We then used plt.streamplot to plot the phase portraits.

As it can be seen from the figure 15, the trajectory converges to the origin as expected from the result in previous sub part. The phase portraits represents a linear vector field.

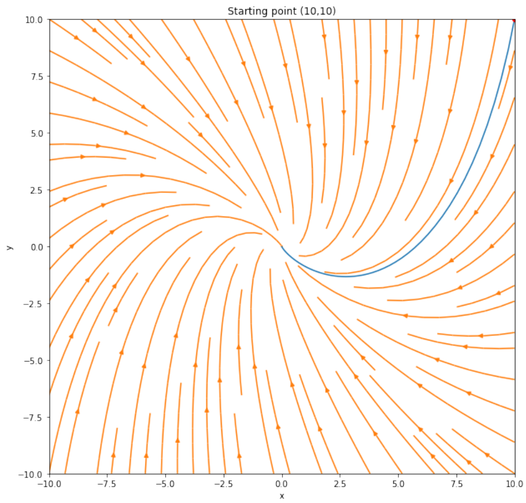


Figure 15: Trajectory for intital point (10,10) and Phase portrait for $T_{end} = 100$

Report on task 3/5, Approximating nonlinear vector fields

In task 3, we begin by plotting the data points at initial time $t = 0$ labeled as x0 and at a later point after time Δt labeled as x1. The plot of x0 and x1 can be seen in fig 16

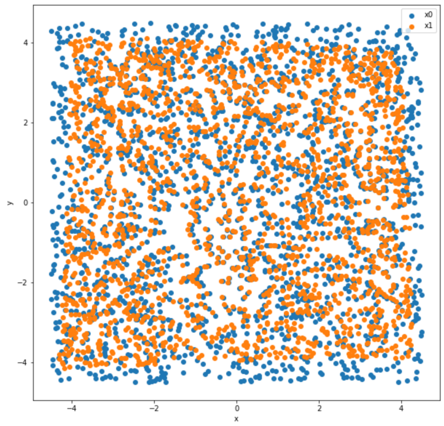


Figure 16: Plot of data at $time = 0$, x0 and after time Δt, x1

Task 3.1

In the first part, we estimated the vector field describing ψ with a linear operator $A \in R^{2x2}$, such that $\psi(t, x) \approx f_{linear}(x) = Ax$.

We first calculated the estimated value of vector field describing ψ by using finite differences formula. To choose the value for Δt, we tested different combinations of Δt and t_end to find out which pair gives the minimum mean square error of approximated value of x1. Table 1 shows the different values of Δt and t_end that we tried and the mean squared error obtained for each combination. From the results obtained, we chose the value of Δt as 1.

Δt	T_{end}	MSE
0.1	1	0.0373
0.1	1.5	0.0400
1	1	0.0373
1	1.5	0.0400
1.5	1	0.0373
1.5	1.5	0.0400

Table 1: Table to show MSE values for different combination of Δt and T_{end}

Similar to task 2, we calculate A using least square minimization method. The approximated of x1 obtained is shown in fig 17. As we can see the predictions are not very accurate for a linear approximation.

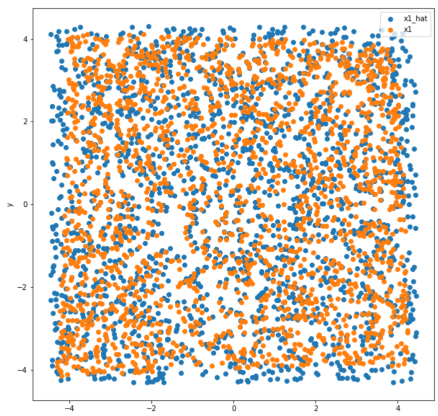


Figure 17: Plot of data at $time = 0$, x0 and after time Δt, x1

Task 3.2

For the next part, we approximated the vector field using radial basis functions. To find out the optimal number of basis functions needed to get a good approximation of the vector field, we tested different values of l between 100 and 1000 with a step size of 25. The error decreases exponentially and we get a very good error at l=400 of approximately 10^{-7}. At l=1000 we get an even lower error rate, but to avoid overfitting we choose l = 400. We kept the value of epsilon to be 1 as the this gave us lower values for errors.

The predicted values of x1 from the approximated vector-field is shown in fig 18. As we can notice, the blue dots ,representing the predicted x1, are completely overlapped by the provided data for x1.

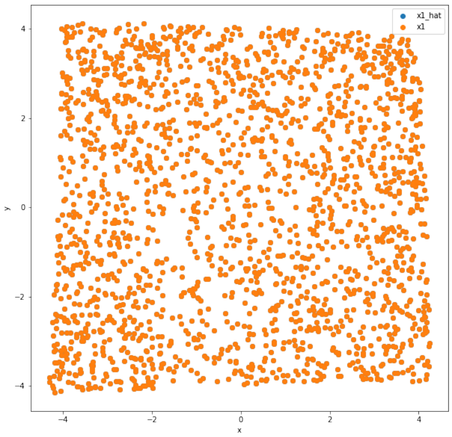
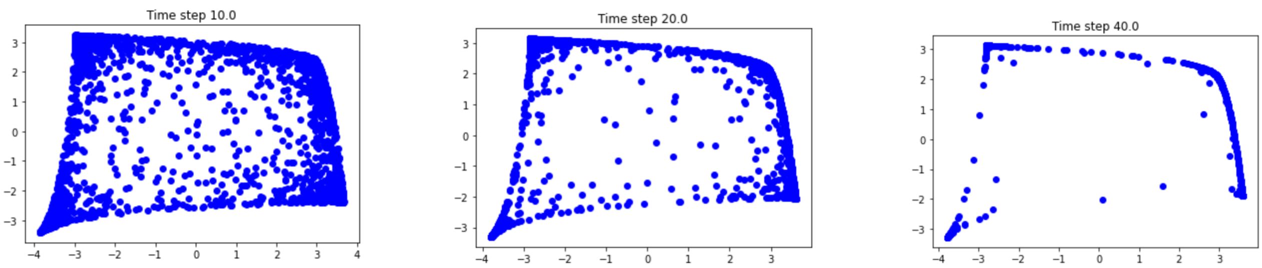


Figure 18: State of system after t = Δt. Orange dots represents the given points x1 . Blue dots represents the predicted points x1 using non-linear approximation of vector field

The errors obtained from non-linear approximation are significantly less than that obtained in linear approximation, suggesting that the vector-field might be non-linear.

Task 3.3

In the third part, we wrote our function to solve the system for a larger time of 300 timesteps. We also plotted the state of the system after every 10 timesteps. Figure 19 shows the state of the system at t=10, 20 and 40. We can observe from the resulting plots, that the system converges to 4 steady states. This is shown in Fig 20 at timestep = 230. The system cannot be topologically equivalent to a linear system because of the existence of multiple steady states.



(a) State of system at time $t = 10$ (b) State of system at time $t = 20$ (c) State of system at time $t = 40$

Figure 19: State of the system at different timesteps

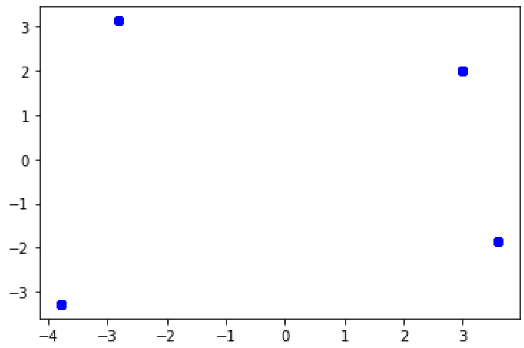


Figure 20: State of the system at time $t = 230$

Report on task 4/5, Time-delay embedding

For this task we began by downloading the takens1_.txt file from moodle. We loaded the data into an numpy array with the np.loadtxt function and stacked an additional column with np.hstack and the use of np.array and reshape.

We then started with exploring the dataset by looking into the values of the dataset and plotting the periodic signal with pyplot.

```
array([[ 2.16837096e+00, -5.46312593e-01,  0.00000000e+00],
       [ 2.17981061e+00, -5.32475177e-01,  1.00000000e+00],
       [ 2.19002807e+00, -5.18940339e-01,  2.00000000e+00],
       ...,
       [ 2.14086777e+00, -5.76113402e-01,  9.97000000e+02],
       [ 2.15555144e+00, -5.60690562e-01,  9.98000000e+02],
       [ 2.16853679e+00, -5.46119652e-01,  9.99000000e+02]])
```

Figure 21: Data Exploration

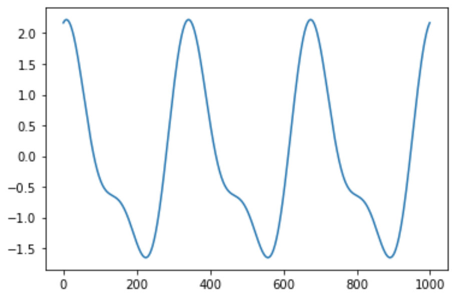


Figure 22: Periodic Signal

Afterwards we plotted the original manifold utilizing the given coordinates.

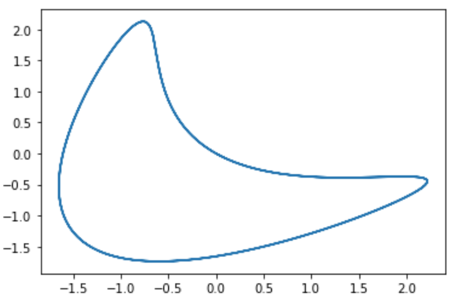


Figure 23: Original Manifold

For the delay Δn we chose Δn = 30, because it looks the most similar to the original manifold, but we also additionally added plots for delays in five row intervals.

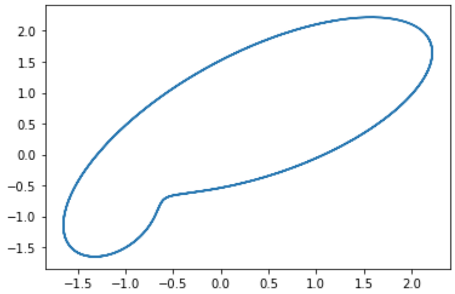


Figure 24: Embedded Manifold Δn = 30

In the lecture it was stated that for a standard circle/shifted cosine function, we needed two points to ensure at which point of the circle we are. Our function is the same, because it is monotonically increasing or decreasing.

According to Takens, for this 1-dimensional manifold, we need 1+2*d = 3-dimensional manifold, which represents the delay embedding with the topologically equivalent property. This means for computing a single point on the embedded manifold we need three points in our original manifold.

We need four coordinates (t - 3Δt, t - 2Δt, t - Δt, t), which equal to two points in the embedded manifold to know at which point of the periodic manifold we are, since the embedded manifold is period as well (topological equivalence).

To ensure the periodic manifold is embedded correctly. We first would need 333 points in our original manifold (= one iteration of our periodic signal) in our embedded manifold we need 333 + 2 to compute the embedding for our first point.

For part two we began by reusing our old code on the Lorenz attractor and plotting the butterfly wings for the given configuration.

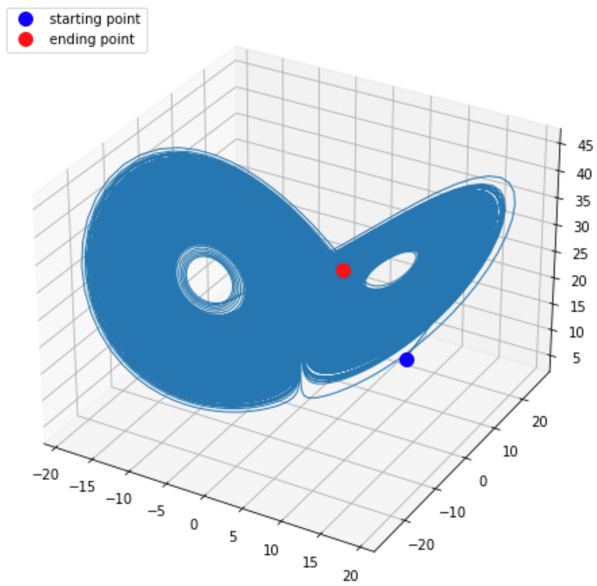


Figure 25: Lorenz attractor starting point (10, 10, 10) parameters: $\sigma = 10$, $\rho = 28$, $\beta = 8/3$

We then proceeded by visualizing the attractor by plotting x_1 = x(t) against x_2 = x(t + Δt) and x_3 = x(t + 2Δt).

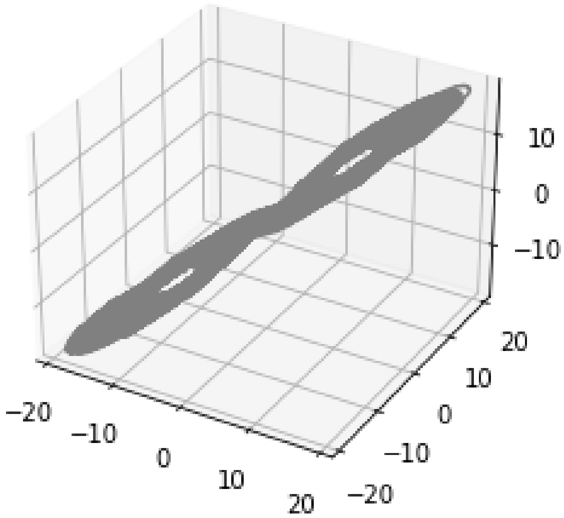


Figure 26: Embedded Lorenz attractor built with the x-component

The same does not work for the z coordinate, because of the missing symmetry in the z-component. The original system has a symmetry of f: (x, y, z) -¿ (-x, -y, z). Meaning that when reconstructing with x as well as y we conserve symmetry. We get only a single loop/wing, because both of the original loops/wings are folded together into one, because of the missing symmetry property under the assumption that reconstruction does not change symmetry [1].

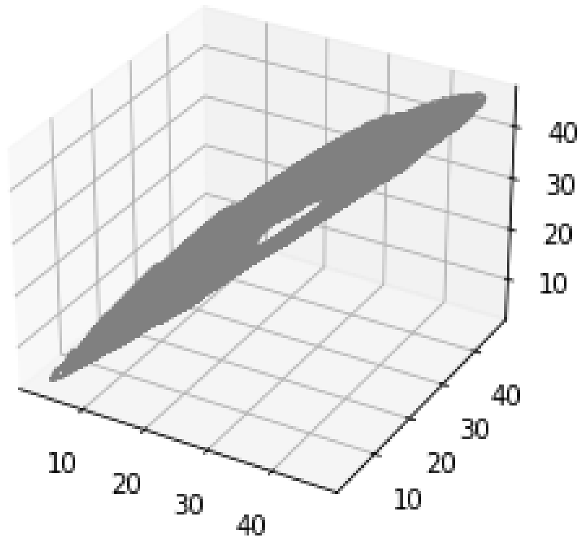


Figure 27: Embedded Lorenz attractor built with the z-component

Report on task 5/5, Learning crowd dynamics

For the first task, we downloaded the dataset MI_timesteps.txt from moodle. Then we loaded the data as numpy array using np.loadtxt function. The set the skiprows parameter of np.loadtxt to 1001 to ignore the burn-in period and the header row. According to Takens, for this 1-dimensional data, we need 1+2*d = 3-dimensions to embed it.

To create the delay embeddings with 350 delays, we created a numpy array of all zeros with the shape data.shape[0]-351 x 1053. Then we looped over all the data in original file, and created a delay embedding.

For this we only took 351 rows at a time and only 2,3, and 4th column and flattend them. This created the embeddings with shape 13650 X 1053.

Then we used the PCA function from sklearn.decomposition and passed the number of principal components to 3.

For the second subtask, we plotted for each of the nine measurement areas our pca components and then scatter plotted them with a color corresponding to the first coordinate of the delay embedding.

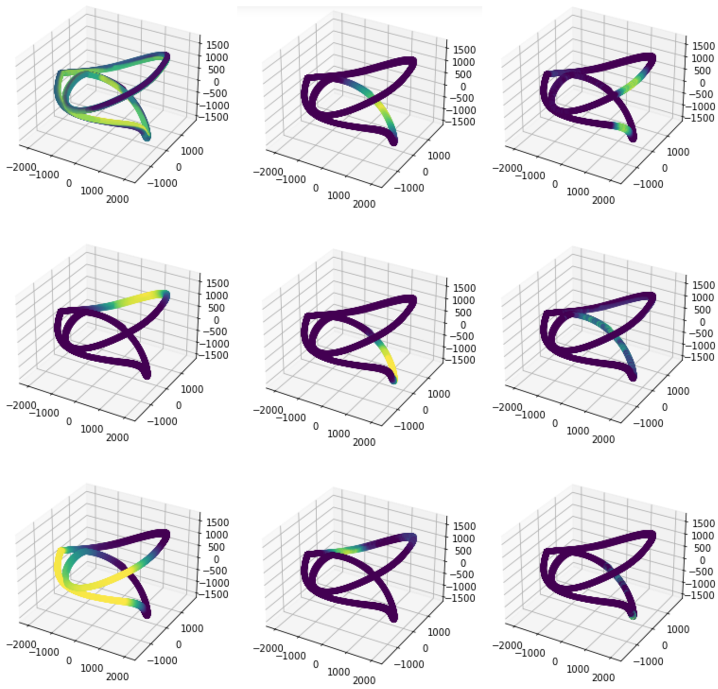


Figure 28: 9 Plots of the PCA space according to measurement area colored the first coordinate of the delay embedding

In the third subtask, we first defined two functions one for the arclength and the second one for the change of the arclength (velocity) (over time does not matter since each measurement is one timestep away from the next one).

The first plot is of the arclength between each pair of points as shown in fig 29 and the second plot represents the velocity calculated as the difference between arclength over a timestep of 1. As shown in fig 30

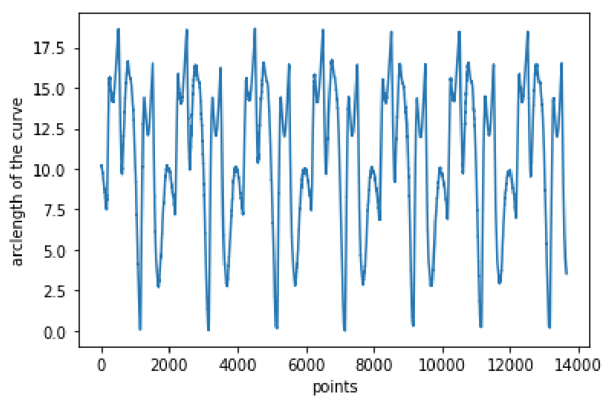


Figure 29: Arclength between pairs of points

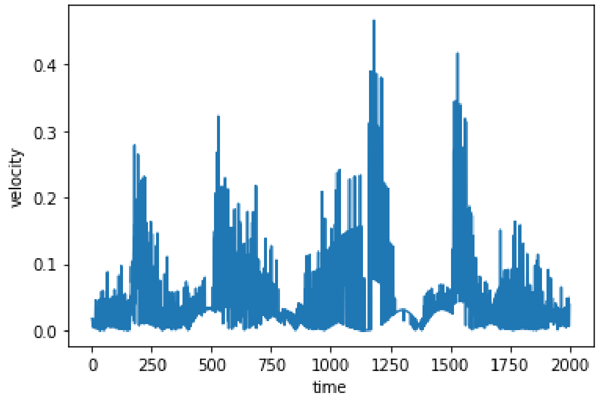


Figure 30: Velocity against arclength

Next we approximate the velocity as a function of arclength using radial basis functions. For this we chose l = 1000 and epsilon = 1. The resulting approximation of velocity function is shown in fig 31

Using this function, we can predict future values of arclengths.

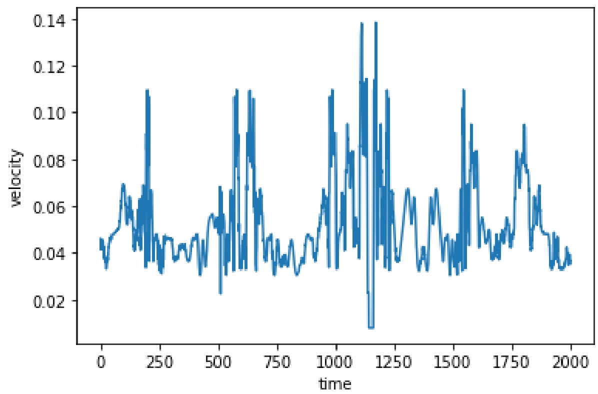


Figure 31: Predicted Velocity against arclength

As we know that the utilization (measurement value 1) is a function of the arclength.

$$f_{util}(s) = C * \phi(s)$$

where, s is the arclength at time t and C is the unknown constant and f_{util} is the measurement value at time t.

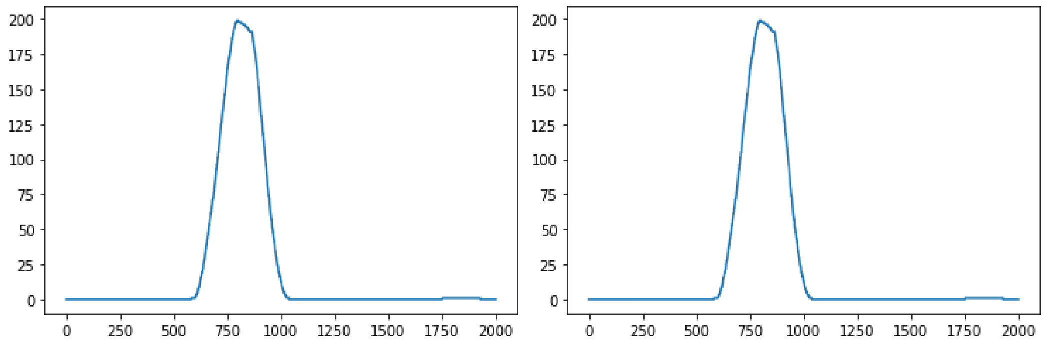
We can approximate this function using radial basis functions. We chose periodic kernel for radial basis functions as shown below,

$$k_{per}(x, x_l) = exp(-2sin^2(\pi|x - x_l|/p)/\epsilon^2)$$

where p is the periodicity of the function to be approximated.

To approximate the function, we chose l = 3000 and period = 2000 as the measurement repeats over a period of approximately 2000 timesteps, and $\epsilon = 2$. The resulting approximation is shown in fig 32b which is very similar to the actual values as shown in fig 32a

Once we have the new predicted values of arclength for future days, we can use this function to predict the measurement values for future days as well.



(a) Plot of actual measurement point 1

(b) Plot of predicted measurement point 1

Figure 32: Actual and predicted data measurement point 1

[1] Dietrich, Jan Philipp. "Phase space reconstruction using the frequency domain: a generalization of actual methods." (2008).