Report for exercise 5 from group A

Tasks addressed: 5
Authors: ALEX PASQUALI (03754113)
NICOLA GUGOLE (03753996)

Last compiled: 2022–01–21
Source code: `https://github.com/AlexPasqua/MLCMS-exercises`

The work on tasks was divided in the following way:

ALEX PASQUALI (03754113)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%
NICOLA GUGOLE (03753996)	Task 1	50%
	Task 2	50%
	Task 3	50%
	Task 4	50%
	Task 5	50%

Report on task 1, Approximating functions

This task consists in approximating linear and non-linear functions. We creatd a Python script called `function_approximation.py` that contains a series of functions to read the data, compute the radial basis functions, approximate linear and non linear data and plot the approximated function over the actual data.

Part 1 This part consists in linearly approximating the linear data contained in `linear_function_data.txt`. To accomplish the task, in `function_approximation.py`, we created the function `approx_lin_func` as a sort of wrapper for numpy's least squares solver (`numpy.linalg.lstsq`): it simply splits the data into points and targets (if the path is provided instead of the actual data, the function reads the file first), then it applies numpy's solver and returns the result. The code is shown in Figure 1, while the plot of the approximated function over the actual data is instead shown in Figure 3, where, as expected, it is possible to see that the data is approximated very well. Using the radial basis functions to approximate linear data is not the best idea because, to reach a semi-decent approximation, we should use a number of basis functions near to the number of points (more is not possible, because we would need more centers than actual points) and a large ϵ. This is shown in Figure 3.

```python
def approx_lin_func(data: Union[str, Iterable[np.ndarray]] = "../data/linear_function_data.txt") -> Tuple[np.ndarray, np.ndarray, int, np.ndarray]:
    """
    Approximate a linear function through least squares
    :param data:
        Either str: path to the file containing the data in the format Nx2, col 0 is the data, col 1 the targets.
        Or Iterable containing 2 numpy ndarrays: points and targets
    :returns: tuple (least squares solution, residuals, rank of coefficients matrix, singular values of coefficient matrix)
    """
    # get coefficients and targets from data
    points, targets = get_points_and_targets(data)
    # solve least square
    sol, residuals, rank, singvals = np.linalg.lstsq(a=points, b=targets, rcond=None)
    return sol, residuals, rank, singvals
```

Figure 1: Function to approximate linear data.

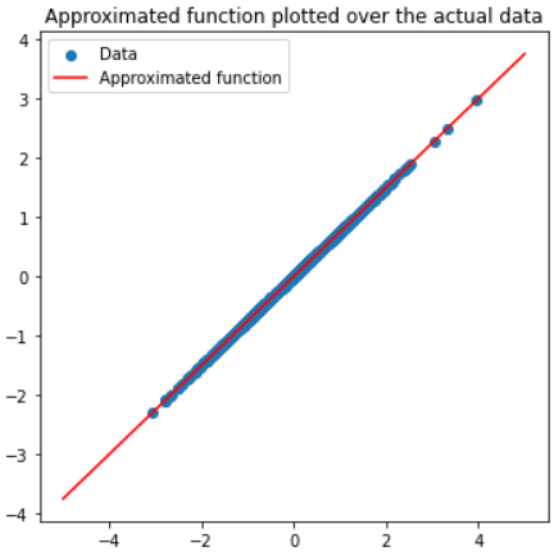


Figure 2: Plot of the linearly approximated function over the actual linear data contained in the file `linear_function_data.txt`.

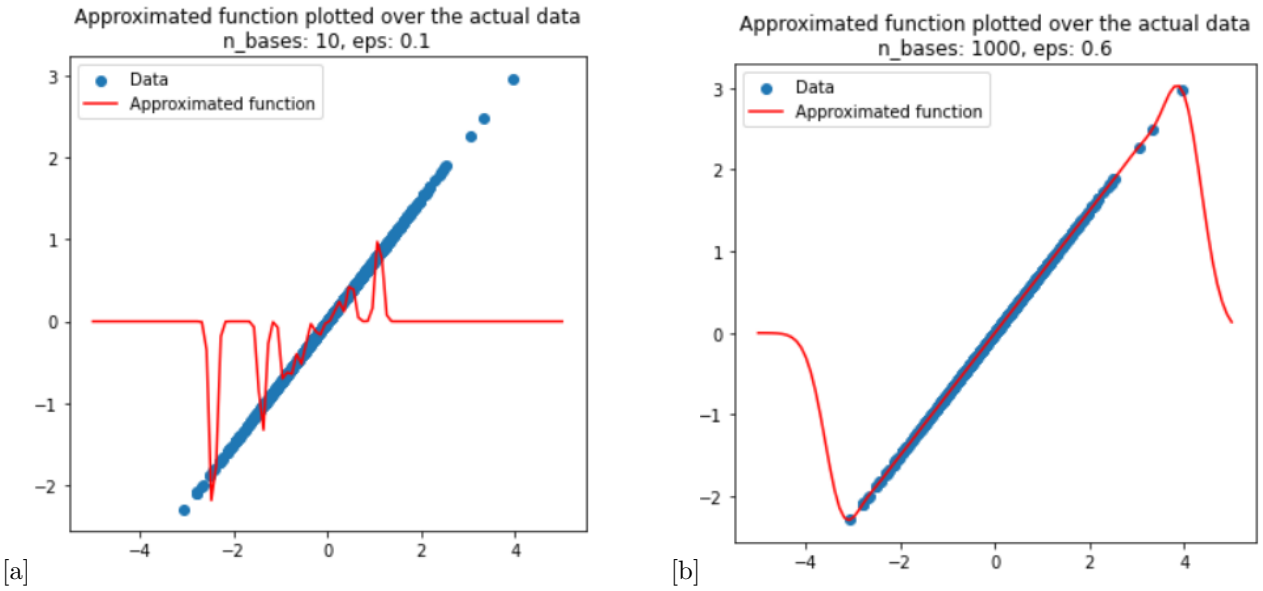


Figure 3: Approximation of linear data with a combination of radial basis functions.

Part 2 The second part consists of performing again a linear approximation, but this time the data, contained in `nonlinearfunction_data.txt`, is non-linear (as the name of the file suggests). The procedure is the same as for part 1, we simply change the path to the data file. The plot of the approximated function over the data is shown in Figure 4, and, as expected, approximating such data with a straight line does not provide good results.

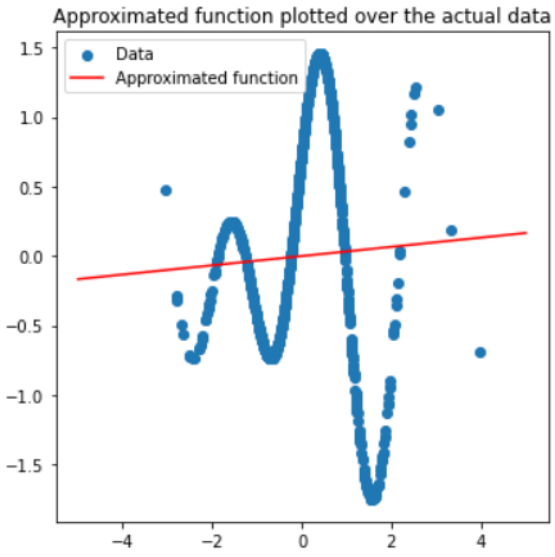


Figure 4: Plot of the linearly approximated function over the actual non-linear data contained in the file `nonlinear_function_data.txt`.

Part 3 This time the non-linear data (again the one in `nonlinear_function_data.txt`) is approximated as in Eq. 1 with a combination of radial basis functions having the form showed in Eq. 2.

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

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

where x_l is the center point of the function, chosen randomly among the data points, and ϵ indicates how wide the Gaussian is.
We implemented a function called `approx_nonlin_func` (Fig. 5) and tried different combination for the values of ϵ and L (i.e. the number of basis functions used for the approximation), which are reported in Table 1. Obviously, a value of 12 for ϵ is way bigger than needed, but it provides an insight about the behavior of the basis functions when this parameter grows a lot. Figure 6 shows how adding more functions is in general a good thing, but of course the approximation gets more costly to compute. The value of ϵ is very important as well, in fact, with the same number of functions, the results can be pretty different. This can be observed in Figure 6, looking at how different the approximation is between the plots on the same row (that correspond to different values of ϵ, but with the same number of functions). Finally, in Figure 6(e) it is shown an example of what a too large value of ϵ can do.

```python
def approx_nonlin_func(data: Union[str, Iterable[np.ndarray]] = "../data/nonlinear_function_data.txt", n_bases: int = 5, eps: float = 0.1,
                       centers: np.ndarray = None):
    """
    Approximate a non-linear function through least squares
    :param data:
        Either str: path to the file containing the data in the format Nx2, col 0 is the data, col 1 the targets.
        Or Iterable containing 2 numpy ndarrays: points and targets
    :param n_bases: the number of basis functions to approximate the nonlinear function
    :param eps: bandwidth of the basis functions
    :param centers: list of center points to compute the basis functions
    :returns: tuple (least squares solution (transposed), residuals, rank of coefficients matrix, singular values of coefficient matrix,
                     centers, eps and phi (list_of_basis))
    """
    # get coefficients and targets form the data
    points, targets = get_points_and_targets(data)

    # evaluate the basis functions on the whole data and putting each basis' result in an array
    list_of_bases, centers = compute_bases(points=points, centers=centers, eps=eps, n_bases=n_bases)

    # solve least square using the basis functions in place of the coefficients to use linear method with nonlinear function
    sol, residuals, rank, singvals = np.linalg.lstsq(a=list_of_bases, b=targets, rcond=1e-5)
    return sol, residuals, rank, singvals, centers, eps, list_of_bases
```

Figure 5: Code of the function `approx_nonlin_func` implementing a function approximation through a combination of radial basis functions.

Parameter	Values
ϵ	0.1, 0.2, 0.5, 0.8, 12
L	3, 5, 8, 10

Table 1: Values tested for ϵ and L.

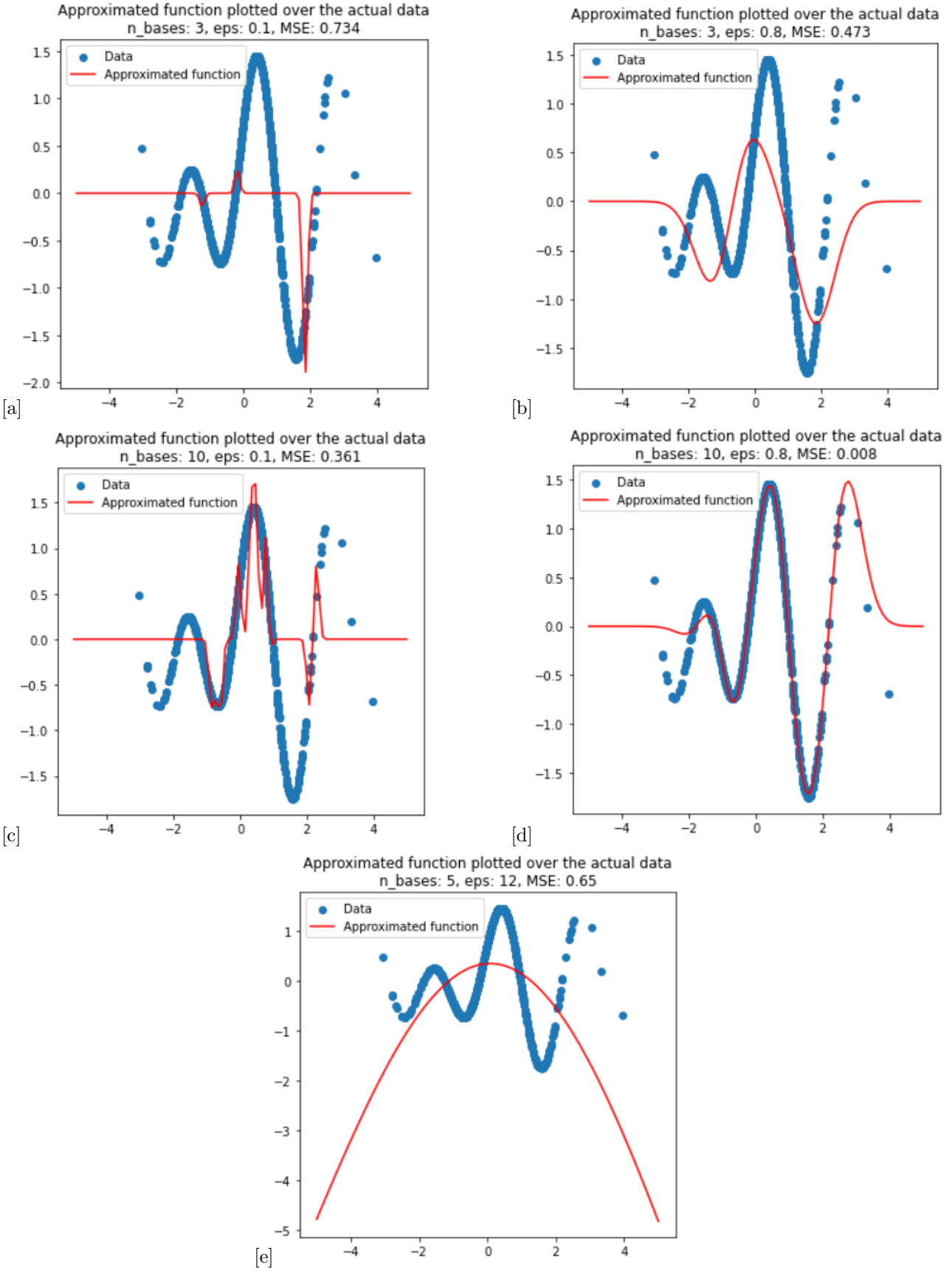


Figure 6: Examples of approximations of the non-linear data (contained in `nonlinear_function_data.txt`) with a varying number of radial basis functions and a varying ϵ.

Report on task 2, Approximating linear vector fields

This task is centered around approximating a given 2 dimensional *linear vectorfield data*. In particular we are given two instants in time for the dataset, `x0` and `x1`, with points in `x0` reaching `x1` after a certain unknown Δt (although from the task text we know the value should be **0.1**).

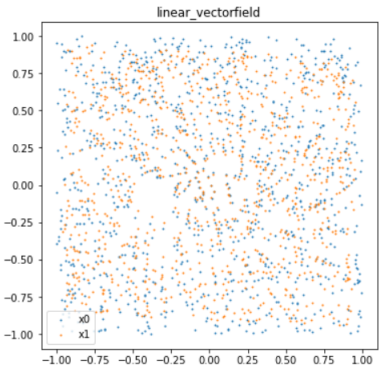


Figure 7: linear vectorfield dataset

Part 1 The first part of *task 2* asks to estimate the linear vector field `v` that leads from `x0` to `x1` such that the MSE error will be minimized in part 2, regarding a trajectory simulation with $\Delta t = 0.1$. With that `v` we can then approximate the matrix `A` solving a least square problem using `x0` and `v`.

To search for such an optimal Δt we decided to test 500 possibilities in an interval ranging from 0.0001 to 0.5, basically approximating `v`, creating `A` and then simulating the trajectory to generate `x1` to be able to assess the MSE. We decided to use `solve_ivp` from `scipy.integrate` as numerical solver. **Figure 8** shows a little part of the search, which winner was $\Delta t = 0.14336$ with an MSE equal to **0.0026804**. Such a difference between the winning Δt and the forecasted one (0.1) was explained in the QA as a derivative of having such an approximation in calculating `v`.

```
attempting with 0.08224789579158316 got mse: 0.0037736408508847303
attempting with 0.08324969939879759 got mse: 0.003714117459297639
attempting with 0.08425150300601202 got mse: 0.0036575467591858837
attempting with 0.08525330661322646 got mse: 0.003603776740917067
attempting with 0.08625511022044088 got mse: 0.0035526644169930712
attempting with 0.08725691382765531 got mse: 0.003504075205561827
attempting with 0.08825871743486974 got mse: 0.003457882361761225
attempting with 0.08926052104208416 got mse: 0.0034139664527351814
attempting with 0.09026232464929859 got mse: 0.003372214872573909
attempting with 0.09126412825651302 got mse: 0.003332521393777736
attempting with 0.09226593186372746 got mse: 0.0032947857521613225
attempting with 0.09326773547094189 got mse: 0.003258913262409406
attempting with 0.09426953907815631 got mse: 0.0032248144617539777
attempting with 0.09527134268537074 got mse: 0.0031924047794929045
attempting with 0.09627314629258517 got mse: 0.003161604230146576
```

Figure 8: small section of trying different `dt` searching for lowest MSE value

Part 2 This part of the task asks to estimate `A` with the chosen Δt to then solve the linear system up to a $T_{end} = 0.1$ and calculate once again the MSE. The MSE is exactly the one written before since the optimal Δt was already discussed and utilized to estimate `A`. In **Figure 9** one can appreciate in orange the trajectories that are moving all the initial points `x0` nearer to the objective points `x1` (in blue). The residual error above the image shows how the found `A` is a good approximation.

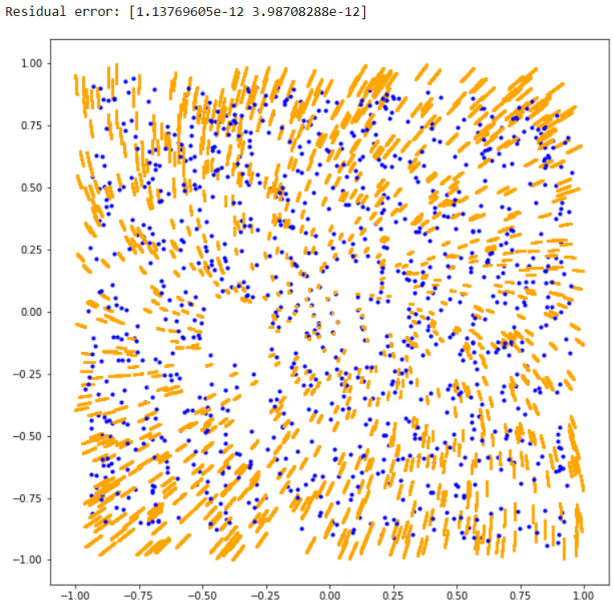


Figure 9: trajectory simulation from `x0` (in orange) lasting $\Delta t = 0.1$, with `x1` scattered (in blue)

Part 3 the final part of the task is graphically explained by **Figure 10**, where the phase portrait of the linear vector field is shown. It is clear to see how the dynamical system locally has a single attractive steady in the origin.

This part of the task is in particular asking to choose a point far away from the initial data, such as `(10,10)` and to solve its trajectory for a long time, $T_{end} = 100$. This fact is shown in the figure by the orange scatter plot, where the far apart point can be seen reaching at big leaps the steady state in the origin, confirming the phase portrait of the vector field.

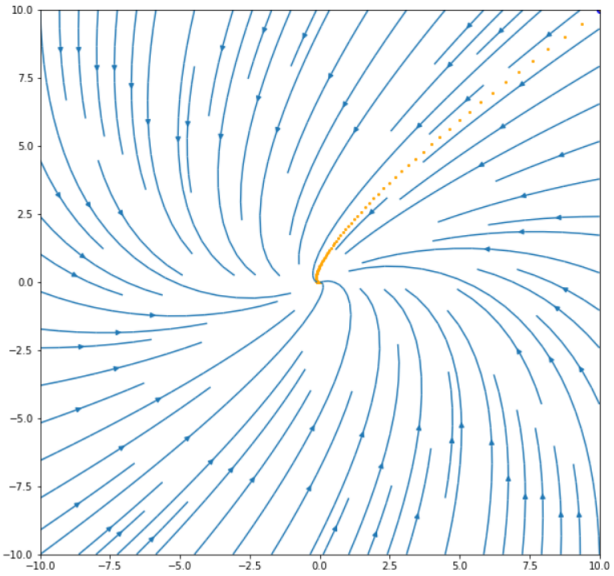


Figure 10: phase portrait of linearly approximated vector field - trajectory of point with `x0 = [10,10]`

Report on task 3, Approximating nonlinear vector fields

This task revolves around working on a given *nonlinear vector field dataset* composed once again of two moments in time, starting points `x0` and points `x1` after an unknown Δt.

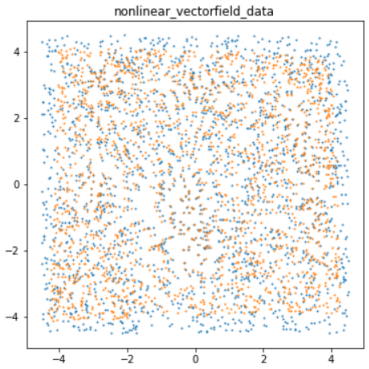


Figure 11: non linear vectorfield dataset

Part 1 Firstly we are asked to estimate the vector field with a linear operator, as we did in the previous task for the *linear vector field dataset*. We therefore fixed a `dt` (to 0.1) to approximate `v` using again the finite difference formula. With the approximation we linearly estimated `A` (with a residual error of `[3890.04548426 3563.97075073]` and we simulated the system up to time **0.5**, finding out that, at end time, the MSE has a value equal to **0.201521**, while the moment where the system got closer (starting from `x0`) to `x1` was at `t = 0.101010`, with an MSE of **0.037288**. This shows that the initial choice of `dt` influences the moment where the system gets closer to `x1`, as also the choice of `dt` changed accordingly the time with best (lowest) MSE. **Figure 12** shows the phase portrait for the linearly estimated vector field, showing a system resembling a *stable node*, with a single steady state that appears to be in the origin.

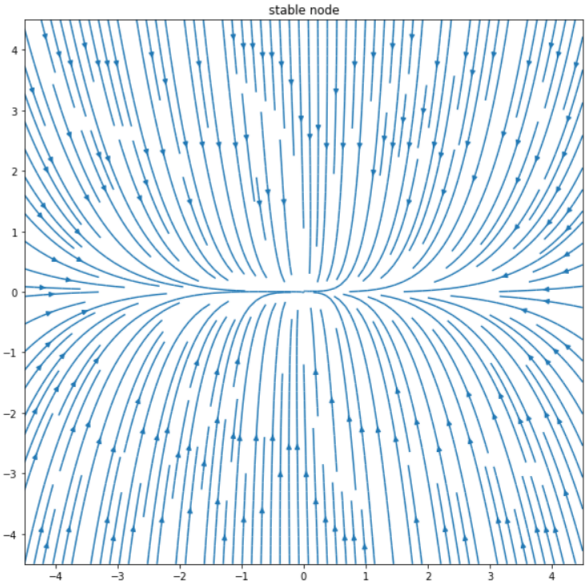


Figure 12: phase portrait of linear approximation for the *non linear vector field dataset*

Part 2 After trying to linearly approximate the vector field, we are asked to approximate it non-linearly, using the same **RBF** function we described before. To find the best parameters, we applied a grid search through the method `find_best_rbf_configuration`, trying different values for `eps` (0.3, 0.5, 0.7, 1.0, 5.0, 10.0, 20.0) and for `n_bases` (trying 20 equally spaced bases numbers between 100 and 1000). To assert the best parameters we performed MSE analysis on each of the configurations, searching for the lowest MSE one can encounter during the trajectory generation of a particular configuration. After trying the process for some times we noticed the best number of bases slightly changed from time to time (probably due to a change of the center points of the radial basis functions), revolving around **500** most of the times, but keeping `eps = 5.0`. Reporting here the final execution results, the best parameters were:

- `n_bases`: 479

- `eps`: 5.0
- `best dt`: 0.11 (once again approximating the chosen `dt`)

The best performing non-linear approximation gives a much better MSE, going down to **0.000314**, showing how this method greatly outperforms the linear counterpart. This is immediately visually clear by comparing the final trajectory points using the best linear approximation and the best non-linear approximation, as reported in **Figure 13**. Therefore one can conclude that the non linear approximation leads to a much nearer solution with respect to the linear approximation, therefore leading to the vector field itself being non linear.

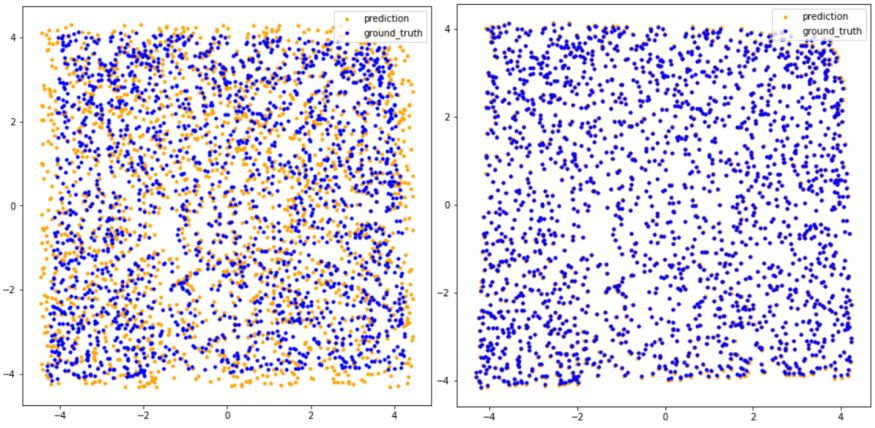


Figure 13: end trajectory points (in orange) against ground truth end points (in blue) - linear vs non linear

Part 3 Eventually, since the results proved the validity of the non linear approximation, we can solve the system starting from all the initial points `x0` for a much longer time (we opted for **50**). The non linear vector field shown in **Figure 14a** already shows five apparent steady states, with four of them being close to the corners (*attracting steady states*) and one being near the origin (*repulsive steady state*). Solving the system with `solve_ivp` effectively leads to discovering the four attracting steady states, as portrayed in **Figure 14b**.

To answer the final question, the non linearly approximated system cannot be topologically equivalent to the linearly approximated one because of their different number in steady states, as dynamical systems theory asserts.

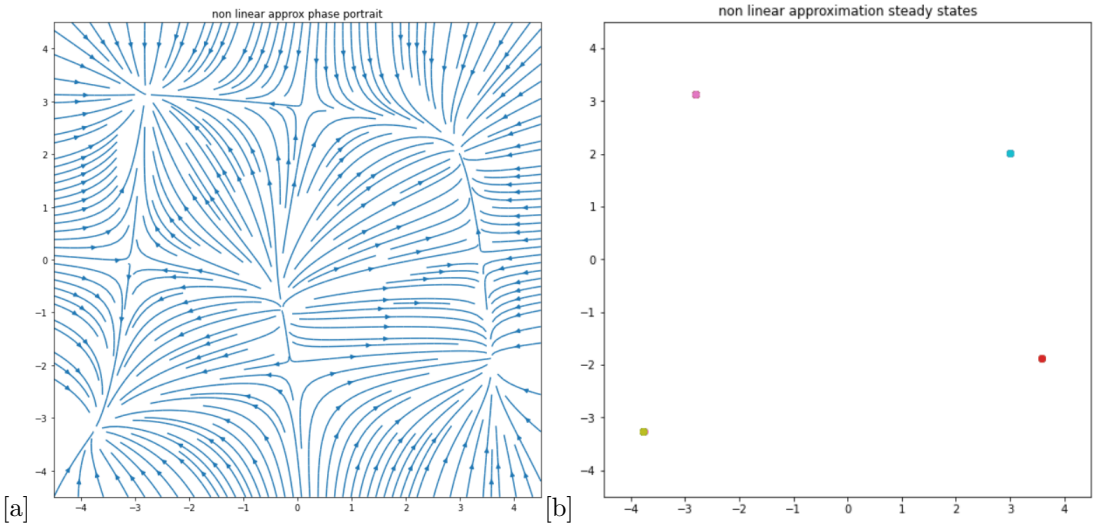
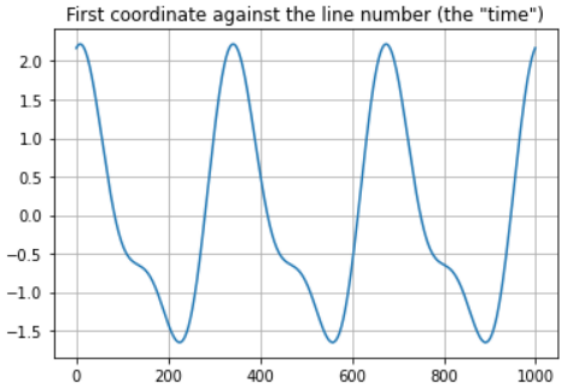


Figure 14: best non linear approximation vector field (a) - *attracting* steady states of said vector field (b)

Report on task 4, Time-delay embedding

Part 1 After reading the dataset `takens_1.txt`, we plotted the first coordinate against the time (i.e. the line number), obtaining what shown in Figure 15(a). Subsequently, we plotted the same data, but this time not against the line number, but against a delayed version of itself, namely $x(t+\Delta t)$. Choosing $\Delta t = 10$ (rows), we obtain the result shown in Figure 15(b). According to Takens theorem, being the manifold of dimension $d = 1$, it is assured to get an embedding using $2d+1 = 3$ coordinates, but as it is possible to see in Figure 15(b), since there are no intersection in the plot, two coordinates are sufficient to embed this manifold, if a proper time delay is chosen.

[a]



[b]

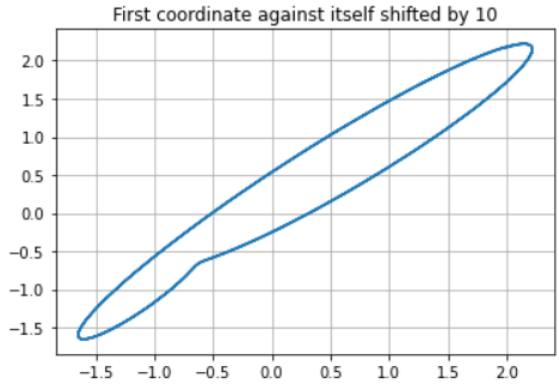


Figure 15: The first coordinate of the data contained in `takens_1.txt` plotted against the time (i.e. the line number) (a) and against a delayed version of itself (b).

Part 2 In this part, we test Takens theorem with the chaotic system called the Lorenz attractor. It is shown in Figure 16(a), setting $\sigma = 10$, $\rho = 28$, $\beta = 8/3$ and the starting point equal to $(10, 10, 10)$. Now we test Takens theorem by choosing a Δt and plotting $x(t)$ against $x(t+\Delta t)$ and $x(t+2\Delta t)$, being x the first coordinate of the Lorenz system. We also try this with the third coordinate: z. The results, with a time delay $\Delta t = 3$ (steps) are shown in Figure 16. As it it possible to observe, using the x coordinate we still get a reasonable shape that resembles the original one, while with the z coordinate this does not happen, so the latter is not suitable to represent the original system through a time delay, at least when using two delays (Δt and $2\Delta t$). This probably happens because for each value of z there are multiple values of the system, and these are found in the two lobes. In some sense, the two lobes overlap when seen from z, and that is why we get a circle-like shape in Figure 16(c).

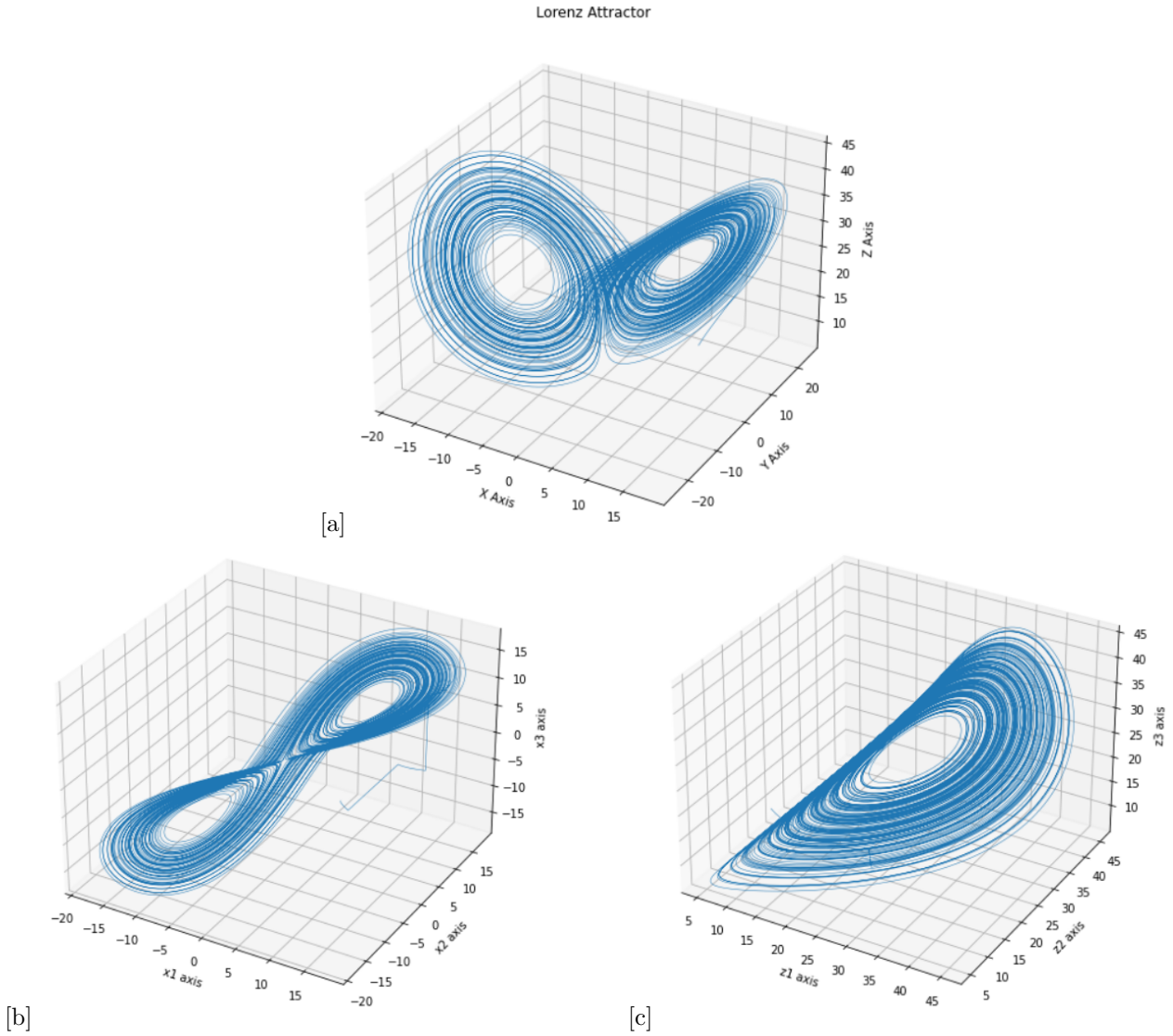


Figure 16: (a) The Lorenz attractor. (b) The Lorenz attractor when plotted using time delays on the x coordinate $(x(t),\ x(t+\Delta t),\ x(t+2\Delta t)$. (c) The Lorenz attractor when plotted using time delays on the z coordinate $(x(t),\ x(t+\Delta t),\ x(t+2\Delta t)$. The chosen time delay is 3.

Bonus We first created a trajectory concatenating the values of the coordinated $x_1(t) = x(t),\ x_2(t) = x(t+\Delta t),\ x_3(t) = x(t+2\Delta t)$. Then, to compute the approximation through the radial basis functions (using our `approx_nonlin_func` (Fig. 5)), we need some points and the corresponding targets. To get them, we started with our trajectory containing 5000 points[1] and used the first 4999 (from the first to the penultimate) as "points" and the second 4999 (from the second to the last) as their "targets". After approximating the function[2] using 100 bases and $\epsilon = 10$, we multiplied the basis functions with the solution matrix from least squares, obtaining a 3-dimensional value for each point in the trajectory. We proceeded creating a scatter plot of them, obtaining something very similar to the embedding of the attractor with the x coordinate. The result is shown in Figure 17

[1] $[[x(t),\ x(t+\Delta t),\ x(t+2\Delta t)]$ for $0 \leq t < 5000]$.

[2] It was easy to obtain good results even approximating the function with a different number of basis functions and/or a different value of ϵ, therefore the values we reported serve simply as an example.

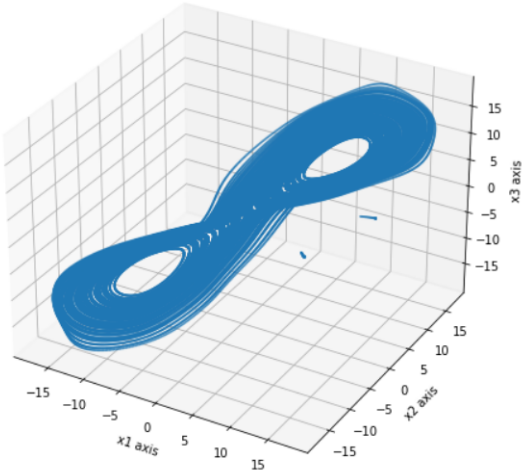


Figure 17: Reconstruction of the x coordinate time delay embedding of the Lorenz attractor after approximating the vector field of the system through a combination of radial basis functions and solving the system.

Report on task 5, Learning crowd dynamics

The last task is centered on learning a dynamical system describing the people flow in nine different areas of TUM Garching, containing the MI Building and the 2 mensas. The given database describes the system over the course of 7 days and is divided as follows: it has a first column containing the *time index* (which is simply a natural number going up one by one and starting from 1) and nine other columns containing the number of people per area at that certain time. In **Figure 18** are therefore shown nine different points at each time step, defining the amount of people per area.

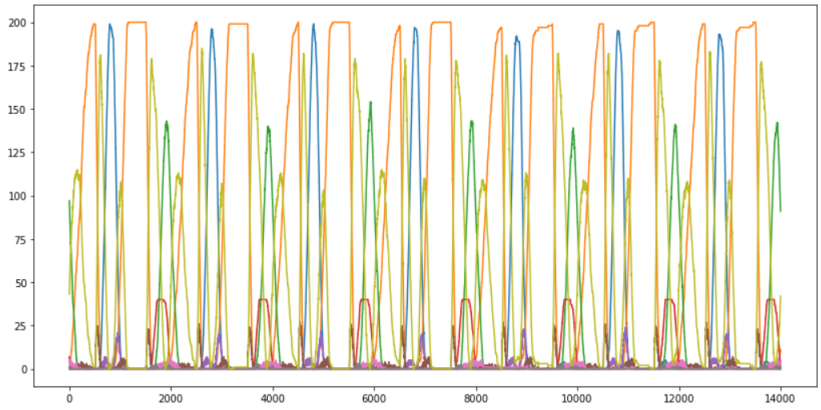


Figure 18: MI building data

Part 1 The first part of the task hints some important dataset properties, such as it being *periodic* (we know the dataset spans seven days and in fact it is easy to notice how **Figure 18** shows for seven times the same pattern) and with no parametric dependence, therefore being a nine dimensional closed loop which can be reduced to be *one-dimensional*. Following **Takens theorem**, since we know that `d=1`, we can assert that it is possible to construct an embedding of the dataset with `2d+1` dimensions, therefore we need **three** in order to get a reasonable state space. As a sidenote, first 1000 rows of the dataset have been discarded since they are regarded as *burn-in period* by the exercise sheet.

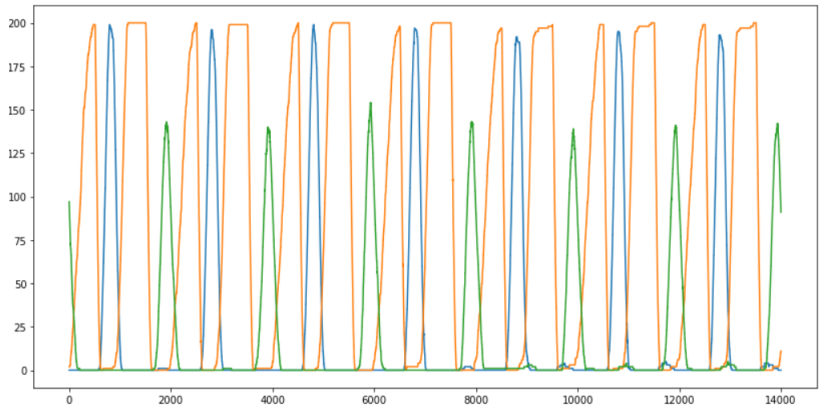


Figure 19: MI building data (only first 3 measurements)

After reducing the nine measurements zones to three (with the results shown in **Figure 19**) we create a matrix as required:

- each point (row) is a *delay embedding* of the dataset containing only the first three measurements (let us call it `df`).
- every *delay embedding* is created by taking flattened windows of 351 consecutive 3D points in `df`.
- M of this points are created by moving the start of the delay by one row. We chose M to have maximum possible value, as we have the shape of `df` and also the length of the window to create a row.

Once this rather big `Mx1053` matrix is created, then we apply PCA to get a matrix maintaining only the **three** principal components required by Takens, getting the reasonable state space to embed the system, which plot is **Figure 20**.

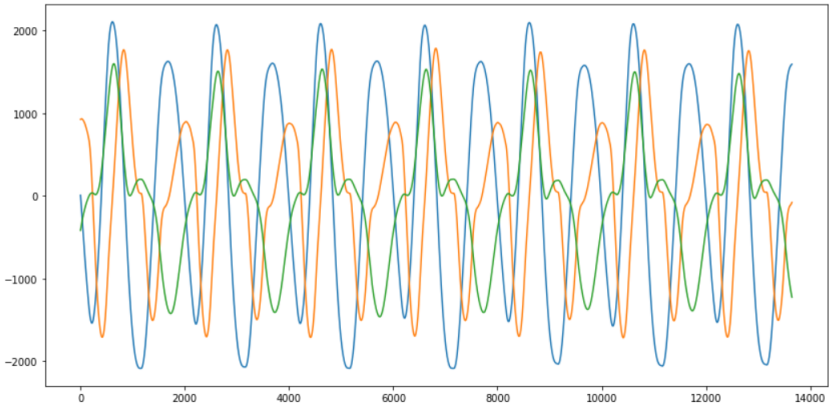


Figure 20: MI building data after PCA dimension reduction to 3 dims

One can appreciate that the reasonable state space is effectively an embedding by visualizing it in 3D (we show it from different angles in **Figure 21**), where the system shows no intersection.

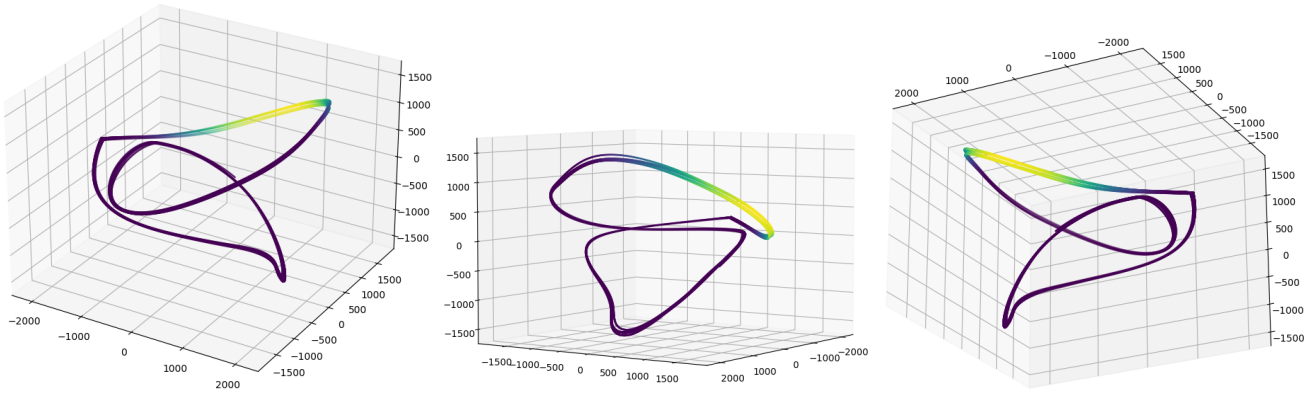


Figure 21: PCA embedding shown from different angulations

Part 2 We created 9 plots of the PCA embedding coloring the points differently depending on the different measurements areas (i.e. columns) of the original data. These are reported in Figure 22. In order to accomplish this, we simply had to pass the different columns as the color parameter to the `scatter` function of `matplotlib`, as shown in Figure **??**.

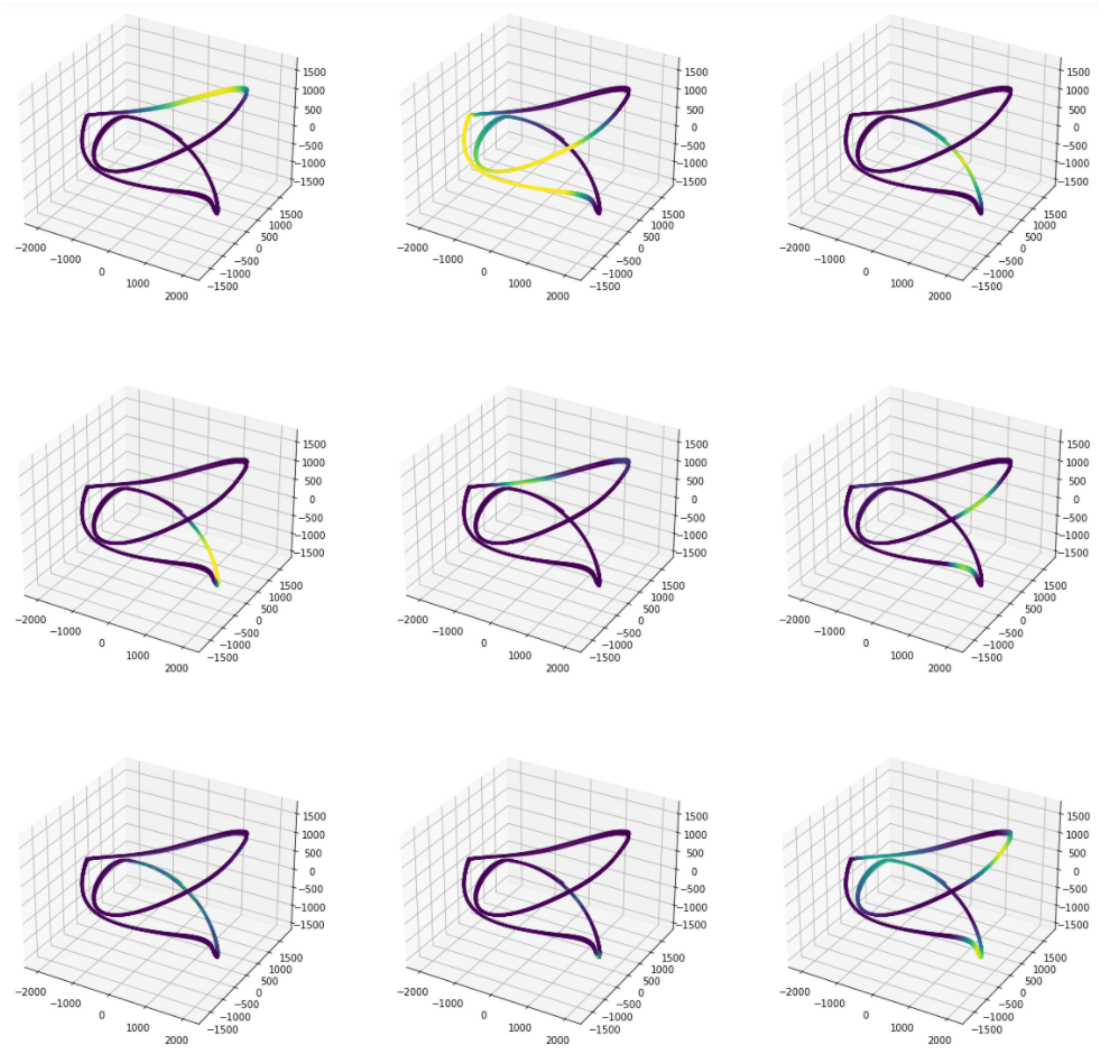


Figure 22: different measurements coloring on the PCA embedding

```
fig, axs = plt.subplots(3,3,figsize=(20,20), subplot_kw=dict(projection='3d'))
for i in range(9):
    axs[i // 3][i % 3].scatter(*x_pca.T, s=1, c=x_original[:M_dataset.shape[0], i])
plt.show()
```

Figure 23: Function to plot the PCA embedding in different colors depending on the different measurement areas of the original data.

Part 3 Now we are required to learn the dynamics on the curve embedded in the principal components, shown in the previous part. To do so we compute the *arclength* of said curve and approximate its change over time, obtaining therefore the speed of *arclength*, i.e. its first derivative in time, which is a vector field on the *arclength* itself. To do so we follow these steps:

- fixate as first point of the arc the first point in the PCA embedding, beginning of the 3D curve.
- iterate through all points (starting from the first point of the arc).
- for each couple calculate the *arclength*: accumulate the L2 norm of consecutive points from the first point up to the `i`-th point.
- divide the *arclength* by `i` (which is the dataset time step, representing also the time passed from starting point to `i`-th point) to get the speed.
- since the data is periodic, cut the vector field to only one period, with result being **Figure 24**.

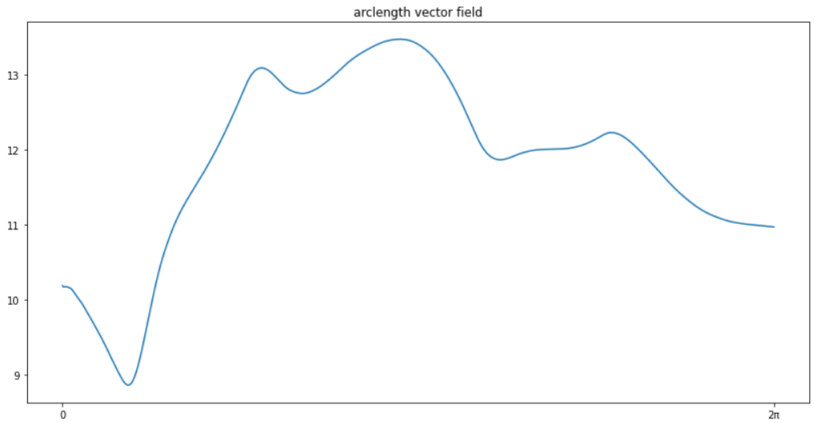


Figure 24: arclength vector field (one full period)

To show the periodicity we can observe **Figure 25**, where the period ending is highlighted by the *red dot*. To find such a period we searched for the nearest point to the starting point (the loop does not exactly pass through the starting point) after a reasonable amount of non-accountable steps. Finding the period let us create such a figure, zeroing the *arclength* accumulation as well as the time passed from the starting point at every period closing.

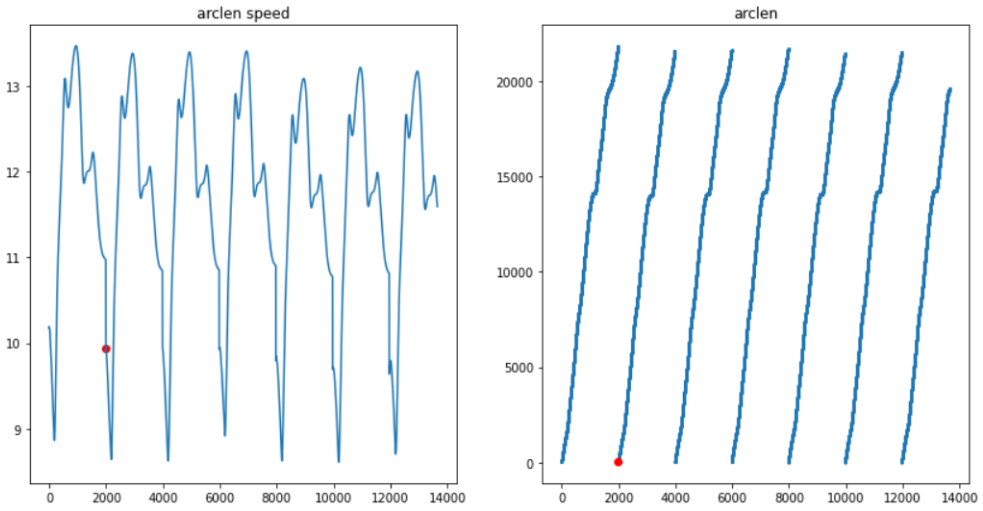


Figure 25: seven times repeating arclength vector field (on the left) and arclength (on the right)

Part 4 As last assignment of the sheet we are required to predict the utilization of the MI building for 14 days, hoping the system learns the curve and correctly estimates future unknown points. To do this we start by taking advantage of the *arclength vectorfield* we created in the last part, although we need a way to integrate it for longer time with respect to only a period of one day. To do so we can approximate the *arclength vectorfield* through the aforementioned **RBF** implementation, treating the arclength data as points and the vectorfield as targets for the least square method.

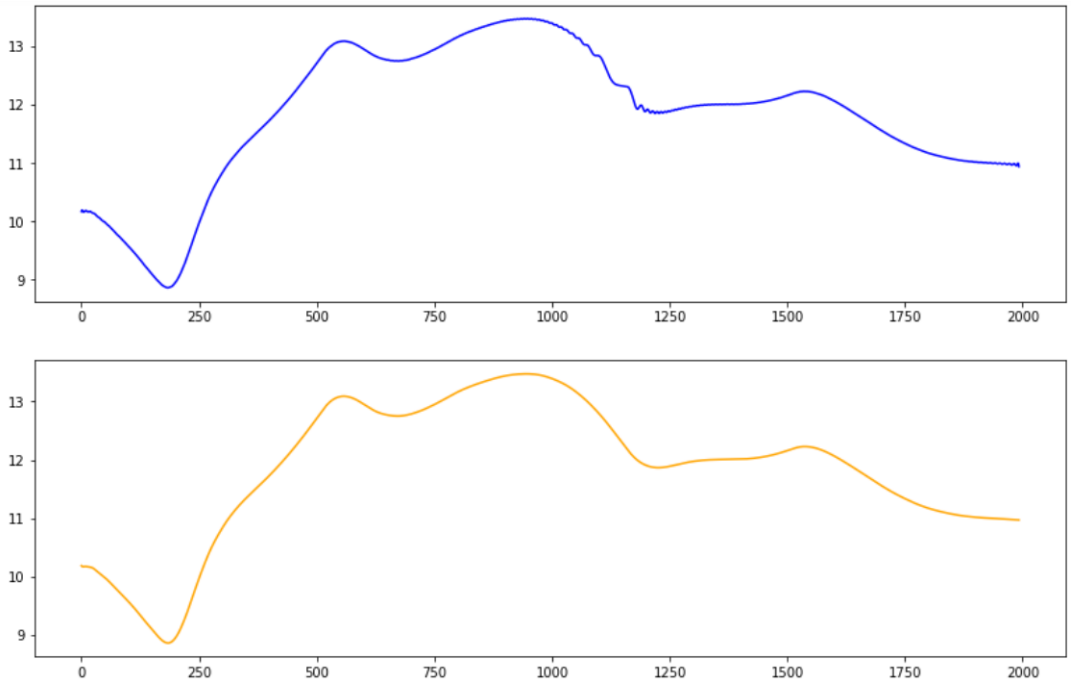


Figure 26: rbf approximation of *arclength vector field* (above) vs the groundtruth (below)

Figure 26 shows the function approximation result. The approximation has been achieved using `eps = 100` and `n_bases = 1000`.

We can continue the path of simulating the MI building utilization by using the *arclength vectorfield* function approximation we just presented, using it to estimate the manifold curve arclength in time. Most importantly we can do so for 14 days, with the results being shown in **Figure 27**.

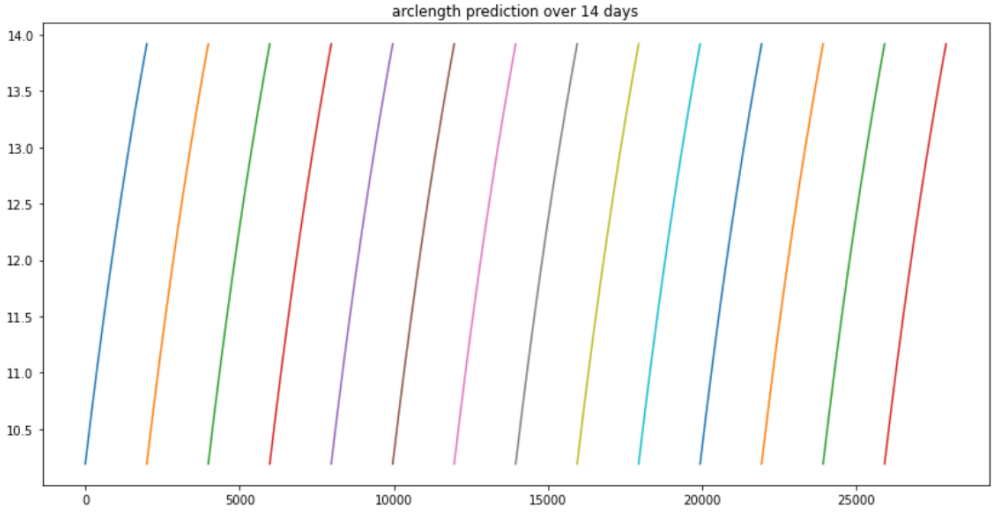


Figure 27: solving the non linearly approximated arclength for 14 days

Once we have the arclength values over time we can map to each arclength the original utilization values since we know how we created them (referring to the time delays) and therefore which arclength corresponds to which utilization value. Having these two paired sets we can construct a radial basis function approximation from arclength space to the utilization value, so we can estimate once again future values. To do so we treat the arclength values as points and the utilization values as targets for the least squares algorithm, effectively finding a function (given by ΦC) which taken an arclength gives the utilization value. With such a function it is now easy to estimate the utilization value for much longer time, as 14 days, as shown in **Figure 28**.

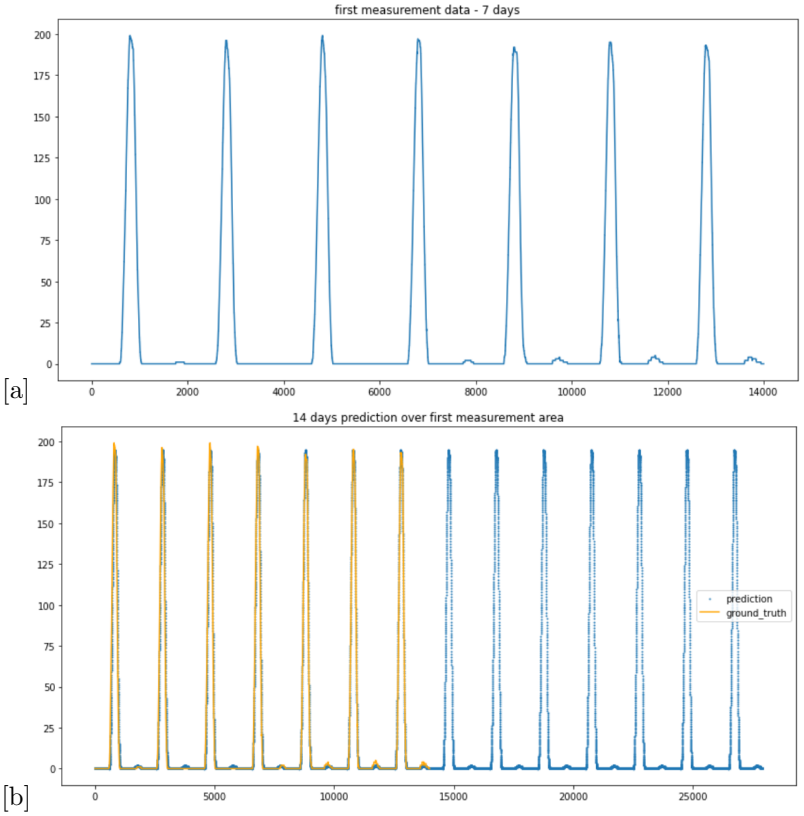


Figure 28: MI building utilization ground truth of 7 days (a) and utilization prediction for 14 days (b)