Using Monte Carlo Markov Chain (MCMC) to identify unknown flux function from observation data

Urszula Maria Starowicz

Abstract

In this project, we address the challenge of identifying unknown flux functions within a traffic flow model using a Monte Carlo Markov Chain (MCMC) approach. The model under consideration describes vehicle density dynamics with a spatially varying resistance coefficient $k(x)$, which significantly influences the traffic behavior. By combining a discrete numerical scheme with MCMC methods, we aim to recover the hidden parameter $k(x)$ from noisy observation data collected at multiple spatial positions. The methodology involves generating an ensemble of parameter realizations and statistically characterizing the posterior distribution through iterative updates based on a random-walk proposal strategy. Simulation results are presented to illustrate how variations in $k(x)$ affect the predicted vehicle densities compared to true observation data, providing insights into the sensitivity and reliability of the proposed inversion technique.

1 Introduction

Traffic flow modeling is essential for understanding and predicting the dynamics of vehicular movement on road networks. In practice, various local factors, such as road conditions and driver behavior, can affect the flow of traffic. These influences are often encapsulated in a spatially dependent flux function $k(x)$, which represents the resistance or facilitation of vehicle motion along the road. However, directly measuring $k(x)$ is challenging, and its exact form is typically unknown.

In this project, we aim to identify the unknown flux function $k(x)$ from observational data of vehicle density collected at discrete positions along a road. The core idea is to combine a numerical scheme for solving the traffic flow model with a Monte Carlo Markov Chain (MCMC) approach. MCMC methods provide a robust statistical framework for parameter estimation, allowing us to generate an ensemble of candidate flux functions and select those that best explain the observed data, despite the presence of measurement noise.

The following sections detail the mathematical model underlying the traffic flow dynamics, the numerical methods employed to solve the governing equations, and the MCMC algorithm used for parameter inversion. Ultimately, our approach seeks to not only recover a plausible form of $k(x)$ but also to quantify the uncertainty in the estimates, thereby enhancing our understanding of the traffic flow behavior under various conditions.

The introduction, and the rest of the project, is based on the knowledge got from the paper *Identification of nonlinear conservation laws for multiphase flow based on Bayesian inversion* [4], *MCMC Methods for Functions: Modifying Old Algorithms to Make Them Faster* [1] and notes from UiO [5].

2 Theory and Methods

In this section, we present the mathematical model governing traffic flow dynamics, described by a nonlinear partial differential equation that incorporates a variable "resistance" coefficient $k(x)$. This coefficient reflects road conditions affecting vehicle speed, and our goal is to estimate its unknown values using observation data on car density at four fixed positions over time. The data is assumed to be noisy, and the parameter estimation is framed as a statistical inverse problem, with the noise modeled as a normal distribution. To solve for the unknown parameters $k(x)$, we employ the Metropolis-Hastings algorithm from the MCMC family, starting with an initial guess and iteratively refining the estimate. We explore the steps of the MCMC method, from generating candidate solutions to computing the acceptance probability based on the fit to observed data. Finally, we summarize the key assumptions and setup used for the parameter estimation process.

2.1 Mathematical Model

In this project we will look into the problem given by following traffic flow model:

$$u_t + (k(x)f(u))_x = 0, \quad \text{for } x \in (0, L), \tag{1}$$

subject to the initial condition:

$$u(x, t = 0) = u_0(x), \tag{2}$$

and the Neumann boundary conditions:

$$u_x|_{x=0} = u_x|_{x=L} = 0 \tag{3}$$

In our model we will use:

- The function is $f(u) = u(1 - u)$
- "Resistance" coefficient $k(x) \in [0.25, 1.75]$
- The considered domain is $x \in [0, L]$, where $L = 4$
- The time period is $t = [0, T]$, where $T = 10$

Before more assumption will be presented, lets explain some more about this model. It is, as mentioned, a simple traffic flow model, but with a twist, that makes it a little less simple. The complicated part it hidden in $k(x)$ "variable". It's purpose is to reflect different circumstances on the road, that cause changes in the speed of the vehicles - if $k(x) > 1$ the flow is faster and if $k(x) < 1$ the flow is slower. But in this project the $k(x)$ is unknown and we will try to guess it using the time series of provided data of observed density of cars at four positions X_i over time period $[0, T]$. This means that $k(x)$ is shown as a piecewise linear function on x. It is represented by the values $(x_i, k_i)_{i=0}^{i=10}$. It is a uniform grid on x of eleven points. Moreover, we assume that $k(x_0) = k(x_{10} = 4) = 1$. Hence, we finally have the task only to identify the unknown values $k_{i\,i=1}^{\,9}$. The plot below shows a random representation of $k(x)$ values.

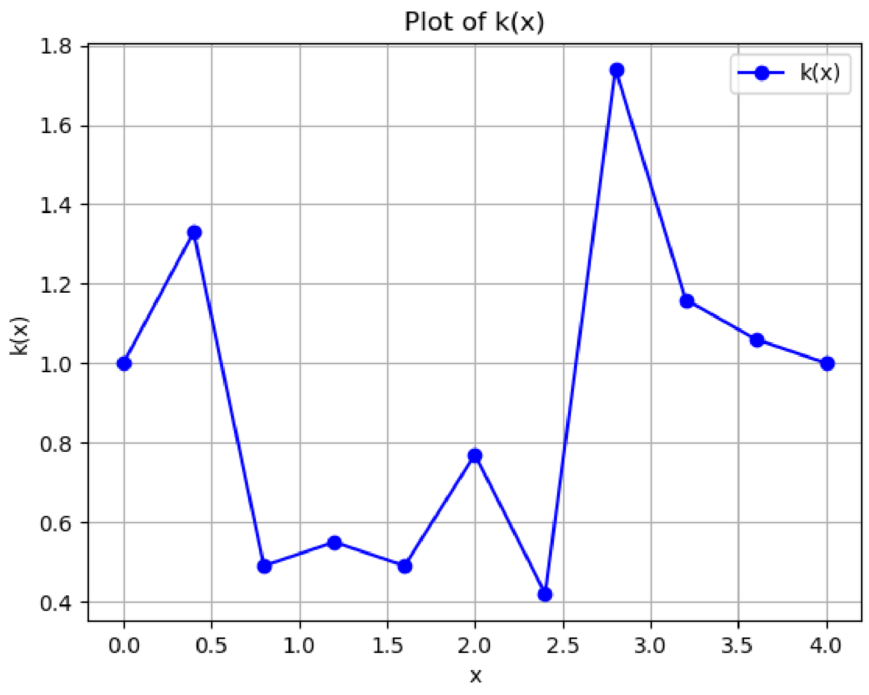

Figure 1: Random $k(x)$ values representation/

$$u(X_i,t)_{i=1}^4 = d$$

In the equation above we described the function for the single data point, but we have the assume that the data is noisy. To visualize this let use this relation with an unknown parameter vector θ - our $k(x)$, to be even more precise $\theta \approx k_i{}_{i=1}^9$ - and the solution operator associated with the mathematical model of the main problem h:

$$d = h(\theta) + \eta \tag{4}$$

The last to explain is the η, which there represents the noise/measurement error associated with the observation data and hidden parameter.

Given the explanation we can dive further into the assumptions:

- The noise η is given as a normal distribution $N(0, \gamma^2 I)$. This notation means that in the normal distribution the mean is 0 and the standard deviation is $\gamma = 0.03$ and I is the identity matrix. The covariance matrix for the measurement error η is $\Gamma = \gamma^2 I$.
- Knowing that, the next assumption is that our predicted observation data $h(\theta) \approx d$ is given by:

  $$h(\theta) = [u_\theta(X_1,t_j), u_\theta(X_2,t_j), u_\theta(X_3,t_j), u_\theta(X_4,t_j)]_{j=0}^N \approx d^{obs}$$

  where $t_j{}_{j=0}^N$ are the discrete times of observation $t_0 = 0$, $t_1, ..., t_{N-1}$, $t_N = T$. We will use a uniform distribution of t_j and values for the variables: $N = 40$, $t_0 = 0$ and $t_N = 10 = T$. Our grid will be set to $M = 600$ grid cells and points were chosen to be:

  $$X_1 = 0.75, X_2 = 1.5, X_3 = 2.25, X_4 = 3.25$$

That represents all of the assumptions made to create our mathematical model. To sum up, we will use provided observation data to guess the $k_i{}_{i=1}^9$, that will be our $k(x)$ in the equation (1). Also we set values to different, necessary variables and assume that the noise is represented by normal distribution. This section was based on the knowledge from lectures and introduction to the project [3].

2.2 MCMC method

To identify the unknown parameter vector θ of the length m in $d = h(\theta) + \eta$ we will use the Methropolis-Hastings method. This means that we will start with an initial guess/distribution/a prior:

$$\theta^{(0)} \approx N(\mu, \sigma^2 I) \tag{5}$$

which is a random normal distribution with mean μ and covariance matrix $C = \gamma^2 I$. As said before I is the identity matrix of size $m \times m$ and m is the length of the parameter vector θ. To get the statistical characterization of that vector, that will provide best representation of the provided data, we will use MCMC algorithm. In mathematical words we will look for a characterization of posterior distribution

$$\theta_* \approx N(\mu_*, {\sigma_*}^2 I).$$

Generating an ensemble of N_{ens}, we will get the estimation of μ_* and σ_*.

The MCMC algorithm applied in this project is biased on a random walk , where we will start with an initial guess $\theta^{(0)}$ and step after step we compute next $\theta^{(k)}$ parameters. The generation of the updated parameter vector is corrected by the mismatch (Loss) between predicted observation $h(\theta^{(k)})$ and the true observation data $d = d^{obs}$.

$$Loss_k \approx ||h(\theta^{(k)} - d||^2 \tag{6}$$

Lets keep in mind that the above equation is what we want to optimize and get a closer look at the algorithm. As said, it is a random walk which purpose is to find a characterization of θ.

Step 1

We will start with setting $k = 0$ and selecting $\theta^{(0)} \approx k_i{}_{i=1}^9$ from a priori distribution $N(\mu, \sigma^2 I)$, where the mean $\mu = 1$ and standard deviation $\sigma = 0.25$. Our $k(x)$ is in the range $[0.25, 1.75]$. We will also make an assumption that θ is in the range $[0.5, 1.5]$ to ensure that generated values for $\theta^{(0)}$ will be probable. Having generated the first set of $k(x)$ we go on to the next step.

Step 2
For the θ from the previous step we propose the ϕ, that is a sightly changed vector θ.

$$\phi^{(k)} = \sqrt{(1-\beta^2)}\theta^{(k)} + \beta\xi^{(k)} \tag{7}$$

Above equation represents the way that the vector θ will be change in the algorithm every time we will be computing the step 2. There $\xi^{(k)} \approx N(0, \sigma^2 I)$ and β values are to be tested and we will start with $\beta \approx 0.05$.

Step 3
This step, I would say, is the hearth of this algorithm and is based on commutating $a(\theta^{(k)}, \phi^{(k)})$. This requires to compute $\Phi(\theta^{(k)})$ and $\Phi(\phi^{(k)})$. But lets explain what what is happening. Firstly, we should get a closer look at what is a.

$$a(\theta, \phi) = min[1, e^{\Phi(\theta)-\Phi(\phi)}]$$

In the equation above the $e^{\Phi(\theta)-\Phi(\phi)} = \frac{P_\phi}{P_\theta}$. The probability of those two vectors are calculated by combining two equations (4) and (5):

$$\eta = d - h(\theta) \approx N(0, \gamma^2 I)$$

$$P(\eta = d - h(\theta); N(0, \gamma^2 I)) = \frac{1}{\sqrt{2\pi\gamma^2}} e^{-\frac{1}{2}\frac{(d-h(\theta))(d-h(\theta)^T)}{\gamma^2}}$$

In the equation below we used $||a||^2 = aa^T$ is the means square error where a is a column vector and a^T is a transpose.

$$\frac{1}{2}\frac{||d - h(\theta)||^2}{\gamma^2} = \Phi(\theta)$$

$$\frac{1}{\sqrt{2\pi\gamma^2}} e^{-\Phi(\theta)} = P_{(\theta)}$$

And the same we have for the ϕ:

$$P(\eta = d - h(\phi); N(0, \gamma^2 I)) = \frac{1}{\sqrt{2\pi\gamma^2}} e^{-\Phi(\phi)} = P_{(\phi)}$$

Having understood the origins of the computation of the loss function, we can see that in the algorithm the better choice is the highest probability. This also means that it will give the best fit to the observation data.

- If $\frac{P_\phi}{P_\theta} \geq 1$ better choice is ϕ because it also accounts for:

  $$||d - h(\phi)|| \leq ||d - h(\theta)||$$

  Then the algorithm update θ is becoming ϕ for the next iteration.

- In other case, when $\frac{P_\phi}{P_\theta} \leq 1$ better choice is θ, so it will not be change.

Basically this means that we have to compute $h(\theta^{(k)}$ and $h(\phi^{(k)}$ and acquire from it predicted data.

Step 4
Next, we will draw a number $i^{(k)}$ in $[0, 1]$ from a uniform distribution. If $i^{(k)} \leq a(\theta^{(k)}, \phi^{(k)})$, then we set $\theta^{(k+1)} = \phi^{(k)}$.

Step 5
Set $\theta^{(k+1)} = \theta^{(k)}$

Step 6
$k \to k+1$

Variable	Values	Description
x	$\in [0,4]$	Domain.
t	$\in [0,10]$	Time series.
N	40	Amount of the observation in time t.
M	600	Number of grid cells.
γ	0.03	Standard deviation for generation of noise η.
η	$N(0,\gamma^2 I)$	Noise of the data.
σ	0.25	Standard deviation for generation of $k(x)$.
μ	1	Mean for generation of $k(x)$.
$k(x)$	$\in [0.25, 1.75]$	The range for generating $k(x)$.
$k_{i\,i=1}^{\,9}$	$N(\mu,\sigma^2 I)$	Generated $k(x)$ values.
β	0.05	Value that will perturb the vector θ.

Table 1: Table with the summary of different starting values.

3 Problem

This section delves into solving optimization problems using MCMC algorithms for parameter identification in complex systems. We begin by evaluating how the use of one dataset versus two impacts the accuracy of the predicted observations and the resulting identification of $k(x)$. Visual comparisons between predicted and true observations form a key part of our analysis, highlighting the influence of data selection. The discussion extends to whether the $k(x)$ obtained from a single dataset differs significantly from that identified using dual datasets, and which might be more reliable when the loss function performs well. We also explore the critical role of the initial parameter vector $\theta^{(0)}$ and propose strategies to minimize its influence. Additionally, the choice of the parameter β and its effects on the learning process are examined. The impact of numerical diffusion, particularly variations in the M factor in the Engquist-Osher scheme flux, is scrutinized for its effect on the algorithm's performance. Finally, the section concludes with suggestions for improving the identification of $k(x)$ and refining the overall optimization strategy.

3.1 Task 1

In this section, we extend our previous work on the conservation law $u_t + f(u)_x = 0$ [6] by incorporating a spatially varying coefficient $k(x)$ into the numerical scheme. Building upon the Engquist-Osher method [2], we modify the discrete formulation to account for the heterogeneity introduced by $k(x)$. The adapted scheme takes the form that integrates the effects of a non-constant diffusion coefficient into the numerical flux. This modification enables a more precise simulation of the system's dynamics, particularly in capturing variable diffusion behavior. By slightly adapting our previous scheme, we ensure that the local variations of $k(x)$ are properly reflected in the solution. Overall, this approach provides a robust framework for both analysis and comparison with observed data.

a) **Discrete scheme to solve the main problem**

Lastly in the previous project task explanation [6] we explore the problem $u_t + f(u)_x = 0$, which differs from our current problem only with the $k(x)$. Thanks to that we can just change a bit our previous numerical scheme - The Engquist-Osher scheme described in the theory for the previous project [2]:

$$u_j^{n+1} = u_j^n - \frac{\Delta t}{\Delta x}(F_{j+\frac{1}{2}}^n - F_{j-\frac{1}{2}}^n),$$

where $F_{j+\frac{1}{2}}$ is a numerical flux. Now extending this scheme according to the hint from the project tasks [3] by the generated $k(x)$, we will receive:

$$\frac{u_i^{n+1} - u_i^n}{\Delta t} + \frac{1}{\Delta x}(k(x_{i+\frac{1}{2}})F^n_{i+\frac{1}{2}} - k(x_{i-\frac{1}{2}})F^n_{i-\frac{1}{2}}) = 0 \tag{8}$$

where $k_{j+\frac{1}{2}} = k(x_{j+\frac{1}{2}})$ and $F^n_{i+\frac{1}{2}} = \frac{1}{2}(f_j + f_{j+1}) - \frac{M}{2}(u^n_{j+1} - u^n_j)$.

b) **Numerical solution with given initial data**

The initial condition $u_0(x)$ is defined as a piecewise function over the domain $x \in [0, 4]$:

$$u_0(x) = \begin{cases} 0.2, & x \in [0, 0.5] \\ 0.4, & x \in (0.5, 1.5] \\ 0.6, & x \in (1.5, 2.5] \\ 0.7, & x \in (2.5, 3.5] \\ 0.4, & x \in (3.5, 4]. \end{cases} \tag{9}$$

The $k(x)$ is generated randomly according to a normal distribution:

$$k_i \sim \mathcal{N}(\mu, \sigma^2 I), \quad \text{with } \mu = 1, \sigma = 0.25.$$

The values k_i are constrained within $[0.25, 1.75]$, with boundary conditions:

$$k(0) = k(4) = 1.$$

The more precise description is in the Mathematical Model section in this project.

The code computes the numerical solution $u(X_i, t_j)$ at specific spatial points X_i and discrete time steps t_j:

- Spatial observation points:

$$X_1 = 0.75, \quad X_2 = 1.5, \quad X_3 = 2.25, \quad X_4 = 3.25.$$

- Time grid:

$$t_0 = 0, t_1, \ldots, t_N = T, \quad \text{where } N = 40, \quad T = 10.$$

The problem is solved on a grid with $M = 600$ spatial cells to ensure accuracy.

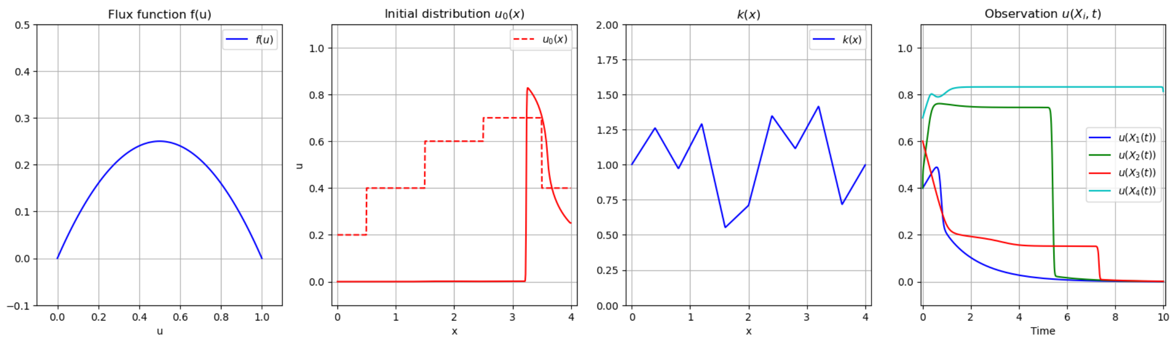

Figure 2: Output of the simulation after *time* = 10

On the figure above we can see that the plot of flux function, initial distribution and distribution after $T = 10$, currently generated $k(x)$ and observation of the points $u(X_1, t_j), u(X_2, t_j), u(X_3, t_j), u(X_4, t_j)$, over time.

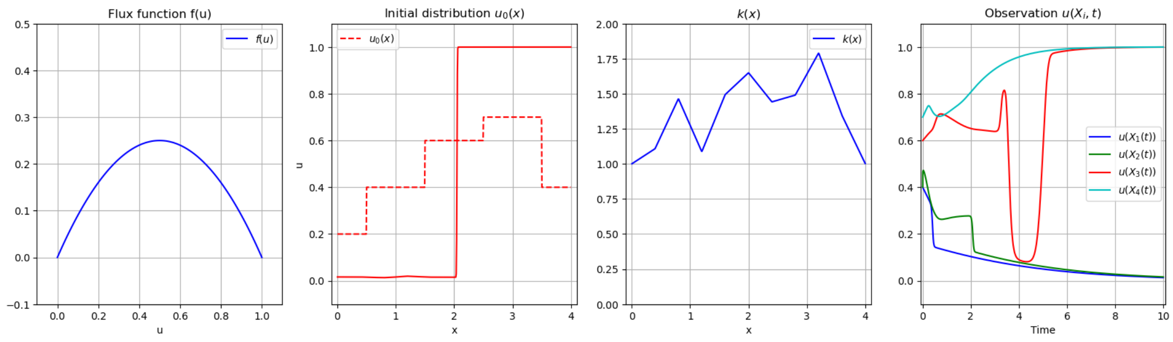

Figure 3: High $k(x)$ values.

c) **Numerical solution with an example initial data**

As other examples were run different values of $k(x)$. All of them with the same $\sigma = 0.25$, the same initial distributions and flux functions. The first one with the high vales of $k(x) \approx N(1.5, \sigma^2 I)$ and $k(x) \in [1, 2]$.

The high $k(x)$ leads to faster diffusion, smoothing out distribution more quickly.

- With a larger diffusion coefficient, sharp gradients in the initial condition are quickly smoothed.
- The solution curves at different times flatten more rapidly and converge to a uniform profile.
- Physically, this behavior represents a medium where heat or mass spreads quickly, making $u(x, t)$ more homogeneous in a shorter time.

Next one is with the low values of $k(x) \approx N(0.5, \sigma^2 I)$ and $k(x) \in [0, 1]$.

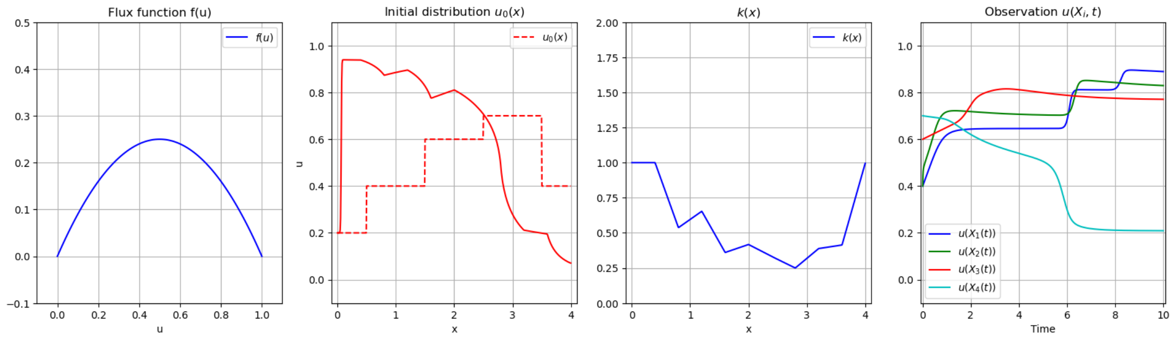

Figure 4: Low $k(x)$ values

Here, $k(x)$ is sampled around a lower mean within an interval $[0, 1]$, making diffusion is weaker.

- Sharp gradients or step-like features in the initial condition persist longer.
- Smoothing is delayed, leaving noticeable differences between adjacent spatial regions.
- Physically, this represents a medium that does not conduct heat or spread mass efficiently.

The last one is in the same range as in the provided $k(x) \in [0.25, 1.75]$, but for generating $k(x)$ was used uniform distribution instead of normal.

In this scenario, $k(x)$ is sampled uniformly. The subplots show a piecewise-linear interpolation that does not cluster around a single value but is instead spread more evenly across the domain.

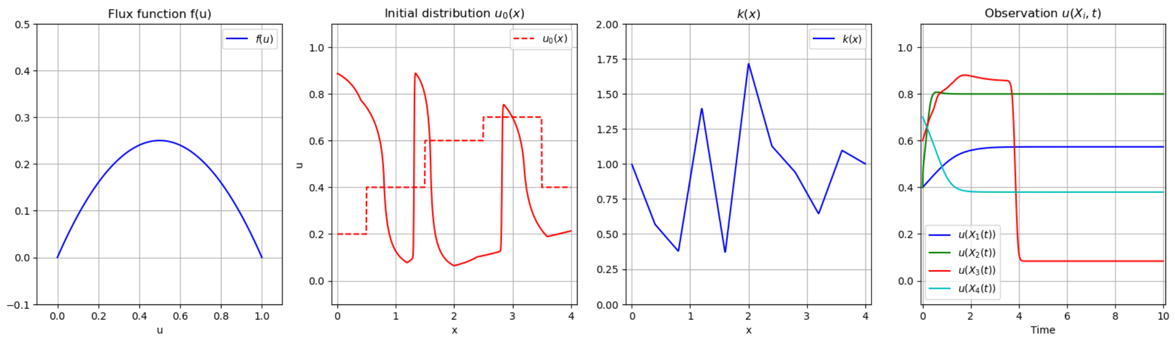

Figure 5: Uniform distribution of $k(x)$

- The solution behavior is intermediate between the high and low $k(x)$ extremes.
- Regions where $k(x)$ is closer to 1.75 diffuse more quickly, while those near 0.25 diffuse more slowly.
- Overall, the solution smooths out but not as rapidly as in the high-$k(x)$ case, and not as slowly as in the low-$k(x)$ case.

To sum up, different versions of the $k(x)$:

- Smoothing Rate:
  - High $k(x) \to$ rapid smoothing of $u(x,t)$.
  - Low $k(x) \to$ slower smoothing, preserving gradients.
- Spatial Heterogeneity:
  - When $k(x)$ varies significantly, different regions of the domain diffuse at different rates.
  - This can lead to multi-regime patterns and wave-like spreading.
- Long-Term Behavior:
  - Over long times, all solutions eventually become smoother.
  - The timescale of this smoothing is heavily influenced by the magnitude and distribution of $k(x)$.

From a numerical stability perspective, explicit schemes require careful handling of $k(x)$; very high values may necessitate smaller time steps to ensure stability, whereas low values can lead to slower convergence. Additionally, designing experiments with various distributions of $k(x)$—including high, low, and uniformly distributed values—allows for evaluating the system's sensitivity to diffusion parameters, which is particularly important for uncertainty quantification and inverse modeling.

Overall, the figures demonstrate that the choice and range of $k(x)$ directly determine the rate of diffusion in the PDE, leading to substantially different solution profiles over the same time horizon.

d) **Numerical solution with provided observation data**

The data file provided by the teacher contains, among other parameters, an observation dataset, where $d = d_{obs} = \{d_1(t_j), d_2(t_j), d_3(t_j), d_4(t_j)\}$. Each d_i is associated with position X_i as described in Section 1.2.1, and the initial data (16) represents the initial distribution of vehicles. The script loads the dataset, extracts the time-series values for d_1, d_2, d_3, and d_4, and visualizes them as four distinct time series corresponding to the four positions X_1, X_2, X_3, and X_4. The x-axis represents observation times, while the y-axis represents the corresponding values of d_i.

On the figure above is plotted representation of the provided real observation data in points:

- $D1 \to u(X_1, t_j)$
- $D2 \to u(X_2, t_j)$

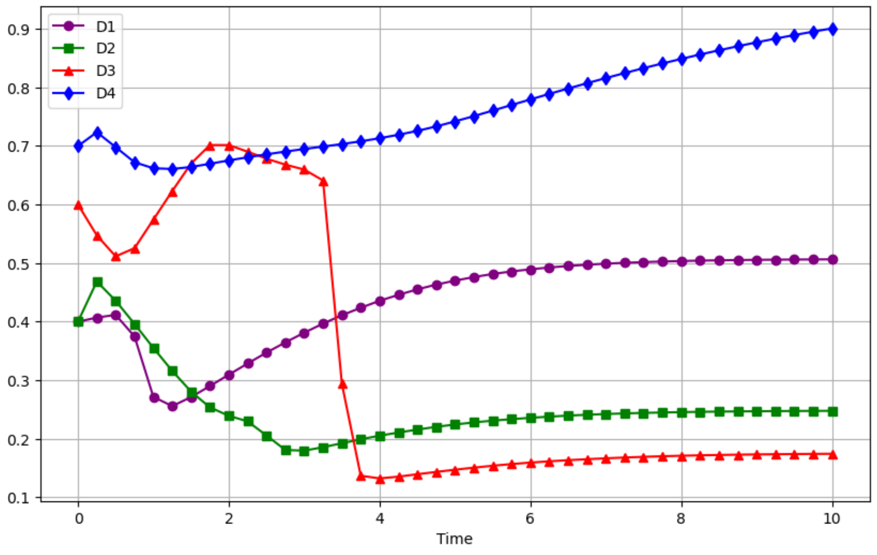

Figure 6: Representation of the provided data

- $D3 \to u(X_3, t_j)$
- $D4 \to u(X_4, t_j)$

e) **Randomly generated $k(x)$ vs the observation data**

In this subsection we will look into three different generation of $k(x)$ and try to compere them to the observed data. Previously in this project we saw that the $k(x)$ have huge impact on the changes in the distribution and observation predicted data. We have explained how the various $k(x)$ generations is changing diffusion and what happens with initial conditions.

Now we will compere predicted data, generated by three different $k(x)$, to the provided data.

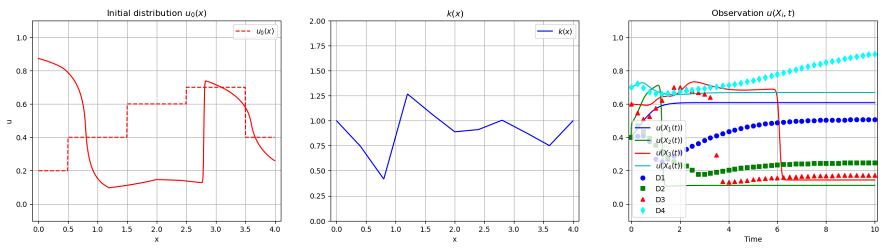

Figure 7: First generation of $k(x)$

.

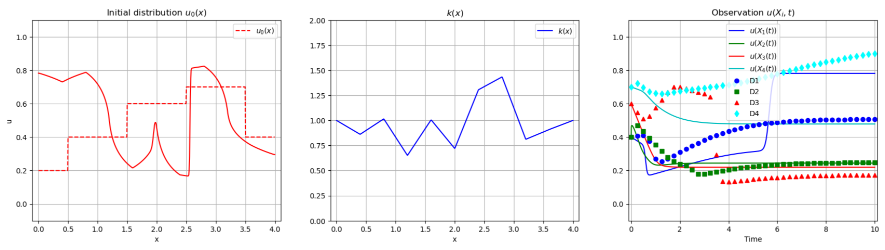

Figure 8: Second generation of $k(x)$

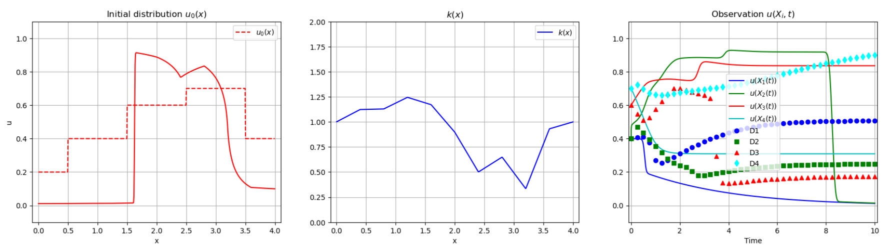

Figure 9: Third generation of $k(x)$

From the plots, each randomly generated $k(x)$ alters how the solution evolves and how it matches (or fails to match) the observed data. Each $k(x)$ controls how quickly or slowly the wave propagates and how steep or diffused the wave-fronts become. If $k(x)$ is higher in certain regions, it can enhance diffusion (or flux), causing the solution to spread more or move faster in those locations. Even though the same initial conditions are used, differences in $k(x)$ lead to distinct solution profiles over time. The wave may arrive at observation points earlier or later, and the peak amplitudes can differ, simply because $k(x)$ changes the local transport/diffusion rate.

The colored lines in the observation plots show how the predicted values at specific points (e.g., $X_1, X_2, \ldots$) evolve over time. With a different $k(x)$, the timing and magnitude of these observation curves can vary significantly.

In some cases (e.g., second generation of $k(x)$), the predicted data may track the provided data more closely. In other cases (third generation), the solution deviates more—reflecting that the particular random $k(x)$ does not match the real (or reference) behavior.

Some $k(x)$ realizations appear to produce a solution close to an "average" or "mean" behavior, while others differ substantially. The amplitude and phase of the solution's waveforms, as well as how sharply they rise or fall, are all tied to local values of $k(x)$.

Overall, each distinct shape of $k(x)$ changes how the wave propagates and diffuses through the domain. This results in different solution profiles and different agreement levels with the measured or provided data.

In conclusion, there exists a $k(x)$ that can predict the provided data better than shown ones. To do this, in the Task 2, we will use MCMC algorithm, described in the theory part od the project.

3.2 Task 2

In this task, we implement a Monte Carlo Markov Chain (MCMC) algorithm to optimize the spatially varying parameter $k(x)$ from our previous simulation. The algorithm begins by randomly generating an initial parameter vector $\theta^{(0)}$ and then iteratively updates it over 1,000 iterations, extracting a new realization every 20 iterations. As iterations progress, the $k(x)$ function stabilizes, indicating convergence toward an optimal solution. Subsequent experiments adjust parameters such as σ and β to enhance convergence and prevent the algorithm from becoming trapped in local optima. These modifications yield progressively lower loss values, demonstrating the effectiveness of the MCMC approach in fine-tuning $k(x)$.

a) **MCMC algorithm**

This task will be about implementing the Monte Carlo Markov Chain algorithm to the simulation from the simulation from the previous task. More about the algorithm is written in the section Theory and Methods.

It will be started by randomly generating $k(x)$ to represent the initial parameter vector $\theta^{(0)}$, which is represented below.

Next, the simulation will be run $1,000$ times ($k = 1, \ldots, 1000$), extracting every 20 the generated value of $\theta^{(k)}$. This process results in an ensemble of 50 realizations of the parameter vector $\theta = \{k_i\}_{i=1}^{9}$. On the figures below was shown couple of the iterations.

Around this iteration the $k(x)$ function stopped changing and was left the same to the end. This means that the any other generated $k(x)$ for over 500 times did not produce smaller loss.

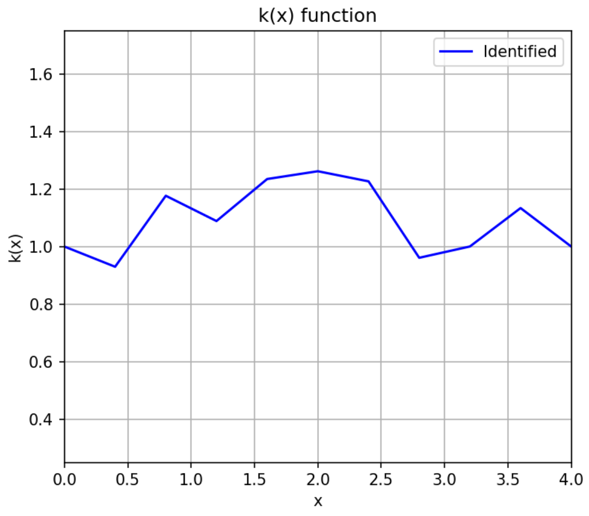

Figure 10: First generated $k(x)$

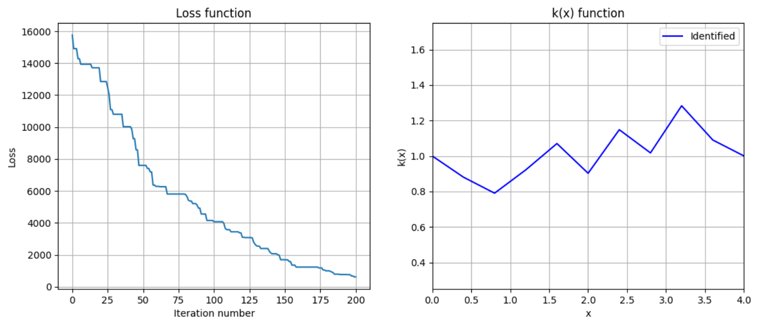

Figure 11: Loss function and $k(x)$ function after 200 iterations.

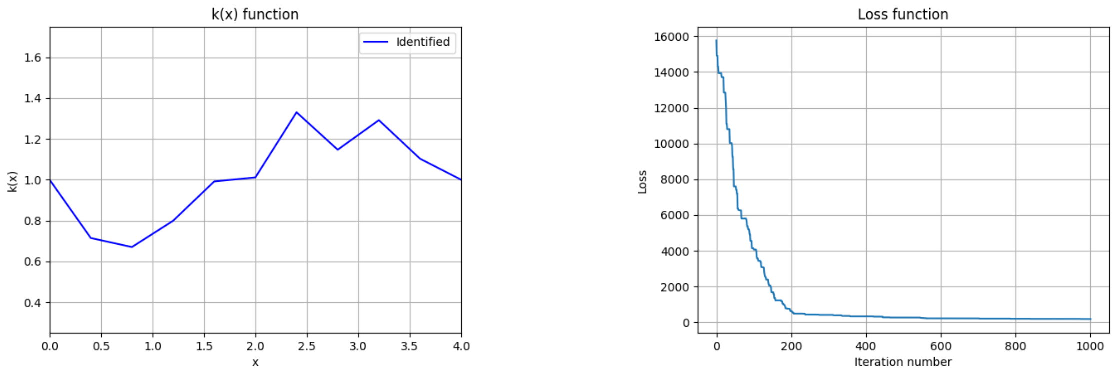

Figure 13: Simulation after 1000 iterations

The loss function is defined by:

$$a(\theta, \phi) = \min\{1, \exp(\Phi(\theta) - \Phi(\phi))\}$$

where

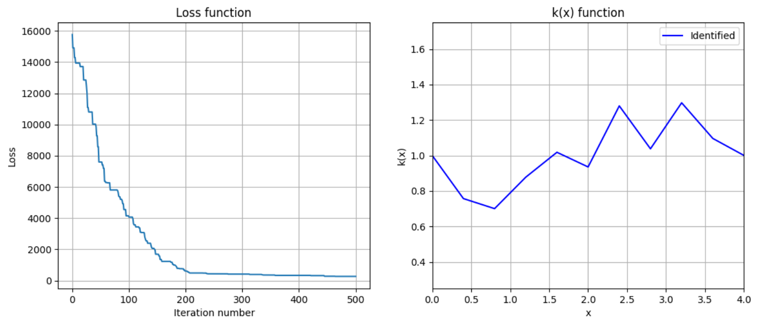

Figure 12: Loss function and $k(x)$ function after 500 iterations.

$$\text{Loss} = \Phi(\theta) = \Phi(\theta; d) = \frac{1}{2}\|\Gamma^{-1/2}(h(\theta) - d)\|^2 = \frac{1}{2}\frac{(h(\theta) - d)(h(\theta) - d)^T}{\gamma^2}.$$

and at the end had value about 655.

After that we will try to start with the better starting $k(x)$ - not randomly generated, but the function that was in the 1000 iteration of the simulation. The results are shown below.

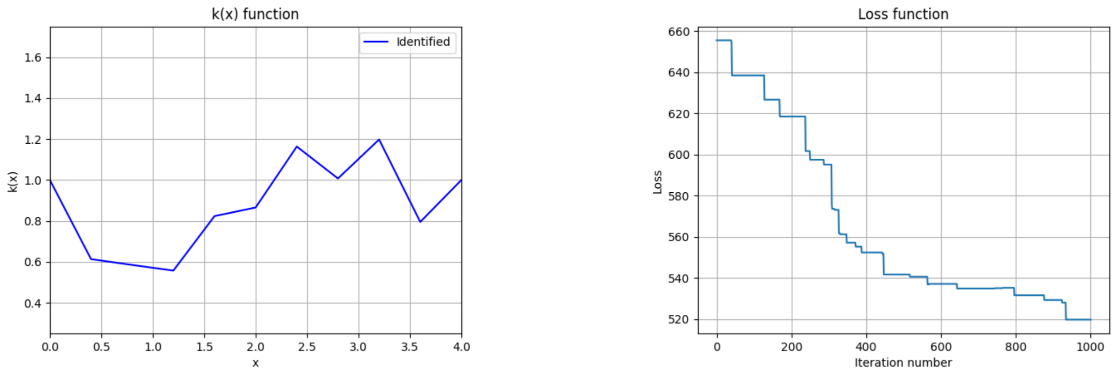

Figure 14: Simulation after 2000 iterations

As we can see the $k(x)$ function did not change a lot. But what happened is that looking at the loss function we can see better that the algorithm is still finding $k(x)$ that are generating data closer to the observed ones. Considering the loss function with those parameters we got from around 655 to the 520.

We will to the same once again - start the algorithm with the $k(x)$ function that was loaded from the end of the previous simulation.

For the third time the algorithm does not have as impressive correction of the loss function and sometimes got the even worse value than the in the previous iterations. But still the algorithm got a bit better value of the loss function - 460.

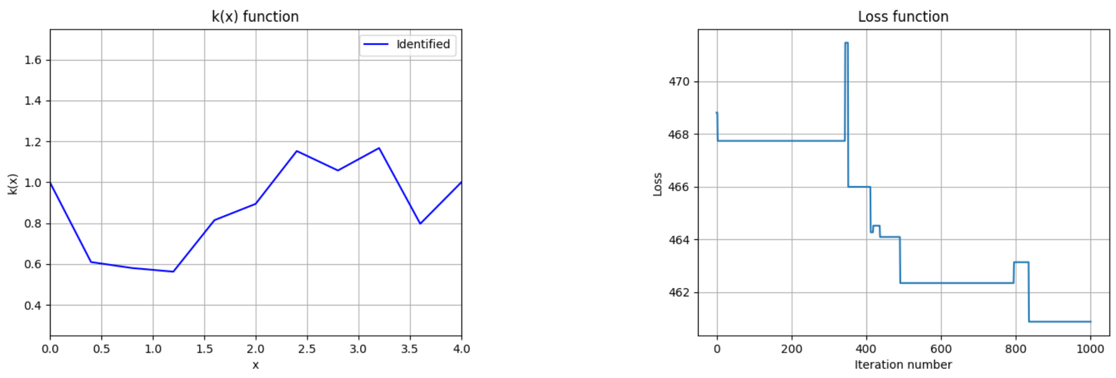

Figure 15: Simulation after 3000 iterations

This time also the difference within the plot of the $k(x)$ is barely visible and we can analyze the results only on the loss function figure. The example above was carried out with the default values given by the professor. To 'play' with the algorithm to maybe get different results we should change values of$\sigma = 0.25$ from the equation (5) and $\beta = 0.05$ from the equation (7). Bigger values may make the algorithm to work faster - check bigger range of $k(x)$ functions.

First to check different variations of the parameters we will try to increase the $\sigma = 0.5$ and $\beta = 0.05$ will stay the same. This should make the algorithm to generate more different $k(x)$ function because we increased the standard deviation in the normal distribution for generating it.

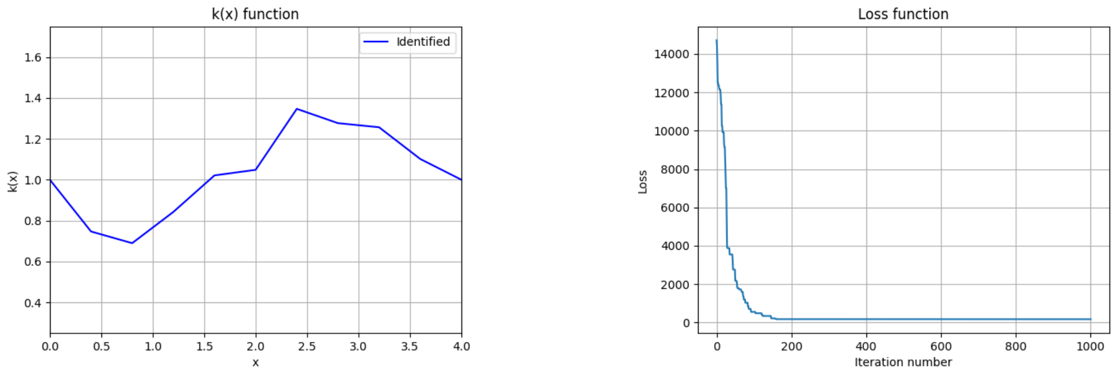

Figure 16: Simulation after 1000 iterations with $\sigma = 0.5$

Using $\sigma = 0.5$ after 1000 iterations we achieve the value of the loss function , which is significantly better than the final value of the loss function with $\sigma = 0.25$.

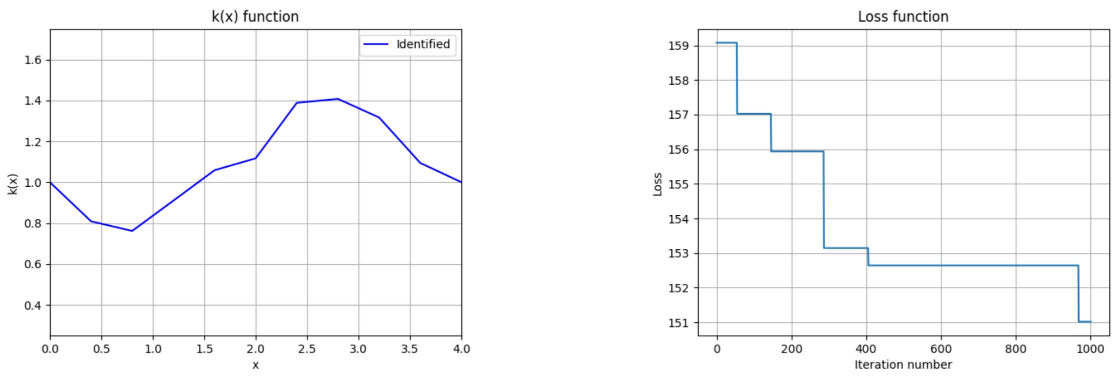

Figure 17: Simulation after 3000 iterations with $\sigma = 0.5$

As we can see above the $k(x)$ function does not visibly changed, but on the figure below we see that the loss function was decreased to about 151.

The same experiment we would run with big $\beta = 0.25$ and $\sigma = 0.25$ from the original example. Below are shown figures with plots of the simulation after 1000 iterations.

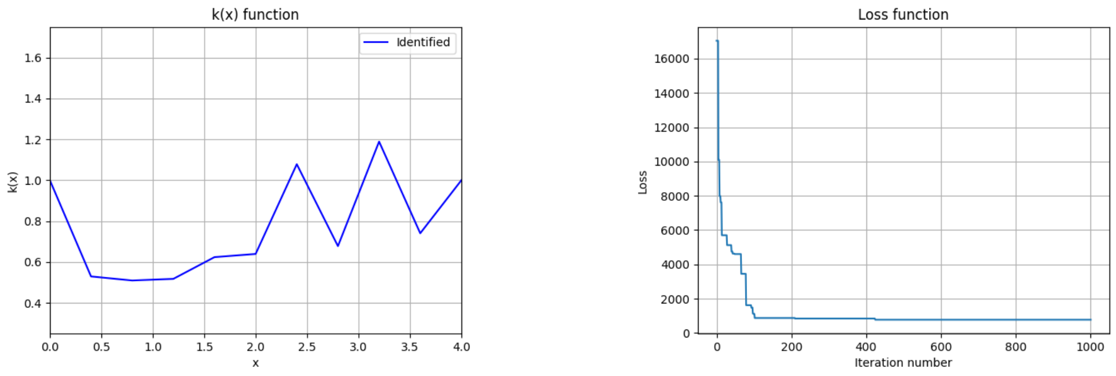

Figure 18: Simulation after 1000 iterations with $\beta = 0.25$

Here the algorithm reached the loss function of value 775, which is the worst it was generated up to this point.

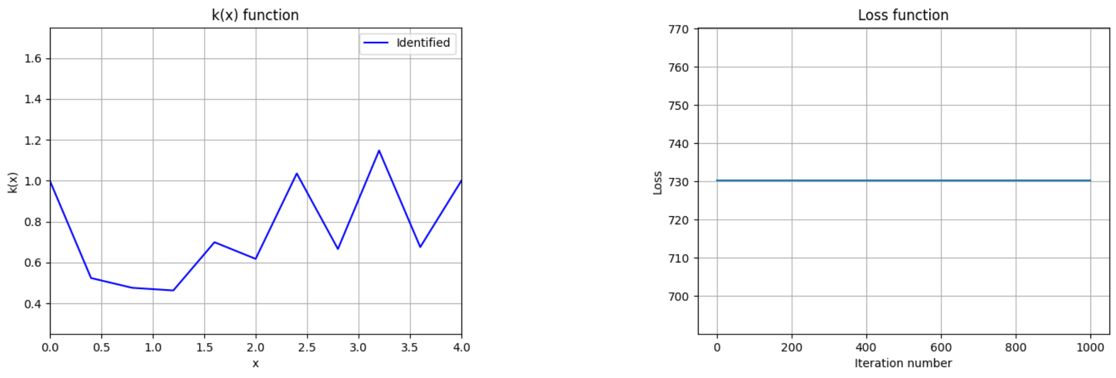

Figure 19: Simulation after 3000 iterations with $\beta = 0.25$

On the figures above, that show results after 3000 iterations, we see that the algorithm decreased the loss function between iterations 1000 and 2000, but after that likely fall into the local optimum and cannot get out of it. At the end the loss function reach only 730.

Next option is to try both parameters bigger than suggested by professor. Given experience from the previous cases, we will choose $\sigma = 0.45$ - close to the previous value, but a bit smaller to balance bigger β. Knowing that $\beta = 0.25$ was unsuccessful and given that σ will be bigger, we will set $\beta = 0.1$. Starting once again with randomly generated $k(x)$ function, we achieve loss function after 1000 iteration to be 167, which is the best result up to this point.

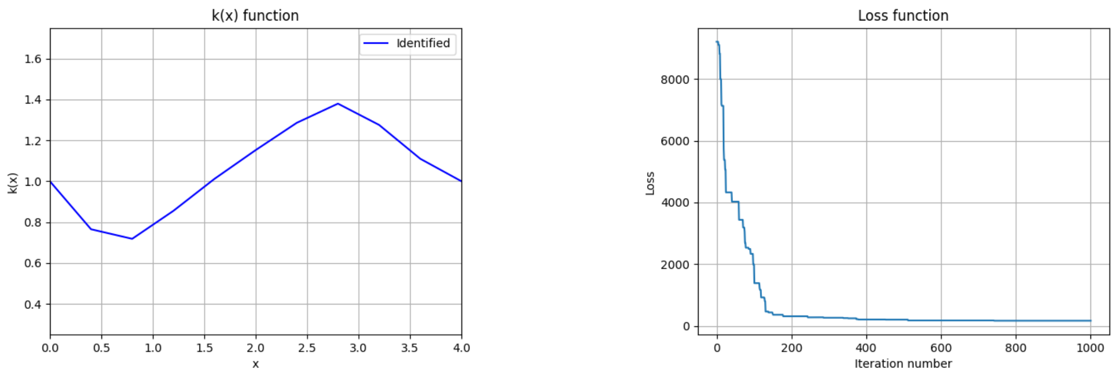

Figure 20: Simulation after 1000 iterations with $\beta = 0.1$ and $\sigma = 0.45$

And final results, after 3000 iterations, are shown below.

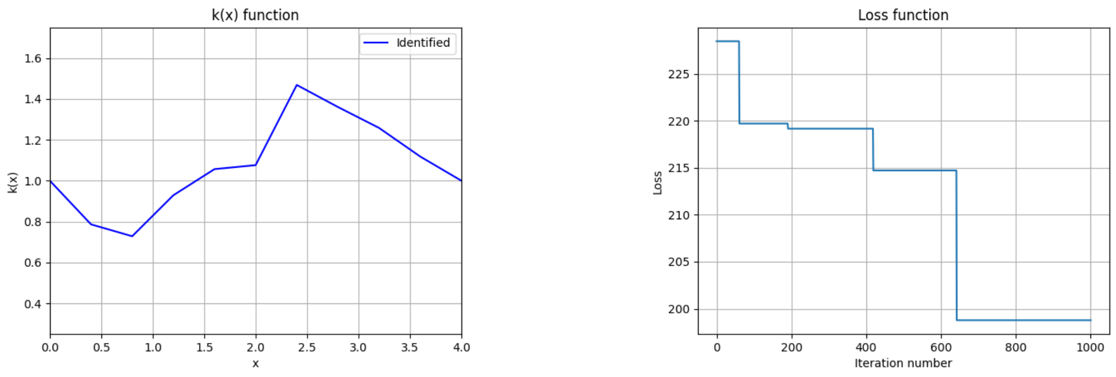

Figure 21: Simulation after 3000 iterations with $\beta = 0.1$ and $\sigma = 0.45$

Combing to parameters this way has given a very good results after the 1000 iterations, but after that algorithm not only did not improve, but got worse up to final 199. Possibly what had happen was that during computation the algorithm firstly found the global minimum or at least smaller local minimum and then jump out of it and never went back.

On the other hand, the small values may make the algorithm work slower, but it will check smaller range of $k(x)$ values, but it will be more precise.

This time the $\beta = 0.05$ and we used smaller $\sigma = 0.1$.

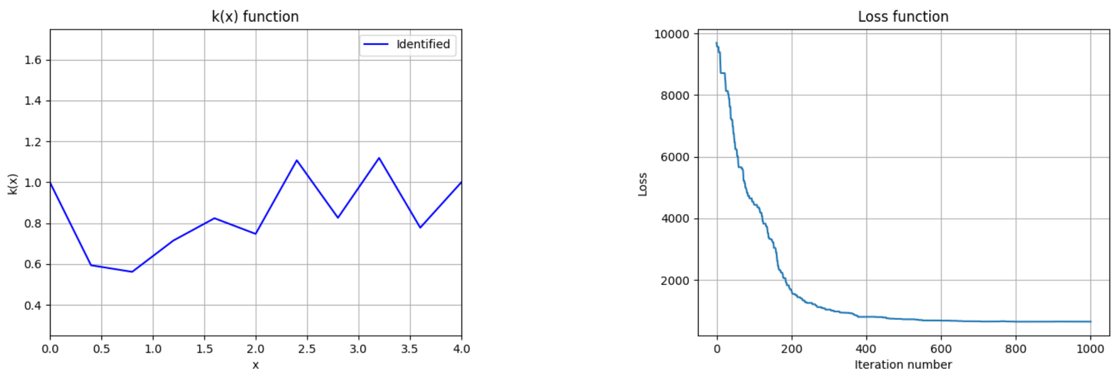

Figure 22: Simulation after 1000 iterations with $\sigma = 0.1$

From the plots above we can see that the loss function does not consists of big jumps, but more small steps going gradually to the small values, ending with the value of the loss function equal about 659.

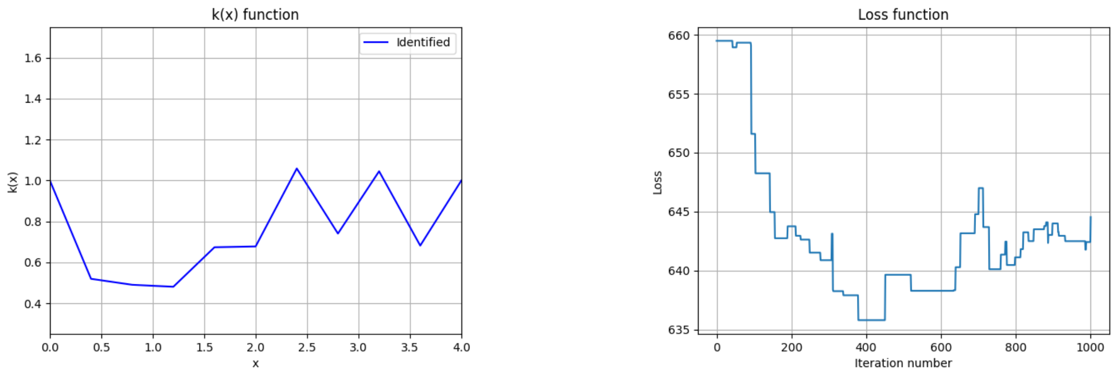

Figure 23: Simulation after 3000 iterations with $\sigma = 0.1$

At the end, the loss function have some problems, reaching the best value 637 at around 2400 iteration. However, getting worse with the nest steps, finishing at 644.

For this experiment the parameters are: $\sigma = 0.25$ and $\beta = 0.01$.

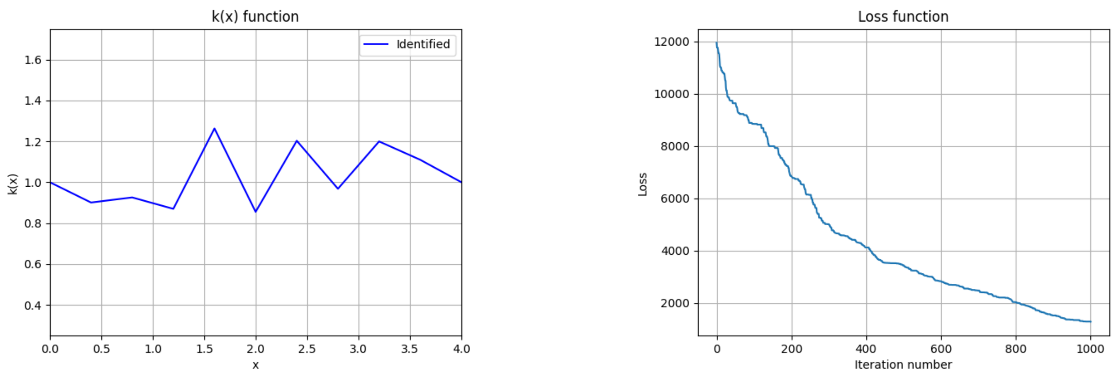

Figure 24: Simulation after 1000 iterations with $\beta = 0.01$

Similarly to the small σ case, the loss function is getting gradually smaller, even slower than before, reaching after 1000 iteration 1273 value of the loss function. After whole simulation we reached 195, but in this case the simulation is more stable. There are smaller jumps visible on the loss function and the algorithm can handle them at the end.

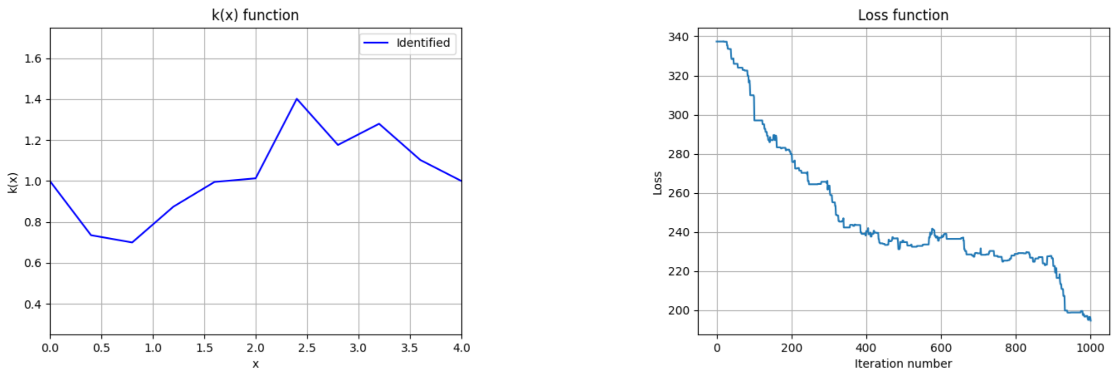

Figure 25: Simulation after 3000 iterations with $\beta = 0.01$

Below is shown the table with gathered all results from the previous experiments.

σ	β	After 1000 iterations	After 3000 iterations
0.25	0.05	655	460
0.5	0.05	168	151
0.25	0.25	775	730
0.45	0.1	167	199
0.1	0.05	659	644
0.25	0.01	1273	195

Both approach have their advantages and disadvantages, but what will happen if we would combine them? Firstly, we run the simulation with bigger values for the first 1000 iterations and then using the generated and saved $k(x)$ we would run it with the small values of the parameters. As we saw on the carried experiments, the big values of the β are not necessarily effective, but big σ got great results. Other case is with the small values, where smaller β got much better results than small σ. Analyzing the table above with the gathered results from the all experiments we see that after 1000 iteration the best combination of the parameters were $\sigma = 0.45$ and $\beta = 0.1$.

Also having β two times smaller would not make a big difference. That is why we can assume that setting bigger value of σ at the beginning of the simulation is crucial to the algorithm because then it can check more various $k(x)$. The last experiment will be carried according to the results from the table to get the best possible values. For the first 1000 iterations the parameters will be $\sigma = 0.45$ and $\beta = 0.1$.

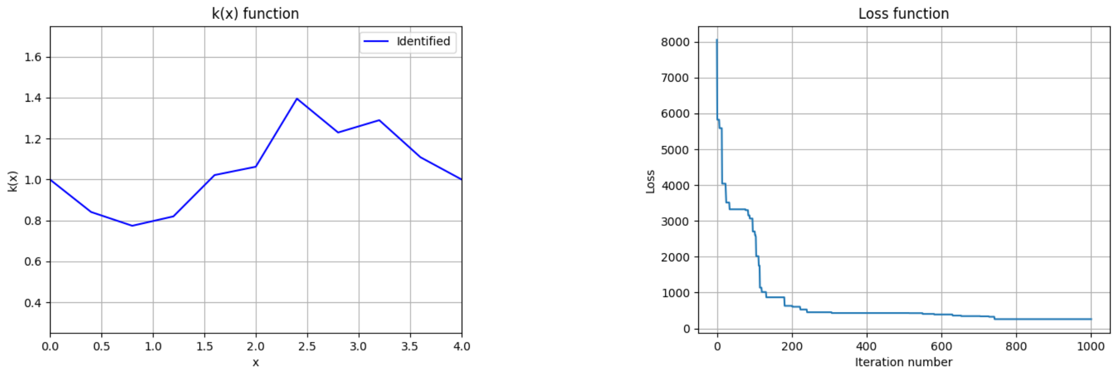

Figure 26: Simulation after 1000 iterations.

After only 1000 iteration the algorithm has reached value of the loss function 258. Next, to make it decrease less drastically and possibly not jump out of the found minimum, we will set for the next 1000 iterations $\sigma = 0.25$ and $\beta = 0.01$. Having done that, we hope that in the first stage algorithm has checked found the global optimum and now it will not jump out of it, but get deeper inside.

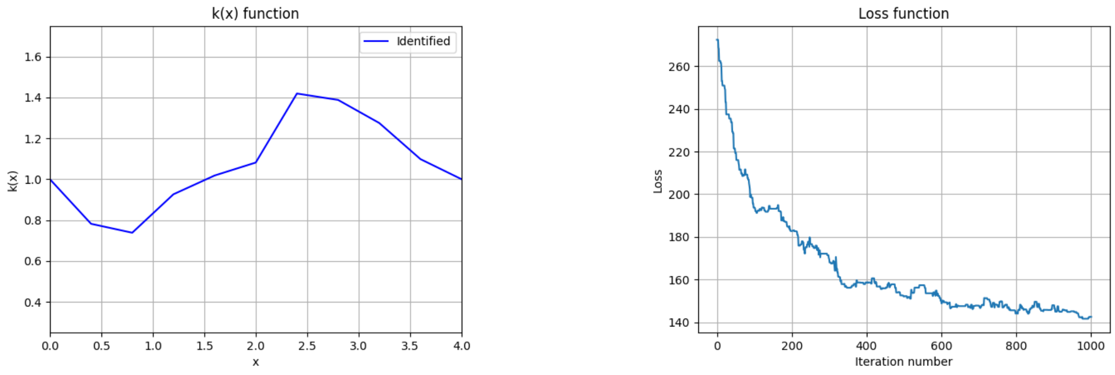

Figure 27: Simulation after 2000 iterations.

This time the simulation decreased the loss function to the 142. The loss function for the first 400 iteration this time is getting smaller quite strongly and after that is improving just slightly. Given that for the last 1000 iterations the parameters will be even smaller, to avoid jumping out of the found minimum. Hopefully, the algorithm will go deeper into it and lost itself.

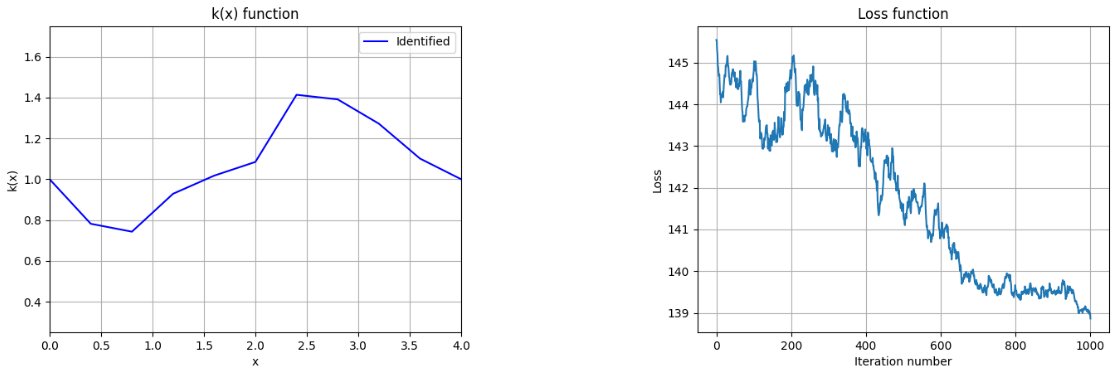

Figure 28: Simulation after 3000 iterations.

The algorithm struggled a bit, but finally improved to 138, which is the best value that was generated. Only question we should ask ourselves is if we need to run the simulation 3000 times instead of stopping at 2000 with the result that is worse only by 4.

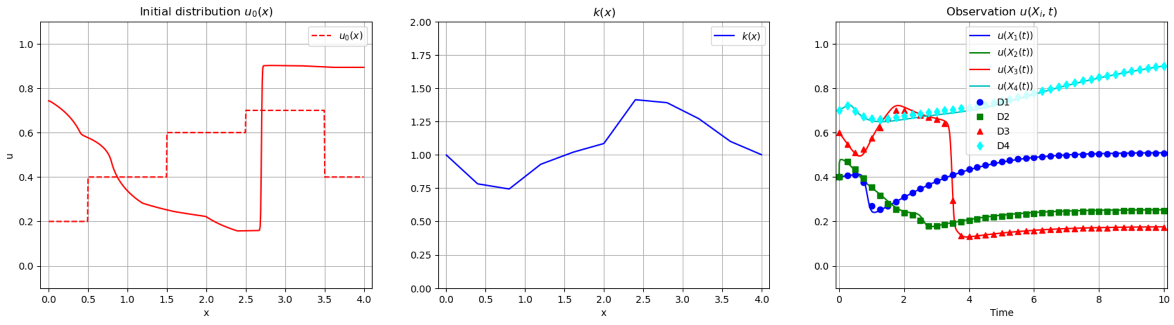

Figure 29: Predicted data with optimized $k(x)$

Bellow is shown the table with the summarization of the final run of the simulation.

Number of iteration	σ	β	Loss value
1000	0.45	0.1	258
2000	0.25	0.01	142
3000	0.15	0.005	138

As we can see above the optimized $k(x)$ made the predicted data identical to the observation data, for which we can give ourselves a pat on the shoulder.

b) **Predicting data with two datasets**

The second dataset has the initial distribution of:

$$u_0(x) = \begin{cases} 0.1, & x \in [0, 0.5] \\ 0.3, & x \in (0.5, 1.5] \\ 0.7, & x \in (1.5, 2.5] \\ 0.4, & x \in (2.5, 4]. \end{cases} \tag{10}$$

In this task we will combine two datasets to train the $k(x)$ function. Firstly, lets see what will be the loss function we would use already optimized $k(x)$ for the second dataset with the parameters advised by professor $\sigma = 0.25$ and $\beta = 0.05$.

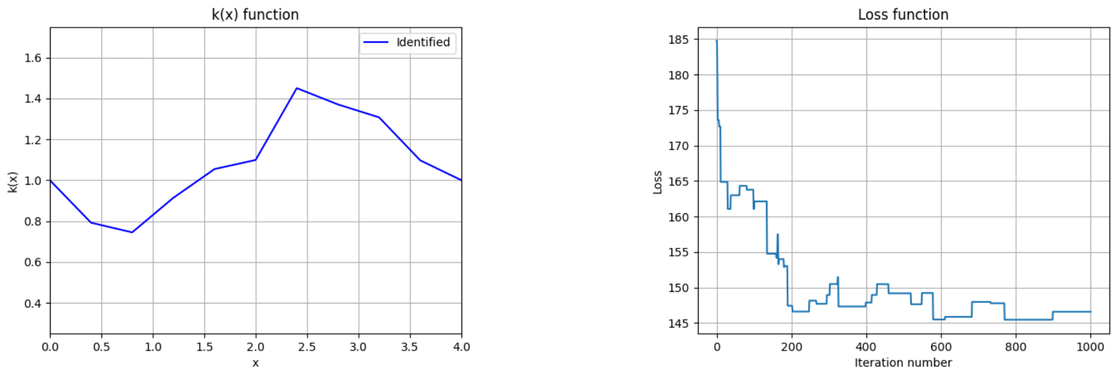

Figure 30: Simulation with different datasets.

Using second dataset allows the algorithm to find faster better results. Probably this happens because algorithm can check different values than before.

Next run was with both datasets and with the values of parameters provided by the professor $\sigma = 0.25$ and $\beta = 0.05$.

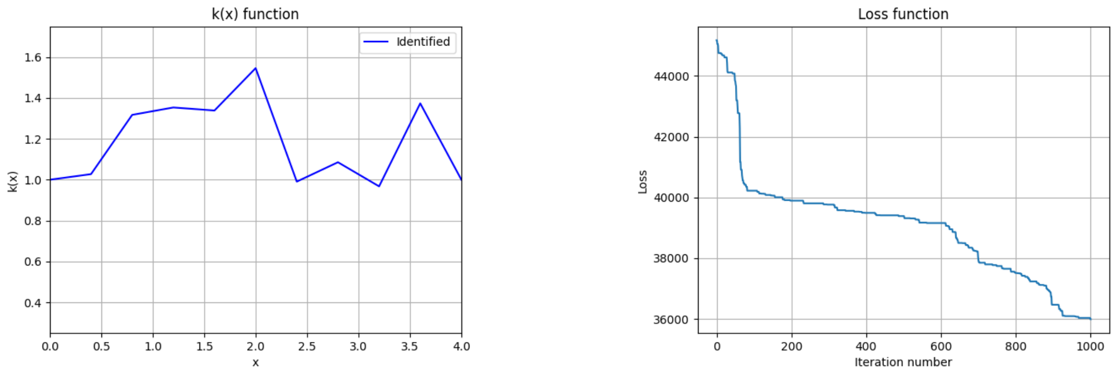

Figure 31: Simulation with two combined datasets after 1000 iterations.

Looking at the loss function on the figure above, firstly we can notice that the starting point of combined loss is much bigger than when it was only one dataset. Using only one dataset the loss at the beginning was about 8000 up to 16000, but this time the combined starting loss was over 45000. After the 1000 iterations the loss was about 36000.

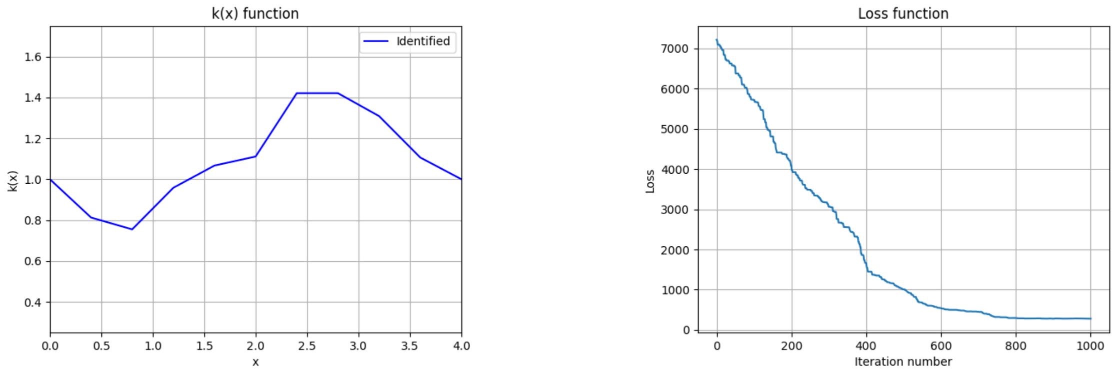

Figure 32: Simulation with two combined datasets after 3000 iterations.

In the third part of the simulation, the loss function started from the level it was starting with only one dataset - around 7000. There up to the 2400 iteration the loss function is close to being linear and after is decreasing in more exponential way. At the end it reached 278, which is very good final loss. With those parameters and only one dataset the algorithm after 3000 reached 460, which is close to double of the loss achieved now. But also before this setup was not the most successful one, so lets try running the simulation with the final combinations of the σ and β from the previous section. For the first 1000 iterations we will use $\sigma = 0.45$ and $\beta = 0.1$.

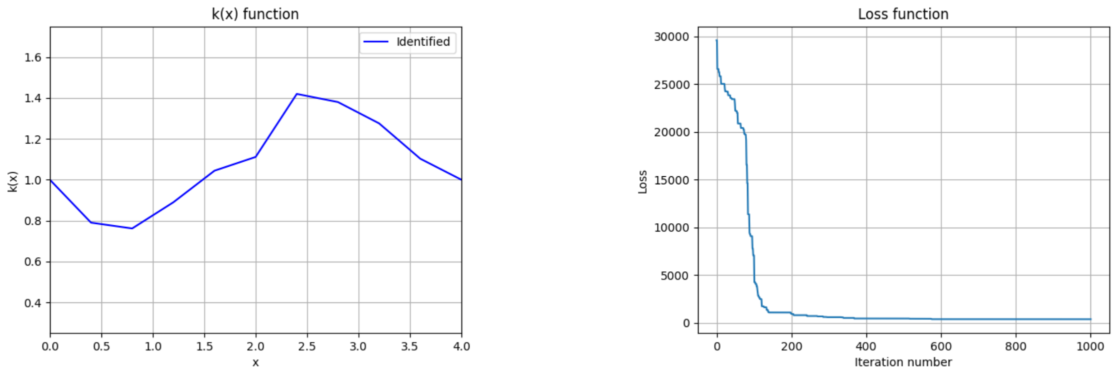

Figure 33: Simulation with two combined datasets after 1000 iterations with optimized parameters.

This is an excellent run, that minimized the loss nearly 100 times, from 30000 to over 300. Next part will be run with $\sigma = 0.25$ and $\beta = 0.01$.

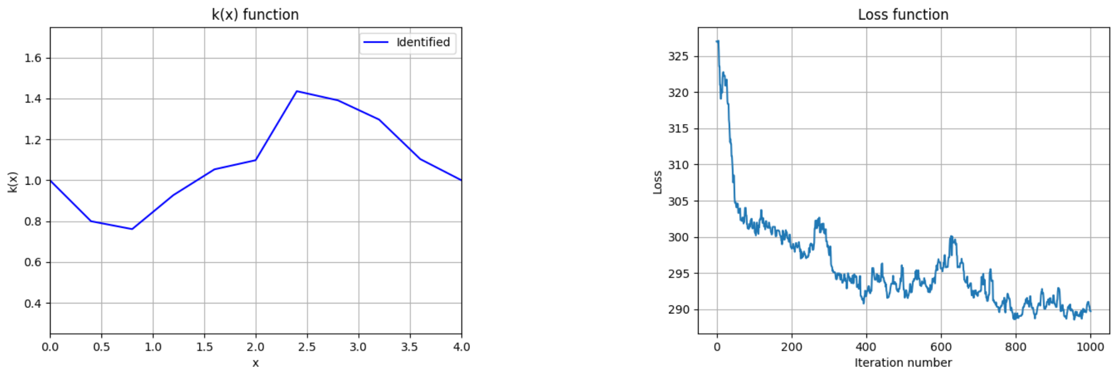

Figure 34: Simulation with two combined datasets after 2000 iterations with optimized parameters.

After this run the simulation reached 289. The last part will be run with parameters σ and β.

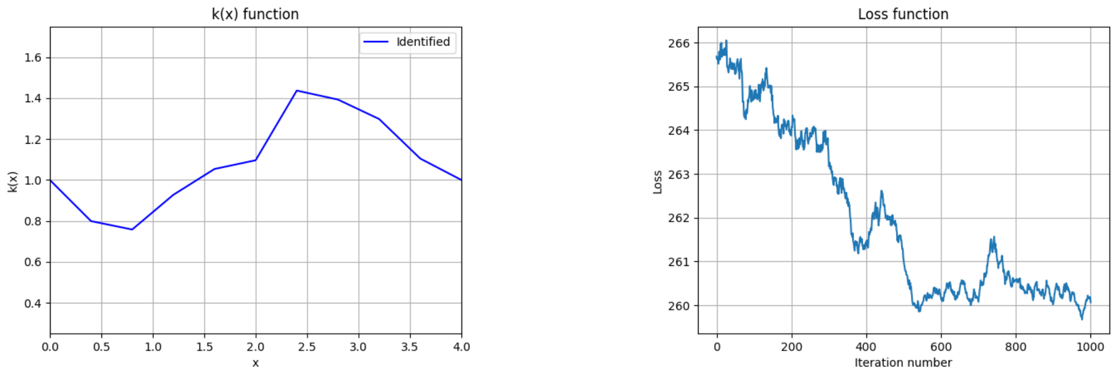

Figure 35: Simulation with two combined datasets after 3000 iterations with optimized parameters.

After the last part of the simulation the loss reach 260. Final table with the results of the simulation of the algorithm using optimized parameters and combined datasets.

Number of iteration	σ	β	Loss value
1000	0.45	0.1	359
2000	0.25	0.01	289
3000	0.1	0.005	260

The same as before the loss got decreased with every step, but this time the last part result with bigger difference than before, but still not big enough to be sure that the last run is necessary. As the last part of this project lets see how the optimized $k(x)$ function on combined datasets fit the observed data.

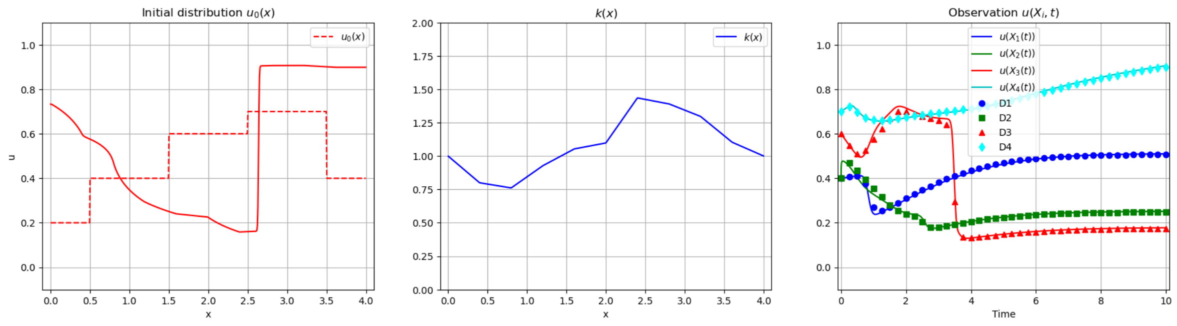

Figure 36: Predicted data with optimized $k(x)$

The fit is similar to the fit achieved with the only one dataset, sometimes a bit worse. Which can be explained that having bigger amount of data make it harder for the algorithm to handle optimization enough, as we saw on the simulation with the first set of the parameters. There was not enough iteration to find the optimum. In the second run it was not the option, having that the loss function sometimes jumps up already. In this case we can assume that maybe it got stuck in different local minimum - created by the new data and could not get out. However, we could also get lucky with the simulation with only one dataset.

Summing up, having more data is not always a better option. It can make work more difficult, but having not enough data makes work impossible. It is important to find the amount of data that is suitable for given task and if possible try to visualize the optimized function.

References

[1] S. L. Cotter, G. O. Roberts, A. M. Stuart, and D. White. Mcmc methods for functions: Modifying old algorithms to make them faster. *Statistical Science*, 28:424–446, 2013.

[2] S. Evje. Basic theory for nonlinear conservation laws + exercises. *Lecture notes UiS*, 2025.

[3] S. Evje. Using monte carlo markov chain (mcmc) to identyfy unknown flux funtion from observation data. *UiS Lecture Notes*, 2025.

[4] S. Evje, J. H. Skadsem, and G. Nævdal. Identification of nonlinear conservation laws for multiphase flow based on bayesian inversion. *Nonlinear Dynamics*, 111:18163–18190, 2023.

[5] S. Mishra, U. S. Fjordholm, and R. Abgrall. Numerical methods for conservation laws and related equations. *UiO Lecture Notes*, 2023.

[6] U. Starowicz. Modeling traffic flow. *Project for MOD600*, 2025.