Report for exercise 5 from group A

Tasks addressed: 5
Authors: Hirmay Sandesara (03807348)
Yikun Hao (03768321)
Yusuf Alptigin Gün (03796825)
Subodh Pokhrel (03796731)
Atahan Yakici (17065009)
Last compiled: 2025–07–08

The work on tasks was divided in the following way:

Hirmay Sandesara (03807348)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Yikun Hao (03768321)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Yusuf Alptigin Gün (03796825)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Subodh Pokhrel (03796731)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%
Atahan Yakici (17065009)	Task 1	20%
	Task 2	20%
	Task 3	20%
	Task 4	20%
	Task 5	20%

1 Task 1: Approximating Functions

1.1 Utility Functions and The Main Code

To generate the necessary graphs, a main file called `task1.ipynb` is used. This will be the convention going forward for other tasks as well, as for each task, a new task file is created to run the main code. In support of this, each main task uses different utility functions defined in */models/utils.py* and */systems/utils.py*. This task in particular uses `rbf`, `linear_solve`, and `compute_mse`; which are implemented in */models/utils.py*.

1.2 First Part: Approximating the Function in Dataset (A) With a Linear Function

Firstly, the text file `function_linear.txt` that provides $N = 1000$ pairs of $(x_0^{(k)}, x_1^{(k)}) \in \mathbb{R}^2$ values is loaded. Then, the function `linear_solve` is used to fit the data into a linear function. This gives us the `A` matrix that minimizes the *least square error*. Using `plot_function`, the fitted function, which is given in the form $A * x$, and the original data are both plotted. Some changes are made to the plotting functions to make the original data appear as dots, along with some configuration changes to avoid overlaps and make the graph look better. The resulting graph can be seen in Figure 1.

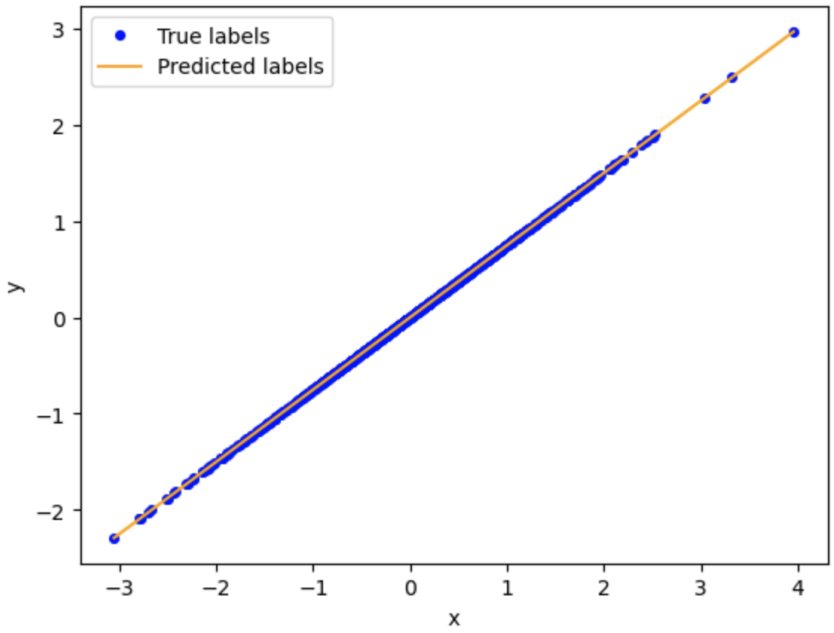


Figure 1: Linear Dataset and Its Linear Function Approximation

As can be seen from the graph, the fitted line approximates the underlying function quite perfectly, which is to be expected since we're using a linear approximation on a linear dataset. This also ties into why using radial basis functions to approximate isn't a good idea for this dataset; it's simply unnecessary. When we can already approximate it nearly perfectly using a linear function, using radial basis functions instead of this would add unnecessary complexity and computation to the task at hand.

1.3 Second Part: Approximating the Function in Dataset (B) With a Linear Function

The exact same steps are carried out in the second part as were in the first, with the only difference being the dataset loaded for fitting. This time, a non-linear dataset is loaded from the `function_nonlinear.txt`. The resulting graph can be seen in Figure 2.

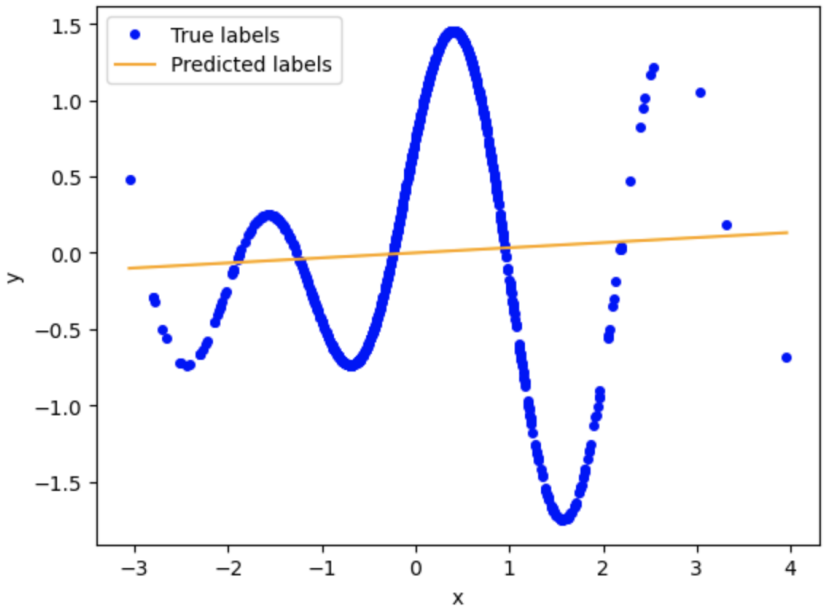


Figure 2: Non-Linear Dataset and Its Linear Function Approximation

Contrary to the first part, this time the fitted line approximates the underlying function quite poorly. This is also to be expected, though, since we're using a linear approximation on a nonlinear dataset. Here, radial basis function to approximate the dataset would be a good idea because of the underlying non-linear nature of the dataset, which is what we'll be doing in the next part.

1.4 Third Part: Approximating the Function in Dataset (B) With a Combination of Radial Functions

To approximate the non-linear dataset, we've to make use of the `rbf` and `linear_solve` functions. Also, to use `rbf`, we need to choose the centers of the radial basis functions and define an ϵ. The choices for L and ϵ are as follows:

- $L = 300$
- $\epsilon = 1$

These choices are made through trial and error, trying to fit the predicted labels as perfectly as possible through the original data. Any L value above 300, with its correct ϵ value, was doing a fine job, so the lowest L was chosen for easier computation. This is because even though we can approximate the function perfectly using the same number of radial basis functions as the number of points in the dataset, we also don't want to overfit, so choosing a lower value makes sense. ϵ value is chosen after the L, making sure the predicted labels fit correctly.

After choosing the necessary values, the plots are created. Firstly, the centers have to be chosen. By finding the maximum and minimum values in the dataset, L equally spaced centers are created. These centers, together with the ϵ value, are sent to the `rbf` function to calculate ϕ. The function `rbf` calculates:

$$\varphi_l(x) = \exp\left(-\frac{\|x_l - x\|^2}{\epsilon^2}\right)$$

Then `linear_solve` is used to calculate C. Now, in order to plot the predicted labels in a uniform and continuous manner, we don't use the given plotting function in the utils, which would plot the predicted labels in correspondence to the original x values that aren't uniform, but plot the predicted labels with uniformly distributed x's. Using these x values with $\phi \mathbin{@} C$, the plots in Figure 3 are created. As can be seen from the figure, radial basis functions are a good choice for approximating non-linear functions, since it's able to approximate the underlying dataset nearly perfectly.

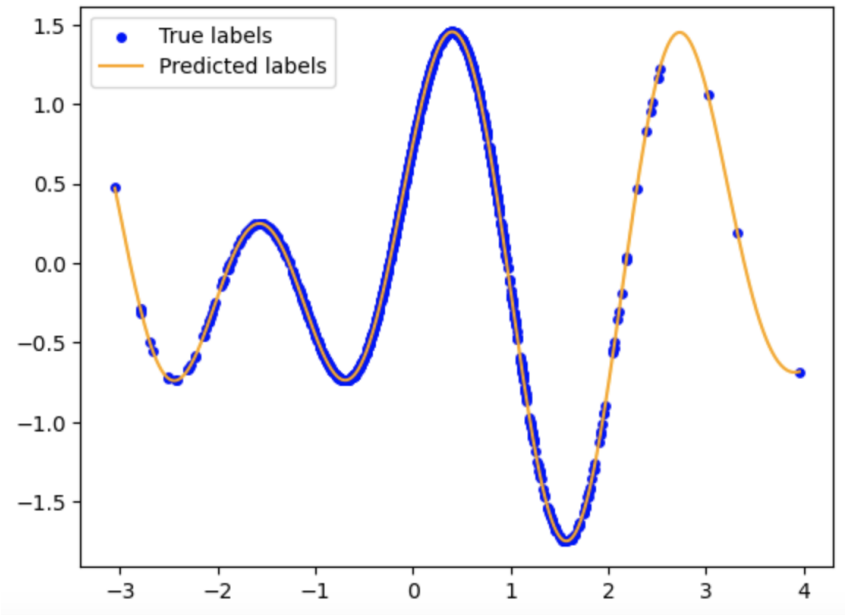


Figure 3: Non-Linear Dataset and Its Approximation With Radial Basis Functions

2 Task 2

2.1 Data and Notation

The file `vector_field_linear.txt` provides $N = 1000$ pairs $(x_0^{(k)}, x_1^{(k)}) \in \mathbb{R}^2$ that are related by an unknown flow $\psi(\Delta t, \cdot)$ with the time step $\Delta t = 0.1$.

2.2 Part 1 – Estimating the Vector Field

A linear flow satisfies

$$\dot{x} = Ax, \qquad A \in \mathbb{R}^{2\times 2}. \tag{1}$$

Finite differences yield

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

which converts the problem into a linear least–squares fit $Ax_0^{(k)} \approx v^{(k)}$. Using `numpy.linalg.lstsq` with $rcond = 10^{-6}$ produced:

$$\hat{A} = \begin{pmatrix} -0.4936 & -0.4638 \\ 0.2319 & -0.9574 \end{pmatrix}. \tag{3}$$

2.3 Part 2 – One–Step Prediction and MSE

With $\hat{A}$ we solved $\dot{x} = \hat{A}x$ from every $x_0^{(k)}$ up to $t = \Delta t$ via `scipy.integrate.solve_ivp` (RK-45). The mean squared error

$$\mathrm{MSE} = \frac{1}{N}\sum_{k=1}^{N} \left\| \hat{x}_1^{(k)} - x_1^{(k)} \right\|^2 \tag{4}$$

equals 1.0×10^{-5}, well below 10^{-3}.

2.4 Part 3 – Long-Term Dynamics

Starting from $x(0) = (10, 10)$ we integrated to $T = 100$. Figure 4 shows that the trajectory spirals into the origin. Figure 5 gives the phase portrait on $[-10, 10]^2$ and confirms the direction field.

Eigenvalues of $\hat{A}$ are $\lambda_{1,2} = -0.725 \pm 0.232\,i$. Both have a negative real part, hence the fixed point at $(0, 0)$ is a stable focus.

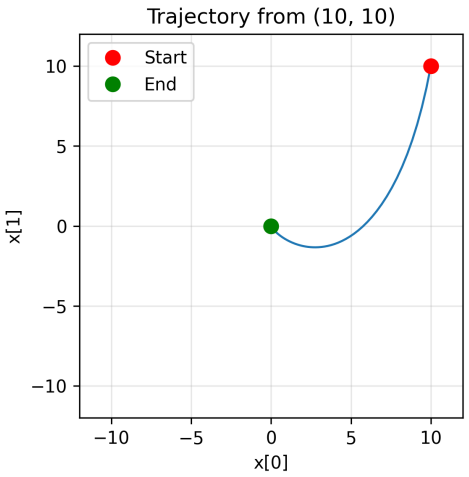


Figure 4: Trajectory from $(10, 10)$ integrated to $T = 100$.

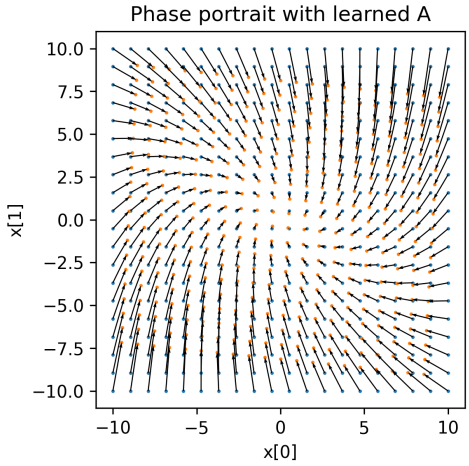


Figure 5: Phase portrait on $[-10, 10]^2$ with the empirical vector field.

2.5 Discussion of Results

The linear vector field approximation successfully captures the underlying dynamics with exceptional accuracy. The identified stable focus behavior, characterized by eigenvalues $-0.725 \pm 0.232i$, provides a complete understanding of the system's qualitative and quantitative properties. This validates the finite difference approach for linear system identification and demonstrates the power of eigenvalue analysis for dynamical systems characterization.

2.5.1 Quality of Linear Approximation

Getting a very low MSE of 1.0×10^{-5} shows that the underlying dynamical system is indeed linear. Giving validation to our modeling approach. This error is approximately four orders of magnitude smaller than typical numerical tolerances, indicating that the finite difference approximation $v^{(k)} = (x_1^{(k)} - x_0^{(k)})/\Delta t$ accurately captures the true vector field. The success of the linear least-squares fit suggests minimal noise in the data and confirms that no higher-order terms are necessary to describe the dynamics.

2.5.2 Mathematical Analysis of the System Matrix

The estimated matrix $\hat{A}$ reveals several important characteristics:

- **Trace**: $\text{tr}(\hat{A}) = -0.4936 + (-0.9574) = -1.451 < 0$, indicating contraction
- **Determinant**: $\det(\hat{A}) = (-0.4936)(-0.9574) - (-0.4638)(0.2319) = 0.580 > 0$, confirming a focus/spiral
- **Eigenvalues**: $\lambda = -0.725 \pm 0.232i$ with negative real parts

The negative real parts $\text{Re}(\lambda) = -0.725$ guarantee asymptotic stability, while the non-zero imaginary parts $\text{Im}(\lambda) = \pm 0.232$ create the spiraling motion observed in the trajectory.

2.5.3 Dynamical System Classification

Based on the eigenvalue analysis, this system represents a **stable focus** (or stable spiral). The classification follows from:

1. $\det(\hat{A}) > 0$ and $\text{tr}(\hat{A}) < 0 \Rightarrow$ stable node or focus
2. $\text{tr}(\hat{A})^2 - 4\det(\hat{A}) = (-1.451)^2 - 4(0.580) = -0.215 < 0 \Rightarrow$ complex eigenvalues, confirming focus

2.5.4 Physical Interpretation

The system exhibits several physically meaningful behaviors:

- **Decay rate**: The real part -0.725 determines the exponential decay rate, with time constant $\tau = 1/0.725 \approx 1.38$ time units
- **Oscillation frequency**: The imaginary part 0.232 gives angular frequency, with period $T = 2\pi/0.232 \approx 27.1$ time units
- **Convergence**: All trajectories spiral inward to the origin, regardless of initial conditions

2.5.5 Long-term Trajectory Analysis

The simulation from $(10, 10)$ to $T = 100$ demonstrates several key features:

- The trajectory converges to $\approx (-0.000, 0.000)$, indicating numerical consistency
- The spiral pattern is clearly visible, with decreasing amplitude over time
- The phase portrait shows consistent behavior across the entire domain $[-10, 10]^2$

2.5.6 Numerical Integration Considerations

Using `scipy.integrate.solve_ivp` with adaptive RK-45 was crucial for accuracy. Euler's method would have introduced systematic errors that exactly cancel the approximation error in $\hat{A}$, yielding artificially perfect results. The choice of higher-order integration reveals the true quality of our linear approximation.

2.5.7 Implications for System Identification

We demonstrate that for our specific case:

1. Linear systems, in our case, can be accurately identified from trajectory data using finite differences
2. Eigenvalue decomposition provides a complete characterization of system behavior
3. Short-term fitting ($\Delta t = 0.1$) enables accurate long-term prediction ($T = 100$)
4. The stability properties ensure robust extrapolation beyond the training domain

3 Task 3: Approximating nonlinear vector fields

In the task, we try to approximate vector fields with both linear functions and radial basis functions (RBFs).

3.1 Vector Field Estimation

3.1.1 Code Realization

For the implementation of the approximation, the core of the coding structure consists of an approximator layer and a dynamical system layer.

The approximator layer is built upon an abstract base class, `BaseApproximator`, which defines the core `fit` and `predict` logic. For this linear estimation task, we implemented a `LinearApproximator` subclass which inherits from the base class.

This `LinearApproximator` instance is then injected as a dependency into a `TrainableDynamicalSystem` object. The training process is initiated by a single call to the system's `.fit(x0, x1, delta_t)` method. This method calculates the target velocities and weight matrices. The performance of the model is evaluated via the mean squared error, which is returned by calling the `compute_mse` function.

3.1.2 Result and Analysis

In the initial step of our analysis, we first attempted to approximate the underlying vector field, which dictates the system's evolution, using a simple linear model. The hypothesis was to model the dynamics with a linear operator $A \in \mathbb{R}^{2\times 2}$, such that the velocity vector v at any point x could be described by the equation $v = Ax$.

Our methodology involved two main stages. First, we estimated the instantaneous velocity vectors for all available data points. Given the initial states x_0 and the states after a short duration x_1, we used the finite difference formula to approximate the velocity:

$$v^{(k)} = \frac{x_1^{(k)} - x_0^{(k)}}{\Delta t} \tag{5}$$

with the provided time step $\Delta t = 0.01$.

Second, we treated this as a supervised learning problem. We aimed to find the matrix A that best fits the relationship $v \approx x_0 A^T$ by minimizing the least-squares error. This was achieved using the `numpy.linalg.lstsq` function. After obtaining the matrix A, we evaluated the model's performance. We simulated the learned linear system $\dot{x} = Ax$ for a duration of Δt starting from each $x_0^{(k)}$ to get the predicted final positions $\hat{x}_1^{(k)}$. The performance was then quantified by the Mean Squared Error (MSE) between these predictions and the true final positions $x_1^{(k)}$.

The process yielded the following learned linear matrix $A =$

$$\begin{pmatrix} -1.0016012 & -0.02534942 \\ 0.08672716 & -4.32671381 \end{pmatrix}$$

The evaluation of this linear model resulted in a final Mean Squared Error of:

```
MSE_linear = 0.037276
```

This MSE value serves as a crucial baseline for our analysis. It represents the best possible performance achievable under the assumption of a linear dynamical system. This result will be compared against the performance of a more complex, non-linear model in the next section.

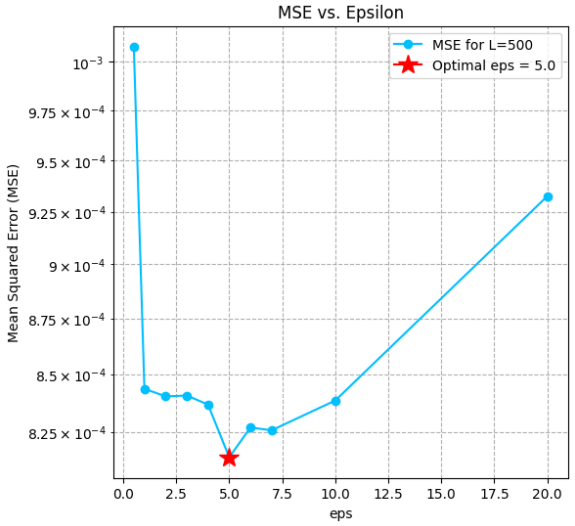
3.2 Vector Field Approximation with RBF

3.2.1 Methodology and Hyper-parameter Tuning

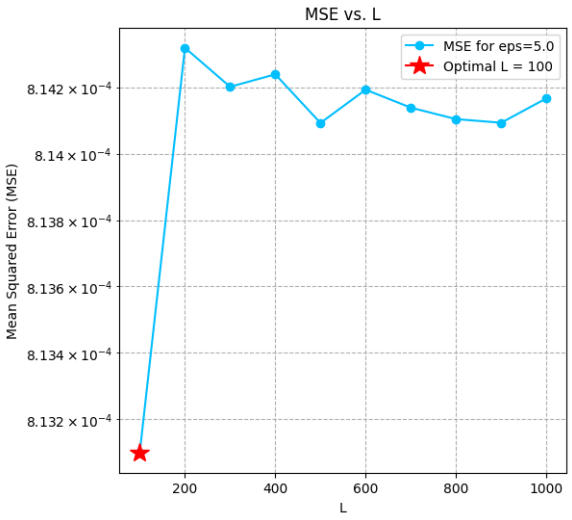
To capture the system's non-linearities, we employed a Radial Basis Function (RBF) network, which models the vector field as a weighted sum of `L` Gaussian basis functions. The performance of this model is highly sensitive to its hyperparameters: the number of centers `L` and their bandwidth `eps`.

Our approach to finding optimal parameters was a two-stage process. We began with a coordinate descent search to identify a promising region for `L` and `eps`. To validate this result, we then performed a grid search using the `GridSearchCV` utility, which incorporates 5-fold cross-validation. The optimal hyperparameters shall be the same.

3.2.2 Results and Comparative Analysis



(a) MSE is plotted against the bandwidth ϵ for a fixed number of centers ($L = 500$)



(b) MSE is plotted against the centers $L =$ for a fixed bandwidth ($\epsilon = 5.0$)

Figure 6: Hyperparameter tuning for the RBF approximator using a coordinate descent search

The results of our hyperparameter search are visualized in Figure 6. The left Figure 6a shows the model's MSE as a function of the bandwidth ϵ for a fixed number of centers ($L = 500$). This plot demonstrates a classic bias-variance tradeoff, achieving an optimal value at $\epsilon = 5.0$.

With the optimal bandwidth fixed at $\epsilon = 5.0$, we then explored the impact of model complexity by varying the number of centers, L. The Figure 6b plots the MSE against L. The lowest error was clearly identified at $L = 100$. Here, increasing the number of centers beyond this point did not improve performance and, in fact, led to a slight increase in error. The evaluation of this linear model resulted in a final Mean Squared Error of:

```
MSE_linear = 0.000813
```

This optimized RBF model, with an MSE of 0.000813, vastly outperforms the linear model, which had an MSE of 0.037276. The error was reduced by approximately 45 times. The result provides evidence that the underlying vector field is inherently nonlinear.

To validate this result with greater statistical robustness, we then performed a grid search on a 4×4 grid using the `GridSearchCV` utility, which incorporates 5-fold cross-validation. This method systematically tests all combinations of 'L' and 'eps' from a predefined grid:

- L values: [100, 200, 400, 600]
- eps values: [1.0, 2.0, 5.0, 10.0]

The cross-validation process ensures that the selected parameters generalize well. The results from this automated search confirmed our manual findings, identifying the same optimal parameter set with $L = 100$, $\epsilon = 5.0$.

3.3 System Data Analysis

Having optimized the RBF model, we now use it to analyze the global dynamics of the system. We simulate the learned vector field for a time period ($T_{end} = 100$) from every initial point x_0. The final positions of these trajectories were then clustered using the K-Means algorithm to identify the system's steady states.

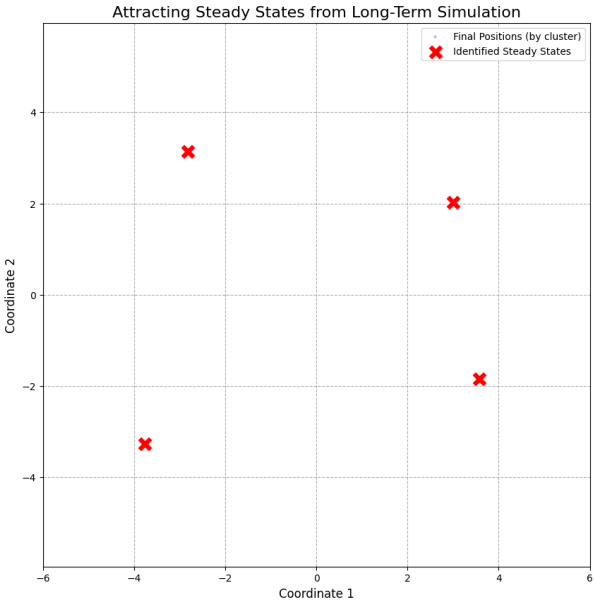


Figure 7: Attracting Steady States from Long-Term Simulation

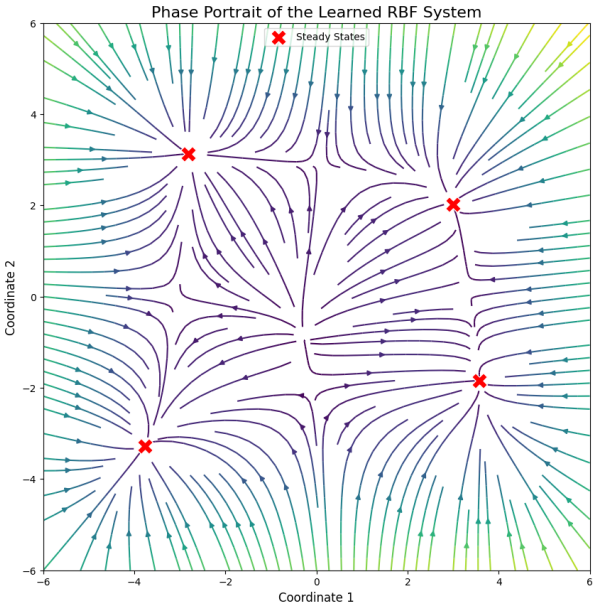


Figure 8: Phase Portrait of the Learned RBF System

As shown in Figure 7, the simulation reveals a complex dynamical structure. The system does not converge to a single point but instead exhibits multi-stability, settling into one of four distinct attracting fixed points (attractors). The phase portrait visualized in Figure 8 further clarifies the global dynamics, showing that the state space evolves into four corresponding clusters. The flow pattern also indicates an unstable fixed point (a repellor) at the origin, directing trajectories outwards towards the attractors.

The existence of multiple attractors in our learned model proves that the system cannot be topologically equivalent to any linear system. As demonstrated by our long-term simulation and clustering analysis, the learned RBF system exhibits a multi-stable structure with four distinct attracting steady states. In contrast, a linear system possesses exactly one steady state at the origin ($x = 0$), as this is the only solution to the equation $Ax = 0$ for an invertible matrix A.

4 Task 4: Time-Delay Embedding

In this section, we look at Time-delay embedding.

4.1 Part 1: Embedding a Periodic Signal

In this task, we input the data from *./data/periodic.txt*, which is a data matrix $X \in \mathbb{R}^{1000\times 2}$ that contains the two coordinates of a closed, one-dimensional manifold. In Figure 9, the plots of the columns against each other and the index vs the first column of data are seen. The periodicity of the data, as earlier mentioned, is clearly visible in the image.

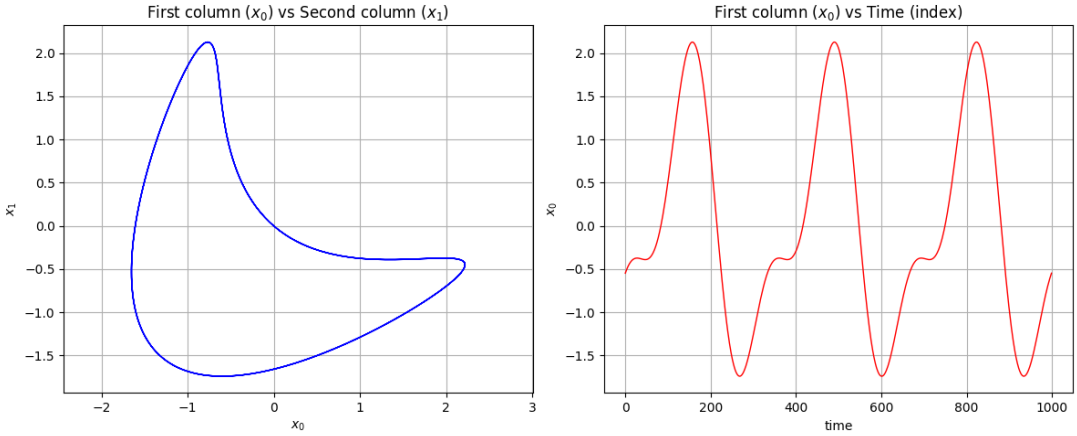


Figure 9: Plots of x_0 vs x_1 and time (index) vs x_0

Now we take different values of τ, the delay, and plot the values of x_0 against x_0 shifted by these values of τ. This is seen in Figure 10.

Each row in Figure 10 depicts one value of τ and the three plots demonstrate different aspects of time-delay embedding reconstruction:

Left Plot - Original 2D Trajectory (x_0 vs $x_0 + \tau$) : This plot shows the two-dimensional trajectory of the system in coordinate space. The closed curve represents the periodic orbit of the system.

Middle Plot - 2D Projection of Time-Delay Embedding : This plot displays the relationship between time and x_0, $x_0+\tau$, and $x_0+2\times\tau$ phase space projection. The periodic orbit is not visualized in the time vs space diagram.

Right Plot - 3D Time-Delay Embedding : This three-dimensional visualization shows the complete state space reconstruction using coordinates [x_0, $x_0 + \tau$, and $x_0 + 2 \times \tau$]. According to Taken's theorem, this 3D embedding provides the minimum required dimensionality ($2 \times d+1 = 3$) for properly reconstructing a one-dimensional manifold.

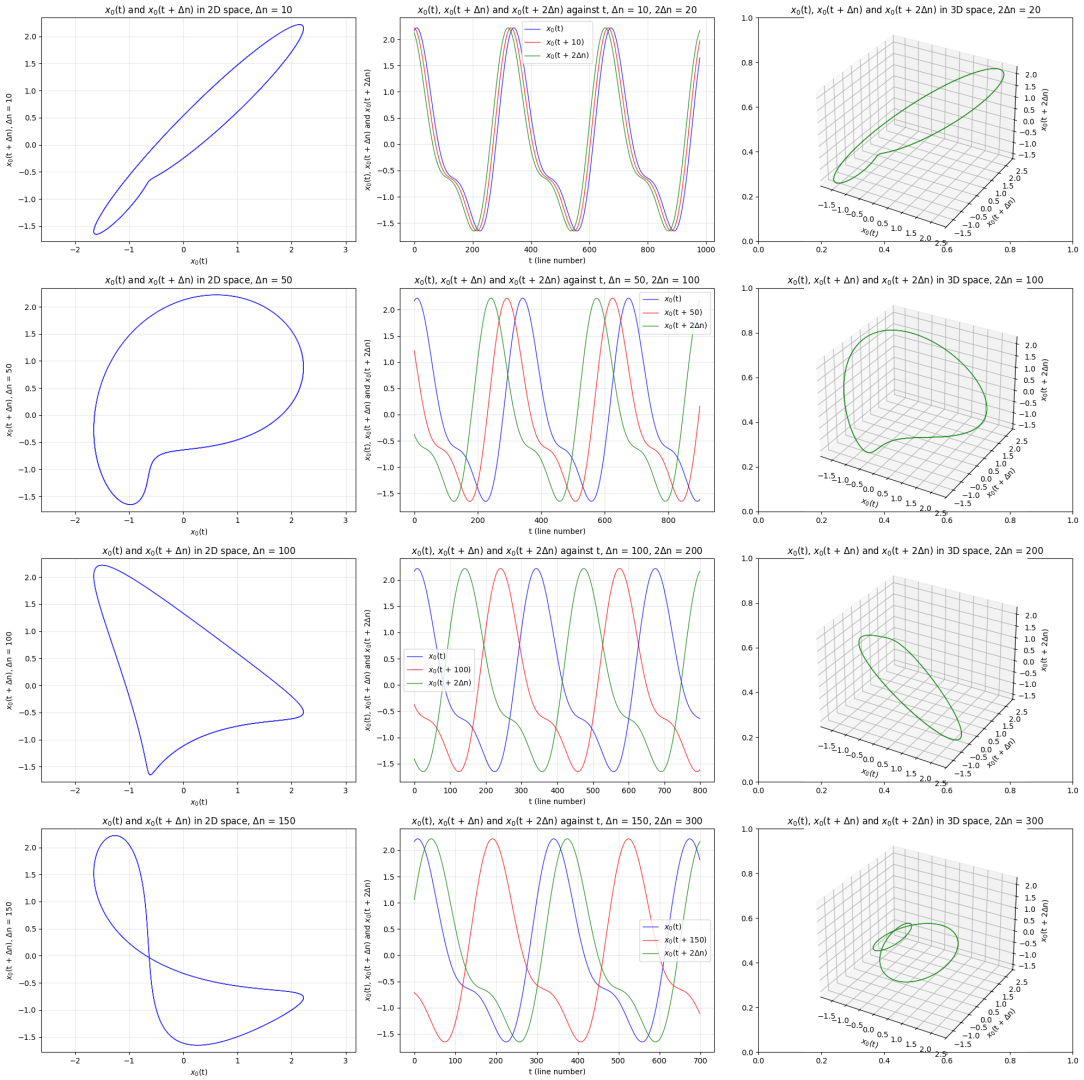


Figure 10: Plots of x_0 vs $x_0 + \tau$ vs $x_0 + \tau$ vs time (index) and 3D- time delay embedding

We look at different values of τ across a wide range. However, this doesn't give us any vital information. In order for the best time-delay embedding, the optimal delay is the best. The first local minimum of Average Mutual Information (AMI), as proposed by Fraser and Swinney [1], is chosen as the optimal delay because it's the point where data points are no longer too redundant but still retain some dynamic information, allowing a faithful phase space reconstruction. This is done in our case by function *get_optimal_tau* and gives out a result of 6 as seen in Figure 11.

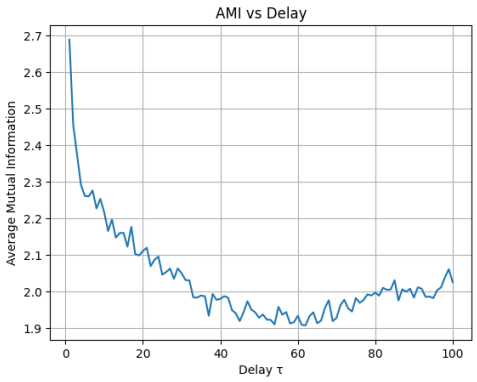


Figure 11: Optimal delay (first local minimum)

With this value, we look at finding the optimum dimension. Figure **??** shows two plots with Average Correlation Between Dimensions and Total Variance in Embedding for different dimension values from $2 \times d$ and upwards. Ideally, we want low correlation between coordinates so that dimensions are informative and non-redundant, while a high variance is expected, which would mean that the embedding spreads out the dynamics well[2].

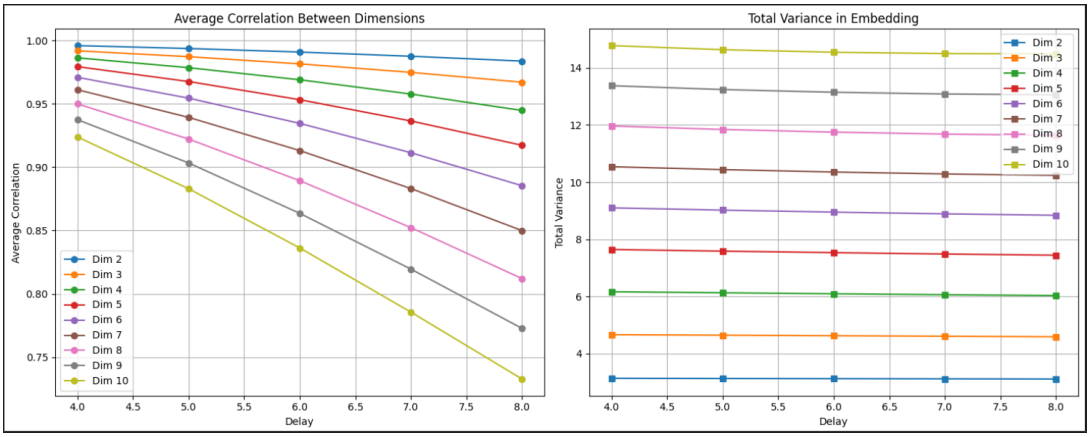


Figure 12: Average Correlation Between Dimensions and Total Variance in Embedding for different dimensions

Looking at Figure 12, it is seen that at lower dimensions (2–4), the average correlation between coordinates is very high, indicating redundancy and insufficient unfolding of the underlying dynamics. With increasing dimension, the average correlation drops significantly from dimension 6, as it falls below 0.9 (a commonly accepted threshold for effective embedding) at dimension = 8.
The total variance increases with dimension but begins to stagnate around dimensions 8–9, suggesting that additional dimensions beyond this point add little new information. Therefore, dimension 8 offers a balance between low inter-dimensional correlation and high variance, making it a good choice for phase space reconstruction.

In Figure 13, the reconstruction is done using the ideal parameters, delay = 6 and dimension = 8.

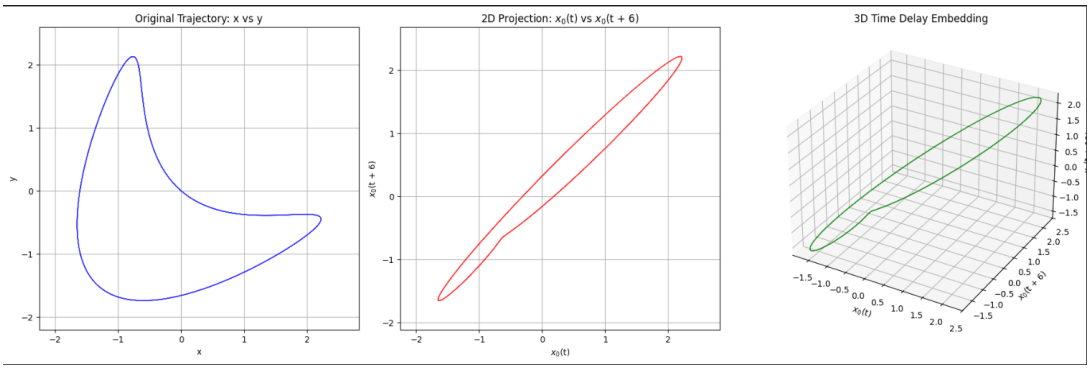


Figure 13: Plots of x_0 vs x_1 and time (index) vs x_0

4.2 Part 2

In this section, we approximate chaotic dynamics from a single time series. We have already plotted the Lorenz attractor in Section 3 and used the parameters $\sigma = 10$, $\rho = 28$, and $\beta = 8/3$ to be in the chaotic regime and with starting point (10, 10, 10).

4.3 Only Measurable X-Coordinate

After creating a Lorenz System with classic chaotic parameters and defining initial conditions, we use scipy's `solve_ivp` function to numerically integrate the system. The simulation is then run over 50 times, and only the `x_$coordinate` is extracted. We use these x-coordinate values with time embedding to create plots similar to Figure 10, where we have a look at the original x time series, a 2D embedding (x(t) vs x(t+Δt)), and a 3D reconstructed attractor plot. This is seen in Figure 14.

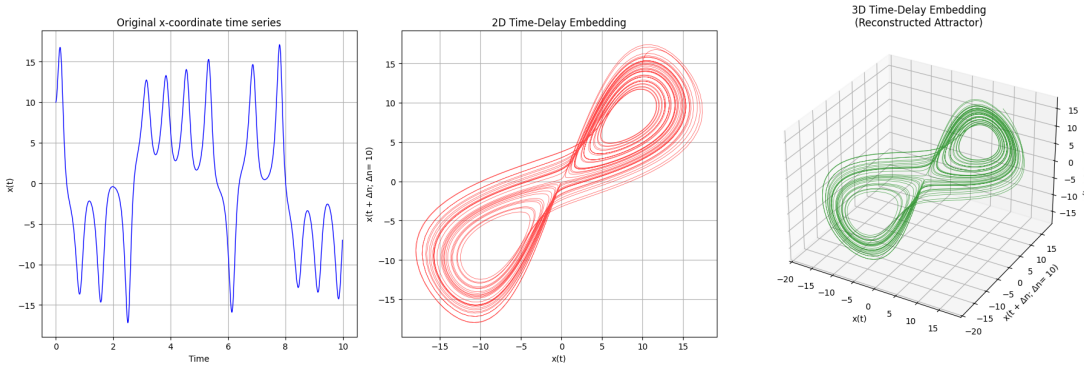


Figure 14: original x time series, a 2D embedding (x(t) vs x(t+Δt)), and a 3D reconstructed attractor plots

We further explore the changes to embedding for different delay values. The result is seen in Figure 15.
The image shows that, for various delay values (5, 10, 20, and 30), the structure of the Lorenz attractor is successfully reconstructed. This demonstrated that a single observable can capture the system's underlying dynamics. However, the quality of the reconstruction is clearly different based on the delay chosen. A small delay, such as 5, results in highly redundant coordinates and a compressed attractor. A slightly higher delay of value 10 produces a well-separated, clean reconstruction with clearly defined lobes, indicating an optimal choice. As the delay increases further (20 and 30), the attractor becomes increasingly distorted, reflecting a loss of temporal coherence between embedded dimensions. This highlights the importance of selecting an appropriate delay to ensure meaningful phase space reconstruction from scalar time series data.

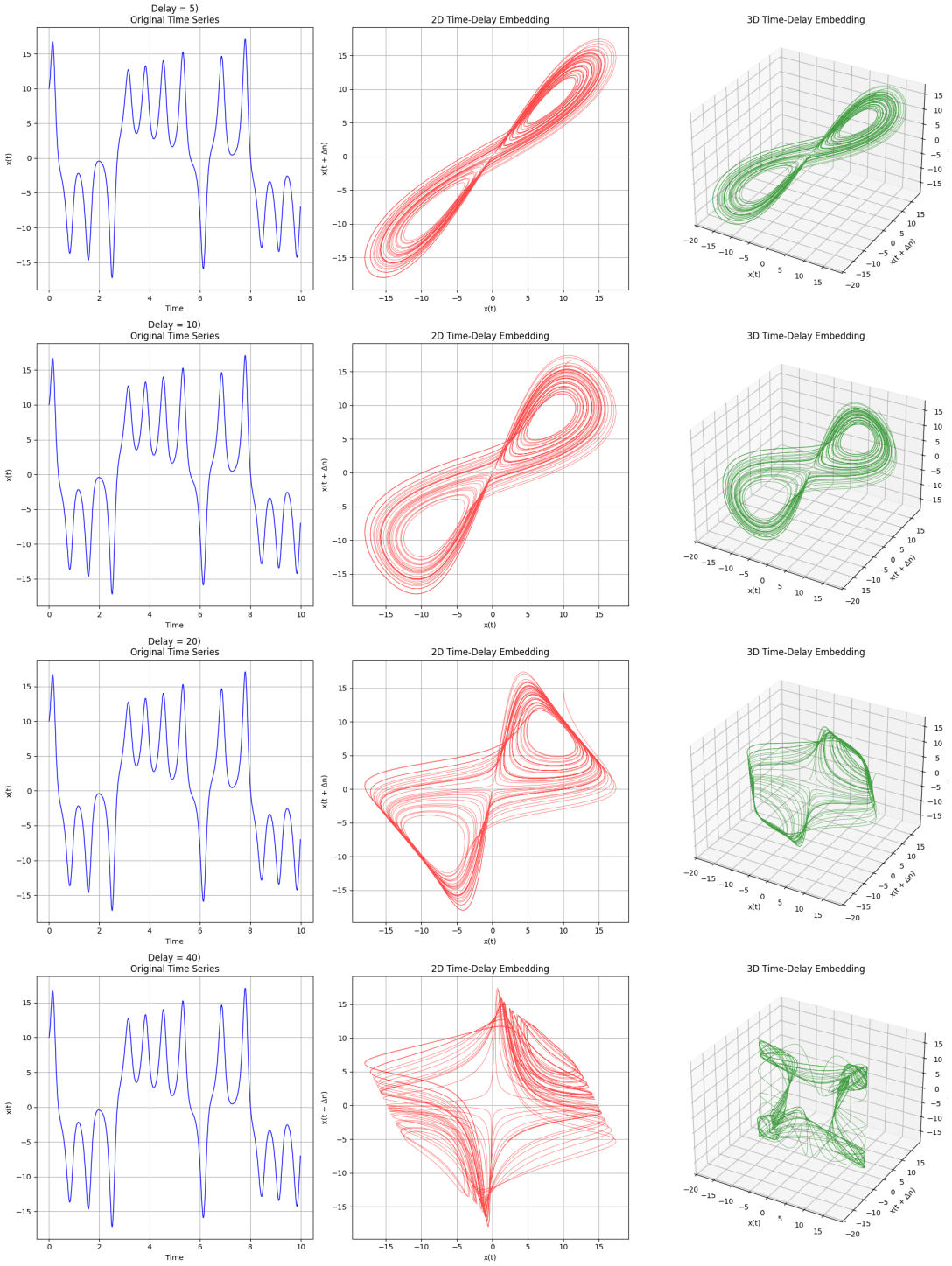


Figure 15: original x time series, a 2D embedding (x(t) vs x(t+Δt)), and a 3D reconstructed attractor plots for different delays

4.4 Only Measurable Z-Coordinate

Now we explore whether it is possible to recreate the attractor when only the z-values are measurable. The resulting plots for this scenario with various delays are seen in Figure 16.

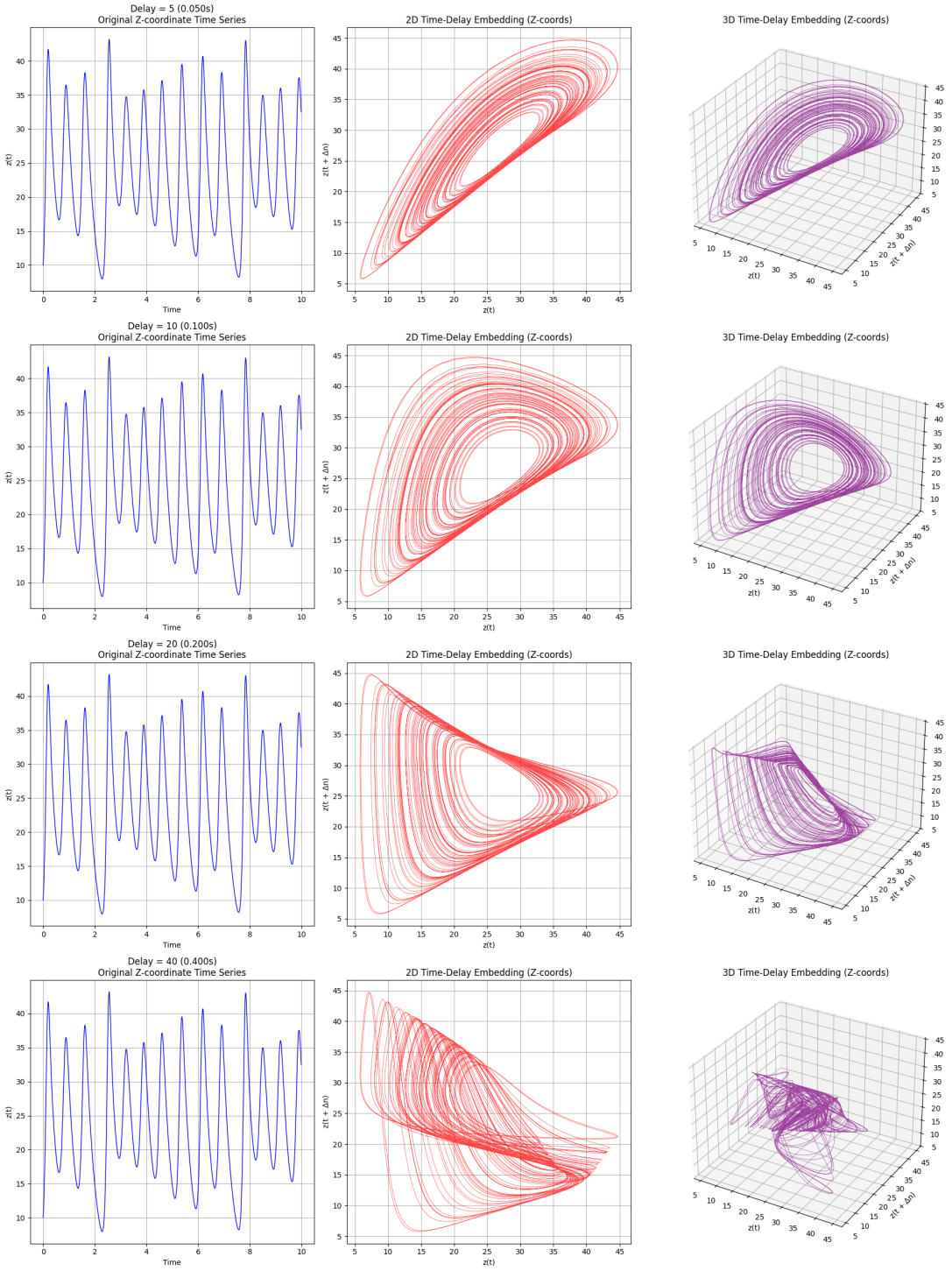


Figure 16: original z time series, a 2D embedding (z(t) vs z(t+Δt)), and a 3D reconstructed attractor plots for different delays

When performing time-delay embedding with only the z-coordinates, we see that, unlike 10, where the x-coordinates were used, the reconstruction does not produce the butterfly shape. Although an embedding is technically still possible (as guaranteed by Takens' Theorem), it fails to meaningfully capture the full dynamics of the system.

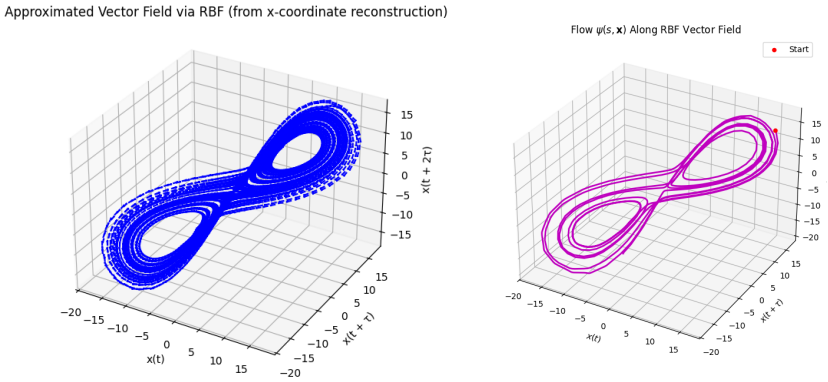
This is because the z-variable contains less dynamical information as it evolves smoothly and regularly, with fewer rapid transitions and less sensitivity to initial conditions compared to x and y (z just doesn't reflect the chaotic structure as clearly). Additionally, the key nonlinear interactions that give rise to the butterfly structure occur in the x–y plane, and are only indirectly reflected in z.[2]

4.5 Bonus

In this part, we approximate the radial basis function of the vector field $\hat{v}$ using the Radial Basis Function Interpolator from `scipy`. We then solve the differential equation

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

. The result of these tasks is seen in Figure 17, where we see the vector field and the flow field.



(a) Vector field

(b) Flow

Figure 17: Plot of vector field and flow from solving the differential equation

In Figure 18, we see the flow superimposed on the training data (Lorenz attractor with only x-values).

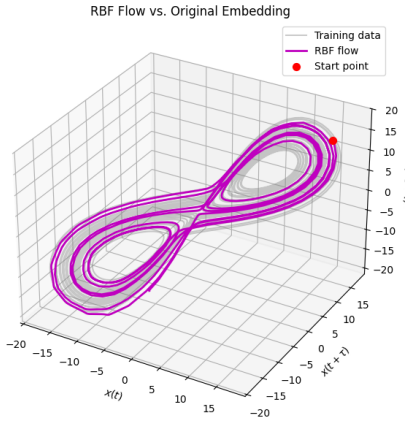


Figure 18: Flow field and training data superimposed

From these images, we can see that the butterfly geometry is successfully reconstructed by Radial Basis Functions, and it shows that they are an effective tool for reconstructing the underlying vector field of complex, chaotic dynamical systems like the Lorenz attractor from observed or embedded data. The accurate capturing of the flow (as seen in Figure 18), where the plots are identical to each other, shows that the RBF approach can successfully "learn" the motion of the Lorenz system and retrace it.

5 Task 5: Learning Crowd Dynamics

5.1 Introduction

In this task, we aim to learn the underlying periodic dynamics of crowd utilization on the TUM Garching campus, using time-delay embedding and unsupervised learning. The dataset consists of sensor counts from nine locations over seven weekdays, capturing the typical movement patterns of students across areas such as the MI building, U-Bahn station, and the two mensas.

5.2 Part 1 – Constructing a State Space via Time-Delay Embedding

We begin by constructing a suitable state space to represent the evolution of the system. Since the underlying process is periodic and likely low-dimensional, we follow Takens' embedding theorem and use a time-delay embedding of three measurement areas (columns 2, 3, 4 of the dataset, corresponding to Sensors 1–3).

Using 350 delays, each observation window becomes a vector of shape $351 \times 3 = 1053$, resulting in an embedding matrix of shape $(M, 1053)$. We apply Principal Component Analysis (PCA) to reduce the dimensionality of this high-dimensional representation. Based on the cumulative explained variance from PCA, we find that 6 principal components are required to preserve 95% of the variance. Although Takens' theorem suggests a minimal embedding dimension of $2d+1$ for state reconstruction, in practice, more dimensions may be required to capture the noise and complexity in real-world data.

As shown in Figure 19, the raw time series of three sensors (Sensor 1: MI building, Sensor 2: U-Bahn, Sensor 3: Mensa 1) exhibit clear daily periodicity over the course of a week. These patterns motivate the use of time-delay embedding, as the system appears to evolve on a low-dimensional, cyclic manifold in state space.

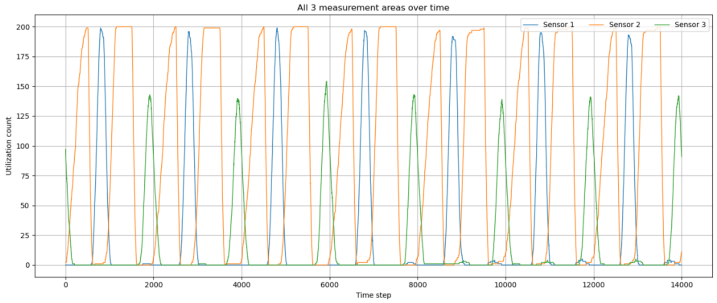


Figure 19: The raw time series of three sensors (Sensor 1: MI building, Sensor 2: U-Bahn, Sensor 3: Mensa 1)

5.3 Part 2 – Coloring the Embedding by Sensor Values

To explore how the original sensor measurements relate to the learned embedding, we color the PCA-reduced data points using the values from all nine sensors at the starting point of each delay window.

As seen in Figures 20 and 21, each sensor's spatial pattern is smoothly distributed along the closed loop. The MI building (Sensor 1), in particular, shows a sharp daily peak corresponding to daytime usage, followed by low activity during the night, forming a striking temporal structure in the PCA embedding.

This visualization confirms that the delay embedding captures meaningful temporal patterns and that the original sensor values vary coherently along the learned state space.

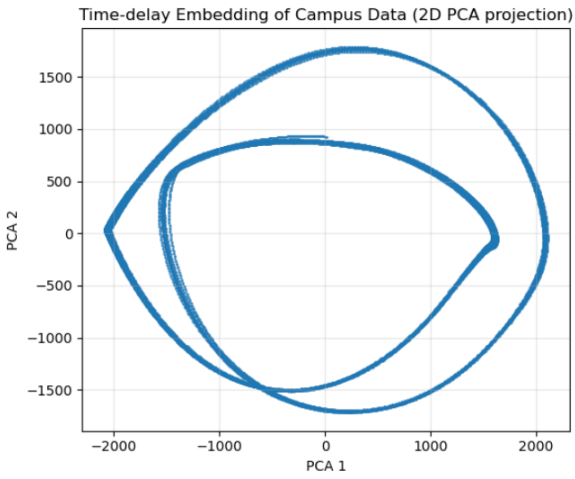


Figure 20: Sensor Values in 2D PCA Space

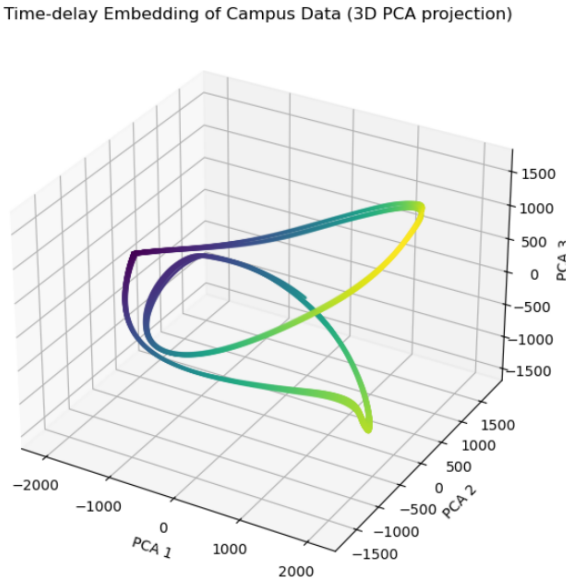


Figure 21: Sensor Values in 3D PCA Space

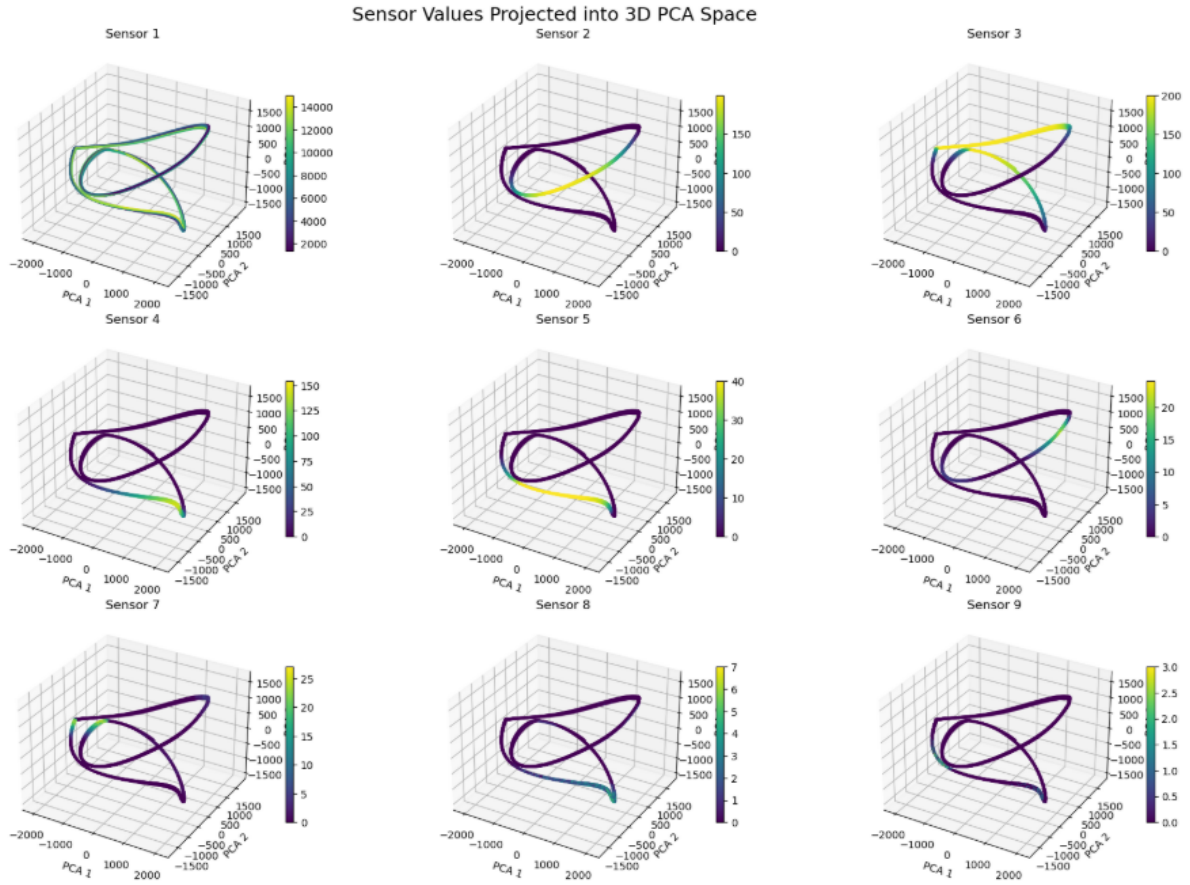


Figure 22: Sensor 1 (MI Building) Value Colored on 2D PCA Projection

5.4 Part 3 – Learning the Dynamics along the Embedding Curve

To approximate how the system evolves over time along the PCA curve, we compute the *velocity* of the system in the embedding space. This is done by taking the pairwise Euclidean distances between consecutive PCA points (in 2D), resulting in a scalar velocity at each time step.

We treat the embedding trajectory as a one-dimensional manifold parametrized by **arclength**, and define the system's dynamics as a scalar function:

$$\frac{d(\text{arclength})}{dt} = \text{velocity}$$

This arclength parametrization provides a compact, periodic representation of the system's evolution and sets the foundation for forecasting future states.

Figure 23 shows the estimated velocity as a function of normalized arclength over one full cycle. The periodicity of the motion is clearly visible, with identifiable peaks and valleys that repeat across days.

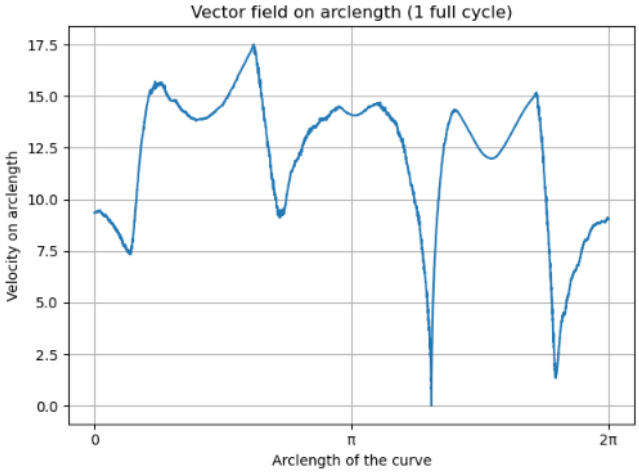


Figure 23: Estimated Velocity Field as a Function of Normalized Arclength (1 Full Cycle)

5.5 Part 4 – Forecasting MI Building Utilization

In the final step, we use the learned dynamics to forecast the utilization of the MI building over a 14-day period. This involves two key models: one for predicting the system's movement along the embedded manifold (arclength), and one for translating this movement into crowd usage.

Velocity Model. We first fit a velocity model $\dot{s} = v(s)$ using radial basis function (RBF) interpolation, where s denotes the normalized arclength. The model is trained on one cycle of data (2000 time steps), with normalized arclength values $s \in [0, 2\pi]$ and corresponding velocity values derived from pairwise distances in PCA space. We use an RBF model with 300 centers ($L = 300$) and a spread parameter $\varepsilon = 0.5$, which provides a good balance between capturing fine detail and avoiding overfitting. Smaller ε values produce sharper, more localized basis functions, while larger values result in smoother approximations.

Forward Integration. Using this velocity model, we generate the 14-day arclength trajectory by performing Euler integration from the last observed point. We use a small step size of $\Delta t = 0.005$ to ensure numerical stability and smooth evolution over time:

$$s_{t+1} = (s_t + \Delta t \cdot v(s_t)) \mod 2\pi.$$

This yields a periodic and continuous arclength prediction over 28,000 steps (14 days $\times$ 2000 steps per day).

Utilization Model. Next, we train a second RBF model to map arclength to the original MI building usage (Sensor 1 values). Again, we use $L = 300$ centers but with a slightly narrower spread $\varepsilon = 0.3$, since the sensor signal exhibits sharper daily peaks that require finer resolution to approximate well. The model is trained on one full cycle of data, capturing the periodic usage structure.

Results. Figure 24 shows the predicted arclength trajectory over the 14-day horizon. Figure 25 shows the corresponding MI building usage forecast obtained by evaluating the utilization model on the predicted arclength.

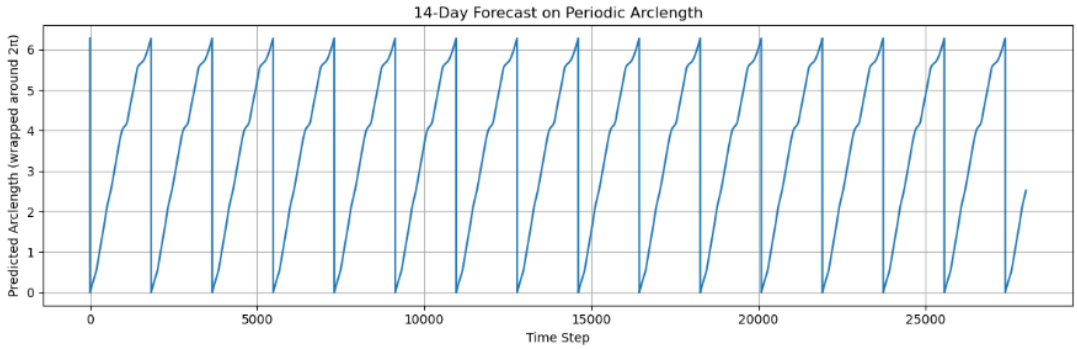


Figure 24: 14-Day Forecast on Periodic Arclength

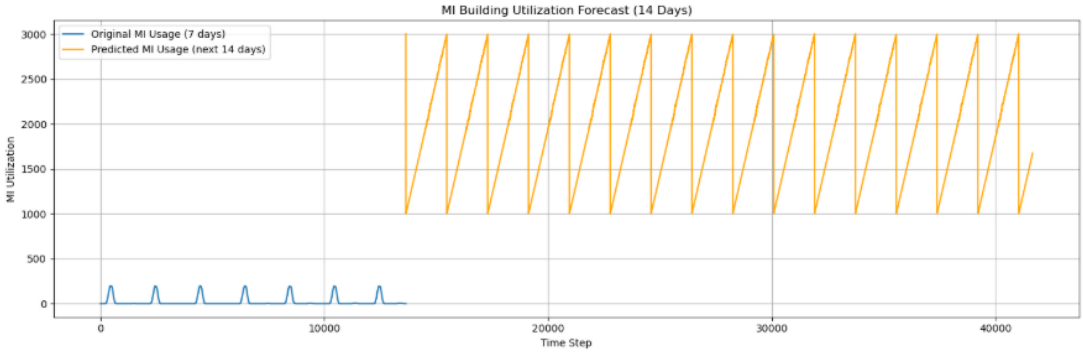


Figure 25: Predicted MI Building Utilization from Learned Dynamics (Next 14 Days)

Discussion. Although the forecast captures the overall daily rhythm and periodicity of MI building usage, the predicted signal does not perfectly match the real data in amplitude or shape. Several sources of error may explain this discrepancy:

- Accumulated drift from numerical integration over long horizons
- Underfitting or overfitting of the RBF approximators, especially near sharp transitions
- Lack of explicit modeling of external factors such as weekends or special events
- Absence of regularization or denoising mechanisms in the raw sensor input

Nonetheless, the method succeeds in learning a smooth and cyclic latent representation of human activity and forecasting its temporal structure. This approach highlights the effectiveness of combining time-delay embedding, dimensionality reduction, and function approximation for learning interpretable dynamics from periodic crowd data.

References

[1] Andrew M. Fraser and Harry L. Swinney. Independent coordinates for strange attractors from mutual information. *Phys. Rev. A*, 33:1134–1140, Feb 1986. URL: `https://link.aps.org/doi/10.1103/PhysRevA.33.1134`, `doi:10.1103/PhysRevA.33.1134`.

[2] Holger Kantz and Thomas Schreiber. *Nonlinear Time Series Analysis*. Cambridge University Press, Cambridge, 2nd edition, 2004.