Report for exercise 5 from group C

Tasks addressed:	5
Authors:	Alejandro Hernandez Artiles (03785345) Pavel Sindelar (03785154) Haoxiang Yang (03767758) Jianfeng Yue (03765255) Leonhard Chen (03711258)
Last compiled:	2024–03–02
Source code:	https://github.com/alejandrohdez00/Exercises-MLCMS-Group-C/tree/main/Exercise-5

The work on tasks was divided in the following way:

Alejandro Hernandez Artiles (03785345)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	100%
	Task 5	0%
Pavel Sindelar (03785154)	Task 1	0%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	100%
Haoxiang Yang (03767758) **("Project Lead")**	Task 1	0%
	Task 2	100%
	Task 3	0%
	Task 4	0%
	Task 5	0%
Jianfeng Yue (03765255)	Task 1	0%
	Task 2	0%
	Task 3	100%
	Task 4	0%
	Task 5	0%
Leonhard Chen (03711258)	Task 1	100%
	Task 2	0%
	Task 3	0%
	Task 4	0%
	Task 5	0%

Report on task 1, Approximating functions

In this task we want to implement basic approximation algorithms using linear least squares and radial basis functions. We demonstrate some basic use cases on (A) a linear data set `linear_function_data.txt` and (B) a non-linear data set `nonlinear_function_data.txt`. Both data sets contain 1000 data points in $\mathbb{R}^2$.

Implementation The implementation is split into a python notebook `task_1.ipynb` for conducting the experiments and `utils.py` which contains all function implementations. As general note our own implementations can achieve similar results as the library implementation. We believe that our implementation is numerically unstable and hence have different hyper-parameters, since the library implementation uses a LU-decomposition. Here is a short overview on the functions provided by `utils.py`:

Function	Description
`load_dataset(...)`	Load dataset by line. Has an option for sorting and choosing the string to splice by.
`prepare_data(...)`	A helper function that splits data into x, y and adjusts the shape.
`expand_data(...)`	Turns $x \in \mathbb{R}$ into $x \in \mathbb{R}^n$ by applying a list of functions on x.
`lstsq_direct(...)`	A helper function with direct implementation of the closed-form solution for least squares.
`linear_lstsq(...)`	A wrapper function that uses input data to approximate a function using linear least squares. Has an option to use and library implementation instead.
`rbf(...)`	A helper function to calculate a Gaussian radial basis function.
`rbf_lstsq(...)`	Wrapper function that uses input data to approximate a function using radial basis functions. Has an option to use and library implementation instead.
`plot_comparison(...)`	Plot 2 different functions for comparison.

Part 1: Approximate dataset (A) with a linear function. In this task we approximate a linear function on the dataset `linear_function_data.txt` using linear least squares. After importing the dataset it was sorted in ascending order by its first feature. We will approximate a coefficient matrix A from the data by a linear function with bias $b = 0$:

$$\hat{f}(x) = Ax + b$$

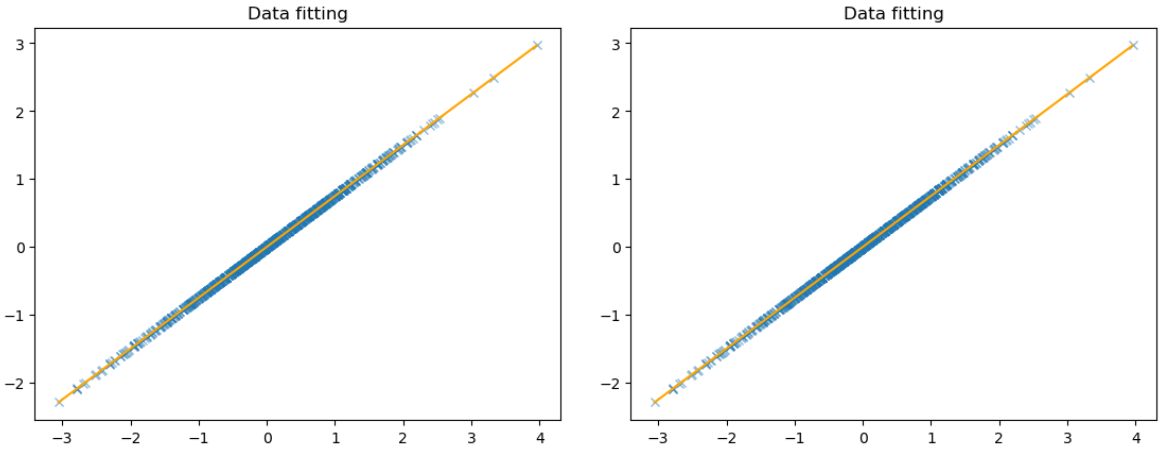
In Figure 1 we plotted the results. We can immediately observe that linear least squares can easily capture the function in the linear data. The blue crosses are the data points from `linear_function_data.txt`. The approximated function is plotted in orange. Figure 1(a) shows the results of using our own implementation of the linear least squares. Note that our implementation does not use the parameter `cond`, which cutoffs singular values smaller than `cond` times the largest singular value of A. By default `cond` is always set to 1.0. Figure 1(b) shows the results after using the library implementation. Additionally we included one sample execution time. Our implementation was faster, since it has less features compared to the library function. In the following tasks we will use the library implementation, since we assume it is more numerically robust.

The results for the coefficient matrix A is as follows:

$$A_{\text{direct}} = 0.75000024, \quad A_{\text{library}} = 0.75000024$$

We used the following parameters:

	`bias`	`cond`	execution time
Figure 1(a)	False	/	0.000165
Figure 1(b)	False	1.0	0.042042



(a) Our implementation (b) Library implementation

Figure 1: Plots of approximated functions on linear data

Part 1.1: Why is it not a good idea to use radial basis functions for dataset (A)? While linear least squares works perfectly fine on linear data, we might think of using a more powerful model, such as radial basis functions (RBF), to fit out data. This is generally a bad idea, if we can safely assume that our data is linear. There are many downsides to using a more powerful model like radial basis functions. The main downside in our case of Gaussian RBFs is that we need to tune our hyper-parameters L and ϵ. Upon further analysis summarize what parameters are good

1. We know that using a composition of non-linear transformations will likely result in a non-linear function if we use $L > 2$. Therefore we can capture linearity only in the boundaries of our sample data.
2. If we choose bad hyper-parameters then a bad i.e. non-linear fit will be visible within the range of the sample data. This can be see in figures 2(a)(b). But even good fit for $L > 2$ will likely be non-linear. Hence it fits linear data incorrectly.
3. The optimal choice would be $L = 2$ and then trying to find a good ϵ. But this ϵ can get quite large and overall it is a less efficient representation of linear data than linear least squares. A good fit using RBFs can be seen in figures 2(c)(d).

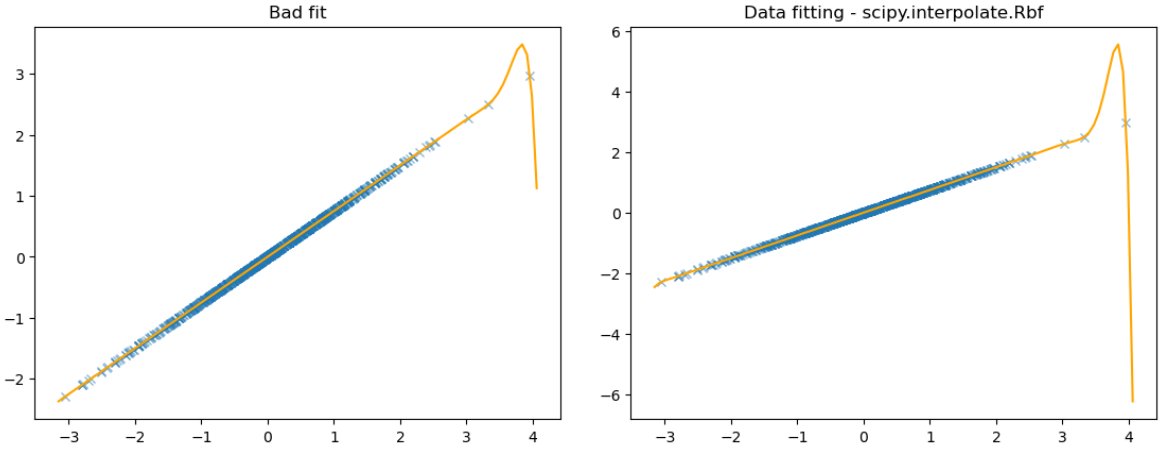
In conclusion it is clear that theoretically RBFs are able to perfectly fit the data, but when used we need to tune hyper-parameters. Choosing bad hyper-parameters can be seen in figures 2(a)(b) and for $L > 2$ the approximation is likely non-linear. Even if we can choose good hyper-parameters, ϵ can grow very large and it is overall a very inefficient representation of compared to linear least squares.

The results for the coefficient matrix C (of our implementation 2(a)(c) are as follows:

$$C_{\text{Bad}} = (-1183008.2, -4276724.5, 25944456.4, ...), \quad C_{\text{Good}} = (-94.999, 95.341)$$

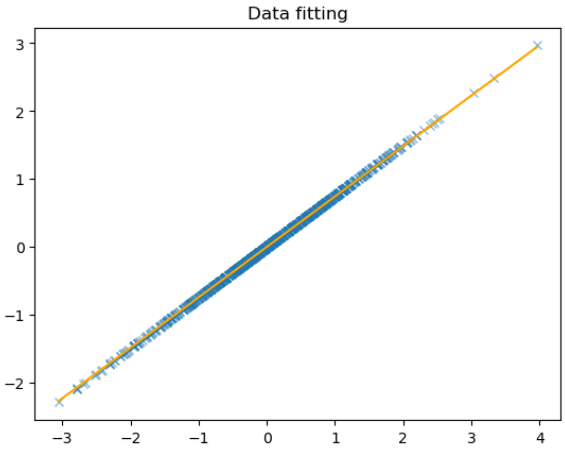
We used the following parameters:

	L	ϵ
Figure 2(a)	100	1.2
Figure 2(b)	100	1.2
Figure 2(c)	2	0.1
Figure 2(d)	2	1

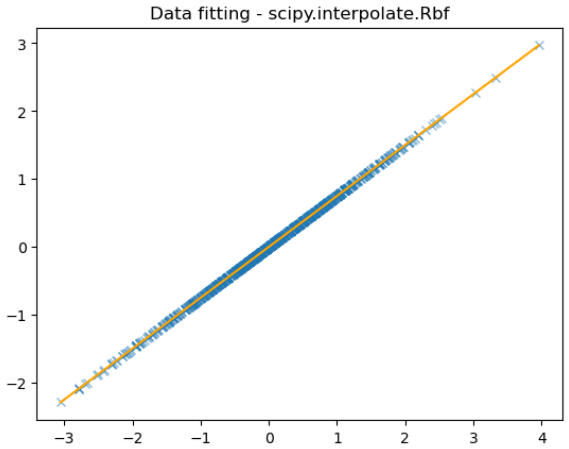


(a) Our implementation

(b) Library implementation



(c) Our implementation



(d) Library implementation

Figure 2: Plots of approximated RBFs on linear data

Part 2: Approximate dataset (B) with a linear function. In this task we will try to approximate the non-linear dataset `nonlinear_function_data.txt` with linear least squares. Again we have pairs of x, y coordinates and sorted the data by its first feature x. We will approximate a coefficient matrix A from the data, but this time we included the bias b for the linear function $\hat{f}(x) = Ax + b$.

In Figure 3 we have plotted the results only using our implementation of least squares. As expected the linear least squares is only able produce a linear function as approximation. Even though the resulting bias is

very small we added it this time.

(As an additional remark we also extended the linear least squares to a Taylor series with polynomial of degree 13. In the python notebook `task_1.ipynb` under task 1.2 it is interesting to see that we can achieve a nice fit using few polynomials. It is important to note that this model makes the assumption that the data has a certain polynomial behaviour.)

The results for the coefficient matrix A is as follows (note that bias b is the second value in A):

$$A = (0.02873498, 0.11114247)$$

We used the following parameters:

	`bias`	`cond`
Figure 3	True	/

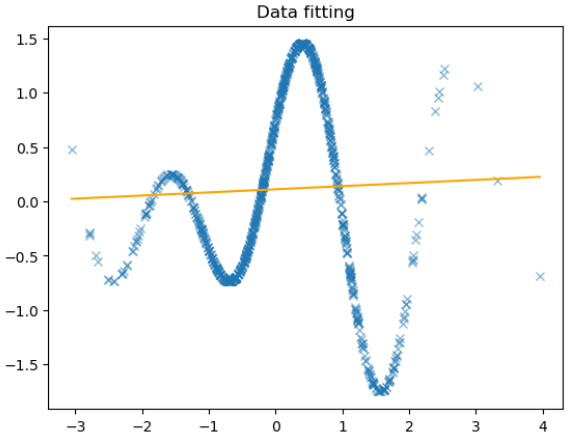


Figure 3: Plots of approximated functions on linear data

Part 3: Approximate dataset (B) with a combination of radial functions. In this final part we are going to approximate the non-linear dataset using radial basis functions (RBF). Specifically here we are going to use a Gaussian RBF of the form:

$$\phi_l(x) = \exp\left(\frac{||x_l - x_k||^2}{\epsilon^2}\right) \tag{1}$$

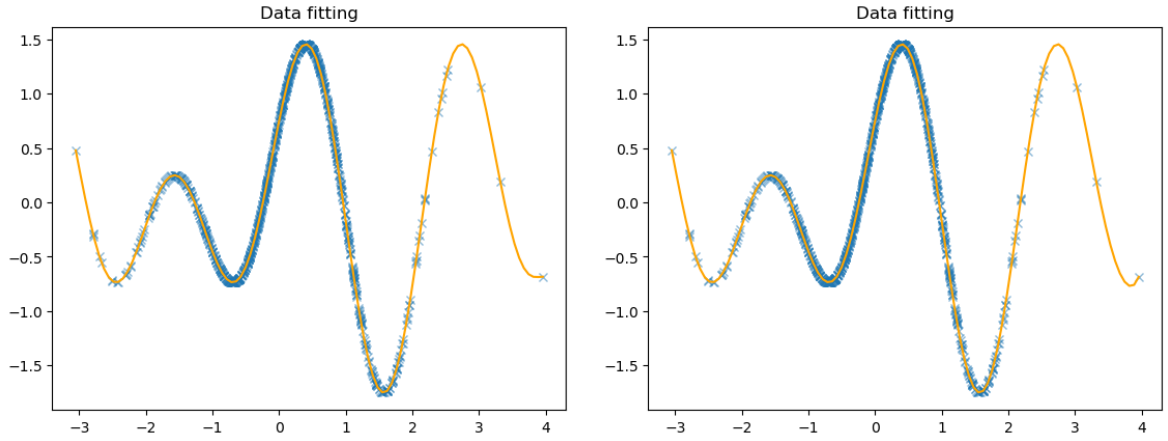
In figures 4 we see an approximation using RBFs. Points were uniformly distributed and we receive a very smooth approximation of the dataset. On the left 4(a) is our own implementation, while the plot 4(b) uses the `scipy` library implementation of RBFs. In our implementation it is possible to receive the coefficient matrix C.

The results for the first 3 entries of the coefficient matrix C are as follows:

$$C = (483730.3, 2724108.7, -11428130.9, 5181098.3, 9651220.1)$$

We used the following parameters:

	L	ϵ
Figure 4(a)	100	1.4
Figure 4(b)	100	1.8



(a) Our implementation (b) Library implementation

Figure 4: Plots of approximated functions on non-linear data

Discuss how and why you chose the values of L and ϵ for the radial basis functions? As we have seen so far L controls the amount of center points. This amount should be higher if the function is more complex. The non-linear dataset resembles a high-degree polynomial (Figure 4), hence choosing a decently large L makes sense.

On the other hand as seen in part 1.1 choosing a smaller L is appropriate when dealing with less data such as linear function data. Therefore we choose L=2 since we are dealing with a linear function. Overall the number of center points L should be picked depending on the complexity of the functions curvature.

The hyper-parameter ϵ controls the bandwidth and inside the Gaussian RBF it acts as a scaling constant in the exponent. Intuitively it controls regularization strength and higher values force over-fitting. Generally choosing ϵ can vary by case, but a good rule of thumb could be to choose it as small as possible to prevent over-fitting.

Additionally using ϵ or ϵ^2 for the Gaussian RBF makes no difference from a theoretical standpoint. But when taking tuning of hyper-parameters into consideration ϵ^2 values increase quadratically and are better suited when trying to over-fit quickly. On the other hand a single ϵ is better for fine-tuning or working with data that is sensitive to regularization.

Report on task 2, Approximating linear vector fields

For this task, we have two linear vector field datasets, each containing 1000 rows and two columns, for 1000 data points x_0 and x_1 in two dimensions.
Approximating vector fields, associated with a vector to every point in the domain space, can describe the direction and rate of change of the state of the system at all the points. Here we introduce the finite-difference formula

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

, x_0 means points at time t=0 , and points x_1 means points at time t = Δt, at a short time later.

Part one we estimate the linear vector field used to generate the points x_1 from the points x_0. The implementation contains two files `task2.ipynb` are testing results and pictures. `utils.py` is the auxiliary file used to plot output and calculate the trajectory of the point's movement. We use the introduced formula above, we can have the equality as

$$v(x_0^{(k)}) = v^{(k)} = Ax_0^{(k)}$$

. This figure 5 shows a visualization of the original dataset distribution.

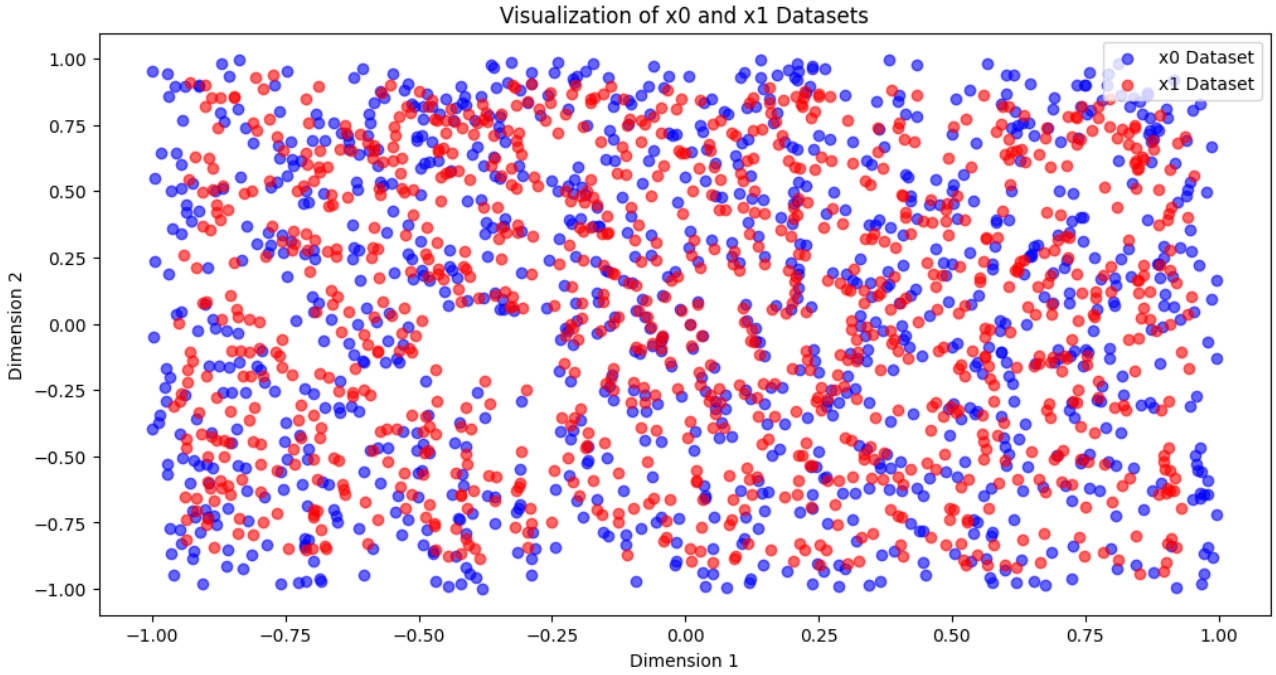


Figure 5: Initial data distribution

We are trying to find the optimal coefficient matrix $A \in R^{2x2}$ to represent the mapping relation between these two datasets. After applying the finite difference method, the estimated field vector can be found. We put the estimated field vector into the function `np.linalg.lstsq` the default Numpy function to have estimated coefficient A.

Part two This task is to solve the linear system and then compute the mean squared error. The estimated A is used to calculate the predicted x_1, the implementation function in `utils.py` called trajectory. it will receive the the data position at time t = 0, and the data position after the Δt time, also the linear function is also needed. Figure6 showed the trajectory of the after the movement of the delta time. Red dots mean the original poison and green lines are the trajectory of each point. The final result error is $2.83210804e + 04$. MSE is 0.32910430761537535. which is quite big in this case. which results in trajectory lines being longer and bigger.

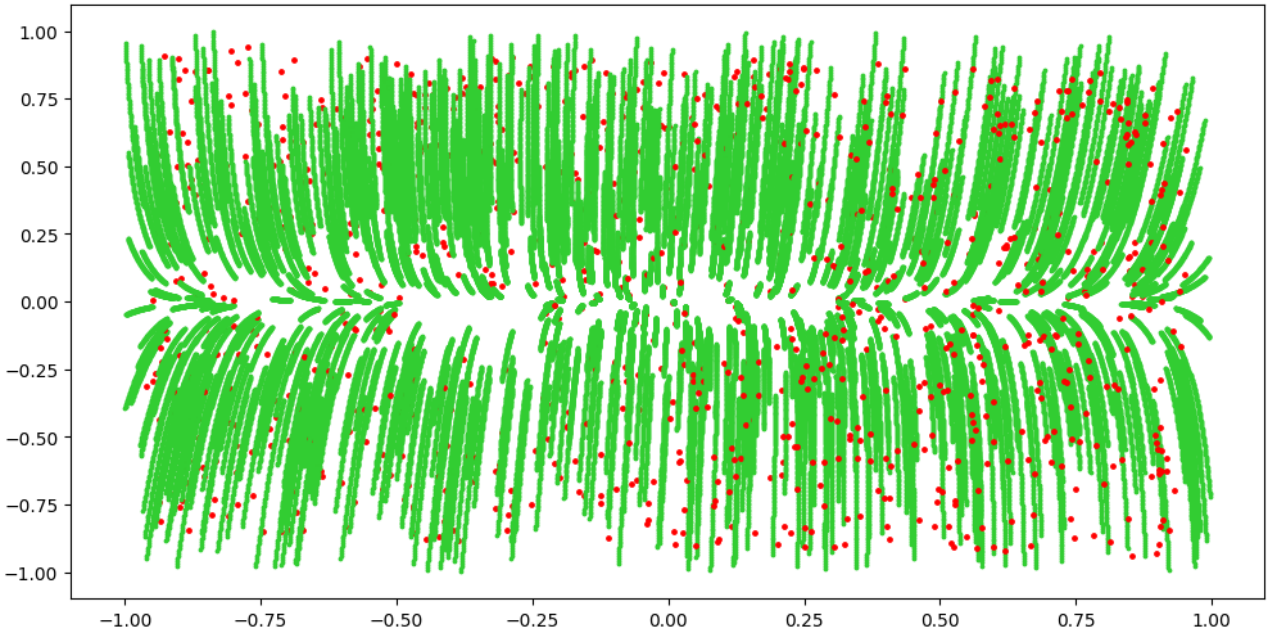


Figure 6: Trajectory from x_0 to x_1 with $\Delta t = 0.1$

Part three The last part of the task is to visualize the trajectory as well as the phase portrait. 7 where you can find the phase portrait of the linear field vector. The direction of each blue line is observed, and the dynamical system looks. The initial point $10, 10$ is set far outside the initial data, we solve this linear system with approximated matrix A, for time $T_{end} = 100$. The figure 7 demonstrates the dynamic system of all trends from the surrounding area into the middle area which is the reflection of the 6 but more clearly. And the orange scatter plot shows the far apart point will reach at the stable state at the end of the time phase. The implementation function is `create_phase_portrait_matrix` in `utils.py`

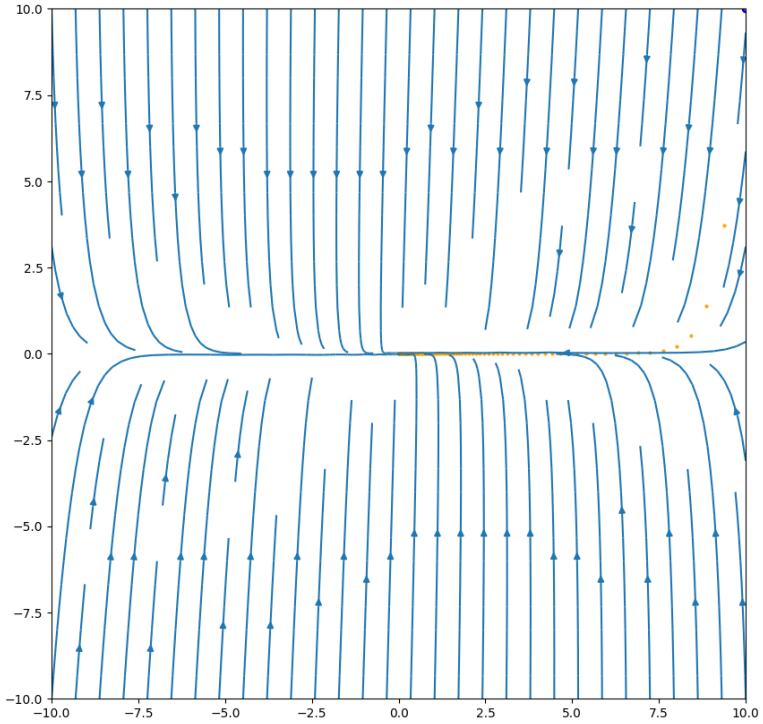


Figure 7: Phase portrait of $x_0 = [10, 10]$ and $t_end = 100$

Report on task 3, Approximating nonlinear vector fields

Our goal in this task is to explore and understand the underlying dynamics of a process. This process is characterized by two specific datasets: *nonlinear_vectorfield_data_x0.txt* and *nonlinear_vectorfield_data_x1.txt*. Each dataset comprises 2000 data points, organized into two columns, within the two-dimensional domain $[-4.5, 4.5]^2$. The dataset *nonlinear_vectorfield_data_x0.txt* contains the initial positions of these points, whereas *nonlinear_vectorfield_data_x1.txt* presents the same points after a transformation through an unknown evolution operator ψ. This transformation is mathematically represented as:

$$x_1^{(k)} = \psi(\Delta t, x_0^{(k)})$$

for $k = 1, \ldots, N$, where $N = 2000$ and $\Delta t = 0.01$. The data represented as scatter points can be seen in Figure 8.

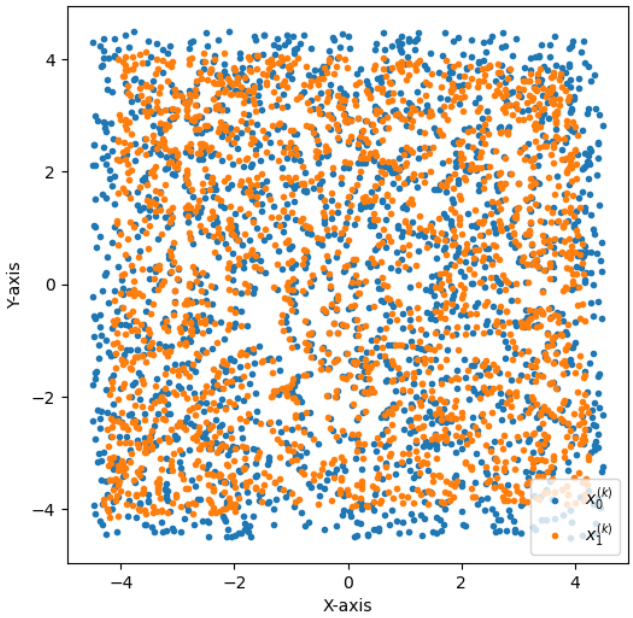


Figure 8: scatter points

Part 3.1: Estimate the vector field with a linear operator This sub-task centered on approximating the vector field described by ψ with data from two databases. This is formulated as:

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

To determine the vector field, we computed $\hat{v}^{(k)}$ for each data point using the formula:

$$\hat{v}^{(k)} = \frac{x_1^{(k)} - x_0^{(k)}}{\Delta t} \tag{2}$$

Here, $x_1^{(k)}$ and $x_0^{(k)}$ are the known end points and initial points of the vectors, respectively. The term Δt represents the small time interval over which the changes in the vectors are observed.

Then we aimed to establish a relationship where the vector field $\hat{v}^{(k)}$ is approximated by the product of a linear operator A and the input point $x_0^{(k)}$, expressed as $\hat{v}^{(k)} = Ax_0^{(k)}$. To get A, we employed the `np.linalg.lstsq` function from NumPy, a robust tool for computing the least squares solution to a linear matrix equation. The obtained value of A is

$$A = \begin{bmatrix} -1.0016012 & 0.08672716 \\ -0.02534942 & -4.32671381 \end{bmatrix}.$$

We solved the differential equation $\frac{dx}{dt} = Ax$ using the `solve_ivp` function over a small interval Δt from 0 to 0.02 with 1000 steps. The goal is to find the approximate end points $\hat{x}_1$ for each initial point x_0, aiming to achieve the smallest mean squared error (MSE).

In our investigation, we found that the smallest mean squared error (MSE) is achieved at the 510th step of our calculation. This corresponds to a Δt value of 0.0102. At this specific time interval, our computed approximate end points $\hat{x}_1$ are as close as possible to the known end points x_1, resulting in an MSE of 0.01864.

We plotted the estimated points $\hat{x}_1$ in green on the scatter plot, resulting in Figure 9. The results are observed to be acceptably adequate.

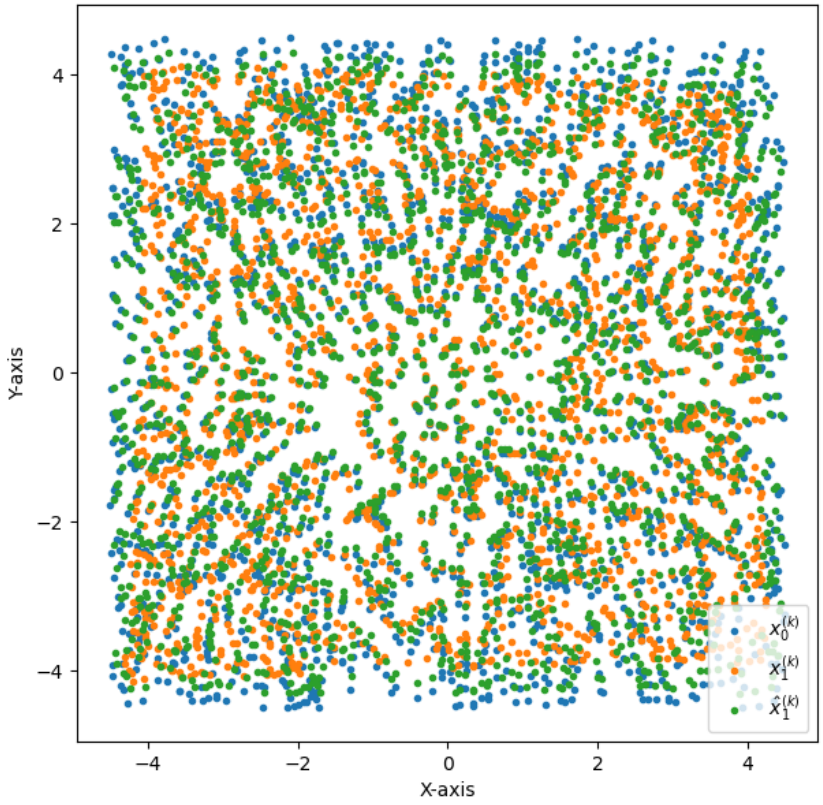


Figure 9: scatter points

Part 3.2: Approximate the vector field using radial basis functions Now, our objective is to approximate the vector field using radial basis functions (RBFs). The task involves determining an optimal number of centers, ranging between 100 and 1000, for the RBFs. The goal is to achieve an approximation such that

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

This approximation aims to capture the underlying dynamics of the process more effectively than linear methods, leveraging the flexibility and adaptability of RBFs.

In this task, we not only attempted the library function `scipy.interpolate.RBFInterpolator` to approximate the vector field but also used our own implementation, `rbf_lstsq`. Their performances are nearly identical. The centers for the radial basis functions are randomly selected from the original data points. The labels of these center points correspond to the vectors2 representing the changes in position.

When using different values of ϵ and varying numbers of center points L, the estimated vector field exhibits variations. To ensure that the approximate endpoints $\hat{x}_1$ are relatively close to the known endpoints x_1, i.e., to find a smaller value of MSE$(\hat{x}_1, x_1)$, we will employ a grid search method to explore values for ϵ and L.

We utilized a heuristic approach to roughly determine the value of ϵ. ϵ was computed as the average distance between all pairs of center points, yielding an approximate value of 5. Consequently, we fixed ϵ at 5 and varied

L from 100 to 1000 with a step size of 100 to obtain the corresponding Mean Squared Error (MSE). The results can be observed in Table 1.

L	MSE
100	0.005498255110276839
200	0.0004429862947538187
300	0.00034369626425355187
400	6.21790385068608e-05
500	1.3091339099161506e-05
600	2.3758776666557353e-06
700	3.2635934980513923e-06
800	1.7102134548916818e-05
900	2.342844970228924e-06
1000	5.269156875542863e-07

Table 1: MSE for different values of L with $\epsilon = 5$

From the table, it can be observed that as L gradually increases, the obtained MSE decreases. Therefore, within the available range, L of 1000 is the most favorable choice.

Next, we fix L at 1000 and further explore the values of ϵ. Using the same method, we conducted a search for ϵ in the range of 0.01 to 200. We observed that the MSE achieves smaller values within the interval of ϵ from 0.2 to 0.5. Since the selection of center points is random, the optimal ϵ may vary. However, within this range, the choice of ϵ has a minimal impact on the approximation results. Therefore, we choose to set ϵ to 0.25.

When L is 1000 and ϵ is 0.25, the resulting MSE is 7×10^{-12}. This is significantly smaller compared to the MSE obtained using a linear operator to estimate the vector field. The estimated $\hat{x}_1$ is plotted together with the original x_0 and x_1 in Figure 10. It can be observed from the graph that all x_1 points are covered by $\hat{x}_1$. This indicates a quite satisfactory result for the approximation.

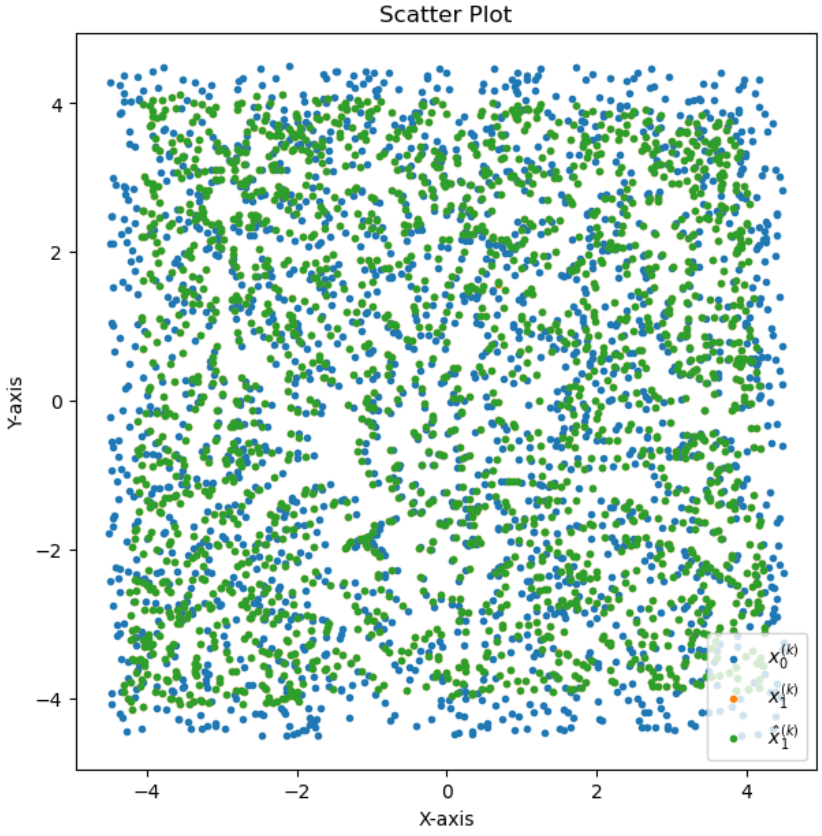


Figure 10: scatter points

In addressing the question of whether the vector field is linear or nonlinear, although there are multiple approaches to make such a determination, this question closely follows the analysis of MSE. Therefore, we will assess whether the vector field is linear or nonlinear from the perspective of MSE.

The results suggest that the vector field is more likely to be nonlinear. The RBF (Radial Basis Function) approximation significantly outperformed the linear approximation, indicating that a linear model is less likely to accurately capture the underlying behavior of the vector field. The superior performance of the RBF approximation implies a higher likelihood of nonlinear relationships within the vector field.

Part 3.3: states of the system After approximating the vector field using radial basis functions (RBF), we can visualize the vector field using the obtained RBF. The specific process is as follows: for the x-coordinate, 50 uniformly distributed points are generated within the range of -4.5 to 4.5; similarly, for the y-coordinate, another set of 50 uniformly distributed points is generated within the same range. In this way, a 2D grid containing 50x50 points is created, evenly distributed within the specified range. These points are then fed into the RBF, resulting in a series of vectors. By plotting these vectors on a 2D graph, we obtain a visualization of the vector field. The visualization is depicted in Figure 11.

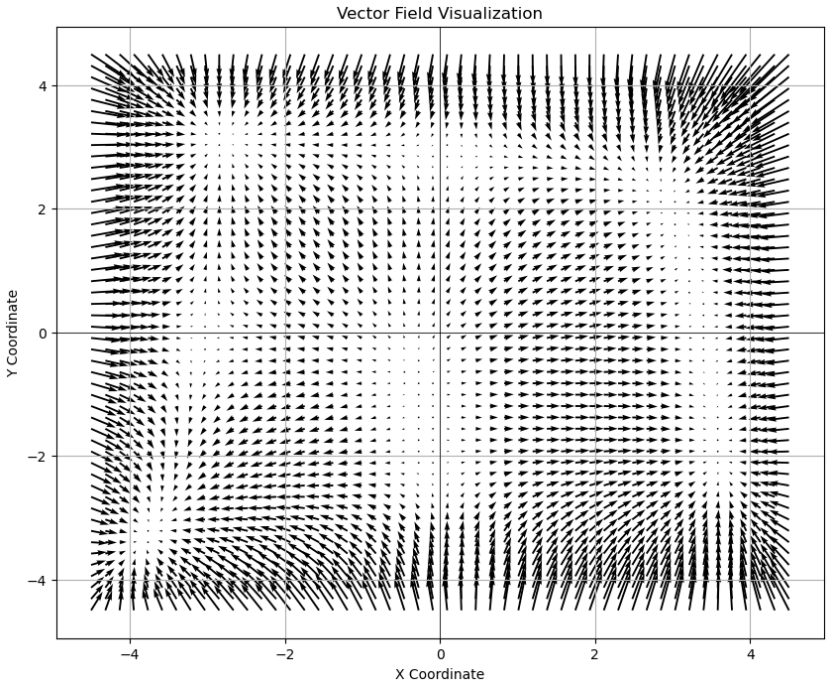


Figure 11: vector field

The steady states of a nonlinear vector field are the points where the vector field becomes zero, indicating that there is no change in the system at those points.

Seeking numerical solutions where the radial basis function (RBF) evaluates to zero can be challenging. However, we can pursue an analytical solution. We generated 1000 uniformly distributed points for both the x and y coordinates within the range of -4.5 to 4.5. This results in a 2D grid containing 1000x1000 points. Subsequently, these points are input into the RBF, and when the resulting function values approach zero vector, we consider these points as steady states. We set the threshold to 8×10^{-4}, indicating that a vector is considered a zero vector when both its x and y components are below this threshold simultaneously. To better visualize the obtained points, we plotted them on the previous image, resulting in Figure 12.

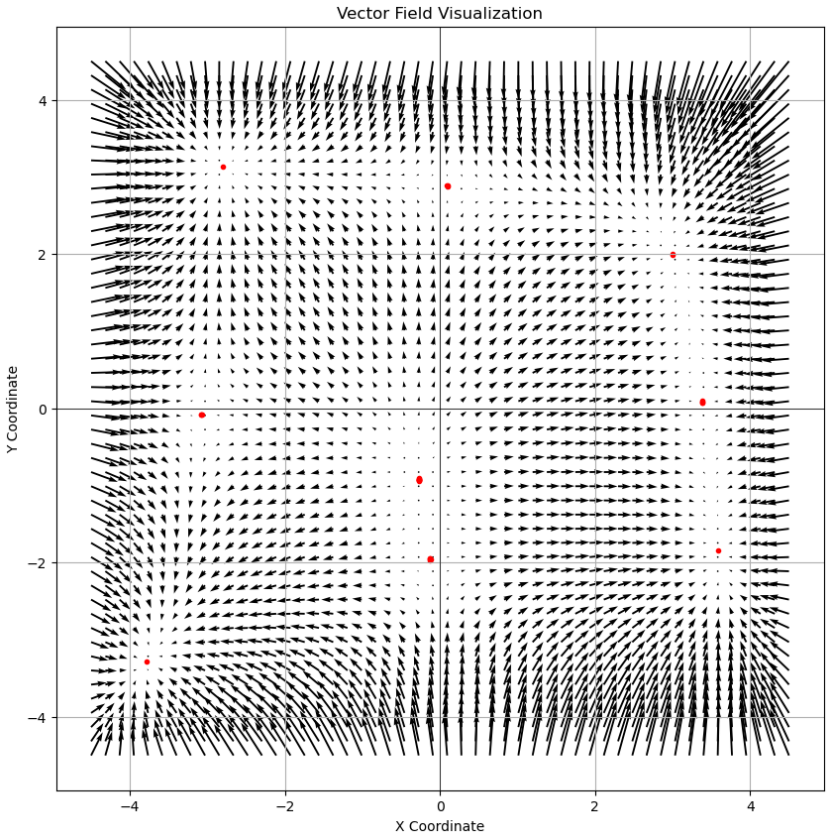


Figure 12: vector field

From the figure, we intuitively observe that the dynamical system has 9 steady states. In fact, among 1,000,000 input points, 28 meet the specified conditions. Some of them are quite close to each other, so it appears as if there are only 9 points. To obtain the states, we first retain one significant digit for the numerical values of these 28 points. Then, we eliminate the duplicates. Finally, we get 9 steady states are as follows:

$$(-3.1, -0.1), (-2.8, 3.1), (-0.3, -0.9), (-0.1, -2.0), (-3.8, -3.3), (0.1, 2.9), (3.0, 2.0), (3.4, 0.1), (3.6, -1.8)$$

In the previous exercise, we discussed the phase portraits exhibited by the linear system under different A. However, this system, with 9 steady states, is completely distinct from any previous linear ones. Additionally, assuming that this system is represented accurately by the obtained radial basis function (RBF), it can be expressed as $\hat{f}_{\text{rbf}}(x(t)) = C\phi(x(t))$. Due to the introduction of the nonlinearity in the function $\phi(\cdot)$, $C\phi(x(t))$ cannot be obtained from $Ax(t)$ through a linear transformation. However, $\hat{f}_{\text{linear}}(x(t)) = Ax(t)$ represents a linear system. Thus, it can be concluded that this system cannot be topologically equivalent to a linear system.

Report on task 4, Time-delay embedding

In this task we will explore embeddings using time-delays, primarily based on Takens theorem. The functions used for this task can be found in `utils.py`, being some of them the Lorenz system's functions used in Exercise 4. The experiments and the results for this task can be found in the notebook `task4.ipynb`.

Part 1 Now, for this part of the task we will load the `takens_1.txt` dataset and will work with it. This dataset contains a matrix $X \in \mathbb{R}^{1000\times 2}$ where the two columns of the matrix are the two coordinates of a closed, one-dimensional manifold. The visualization of the raw data can be observed in Figure 13.

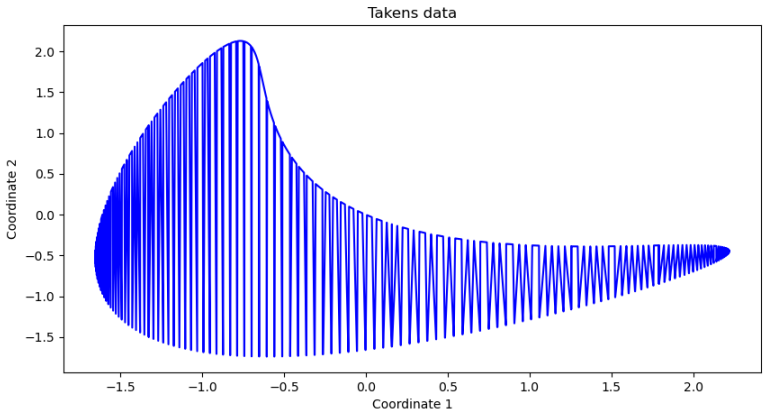
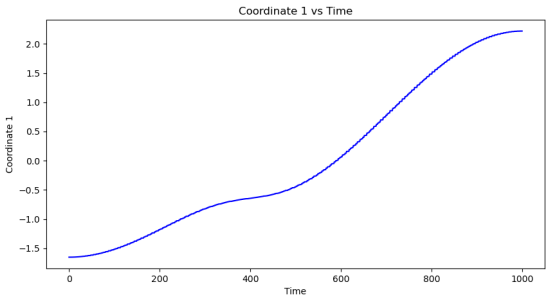
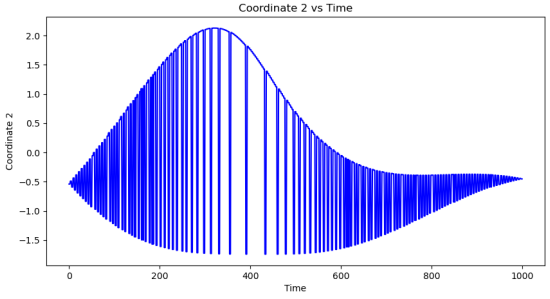


Figure 13: Plotted dataset

Supposing that each row number represents "time," we can plot the first and second coordinates of the data against time (Figure 14).



(a) Coordinate 1 vs Time

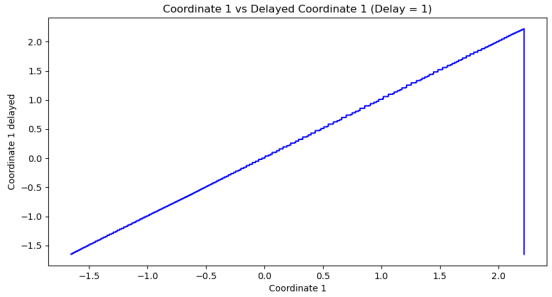


(b) Coordinate 2 vs Time

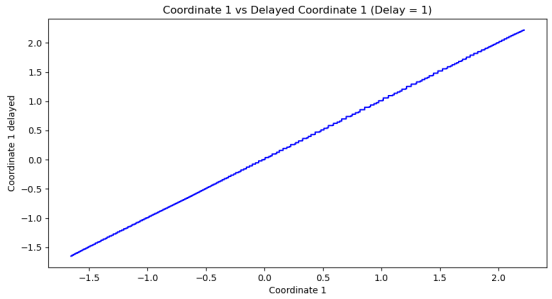
Figure 14: Coordinates of the dataset against the row number (time)

However, in this task, we will focus on the first coordinate, i.e., Figure 14(a). Based on the Takens theorem, we will test some delays Δt, equivalent to Δn rows in this case, to create an approximated version of an embedded representation of the original time series in a higher-dimensional space.

To generate delayed vectors, we employ the `numpy.roll` function, facilitating the shift of array elements along a specified axis. This function seamlessly reintroduces elements that surpass the last position back to the initial position. Consequently, employing a roll of -1, equivalent to a delay of 1 row, results in the initial row of the original array occupying the final position in the delayed array. This behavior manifests as a sudden drop at the plot's conclusion, as illustrated in Figure 15(a). To enhance the plot, the decision is made to exclude the last data points, as the information loss is negligible. This adjustment yields the improved plot depicted in Figure 15(b). Additionally, to eliminate this drop from the plot for any value of *delay*, it is necessary to skip the last *delay* data points, where $delay \in \mathbb{N}$.



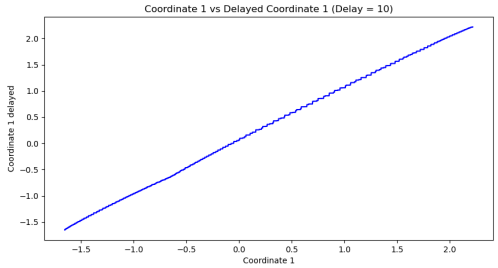
(a) Plotting the last point



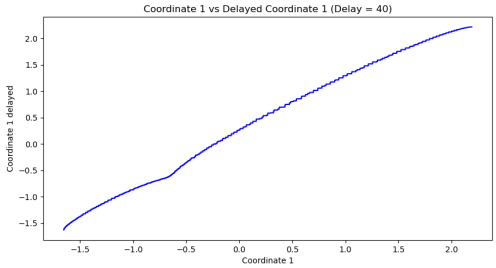
(b) Not plotting the last point

Figure 15: Coordinate 1 against its delayed version (delay = 1)

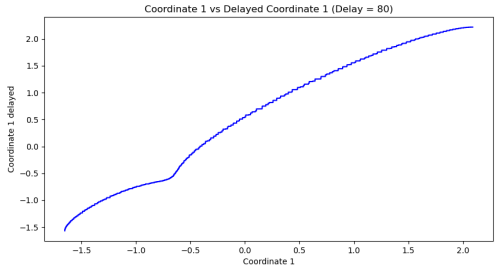
We can observe in Figure 15(b) an increasing straight line, that suggests a strong linear correlation between each data point and its immediate successor in the time series. This pattern might indicate that the first coordinate is changing steadily with each subsequent data point. However, this line does not reveal too much of the underlying behaviour of the original first coordinate. We can try to increase the Δn to test how the increment of the delay changes the captured patterns in the system.



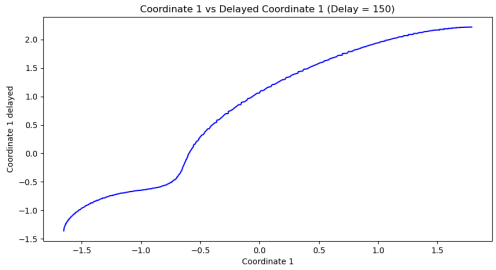
(a) Delay = 10



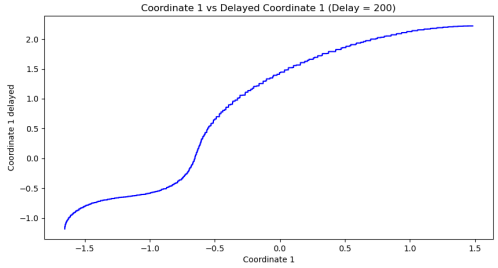
(b) Delay = 40



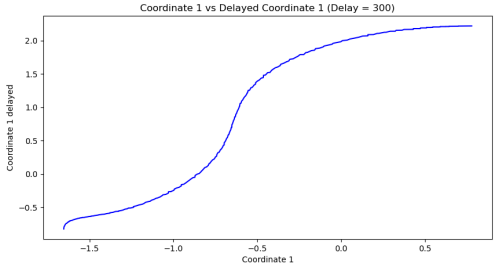
(c) Delay = 80



(d) Delay = 150



(e) Delay = 200



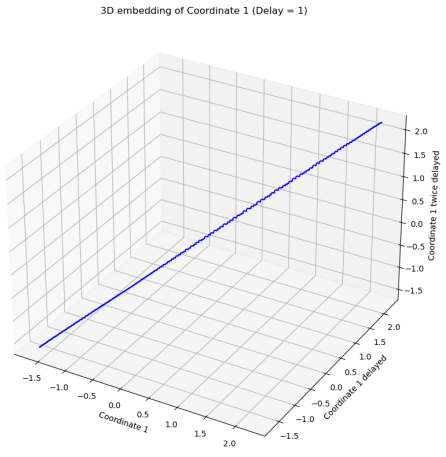
(f) Delay = 300

Figure 16: Coordinate 1 against its delayed version for different delay values

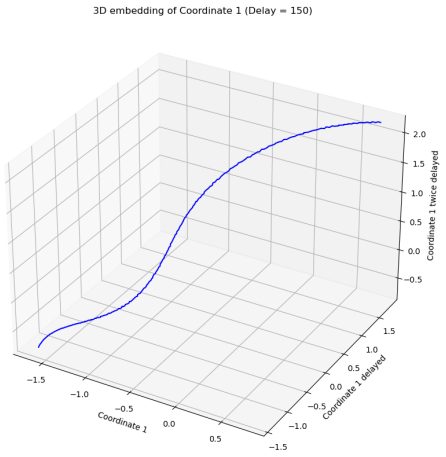
Now, we analyze the plots generated for different delay values. In Figure 16(a), where the delay is set to 10, the resulting graph maintains a straight line. Moving to delays of 40 and 80 (Figures 16(b), 16(c) and 16(d)), curvature becomes evident, especially at the beginning. This observation suggests that these delays are more effective in capturing certain nonlinear features of the system, and the overall structure of the original first coordinate plot is adequately approximated, primarily for $delay = 150$. However, delays of 200 and 300 (Figures 16(e) and 16(f)) introduce even more pronounced curvature. While this might capture additional complexities, it deviates from the original first coordinate plot, potentially introducing noise to the embedding. Additionally, using delays of 200 and 300 results in the loss of information as they involve plotting fewer points.

Takens' theorem states that we need $2d + 1$ dimensions to faithfully embed the dynamics of a dynamical system with an intrinsic dimension d [2]. In other words, to capture the essential features and reconstruct the original system, the embedding space should have a dimension equal to or greater than $2d + 1$, but it could be less in some scenarios [1].

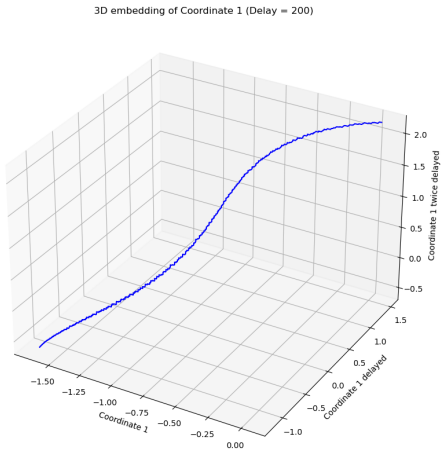
Therefore, in our case, being the dimension of the manifold $d = 1$, we would need 3 dimensions to be sure that we embed the manifold correctly, making the dynamics of the system deterministic according to the theory. Thus, we have experimented using an embedding of 3 dimensions. We use the delay vector of the previous part as second coordinate and another delay vector which doubles the delay of the second coordinate, as the third coordinate.



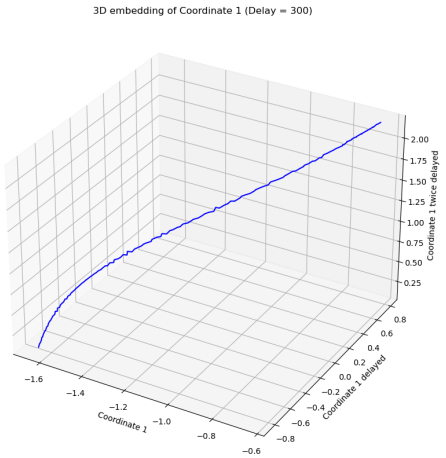
(a) Delay = 1



(b) Delay = 150



(c) Delay = 200

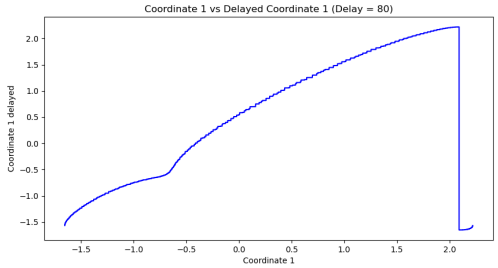


(d) Delay = 300

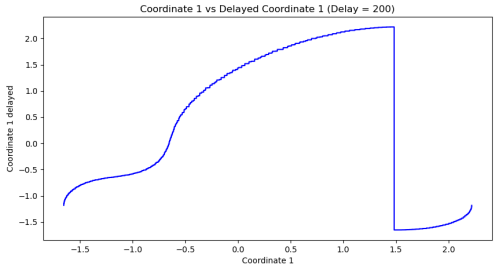
Figure 17: 3D embedding of the system

We can observe now how with $delay = 1$ the obtained representation is still a straight line (Figure 17(a)). However, in Figures 17(b) and 17(c) a more faithful approximation of the original structure is achieved. Finally, for a bigger delay in Figure 17(d) the representation becomes less accurate due to the potential introduction of noise and miss of information, demonstrating the importance of selecting an appropriate delay parameter for the Takens embedding process.

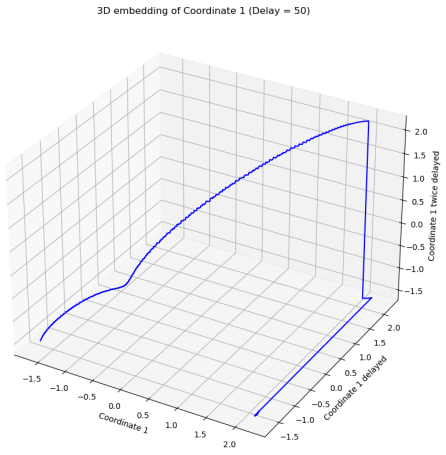
Finally, for the sake of completeness we include additional figures that incorporate the last delay rows (Figure 18). This enables us to better understand the approximation and test our assumption that excluding these rows from the plot does not significantly compromise the graphical representation of the overall structure, even with high delay values, as observed in Figure 18(b). For 3D plots the graphical representation is worsened and also does not provide much information about the overall structure as can be seen in Figure 18(c). In the next part, we will observe that, for more complex systems with greater amounts of data, omitting these rows becomes less critical for visualization, as their proportion in the plot is relatively smaller.



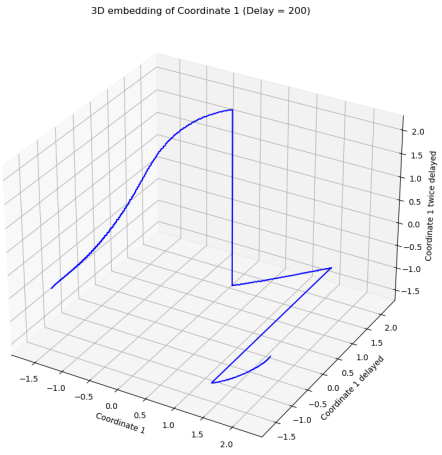
(a) Delay = 80



(b) Delay = 200



(c) Delay = 50 3D Plot



(d) Delay = 200 3D Plot

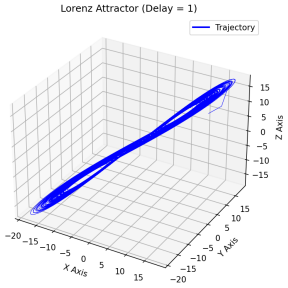
Figure 18: Plots without skipping the last *delay* rows

Part 2 In this task, our objective is to approximate the chaotic dynamics of the Lorenz attractor using the Takens theorem. Since we have previously explored the Lorenz system in Exercise 4, we will skip providing an introduction and explanation of this system in the current task.

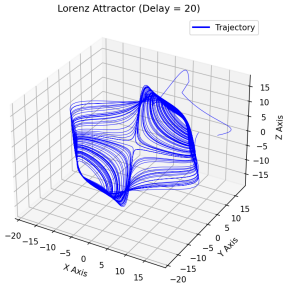
Assuming we only have access to the x-coordinate of the Lorenz system, our goal is to approximate the real system using a 3-dimensional space. According to the Takens theorem, this space enables us to obtain a reasonable understanding of the attractor's shape. The three dimensions in our space are represented by $x_1 = x(t)$, $x_2 = x(t + \Delta t)$, and $x_3 = x(t + 2\Delta t)$, where t is a value greater than 0 that we need to choose.

Unlike the previous part, which was more exploratory, the aim here is to find a good approximation of the system given a 3-dimensional space. Therefore, to select Δt, we have implemented a function `best_delay` inside `utils.py`, instead of relying on visual inspection as done previously. This function determines the optimal Δt value that best approximates the system. The search for this optimal value is conducted within a specified range, limited to computational efficiency considerations (ranging from 1 to 100). The selection criterion is based on minimizing the distance between each point of the original trajectory and its corresponding point on the approximated trajectory, thus the closest approximation will be selected.

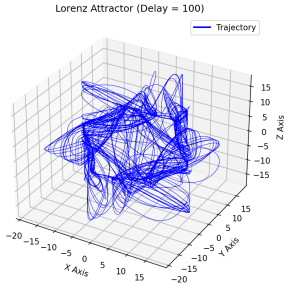
If we opt for a random selection of the delay Δt without using the selection algorithm, the results exhibit significant instability compared to the previous approach. This instability is likely attributed to the chaotic nature inherent in the system. In Figure 19(a), the original structure is recognizable but appears flattened. However, in Figures 19(b) and 19(c), the structure diverges considerably from the original. This observation aligns with our earlier findings in the previous exercise, where we found that the Lorenz attractor, under the given parameters, experiences a chaotic explosion in the distance between the original trajectory and a slightly perturbed one around $t = 20$. Thus, attempting to approximate the system using a completely chaotic delayed version may result in an outcome that does not represent the correct trajectory.



(a) Delay = 1



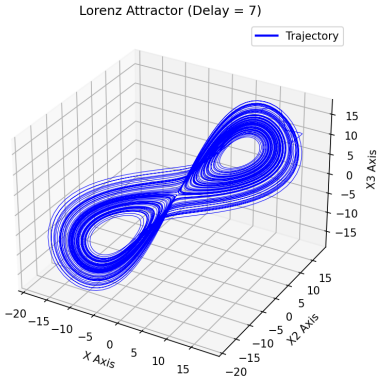
(b) Delay = 20



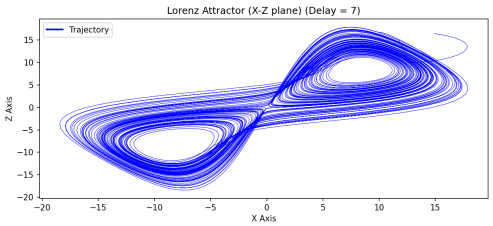
(c) Delay = 200

Figure 19: Effect of delay in the Lorenz approximation

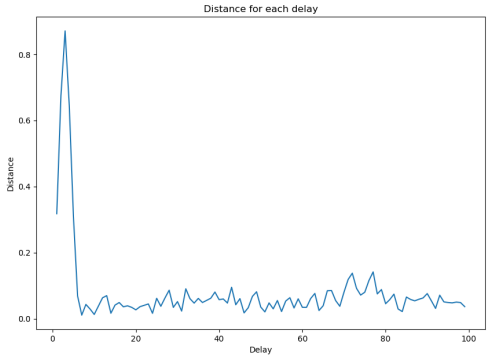
Using the algorithm to select the best delay value we get $\Delta t = 7$. This aligns with our earlier hypothesis, as $7 < 20$ and $2\Delta t = 14 < 20$. Therefore, none of the delayed vectors are yet chaotic with respect to the original vector. As we can see in Figure 20 the shape of the attractor can be approximated in a correct way, allowing us to get an intuition of the system behaviour only using the x-coordinate.



(a) 3D plot



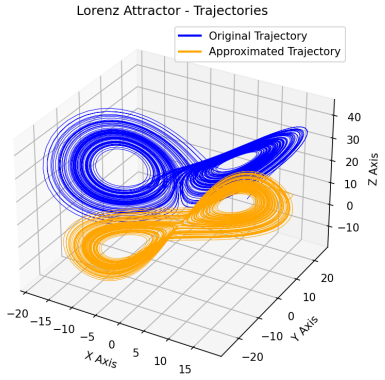
(b) XZ planes plot



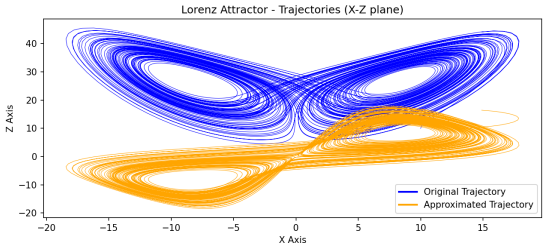
(c) Distance for each value of Δt to the original trajectory

Figure 20: Lorenz approximation

An intriguing observation is that the algorithm computes a similar distance for values of Δt such as 20 or 100 (Figure 20(c), even though we observed that their shapes differ significantly from the real trajectory. This phenomenon may be attributed to the inherent chaotic nature of the system as well as the distance being composed of small peaks and valleys rather than a straight, linear relationship. Also, already in the previous Exercise we observed that, in spite of two Lorenz systems being very similar in structure, their trajectories differed greatly when the distance between them was measured. Therefore, the euclidean distance may not be the best method to obtain the delay overall.



(a) 3D plot



(b) XZ planes plot

Figure 21: Comparison between the original trajectory and the approximation

In Figure 21, we can compare and conclude that the shape of the original system can be reasonably represented using only the x-coordinate and the delayed vectors. However, while the overall shape appears similar, a closer inspection in Figure 21(a) reveals distinct differences in curvature and folding. Additionally, in Figure 21(b), we notice that the approximation is positioned at a lower value, and its shape is not completely symmetrical with respect to the line $X = 0$ as seen in the original shape.

In Part 2 of this task, it's important to note that we haven't excluded the last *delay* rows when plotting. Now, we have much more data and therefore the plot is barely affected by it, as we can see in Figure 22. The only thing that changes compared to Figure 20(b) in the plot is a little trajectory in the upper-right corner.

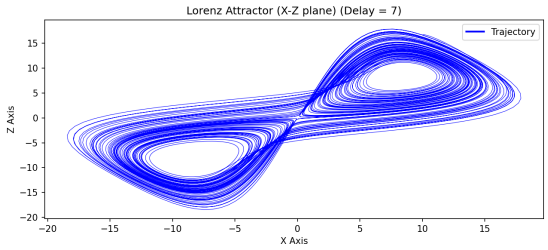
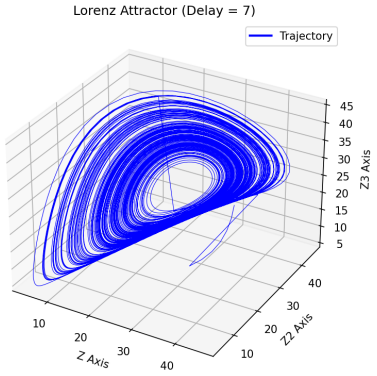
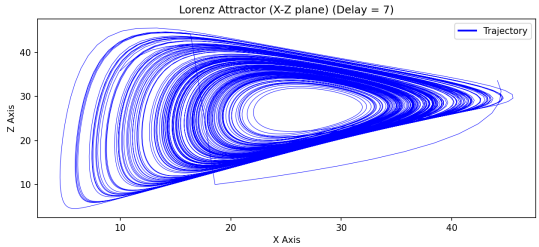


Figure 22: Approximation of Lorenz system skipping the last *delay* rows

Finally, we can try to approximate the Lorenz system using only the z-coordinate now, applying the same method as before. However, we can note in the results obtained in Figure 23 that the approximation does not resemble the original shape.



(a) 3D plot



(b) XZ planes plot

Figure 23: Approximation of the Lorenz system using z-coordinate

One possible explanation for this phenomenon is that the x and y coordinates may be more informative than the z coordinate. Consequently, attempting to reconstruct the system using the z coordinate becomes more difficult. On the other hand, the parameters $\rho = 28$ and $\beta = 8/3$ may play a role, particularly in their impact on z through $\rho - z$ and βz, making z less influential in the system.

Nevertheless, we can try to go even deeper, first reminding the system of equations that rule the Lorenz attractor:

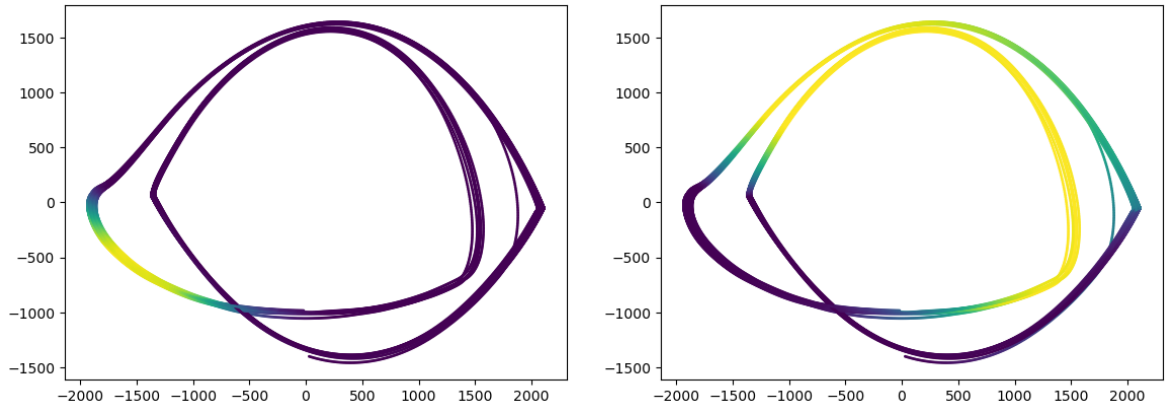
$$\begin{aligned} \frac{dx}{dt} &= \sigma(y - x) \\ \frac{dy}{dt} &= x(\rho - z) - y \\ \frac{dz}{dt} &= xy - \beta z \end{aligned}$$

Analyzing the system we can note that z is determined by xy minus a multiple of z. However, we can notice that z is indeed loosing information, as its value is the same when we have $(x(t), y(t))$ and $(-x(t), -y(t))$, as in the z formula both minus signs are cancelled. Therefore, we can assert that z is less informative than x and y due to its lack of sensitivity to the sign of the solution. This lack of uniqueness in the mapping of z may prevent a clear reconstruction of the original solution when only the z-coordinate is considered. This assertion finds backing in a visual criterion: the z representation closely resembles one of the symmetric halves that form the Lorenz attractor. Then, we can hypothesize that the results of Figure 23 might be illustrating a collapsed solution between the original trajectory and the sign-reversed trajectory, of which z cannot distinguish and therefore, properly represent.

Report on task 5, Learning crowd dynamics

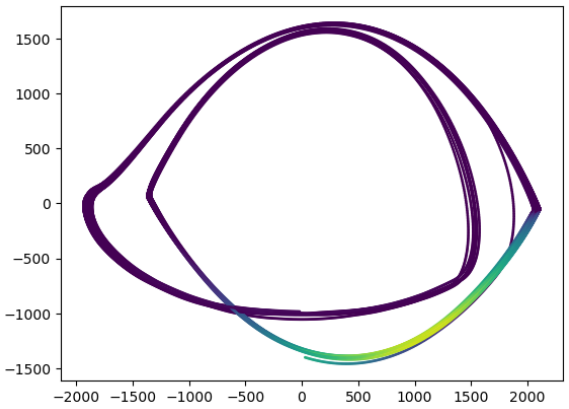
First we create the dataset by using delay embeddings. We use 350 delays in order to make sure we get even distant relations. If the data forms a 1d loop, a 1d manifold, Taken's theorem says we should be able to reduce the dimensionality at least to $2 \times 1 + 1 = 3$ dimensions. After applying PCA we can see this is truly possible, 2 dimensions suffice.

When we scatter-plot the PCA data we can see that it indeed forms a 1d loop 24. When we color the points by area utilization we can see that each building area has very specific parts of the curves where it's usage peaks. This shows us that we can use the curve's traversal to estimate area utilization 24 25 26.



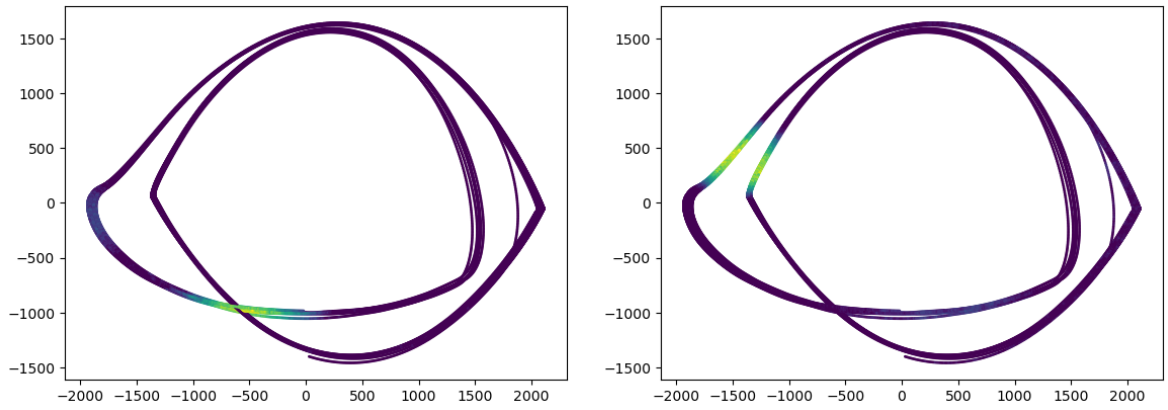
(a) area 1

(b) area 2



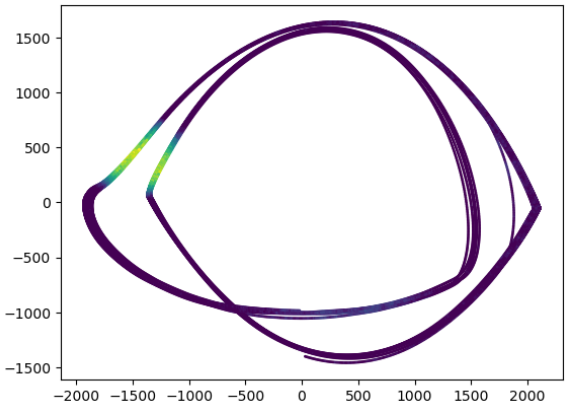
(c) area 3

Figure 24: The area occupations shown on the PCA'd data part 1



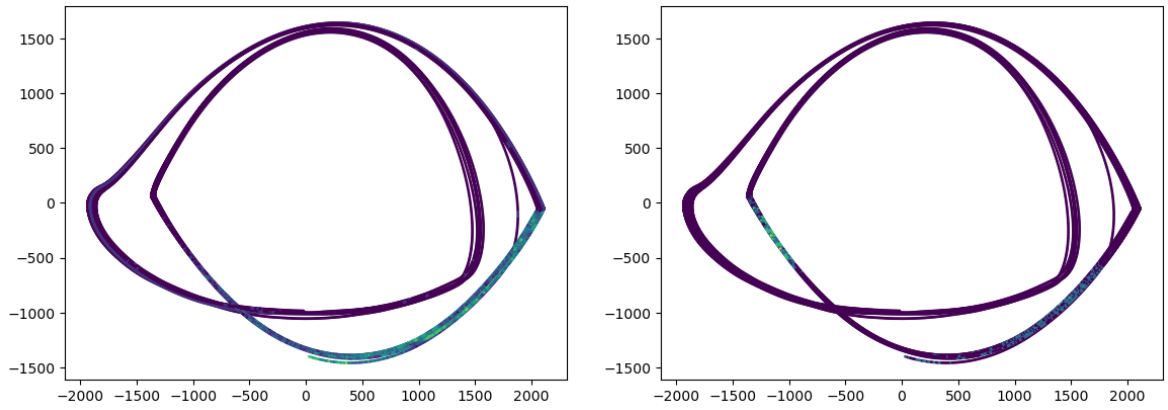
(a) area 4

(b) area 5



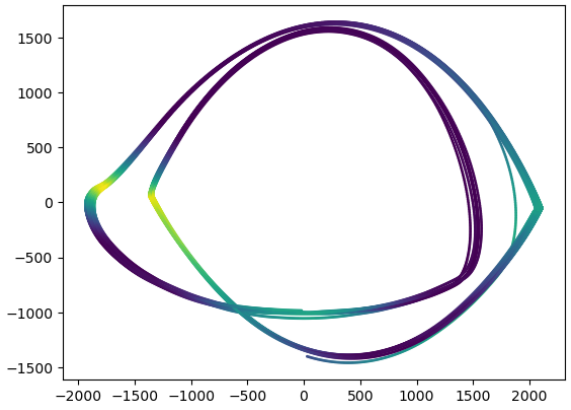
(c) area 6

Figure 25: The area occupations shown on the PCA'd data part 2



(a) area 7

(b) area 8



(c) area 9

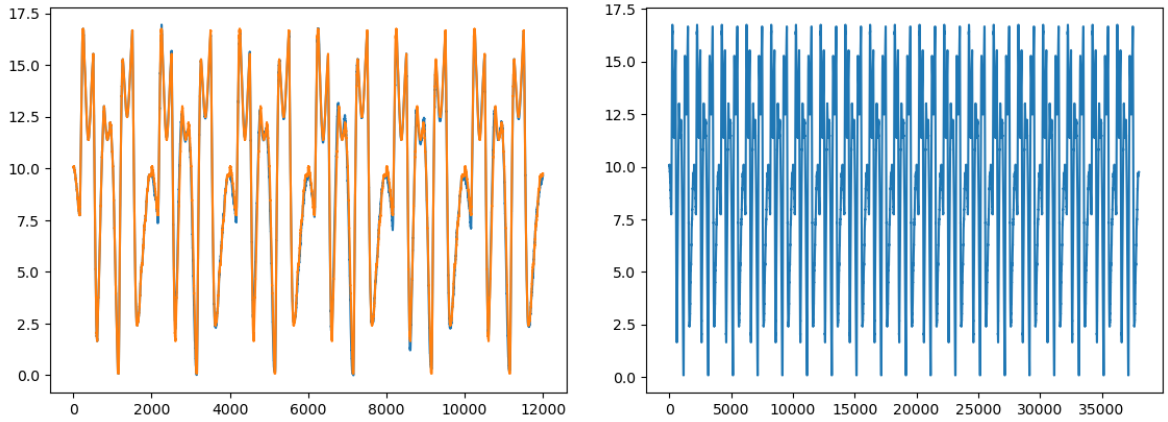
Figure 26: The area occupations shown on the PCA'd data part 3

Calculating the arclength deltas can be done simply by calculating the distance between subsequent embeddings and dividing by the time distance between these embeddings (this is always one since the data is continuous). Calculating arclength traversed can be done by taking a cumulative sum of arclength deltas.

Most machine learning approaches require the data to lie in one specific interval. This is especially true for radial basis splines where each spline only covers a specific interval with non-negligeble values. If we are to predict into a before unknown range of time we therefore need to somehow take the data and represent it in a time independent manner.

The easiest way to do this is to learn a period l of the data by sampling and trying different values. The easiest way to determine this period is to define a function r that takes the first l items and repeats this subsequence to create a sequence as long as the training input. Then we can take the $l2$ loss between the target sequence and sequences parametrized by l and take the l parameter with the lowest loss.

This immediately gives us a relatively good approximation of the dependency between the arc-length delta and the time 27.



(a) Fitting the original function blue is the original orange is the fit

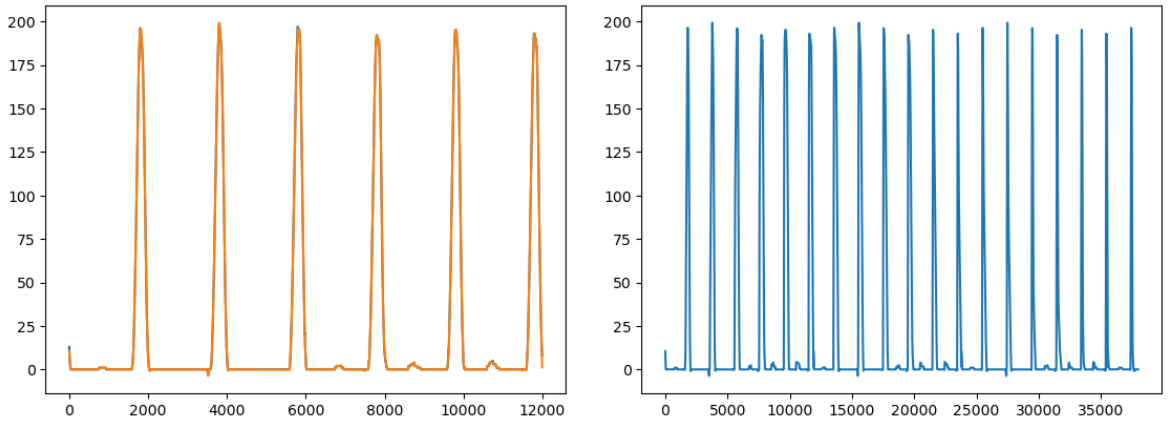
(b) Predicting into the future

Figure 27: Plots of approximating the arclen derivative

We could directly use this to learn the dependency between the arc-length delta and the first measurement area utilisation delta. This is a bad idea because in a cumulative sum even small errors accumulate over time.

The solution we chose in the end was to use the arc-length traversed as the data and the first measurement area utilisation as labels. To calculate the test data beyond the measured area we use the aforementioned time and arc-length delta dependency and cumulative sum the arc-length delta to get the arc-length traversed 28. In order to remove the before mentioned issue with the data intervals we modulo the test data with the largest input data-point.

We used the radial basis function linear regression to get the predictor. The hyper-parameters were chosen by grid-search. Values of L were from range $[2, 5]$ and ϵ from $\{1 + 200 \times x | x \in [1, 24]\}$. Again the $l2$ distance between sequences was used as the loss. The search didn't seem to help too much as when the loss is multiplied by -1 (ie. taking the maximum loss rather than the minimal one) the result does not significantly change.



(a) Fitting the original function blue is the original orange is the fit

(b) Predicting into the future

Figure 28: Plots of approximating area 1

References

[1] Krzysztof Barański, Yonatan Gutman, and Adam Śpiewak. A probabilistic takens theorem. *Nonlinearity*, 33(9):4940, 2020.

[2] Cosma Rohilla Shalizi. Methods and techniques of complex systems science: An overview. *Complex systems science in biomedicine*, pages 33–114, 2006.