Report for exercise 5 from group C

Tasks addressed: 5
Authors: Aleksi Kääriäinen (03795252)
Danqing Chen (03766464)
Julia Xu (03716599)
Joseph Alterbaum (03724310)

Last compiled: 2024–09–27

The work on tasks was divided in the following way:

Aleksi Kääriäinen (03795252)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Danqing Chen (03766464)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Julia Xu (03716599)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%
Joseph Alterbaum (03724310)	Task 1	25%
	Task 2	25%
	Task 3	25%
	Task 4	25%
	Task 5	25%

Report on task 1, Approximating functions

1 Approximating functions

1.1 Approximating a linear dataset with a linear function

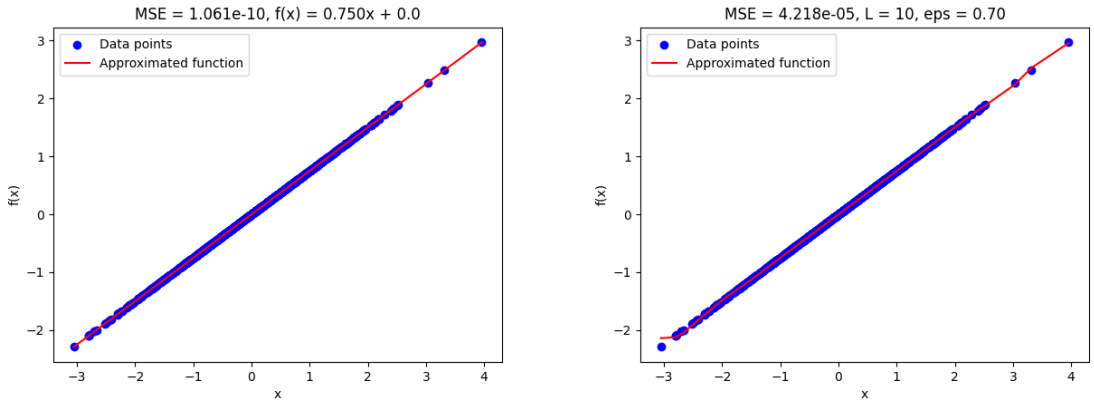
In the first part of this task, we had to approximate the function in dataset A with a linear function. This was done by solving the least squares minimization problem:

$$A^T = (X^T X)^{-1} X^T F$$

In our case, X is the first column of dataset A with padded with zeroes to set the intercept term of the linear function to 0, and F is the second column of the dataset. The least square minimization problem was solved with `scipy.linalg.lstsq`. Additionally, the dataset was also approximated with a nonlinear approximator, namely using a set of radial basis functions with the form

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

We used a simple machine learning model that minimizes the mean square error to learn the best ϵ and L values from a predetermined set of hyperparameters. The full script can be found in `./task1.ipynb`. Some pruning that is not shown in the script was done to decrease the size of the hyperparameter spaces.



(a) Linear approximation (b) RBF approximation

Figure 1: Plots of same dataset approximated with two methods

Figure 1 the plots of the two approximations. Note that the mean square error of the linear approximation is 5 magnitudes of ten smaller than the radial basis function approximation's mean square error. The linear approximation results in a smaller total loss, since linear models have a global optimum, and the weights that produce the global optimum are calculated by solving the least squares minimization problem. Nonlinear models do not always result in the global optimum solution, and thus do not necessarily produce similar results when trained on the same linear dataset. This shows why using radial basis functions for dataset A is not a good idea: the cost of computation is higher, and the results are not as good.

1.2 Approximating a nonlinear dataset with a linear function

In the second part of the task, we had to approximate a nonlinear dataset with a linear function. The method used is precisely the same as in part 1.

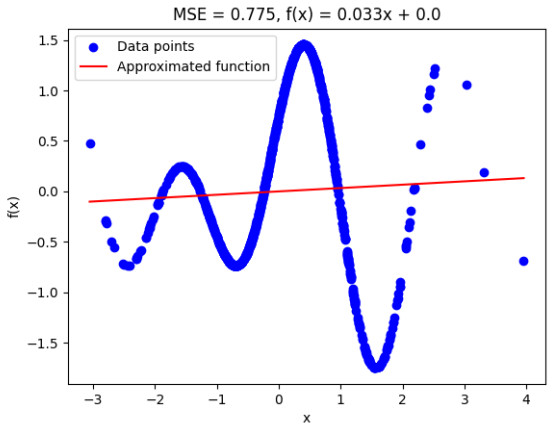


Figure 2: Linear approximation of a nonlinear dataset

Figure 2 shows the linear approximation of the nonlinear dataset. Expectedly, the results for the linear approximation are very poor, the prediction does not fit the dataset at all, and the loss is really high. This shows that to get good results, the used model has to be suitable for the data used.

1.3 Approximating a nonlinear dataset with a nonlinear function

In the third part, we had to approximate the same nonlinear dataset as in part 2, but this time with a nonlinear function. As in part 1, the nonlinear approximation is done using a set of radial basis functions. We also used the same machine learning model to find the most suitable hyperparameters that minimize the mean squared error of the model. Some pruning was conducted to find a suitable range for the parameters L and ϵ, and then the model was trained by iterating over the parameters. After pruning the spaces, we ended up using the range $[2, 10]$ for L and $[0.1, 2]$ for ϵ.

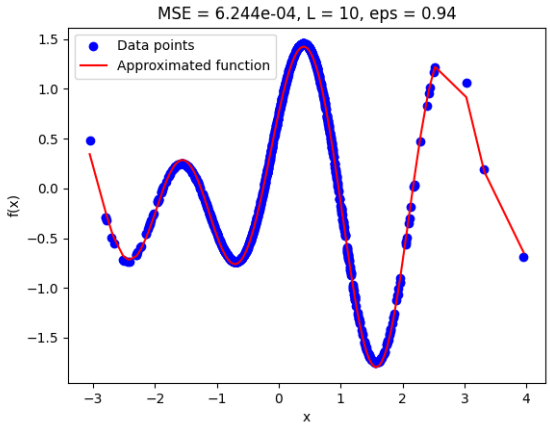


Figure 3: Nonlinear approximation of a nonlinear dataset

Figure 3 shows the results of the trained model. The parameters that produced the best results were $L = 10$ and $\epsilon \approx 0.94$. During pruning, we found out that using any more than 10 radial functions resulted in heavy overfitting of the data.

Report on task 2, Approximating linear vector fields

2 Approximating Linear Vector Fields

The second task of this exercise sheet is divided into three parts: (1) estimate the linear vector field that was used to generate the points x_1 from the points x_0, (2) solve the linear system and compute the mean squared error, and (3) choose the initial point (10, 10) and solve the system including a visualization of the trajectory as well as the phase portrait.

2.1 Estimation of Linear Vector Field

Firstly, the finite-difference formula (1) is used to estimate the vectors $v^{(k)}$ at all points $x_0^{(k)}$, with a time step $\Delta t = 0.1$:

$$\hat{v}^{(k)} = \frac{x_1^{(k)} - x_0^{(k)}}{\Delta t} \tag{1}$$

Since we are dealing with a linear system, an appropriate supervised machine learning technique is **linear regression**. Therefore, the matrix $A \in \mathbb{R}^{2\times 2}$ can be approximated via minimizing the least-squares error:

$$\hat{A}^T = (X^T X)^{-1} X^T V \tag{2}$$

With $x_0^{(k)}$ corresponding to X and $v^{(k)}$ to V, the approximated matrix can simply be computed by inserting these variables into `utils.least_squares()` which implements the formula (2). For further reference in this task, this approximated matrix being is referred to as $\hat{A}$.

2.2 Linear System Solver

After estimating the vector field, it is time to solve the entire linear system: $\dot{x} = \hat{A}x$ (assumption: $\hat{A} \approx A$) with all $x_0^{(k)}$ as initial points up to a time $T = \Delta t = 0.1$. This can be done by simply calling "`scipy.integrate.solve_ivp`" for each initial point with a linear model and the approximated matrix A.

$$\frac{1}{N}\sum_{k=1}^{N} ||\hat{x_1}^{(k)} - x_1^{(k)}||^2 \qquad \text{, with N = 1000.} \tag{3}$$

Then the mean squared error (3) is calculated resulting in an **MSE** of **4.978778645608117e-06** .

2.3 Visualization and Phase Portrait

Finally, a point (10,10) far outside the initial data is chosen to be initial state for the next linear solver. The same concept as in the sub chapter before is applied with the new initial point variable and $T_{end} = 100$.

Figure 4 shows the visualization of the behaviour of both state dimensions x_1&x_2 and the phase portrait with a blue trajectory of the linear solution with initial point (10,10). Unfortunately we were not able to plot the vector field for the grid [-10,10]x[-10,10] due to plotting problems. The formula for a vector v from the **vector field** at position y however is $\mathbf{v} = \mathbf{Ay}$. If this is repeated for a number of positions from the grid & the resulting vectors v are all plotted at their respective positions y the vector field would be visible. The vector field plot would show that the system converges into the fixed point.
The final steady state for this system is approximately $x_{final} = (0, 0)$.

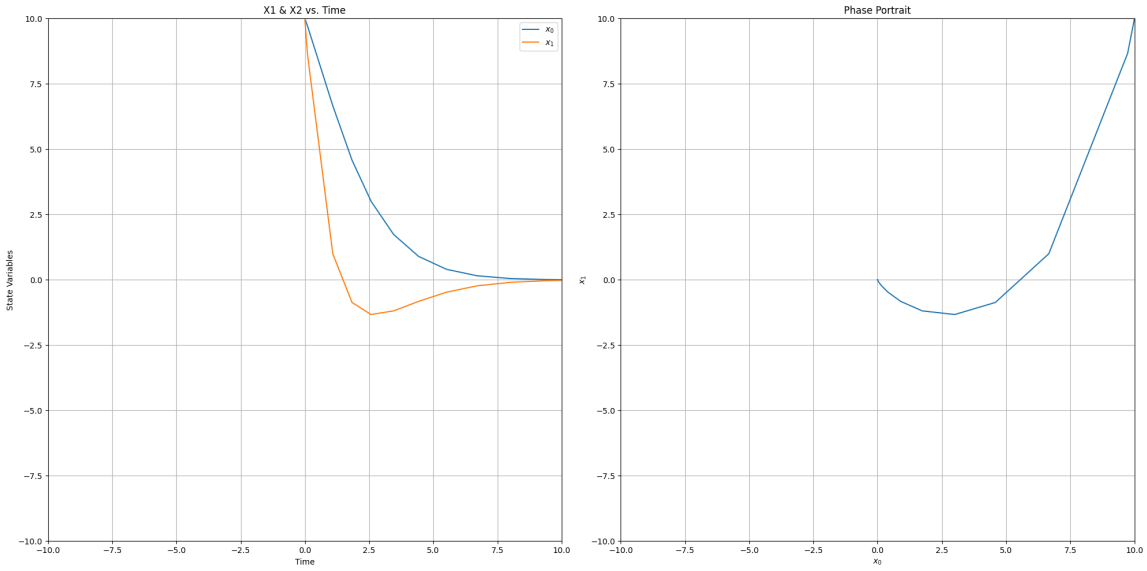


Figure 4: Trajectory and phase portrait of linear system with initial point (10,10)

Report on task 3, Approximating nonlinear vector fields

3 Approximation of non-linear vector fields

The solution to Task 3 is implemented in the Jupyter Notebook "*./src/part1/task3.ipynb*".

3.1 Part 1: Approximation with a linear operator

The first part of this task concerns the estimation of the vector field with a linear operator similar to Task 2. The implementation of this solution uses in particular the function "`least_squares`" & "`linear_model`" from the module "*./src/part1/utils.py*".
To approximate the vector field from the data, we first estimate a target vector field using the finite-difference formula with the two datasets x_0 & x_1, i.e., the states of the system at a time t and a small time step Δt later. The finite-difference formula then yields an approximation which serves as the target $v_k$in a supervised learning setting:

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

(cp. equation (9) on exercise sheet). With this target vector field, we can then proceed to solve the least-squares optimisation problem written in matrix from for the whole datasets X_0 & X_1 as:

$$\min_{\hat{A}} \| \frac{(X_1 - X_0)}{\Delta t} - X_0 \hat{A}^T \|^2$$

From this formulation we can obtain the approximated matrix A with the least-squares algorithm and end up with the result:

$$\hat{A} = \begin{pmatrix} -1.0016 & -0.02535 \\ 0.08673 & -4.32671 \end{pmatrix}$$

Subsequently, we can solve the linear dynamical system from equation (12) on the exercise sheet for a small time-span Δt by using the linear model from "*utils.py*", putting in $\hat{A}$ as the approximated matrix of the dynamical system & integrating it with the method "`solve_ivp`" from the package *SciPy*. With starting points from x_0 we then obtain approximations of the corresponding x_1-values. After we obtained the full approximation of x_1 we can then use the "`mean_squared_error`"-method from the package "*SciKit Learn*" to compute the Mean Squared Error between x_1 & $\hat{x}_1$. The result with a linear model for the vector field is $\mathbf{MSE_{linear} = 0.01863}$.

3.2 Part 2: Approximation with a non-linear operator – Radial Basis Functions

Similar to the first part of the exercise, we approximate the vector field but in this part with a non-linear model, namely Radial Basis Functions. For this we use, among others, the functions "`radial_basis_functions`" & "`nonlinear_model`" from "*./src/part1/utils.py*".
The implementation of the non-linear approximation with Radial Basis Functions can be fine-tuned by specific meta-parameters. These are specified at the start of our implementation and mainly involve the bandwidth ϵ & number of basis functions L as well as the kind of sampling for the basis function centres. For the last parameters, we tried sampling L basis function centres from the d-dimensional unit square and alternatively, which is also common in other implementations, sampling L points from the given dataset, in our case x_0. The sampling is implemented in the function "`sample_basis_centers`" in "*./src/part1/utils.py*" and for all our simulations we chose sampling from the dataset since it yielded much better results.
Again, the target vector field is computed via the finite-difference formula. Subsequently, the function is approximated with the non-linear radial basis functions defined in the lecture and the corresponding slides no. 5 & 6. First the values $\Phi(X_0)$ of all L radial basis functions for each data point in the dataset X_0 are computed and subsequently the least-squares optimisation problem for the new values $\Phi(X_0)$ is solved for the best linear combination represented by $\hat{C}$:

$$\min_{\hat{C}} \| \frac{(X_1 - X_0)}{\Delta t} - \Phi(X_0)\hat{C}^T \|^2$$

Again, in our implementation all computations are implemented directly for matrices.

While the performance of our approximation is highly dependent on the values of L & ϵ, we want to elaborate on the reasons for our final choice in the next subchapter. For now, we just mention that we picked $\mathbf{L = 100}$ & $\epsilon = \sqrt{\mathbf{diameter(X_0)}} = \sqrt{\mathbf{12.4141}}$ to obtain a reasonably good result after simulation. Like before, we solve the dynamical system and calculate the Mean Squared Error (MSE). In case of the non-linear approximation we take equation 13 from the exercise sheet and input our approximated matrix $\hat{C}$ as well as the parameters of the radial basis functions (L, ϵ & the basis function centers x_c) and again solve the system for a time-span Δt. From this we get an approximated state $x_1^{(k)}$ for each initial state $x_0^{(k)}$ and we compute the MSE from this approximation against the underlying true x_1. The best result we achieved was $\mathbf{MSE_{nonlinear} = 0.000407}$.

However, for different settings of L & ϵ the MSE-results differed quite significantly: The MSE was rather stable when varying only L over a fixed value of ϵ. With $\epsilon = diameter(X_0) = 12.4141$ we varied $L \in [100, 1000]$ and only observed changes in the 5th decimal place around $MSE_{nonlinear} = 0.00045$. For $\epsilon = \sqrt{diameter(X_0)}$ they also changed around the 5th decimal place but this time around $MSE_{nonlinear} = 0.00041$. In contrast, when varying ϵ with a fixed L the MSE changed quite a lot. When setting $L = 100$ & picking $\epsilon \in (0.5, 1, \sqrt{diameter(X_0)}, diameter(X_0))$ we observe the following values for the MSE between x_1 & $\hat{x_1}$: $(0.00621, 0.00103, 0.00041, 0.00045)$. We can conclude that ϵ can be set too small, e.g., 0.5 or 1, in which case the error becomes significantly larger but for bigger ϵ, like 3.5 or 12.4, the error only changes in very small margins.

Compared to the MSE for the linear approximation, the error for nonlinear approximation with radial basis functions and good settings of ϵ & L is significantly lower, i.e., 10^{-2} smaller. From this we conclude that the underlying **vector field** is **non-linear** since the approximation by a linear function was much worse than a nonlinear estimation with Radial Basis Functions.

3.2.1 Choosing the values of L & ϵ

While the decision on the final values of ϵ & L was based on the MSE of x_1 & $\hat{x}_1$, the range from which we picked was based on more theoretical considerations. For L the range of $[100, 1000]$ was pre-specified in the exercise. The upper bound is based on the maximum number of samples in the dataset: Since we are in a supervised learning environment, we do not want to overfit our training data. If we chose $L = N$, i.e., the number of samples, we would designate one radial basis function for each sample & would probably overfit the given training data x_0 & x_1 heavily. On the other hand, we need to have a sufficient number of basis functions to leverage the advantages of nonlinear approximation capabilities. Therefore, a minimum of 100 is chosen in this task.
For ϵ we had to find a reasonable range by ourselves. Since the parameter ϵ describes the "bandwidth" of the

radial basis functions, or equivalently the range around the center in which most of the integral over the function is present, it is reasonable to look at the data which should be approximated first. In our case, the data is defined on the area $[-4.5, 4.5]^2$ and therefore we exclude very small values below 0.5 for ϵ since those definitely cannot grasp the underlying nature of the whole dataset. To get a more solid estimate of a reasonable choice for our dataset x_0, we go back to the Diffusion Maps implementation from Exercise 3. There we estimated a similar parameter by the diameter (the maximum distance between two samples) of the dataset. Since we could choose either the diameter itself or its square root, we set the diameter itself as the maximum value we try for ϵ. In practice, we observed that $\epsilon = \sqrt{diameter(X_0)}$ yielded the best results, which is reasonable when considering that in our radial basis functions we actually use ϵ^2 instead of ϵ itself. The corresponding implementation can be found in the function "`diameter_dataset`" of the module "*./src/part1/utils.py*".

3.3 Part 3: Dynamic Behaviour of the approximated system

Finally, we want to observe the dynamical behaviour of the approximated system for longer time spans. For this, we set t_{end} to a much bigger value and again solve the system with "`solve_ivp`" from the package "*SciPy*".
In a first iteration we choose $\mathbf{t_{end} = 100}$ to get a feeling for the steady states of the system. Because of computation time we look at the results for the first few hundred samples. From this simulation we see, that there seem to be **4 steady states in the dynamical system** with our approximation:

$$[(-2.818, 3.137), (-3.767, -3.284), (3.006, 2.024), (3.591, -1.8493)]$$

To check if these 4 candidate steady states are all occurring steady states over the whole 2000 samples, we subsequently choose $\mathbf{t_{end} = 10}$ and print all final states which differ significantly (> 0.05) from the closest candidates. With this setting we can see that the simulation of every sample from x_0 ends up in close proximity to one of the four identified steady states and no differing final state is printed.
In a third simulation with $\mathbf{t_{end} = 1000}$ we want to find a close estimate of each of the four steady states. Here, we observe only the first few (50) samples since the computation time becomes significantly larger. As a result we can observe the following **4 final steady states for very long $\mathbf{t_{end}}$**:

$$[(-2.819, 3.150), (-3.773, -3.278), (2.994, 2.055), (3.595, -1.898)]$$

In conclusion, the approximated dynamical system cannot be equivalent to a linear system since a linear system has only 0, 1 or an infinite number of steady states (solutions). Since our approximated system converges into 4 different steady states which is neither 1 nor infinite (e.g., a line of steady states), they are not topologically equivalent.

Report on task 4, Time-delay embedding

---

4 Task 4 – Time-delay embedding

4.1 Part One: Embedding a Periodic Signal into a State Space

Task Description: The first part of this task involves embedding a periodic signal into a state space where each point carries enough information to advance in time. The dataset `takens_1.txt`, located in the data directory, contains the data matrix $X \in \mathbb{R}^{1000\times 2}$. The two columns are the two coordinates of a closed, one-dimensional manifold. Note that the manifold is one-dimensional because it can be mapped locally to a one-dimensional Euclidean space. However, the complete manifold cannot be embedded in a one-dimensional space due to its periodic nature. To start, plot the first coordinate against the line number in the dataset (the "time"). Then, choose a delay Δn of rows and plot the coordinate against its delayed version. According to Takens' theorem, determine how many coordinates are needed to be sure that the periodic manifold is embedded correctly.

The result of the first coordinate against the line number in the dataset can be found in 5.

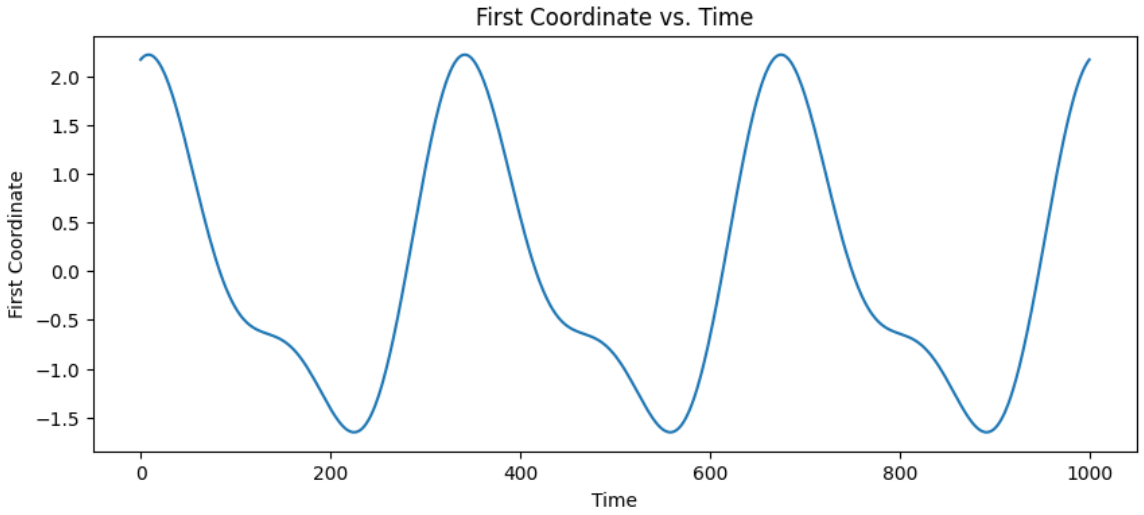


Figure 5

Then, we choose a delay Δn of rows (5, 10, 15, 20, 25) and plot the coordinate against its delayed version, the result can be found in 6.

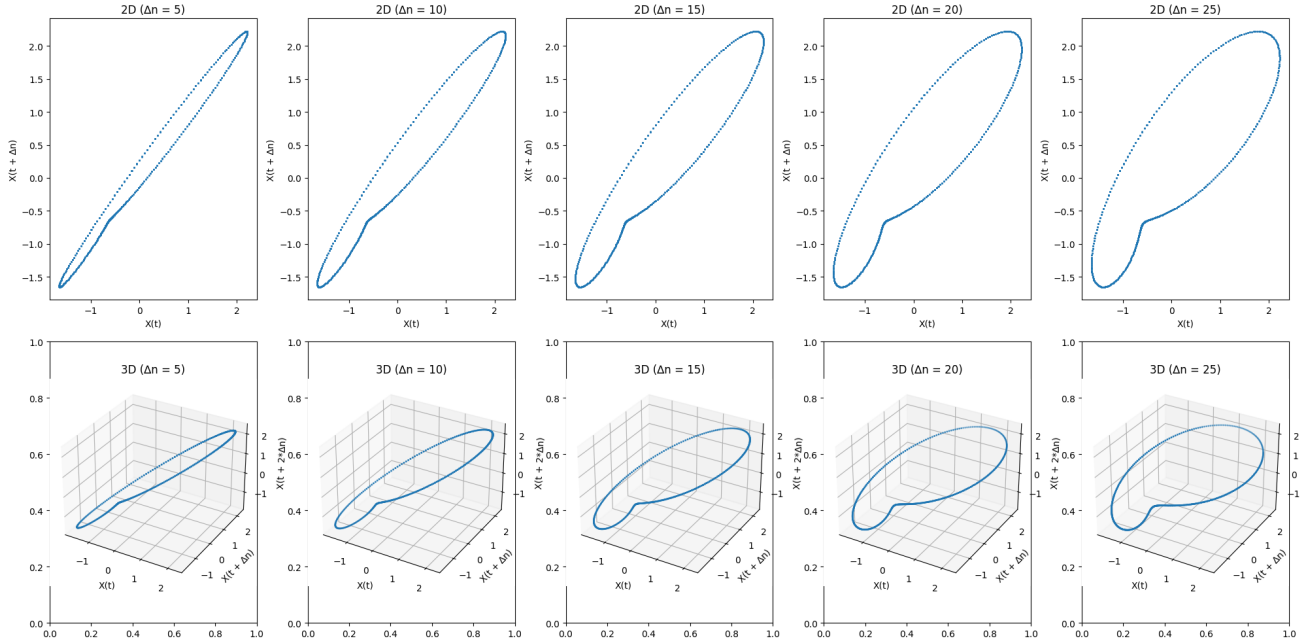


Figure 6

According to the **Takens theorem** and by our calculation, the minimum embedding dimension is $\mathbf{2k+1 = 2*1+1 = 3}$, since the dimensionality of the underlying manifold $k = 1$.

4.2 Part Two: Approximating Chaotic Dynamics from a Single Time Series

Task Description: The second part of this task involves approximating chaotic dynamics from a single time series. Refer back to the Lorenz attractor plotted in exercise three, using the parameters $\sigma = 10$, $\rho = 28$, and $\beta = \frac{8}{3}$ to be in the chaotic regime, starting from the point (10, 10, 10).

In this exercise, test Takens' theorem for this fractal set. If the coordinates in your Lorenz attractor are called x, y, and z, imagine you can only measure the x-coordinate and do not know about y and z. Takens' theorem tells you how you can still get a reasonable idea about the shape of the attractor. Visualize $x_1 = x(t)$

against $x_2 = x(t+\Delta t)$ and $x_3 = x(t+2\Delta t)$ in a three-dimensional plot, for a suitable choice of $\Delta t > 0$. Describe the result in comparison to the attractor in x, y, and z coordinates.

Repeat the same process for the z-coordinate, where an embedding should fail. Discuss why this might be the case.

The illustration for the generated data for Lorenz attractor can be found in 7

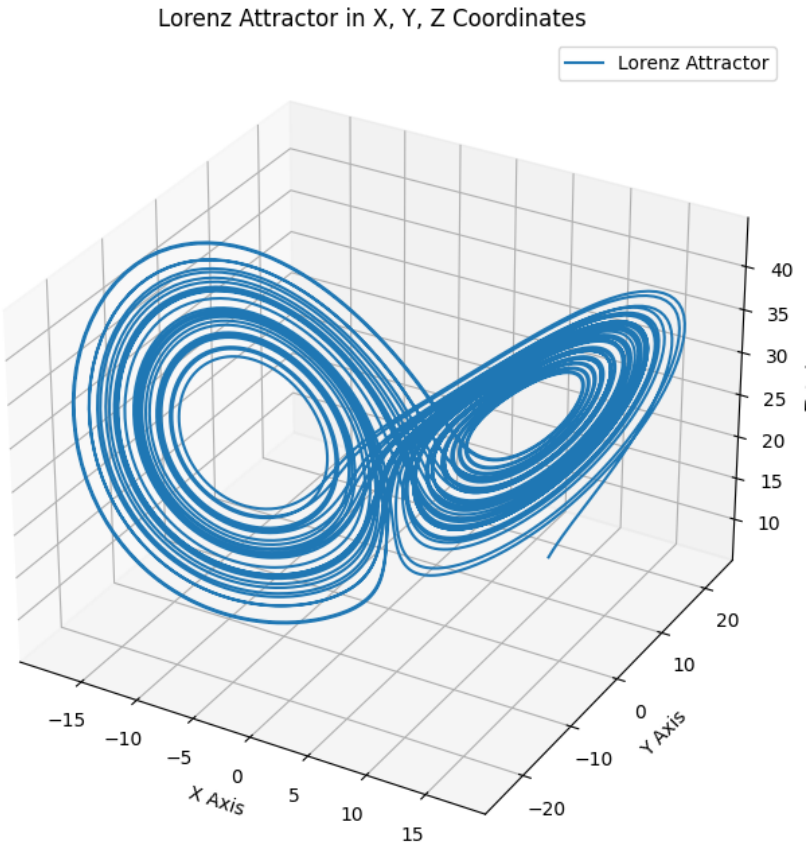


Figure 7

Time-Delayed Embedding of the Lorenz Attractor

In this exercise, we tested Takens' theorem by visualizing the Lorenz attractor through time-delayed embeddings using only the x-coordinate and z-coordinate, respectively.

The Visualization of the time-delayed x coordinates from Lorenz data in a 3-d Euclidean space (first row) and visualization of the time-delayed z coordinates from Lorenz data in a 3-d Euclidean space (second row) can be found in 8. The Δt value are 5, 10, 15, 20, 25.

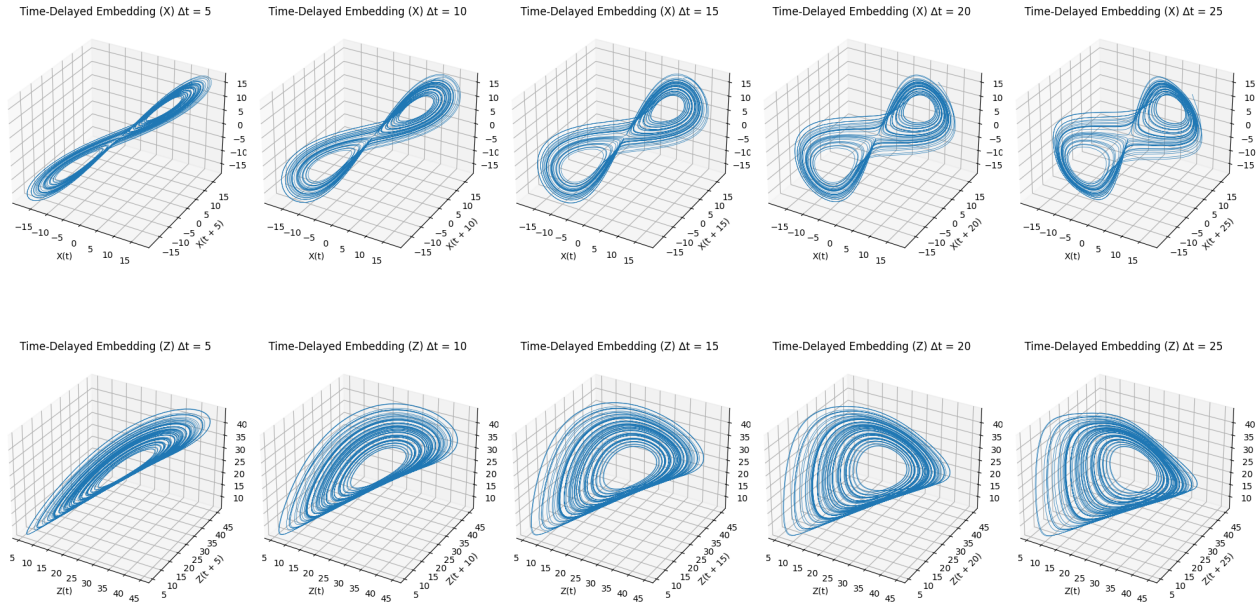


Figure 8

Embedding with x-coordinate

We created three-dimensional plots by embedding $x(t)$, $x(t + \Delta t)$, and $x(t + 2\Delta t)$ with various values of Δt. The results, as shown in the top row of the provided figure, demonstrate that for appropriate choices of Δt, the reconstructed attractor preserves the characteristic shape of the Lorenz attractor. Specifically, the plots for $\Delta t = 5, 10, 15, 20, 25$ all depict the expected double-loop structure of the Lorenz attractor, confirming that the essential dynamics are captured using only the x-coordinate. This aligns with Takens' theorem, which asserts that a higher-dimensional dynamical system can be reconstructed from a single time series through time-delay embedding.

Embedding with z-coordinate

In contrast, the bottom row of the figure shows the time-delayed embeddings using the z-coordinate. The embeddings with $z(t)$, $z(t + \Delta t)$, and $z(t + 2\Delta t)$ for the same values of Δt do not exhibit the clear double-loop structure seen in the x-coordinate embeddings. Instead, the shapes appear more distorted and less representative of the true Lorenz attractor. This failure in embedding with the z-coordinate can be attributed to the fact that z varies less dynamically compared to x and y, making it less suitable for capturing the full complexity of the attractor through time-delayed embedding.

Summary

The findings confirm that the x-coordinate of the Lorenz attractor provides a sufficient basis for reconstructing the attractor's dynamics through time-delayed embedding, as predicted by Takens' theorem. Conversely, embeddings using the z-coordinate do not succeed in capturing the attractor's structure, highlighting the importance of selecting an appropriate variable for time-delay embedding. This exercise underscores the utility of Takens' theorem in reconstructing higher-dimensional dynamical systems from a single time series, provided the chosen observable exhibits adequate dynamical variability.

4.3 Bonus: Estimating the New State Space for the Lorenz Attractor

Task Description: As a bonus task, after estimating the new state space for the Lorenz attractor using the x-coordinate, approximate the vector field $\hat{\nu}$ on it using radial basis functions. Then, solve the differential equation

$$\frac{d}{ds}\psi(s, x_1(t), x_2(t), x_3(t))\Big|_{s=0} = \hat{\nu}(x_1(t), x_2(t), x_3(t))$$

using your approximation with a standard solver (e.g., `solve_ivp`) and compare the trajectories to the training data.

4.4 Methodology

Generating the Lorenz Attractor

- Used parameters $\sigma = 10$, $\rho = 28$, $\beta = 8/3$ with initial conditions $(10, 10, 10)$.
- Solved the Lorenz system using `solve_ivp` to obtain the trajectories for x, y, and z.

Estimating the Vector Field

- Created time-delayed coordinates $x_1 = x(t)$, $x_2 = x(t + \Delta t)$, $x_3 = x(t + 2\Delta t)$ with $\Delta t = 10$.
- Combined these coordinates into a single matrix X and computed the velocity vectors V.
- Reduced the dataset size by sampling every 10th point to address the long computation time during RBF fitting.
- Fitted RBFs to the reduced dataset to approximate the vector field $\hat{\nu}$.

Solving the Differential Equation

- Defined the differential equation using the approximated vector field $\hat{\nu}$.
- Solved the differential equation with initial conditions using `solve_ivp`.
- Compared the resulting trajectories with the original training data.

Choice of L (Number of Centers) To balance the detail of the approximation and computational efficiency, we selected $L = 1000$. This number ensures a detailed approximation necessary for the complex dynamics of the Lorenz attractor while keeping computation feasible. The code ensures L does not exceed the number of available data points:

```
L = min(1000, X_sampled.shape[0])
indices = np.random.choice(X_sampled.shape[0], L, replace=False)
centers = X_sampled[indices]
```

Choice of ϵ (Shape Parameter) We chose $\epsilon = 0.1$ to balance between localization and smoothness. This value was selected to ensure the basis functions are appropriately scaled to model the Lorenz attractor's variations:

```
epsilon = 0.1
rbf_x1 = Rbf(X_sampled[:-1, 0], X_sampled[:-1, 1], X_sampled[:-1, 2], V_sampled[:, 0], function='multiq
rbf_x2 = Rbf(X_sampled[:-1, 0], X_sampled[:-1, 1], X_sampled[:-1, 2], V_sampled[:, 1], function='multiq
rbf_x3 = Rbf(X_sampled[:-1, 0], X_sampled[:-1, 1], X_sampled[:-1, 2], V_sampled[:, 2], function='multiq
```

The following figure shows the comparison of the original and approximated trajectories: 9.

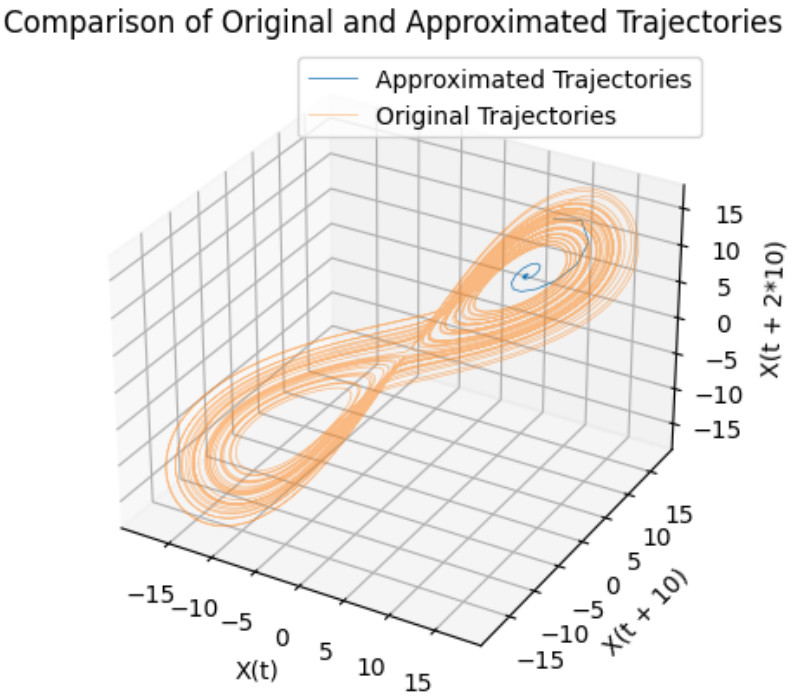


Figure 9

The comparison of the original and approximated trajectories demonstrates that the radial basis function (RBF) approach is capable of capturing the general structure of the Lorenz attractor. However, there are notable differences between the two sets of trajectories:

- The approximated trajectories exhibit a similar overall shape to the original Lorenz attractor, indicating that the RBF approximation was able to capture the main dynamics of the system.
- There are discrepancies between the original and approximated trajectories, particularly in regions where the attractor exhibits more complex behavior. This could be due to the reduced dataset size and the limitations of the RBF approximation.
- The differences between the trajectories highlight the sensitivity of chaotic systems to initial conditions and the challenges in accurately approximating such systems with limited data.

Overall, the RBF approximation provides a useful tool for understanding the dynamics of the Lorenz attractor, but further refinement and additional data may be necessary to achieve a more accurate representation.

Explanation of Changes: Interpretation Section: Added a new section "Interpretation of the Results" to provide insights

Report on task 5, Learning crowd dynamics

5 Learning Crowd Dynamics

5.1 Creation of the delay embedding

In order to explain how many dimensions are needed to embed a reasonable state space of the given system, we need to take a look at the system itself first. As specified in the exercise sheet we can reasonably assume that the manifold, which is the state space, is periodic and without parametric dependencies which results in a 1D-manifold. Since the given dataset has 9 measuring points for each time-step, we can say that we are given a 9-dimensional embedding of a 1D-manifold in this task. Therefore, in Takens theorem we have $\mathbf{d = 1}$ & $\mathbf{k = 9}$. Subsequently, the **number of dimensions** of the time-delay embedding needed to embed the state space is

$2d + 1 = \mathbf{3}$.

The implementation of the delay embedding with 350 delayed values for each data point can be found in the "Task 5.1"-section of the Jupyter-notebook "*./src/part2/task5.ipynb*". While the notebook mainly contains the meta-parameters & pre-processing of the given data, the actual creation of the embedding is realised in the "`create_crowd_embedding`"-function of the module "*./src/part2/utils.py*". Here, the columns from which the embeddings should be built can be specified, as well as the number of delays considered in each column, in our example 350, and the number of data points M that should be constructed from the given data.
After the time-delay embedding is constructed, a Principal Component Analysis with three principal components is done with the *sklearn*-library "*sklearn.decomposition.PCA*". The first three principal components are chosen because formally (from Takens theorem) we would only need three time-delayed embedding dimensions, e.g., one observed dimension with 2 time-delayed observations, to find an embedding of the 1D-manifold state space, which was explained earlier in the section.

5.2 Creation of 9 plots - one for each measurement area

As described in the exercise sheet, there are two variables containing the data points for all measurement areas (`x_original`) and the embedded points in the PCA space (`pca_representation`). For each measurement area a subplot is created with `plt.scatter(*pca_representation.T, c=x_original[:, i])`. The result can be seen in figure 10.

Various plots (measurement areas 3, 4, 7, and 8) show no different coloring in their visualization. While others include multiple colors with the measurement area 2 containing the biggest colored area out of all subplots. Overall, we can see that the PCA-repsentation state is embedding **approximately a closed loop** for all measurement areas which was the expected behaviour.
Therefore, we can state that 3 embedding dimensions according to Takens are actually enough to grasp the 1-dimensional manifold of the data, which is a closed loop.

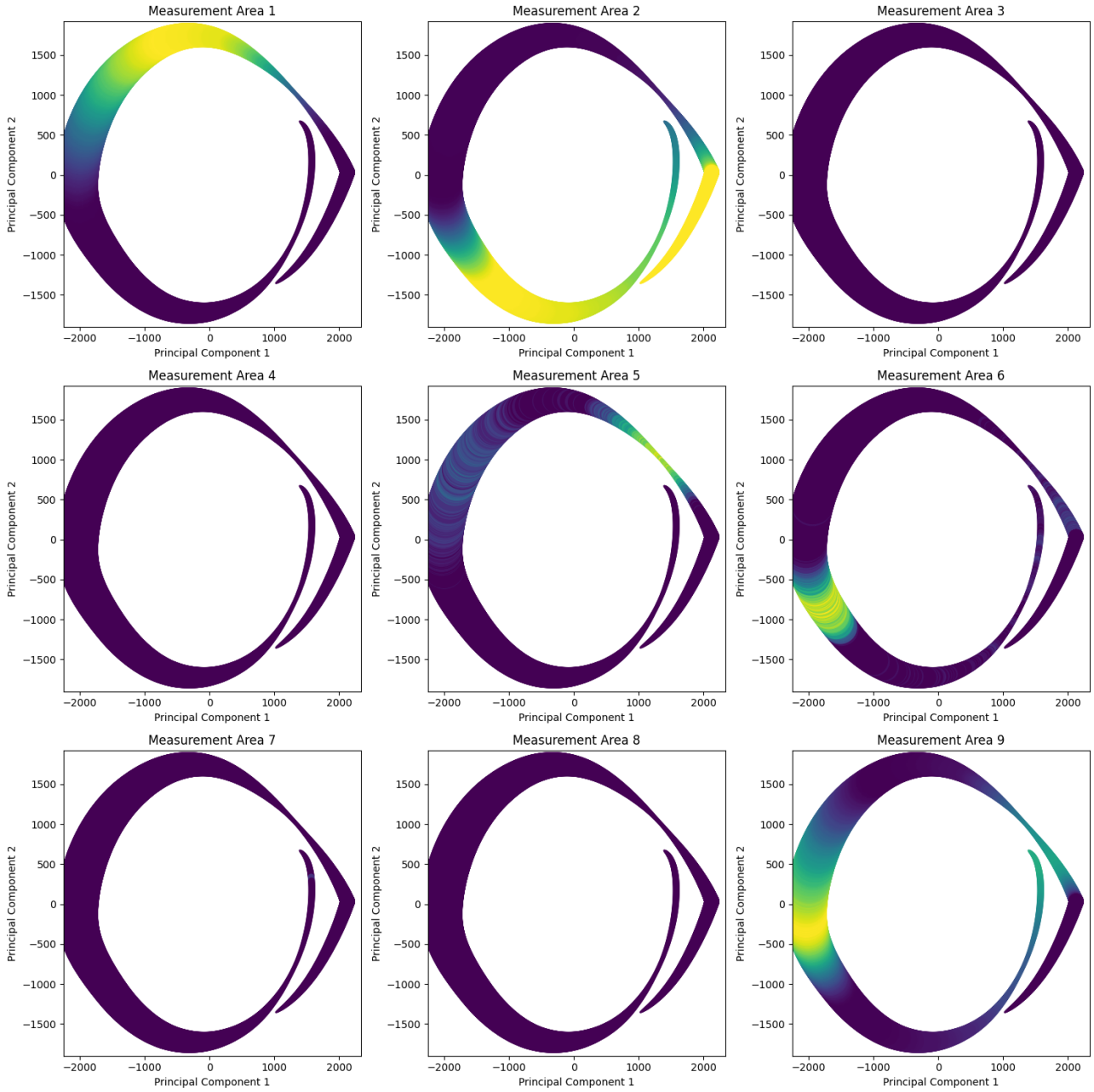


Figure 10: Points by all measurements taken at the first time point of the delays colored for all nine measurement areas

5.3 Learn the dynamics on the periodic curve you embedded in the principal components

The goal of this task is to analyze the dynamics of a periodic curve embedded in the principal components (PCA) space. By leveraging the available time step data, we aim to understand the speed at which the system progresses through the PCA space at each point. This analysis will give us insights into how the system evolves over time in the PCA space, revealing the dynamics and speed at various points along the periodic curve.

The illustration of the result can be found in 11.

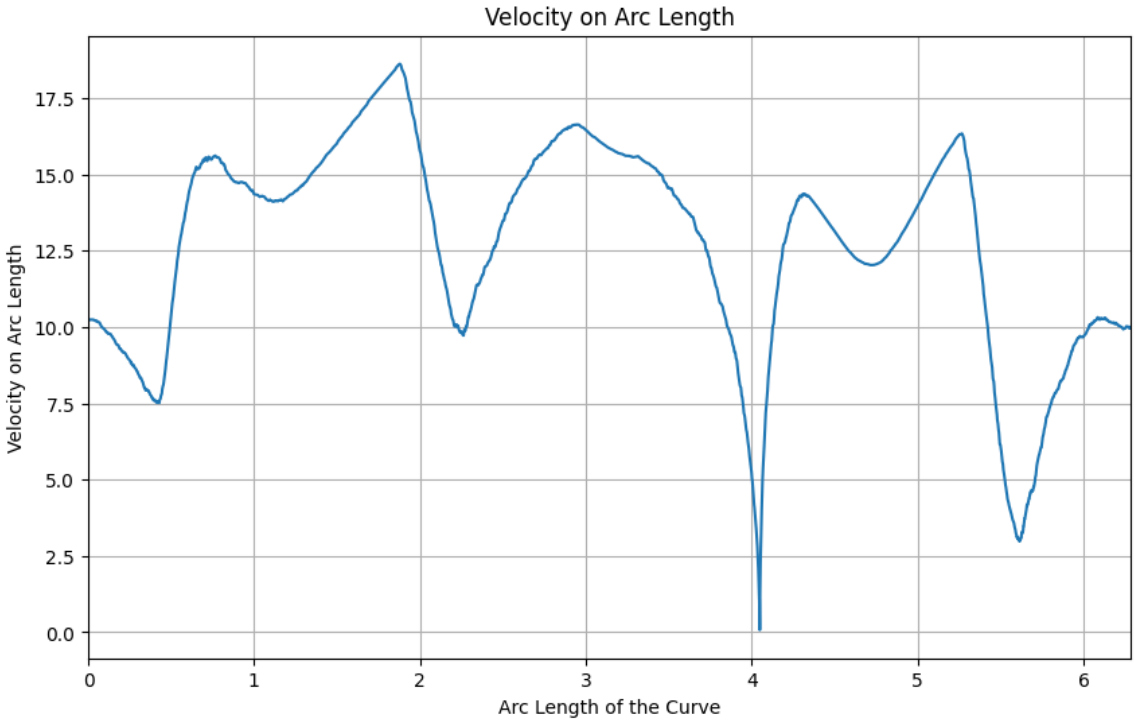


Figure 11

Here the Pseudo code for calculating velocity on arc length in PCA space is also provided, for better understanding.

```
# Step 1: Calculate the arc length in PCA space
# Compute the differences between consecutive points in the PCA space
arc_lengths = np.linalg.norm(np.diff(pca_representation, axis=0), axis=1)

# Step 2: Compute the cumulative arc length
# Start with zero and accumulate the arc lengths
cumulative_arc_length = np.concatenate(([0], np.cumsum(arc_lengths)))

# Step 3: Normalize the arc length to range from 0 to 2 pi
# Scale the cumulative arc length to fit within the range [0, 2*pi]
normalized_arc_length = 2 * np.pi * cumulative_arc_length / cumulative_arc_length[-1]

# Step 4: Calculate the velocity along the normalized arc length
# Use gradient to find the rate of change of the arc length
velocity = np.gradient(cumulative_arc_length)
```

5.4 Prediction of utilization of the MI building

The goal of this task is to predict the utilization of the MI building (first measurement area, first column in the file after the time steps) for the next 14 days. The prediction will be achieved through the following steps:

1. **Learned Vector Field Integration**: Utilize the learned vector field on the one-dimensional, periodic space to predict future arclength values over time.
2. **Radial Basis Function Approximation**: Map the predicted arclength values back to the original utilization values using radial basis function (RBF) approximation.
3. **Comparison with Given Data**: Plot the predicted utilization values over time and compare them with the actual data.

The resulting prediction and comparison will provide insights into the accuracy of the model and the expected utilization trends for the MI building over the specified period.

The illustration of the result can be found in 12.

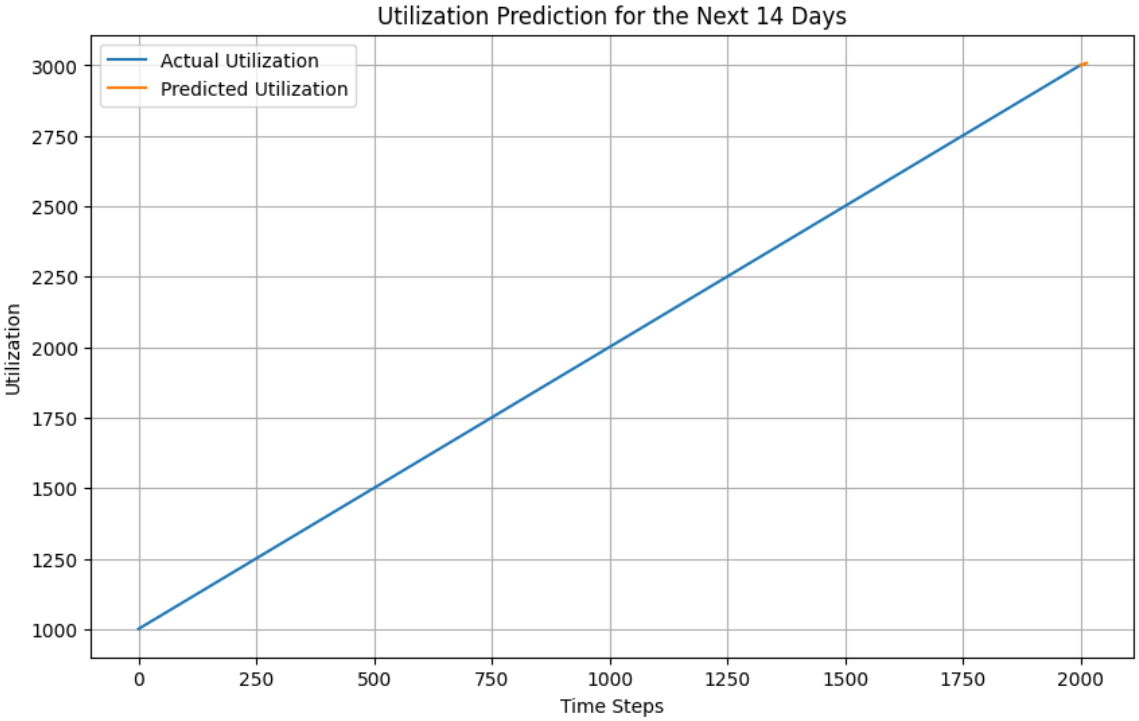


Figure 12

The predicted utilization is just continuing the trend of the actual utilization, which might indicate that the prediction method didn't fully capture the periodic nature of the data or that the integration step didn't work as expected. Due to lack of expertised knowledge and lack of time, we did not decide to pursue the issue.

References

[1] T. Berry, J. R. Cressman, Z. Greguric-Ferencek, and T. Sauer. Time-Scale Separation from Diffusion-Mapped Delay Coordinates. *SIAM Journal on Applied Dynamical Systems*, 12(2): 618649, January 2013.