Project 4: Bayesian Methods: Motion Estimation for Contrast Maximization with Event Cameras

AEROSP 567: Statistical Inference, Estimation, and Learning

Onur Bagoren

December 16, 2021

Abstract

Event cameras are biologically inspired cameras that, in contrast to traditional cameras, register changes in pixel intensity asynchronously. Each change in pixel intensity, referred to as an "event", is triggered by changes in motion in the scene at which the camera is observing; hence, events can be highly informative regarding the dynamics of the scene it is observing. The method highlighted in this project aims to solve the problem of motion parameter estimation of the ego-motion of the camera using probabilistic methods. Posing the problem of estimating motion parameters as an inference problem, this solution provides both information about the motion of the camera while also informing uncertainty of the estimations, which can later be used in tasks of computer vision in applications that require motion parameter estimates, such as scene segmentation or depth estimation. The proposed method utilizes available datasets online and aims to solve linear motion problems. This method proved to be succesful in estimating motion parameters, yet suffered from computational complexity, making it unfavorable to use for online estiamtion problem, but favorable for post processing of the data.

1 Introduction

Event cameras are cameras that respond to pixel intensity changes in the pixel frame of the camera. An "event", occurs once the change of the pixel intensity is over 15%. Each event is represented in the spatio-temporal coordinates, along with a polarity value, which represents whether the pixel value increased, or decreased in intensity. Hence, each event registered by an event camera is represented as a tuple of these values $e_i = (t_i, x_i, y_i, p_i)$. In very simple terms, the product of an event camera can be analogous to a point cloud, only with extra information about the direction at which an object is either approaching the camera or moving away from the camera. A visualization of a collection of events that are triggered by moving objects in front of an event camera is shown in fig. 1.

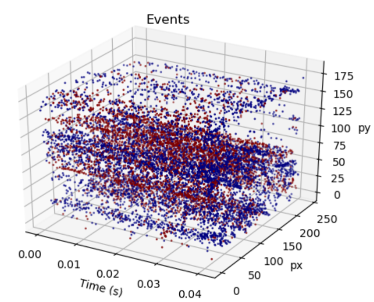

Figure 1: Events caused by moving objects in the scene of the camera

Event cameras additionally offer multiple advantages over traditional cameras. Event cameras have a high dynamic range, which measures up to 140 dB as compared to traditional cameras, which have a dynamic range of 60 dB. More importantly, event cameras offer very low latency, to the order of microseconds. A primary advantage of such low latency measurements prevent event camera observations to suffer from motion blurring. As the changes of pixel intensities also occur asynchronously, ech individual object operating on an event camera can be registered at this low latency. These properties of event cameras can then be used for sample based motion estimation, rather than timing based motion estimation, as it is commonly done with traditional cameras that capture frames at certain time intervals. These advantages have lead to numerous advancements of the use of event cameras in computer vision and image processing applications. As the event camera responds to the changes of objects in the scene it is observing, the dynamics of the scene can be captured. This is where the low latency especially helps, in that accurate representations of the objects that the camera is observing can be captured by the event camera.

A novel, and important, challenge regarding the use of event cameras have been tied to the concept of contrast maximization and parameter estimation with event cameras, which are two areas that are closely related for any event camera data to be used in any complex computer vision or image processing application. This project aims to utilize a probabilistic representation of the data and event generation of the event camera and utilize probabilistic methods in order to perform motion parameter estimation of the camera and observed objects. This challenge is referred to motion estimation. Contrast maximization is a resulting process after motion estimation, which acts as a filter to sharpen the image in order to reduce white noise produced by the sensor itself.

2 Motion Estimation

2.1 Representation of Motion with Event Cameras

Event cameras capture any change in pixel values, which can be seen as any motion that is captured in the frame at which the camera is observing. Hence, following the trace of the events over time is sufficient to characterize the motion parameters of the scene and camera.

Motion estimation with cameras are also analogous to optical flow estimation, which is the characterization of the rate of pixel movement across the image frame. This is typically the method at which such estimation tasks are completed, without information about the camera extrinsic parameters, which describe how the camera frame transforms into the world frame, it is difficult to estimate the actual rate of change.

The task for this project involves characterizing the motion parameters of linear movements of the camera. The model for characterizing such motion in the image coordinate frame is given with (2.1) and eq. (2.2).

$$\mathbf{x}(t_1) = \mathbf{x}(t_0) + \mathbf{v}_x t \tag{2.1}$$

$$\mathbf{y}(t_1) = \mathbf{y}(t_0) + \mathbf{v}_y t \tag{2.2}$$

The motion parameters are hence defined as $\Theta_x = [x(t_0)\ \mathbf{v}_x]^T$ and $\Theta_y = [y(t_0)\ \mathbf{v}_y]^T$.

2.1.1 Existing Solutions

The existing solutions [1], [2] for event camera motion estimation utilizes image processing techniques in order to both find the direction of highest event accumulation. Such solutions aim to group together, or warp, events so that a set of motion parameters Θ can describe the trajectory of the events. A warped event $\mathbb{W}$ can then be characterized according to this definition shown in eq. (2.3):

$$\mathbf{x}_k^{'} = \mathbb{W}(\mathbf{x}_k; t_k; \Theta) \tag{2.3}$$

$$\mathbf{x}_k = (x_k\ y_k)^T \tag{2.4}$$

where Θ represents a set of motion parameters in either the x or y direction of the pixel values. For optical flow estimation, the estimation parameters are computed for small periods of time, which allows representing the warped events as a linear equation, as shown in (2.5)

$$\mathbf{x}_k^{'} = \mathbf{x}_k - (t_k - t_{ref})\theta \tag{2.5}$$

where t_{ref} is the first time at which the first event occurs. The warped events starting from t_{ref} are then projected to a discretized plane of pixels according to (2.6). This projection of events construct an image patch H, which encapsulates information regarding the set of events that fall withing the trajectory constructed by the motion parameter θ. The general expression for the image patch is shown in eq (2.6):

$$H(\mathbf{x}; \theta) = \sum_{k=1}^{N_e} b_k \Delta(\mathbf{x} - \mathbf{x}_k^{'}) \tag{2.6}$$

Where N_e is the number of events, $b_k = 1$ and $\Delta(\cdot)$ represents any function that evaluates the proximity of $x_k^{'}$ to the actual events x_k at time t_k. The selection of this $\Delta(\cdot)$ function is a design choice for the application in question.

Upon the design of this method of encapsulating the events in the image frame, an objective function to evaluate the selected parameters Θ is designed.

The objective function then becomes finding the parameter that maximizes the variance on the image patch H generated by θ, with eq. (2.7).

$$f_{\sigma^2}(\theta) = \frac{1}{N_p} \sum_{i,j} (h_{i,j} - \mu_H)^2 \tag{2.7}$$

where N_p is the number of pixels on H, $h_{i,j}$ represents the number of clustered events projected to pixel location (i, j) and μ_H is the mean number of clustered at a given pixel of H, such that

$$\mu_H = \frac{1}{N_p} \sum_{i,j} h_{i,j} \tag{2.8}$$

Equation (2.7) is one of the many objective functions that have been proven to be successful for these applications [2].

2.2 Proposed Solution to Motion Estimation

The proposed solution utilizes a probabilistic model to update the belief on the motion parameters as data from the event camera is received. This formulation poses the problem as a learning problem on the parameters.

As the dynamics model is linear, the problem unravels as a linear regression problem on the data in order to fit the parameters on the data. There are a multitude of algorithms that can be used to solve this problem. The solution with this project utilizes Gaussian Linear Models and poses it as a linear inverse problem to use Gaussian Processes to learn the parameters.

In order to formulate the problem, it is necessary to define the concepts that will be used in this problem formulation, and how it was used in the implementation of the proposed solution.

2.2.1 Distributions on the Variables

Unlike the solutions described in Section 2.1.1, where the motion parameters Θ_x and Θ_y were treated as deterministic parameters, the solution will approach the parameters as random variables, with probability distributions that correspond to the defined model.

The proposd solution The design choice for the probability distribution of the random variables were defined as normal distributions, where the distribution of parameters Θ_x and Θ_y are shown in eq. (2.9) and (2.10).

$$\mathbb{P}(\Theta_x) \sim \mathcal{N}(\mu_x, \Sigma_x) \tag{2.9}$$
$$\mathbb{P}(\Theta_y) \sim \mathcal{N}(\mu_y, \Sigma_y) \tag{2.10}$$

As part of the dynamic system, the manner in which the event camera captures information and registers events can also be posed as a probabilistic model, due to the probabilistic distributions of the parameters provided in eq. (2.9) and eq.(2.10).

The measurement model for the event camera is a linear model [3], [4]. The generation of events are generated as an increase of contrast in the pixel relative to its previous contrast values, resulting in the intensity increase above a threshold that triggers an event. Hence, the the measurement model can be described as the linear model presented with eq. (2.11), where the measurement noise η is zero-mean with variance that depends on the event camera model that is used to take the measurements (more detail on event camera specifications are provided with [3], Table 1).

$$Y = A\Theta + \eta \tag{2.11}$$
$$\eta \sim \mathcal{N}(0, \Gamma) \tag{2.12}$$
$$A = \begin{bmatrix} 1 & t^{(1)} \\ \vdots & \\ 1 & t^{(N)} \end{bmatrix} \tag{2.13}$$

In eq. (2.11), the matrix A represents the Vandermonde matrix, which has dimensions of $\mathbb{R}^{N\times 2}$, where N is the number of data points that are measured at a time t and 2 is for the number of parameters of Θ. As there are motion parameters to learn in both the x and y direction, the measurement model is then regarded as separate in both directions.

Provided this information, the conditional probabilities between the two random variables representing the measurements and the motion parameters can then be constructed. The measurement model implies a linear Gaussian relation between the parameters due to the Gaussian distribution on the parameters in (2.9) and (2.10); hence, the distribution shown in eq. (2.14) can be constructed.

$$\mathbb{P}(Y|\Theta) \sim \mathcal{N}(\Theta t, \Gamma) \tag{2.14}$$

2.2.2 Linear Gaussian Models

The proposed solution to the problem of motion parameter estimation utilizes a linear Gaussian model. This proposal stems from the idea that the prior distribution on the parameters are Gaussian, as shown in Section 2.2.1, along with the likelihood model being a linear measurement model. From conjugate pairs, it is then known that with a linear likelihood model, the posterior will be of the same distribution family as the prior; in this case a Gaussian distribution, which belongs to the exponential distribution family.

The linear Gaussian model utilizes Bayesian Inference in order to learn the posterior distribution of a parameter, provided an initial distribution of the prior and a given likelihood model.

Bayesian Inference For Bayesian Inference, the learning is done by updating the belief on the parameters θ after observing the data. This belief is a continuous representation that can make predictions between the data that was seen. In order to update this belief representation on the parameters after observing a set of data, Bayes' rule can be used.

$$\mathbb{P}(\Theta|Y) = \frac{\mathbb{P}(Y|\Theta)\mathbb{P}(\Theta)}{\mathbb{P}(Y)} \propto \mathbb{P}(Y|\Theta)\mathbb{P}(\Theta) \tag{2.15}$$

In eq. (2.15),

1. $f_{\Theta|D}(\theta|d)$ is the posterior distribution which represents the distribution of the belief on parameters θ after seeing data d. In the scope of this project, the prior distributions are represented with equations (2.9) and (2.10).

2. $f_{D|\Theta}(d|\theta)$ is the sampling distribution, which represents how the data will be interpreted by the learning model. The sampling distribution is key in how the belief is represented continuously after observing data, as it varies from model to model. Will be represented as $\mathcal{L}(\theta)$. The likelihood model in this project is constructed from the measurement model, which is shown with eq. (2.11)

3. $f_{\Theta}(\theta)$ is the prior distribution, sometimes shown as $f_{\Theta}(\theta|I)$ where I represents the available information. The effectiveness of Bayesian inference depends on the interpretation of I to construct the prior distribution. A good analogy for the importance of the prior is that learning with existing beliefs about the event/topic will impact how the learning will occur with new information

4. $f_D(d)$ is the evidence, which will commonly be used as a normalization factor η as it can be very difficult to compute.

Useful properties of Bayesian Inference:

1. Completing inference on data sequentially or batch does not impact the inference

Proof.

$$\begin{aligned} p(\theta|A,B) &= p(\theta|A)\frac{p(B|A,\theta)}{p(B|A)} && (2.16)\\ &= p(\theta)\frac{p(A|\theta)}{p(A)}\frac{p(B|A,\theta)}{p(B|A)} && (2.17)\\ &= p(\theta)\frac{p(A,B|\theta)}{p(A,B)} && (2.18)\\ &= p(\theta|A,B) && (2.19)\end{aligned}$$

□

The linear Gaussian problem utilizes the fact that the joint distribution between the parameters and the measurement are jointly Gaussian. To show this, it must first be established what the marginal distribution of the random variable for the measurement will be. As the measurement model from eq. (2.11) is just scaling and addition of Gaussian RVs, the marginal distribution for the measurement becomes the expression shown with (2.20).

$$\mathbb{P}(Y) \sim \mathcal{N}(A\Theta, A\Sigma A^T + \Gamma) \tag{2.20}$$

$$\mathbb{C}ov[Y,\Theta] = A\Sigma \tag{2.21}$$

With the marginal distribution of both Θ and Y established, their joint distributions then are formed as (2.22), as their covariance can be expressed as shown in (2.21). The joint distribution is then normally distributed.

$$\begin{bmatrix}\Theta\\ Y\end{bmatrix} \sim \mathcal{N}\left(\begin{bmatrix}\mu_\Theta\\ A\mu_\Theta\end{bmatrix}, \begin{bmatrix}\Sigma_\Theta & \Sigma_\Theta A^T\\ A\Sigma_\Theta & A\Sigma_\Theta A^T + \Gamma\end{bmatrix}\right) \tag{2.22}$$

Finally, in order to get the posterior distribution $\mathbb{P}(\Theta|Y)$, it can be seen that conditioning the parameters on the data will be sufficient. For jointly distributed Gaussian RVs, the conditioning of the variables upon each other yield a closed form expression shown with equations (2.24) and (2.25)

$$\mathbb{P}(\Theta|Y) \sim \mathcal{N}(\mu_\Theta^{post}, \Sigma_\Theta^{post}) \tag{2.23}$$

$$\mu_\Theta^{post} = \mu_\Theta + \Sigma_\Theta A^T \left(A\Sigma_\Theta A^T + \Gamma\right)^{-1}(y - A\mu_\Theta) \tag{2.24}$$

$$\Sigma_\Theta^{post} = \Sigma_\Theta - \Sigma_\Theta A^T \left(A\Sigma_\Theta A^T + \Gamma\right)^{-1} A\Sigma_\Theta \tag{2.25}$$

where y is the data.

Utilizing the closed form expression of the Linear Gaussian, these equations can then be used to perform linear regression on the motion parameters.

2.3 Implementation of the Linear Gaussian Inverse Problem

2.3.1 Partitioning the data

Following the assumptions that are made regarding optical flow estimation, at a short enough time frame, all representations of the object motion can be linearized. Due to the advantage of the extremely high sampling rate of the event camera, within small bursts of times, there is a large amount of data to perform inference on.

Taking advantage of the structure of the data, the data is then segmented into small portions, each portion representing an area to run the regression on to learn the parameters of that time slice. This offers two solutions:

1. Segmenting and operating on less data implies that the size of the Vandermonde matrix, which is inverted with in the GP equations for the posterior moment ($\mu_\Theta^{post}, \Sigma_\Theta^{post}$) computations in eq. (2.24) and eq. (2.25), is smaller. As the inversion of a matrix runs at the time complexity of $\mathcal{O}(n^3)$, the high data regime runs the risk of the algorithm not running fast enough to compute motion parameters within a reasonable time.

2. Transferring this solution to an online problem can then yield the advantage performing the computation at timestamps.

In addition to mitigating the size of the matrices to be inverted, there are also other ways to mitigate with inverting a large dimensional matrix.

Woodbury Matrix Identity is an identity that follows the proposition that

$$(A + UCV)^{-1} = A^{-1} + A^{-1}U\left(C^{-1} + VA^{-1}U\right)^{-1} VA^{-1}$$

This identity can be used in equations (2.24) and (2.25). These equations then become

$$\mathbb{P}(\Theta|Y) \sim \mathcal{N}(\mu_\Theta^{post}, \Sigma_\Theta^{post}) \tag{2.26}$$

$$\mu_\Theta^{post} = \left(A^T\Gamma^{-1}A + \Sigma_\Theta^{-1}\right)^{-1}\left(A^T\Gamma^{-1}y + \Sigma_\Theta^{-1}\mu_\Theta\right) \tag{2.27}$$

$$\Sigma_\Theta^{post} = \left(A^T\Gamma^{-1}A + \Sigma_\Theta^{-1}\right)^{-1} \tag{2.28}$$

The advantage of this representation is that rather than inverting a matrix of the dimension of the data, a matrix of the dimension of the parameters is being inverted - vastly saving computation data for the case of event camera applications, which is a dense data application with few parameters.

For the implementation in the project, there partitioning was done every 7500 data points. For the dataset that was used, this yielded 100 segments to run inference on.

2.3.2 Understanding the form of the data

A large challenge about working with event cameras come in multiple forms. The primary issue is in the form of asynchronous data generation. Asynchronous data generation causes the data at different time steps to be different sizes. In comparison to more traditional sensors, which query for a signals at defined time intervals, the event camera will output an event whenever there is any motion. This poses the problem with working with multidimensional data for the task of estimating a single parameters. Figure 2 displays this phenomenon, that over a time period, multiple points can be present.

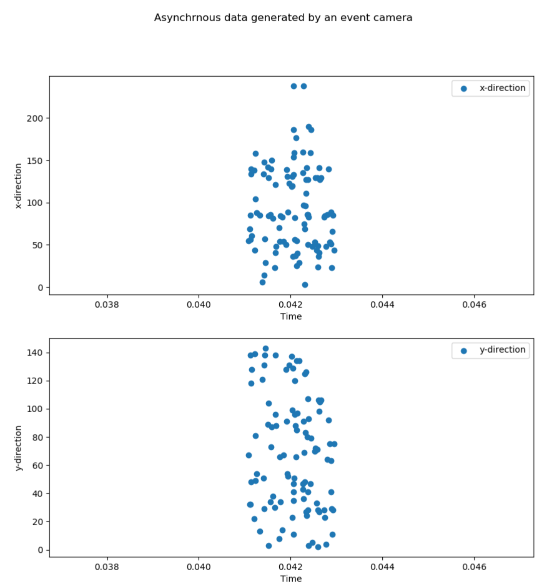

Figure 2: Data generated within a small time window. this figure shows that at different time steps, there are multiple dimensions of data, all representing the same information about the position of an object on the x- or y-axis.

This issue becomes prevalent when forming the Vandermonde matrix in the measurement model setup. The Vandermonde matrix is a representation for the linear transformation of the motion parameters over time. The nature of the a synchronicity of the data then requires that the matrix be varied-dimensional across different time values. The necessity to then iterate over each time step to obtain yields an iterative/sequential solution in comparison to the batch solution.

Shown with eq. (2.19), for Bayesian inference, sequential vs. batch does not make a difference for the inference that is being done, but in terms of runtime, iterative algorithms in comparison to the batch algorithms suffer from the not being vectorized, which greatly improves the computation speed of processes.

Two solutions were attempted in order to mitigate this problem.

1. **Representing ragged data as a matrix with fixed size:** This solution required preprocessing the data to represent it as a matrix with fixed size. The matrix would be of dimension $d \times n$, where d represents the number of unique timestamps that the event camera measured events on, and n representing the maximum number of events that were triggered at a single time step. The entries where the number of data-points $< n$ are then represented as "not a number".

2. **Averaging the measurements:** This method mitigates the varied-dimensional nature of the data by rather than evaluating the likelihood at each data point, evaluating at the mean of the measurements. The assumption made for this solution follows the assumption that due to the linear measurement model and the operation of calculating being a linear operation, the evaluation of the likelihood at a certain time would be equivalent to the measurement of the mean of the measurements at that same time.

For the implementation of the project, the second method was used, as it required less offline time in order to pre-process the data and form a new representation of the data, along with an ease of implementation, as single dimensional data can be input into the measurement model without adjusting dimensional over time and event segment (discussed in Section 2.3.1)

2.4 Converting from camera frame to image frame

The data that is being processed with this algorithm outputs event positions on the image frame. The image frame can be represented as $\mathbf{u} = [u\ v]^T$, where u is the pixels in the x-direction and v is the pixels in the y-direction. The result of running the computations on the pixel frame is the lack of ability to translate the rate of change to more intuitive measures, such as m/s.

A solution for this comes in the form of identifying and knowing the characteristics of the camera that is being used. These are known as the intrinsic parameters of the camera. The intrinsic parameters convey information regarding the focal length in the x- and y-direction, along with the principle point of the camera coordinates (principal point is the offset of the center of the camera frame to the top left corner of the frame, measured in).

By representing the pixel coordinates as homogeneous coordinates, the projection of the pixel coordinates can then be done on the world coordinates. This method follows eq. (2.29), where $\tilde{\mathbf{x}}_c$ are the world coordinates in 3D and $\tilde{\mathbf{u}}$ are the homogeneous image coordinates.

$$\tilde{\mathbf{u}} = M_{int}\tilde{\mathbf{x}}_c \tag{2.29}$$

For this project, the intrinsic values of the camera were provided with the test dataset, which were used to compute the motion parameters to the image plane to compare results with the predictions.

2.5 Informative Priors

As discussed in Section 2.2.2, a good prior is crucial for the inference problem. In the implementation of this project, the partitioning of the data was utilized in order to design informative priors in between segments.

As it is assumed that each partition is the continuation of the partition before it, the prior distribution on the parameters for the GP on the next segment was selected according to the prior before it.

The prior set for the initial segment is $\mathbb{P}(\Theta) \sim \mathcal{N}(0, I)$ for both the x-direction and the y-direction. Selecting this distribution as the prior for each partition will cause issues with the predictions, as assuming not only zero velocity, but also a zero-position at each time step will cause any change in position to impact both parameters at a large scale, yielding inaccurate predictions.

2.6 Confidence Interval Computation

A method to quantify the confidence in the estimation is through a 95% Confidence Interval (CI). In order to evaluate the 95% CI, 10E5 samples from the posterior distribution at segments were taken. Then, a Monte Carlo estimator was used in order to estimate the motion parameters. The setup is as shown in eq. (2.30). In this equation 1_x is the indicator function, which is 1 when the sample generated from the MC estimator is equal to that of the mean of the posterior, otherwise it is 0.

$$S_n = \mathbb{E}\left[g(x)\right] = \frac{1}{n}\sum_{i=0}^{10E5} 1_x \tag{2.30}$$

$$1_x = \begin{cases} 1 \text{ if } \mu_{pred} = \mu_{estimate} \\ 0 \text{ otherwise} \end{cases} \tag{2.31}$$

The CI bounds are then computes as shown in eq. (2.32), where $\frac{\sigma}{\sqrt{n}}$ is the standard error of the MC estimator.

$$\mathbb{P}\left(S_n - \frac{z\sigma}{\sqrt{n}} \leq \mu_\Theta \leq S_n + \frac{z\sigma}{\sqrt{n}}\right) = 0.95 \tag{2.32}$$

3 Results

For the testing of the algorithm, the data representing linear motion from the event camera dataset was used [5]. The dataset was produced from measurements taken with a DAVIS240C event camera, which from Table 1 in [3] can be seen as having measurement noise of 0.1 events per pixel per second.

The results section will analyze the inference run on 4 of the segments. These segments are the first segment, 50^{th} segment and last segment. These were selected in order to display the evolution of the predictions over time, and the impact of informative priors.

For visual familiarity with the environment that the event camera is interacting with, a grayscale image of the environment that the event camera is observing is provided with fig. 3.

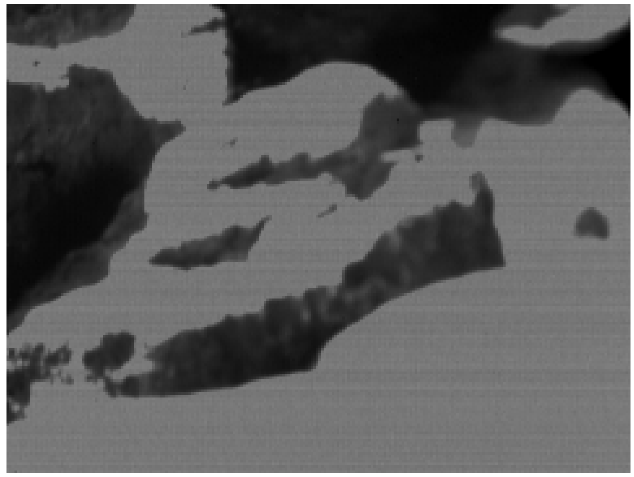

Figure 3: Grayscale image of the environment the event camera observes

3.1 Evaluating the Results

The bench marking of the performance of the algorithm was done against ground truth data, which is provided with the data upon which the algorithm was tested with. the ground truth data for the dataset "$slider_close$" was computed to be $(46.96, 0)\frac{px}{s}$ for the camera motion. This computation was done using eq. (2.29).

3.1.1 First Partition

This section provides figures regarding the posterior distribution after the inference, the prediction in comparison to the ground truth and a visual representation of what the prediction implies for working with the data.

Visualization of the prediction The visualization for interpreting what this prediction implies is helpful to get context on the problem that is trying to be solved. The collection of events are shown in 4. The red line in fig 4 represents the prediction made regarding the trajectory of the events over time, while also being a concise representation of the motion parameters.

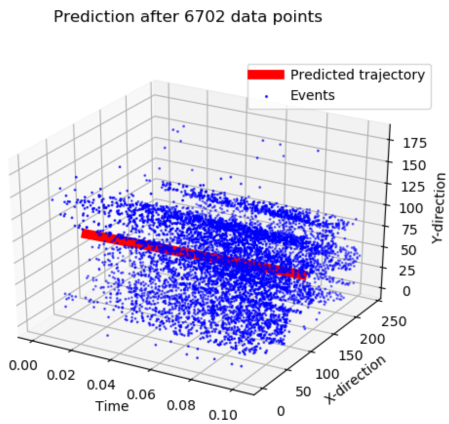

Figure 4: Events generated in the first segment

Figure 5 displays a figure for which the frame aligns with the predicted motion.

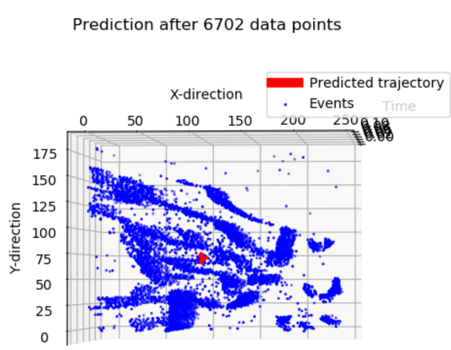

Figure 5: Events aligned according to the prediction made by the algorithm, shown in a 3D representation

A more informative can be seen with figure 6, where the predicted trajectory are shown in 2D plots, for each direction in x and y.

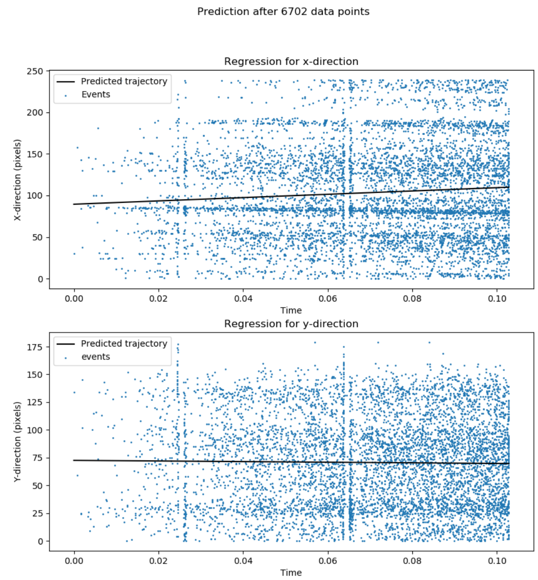

Figure 6: Events aligned according to the prediction made by the algorithm, shown in each dimension of x and y

Ground Truth Comparison The comparison of the predicted velocity from the motion parameters, is compared to that of the ground truth data. As the ground truth data does not provide information regarding the initial position, only the comparison between the ground truth velocity can be done.

Figure 7 displays the predicted velocity against the true velocity at the first segment.

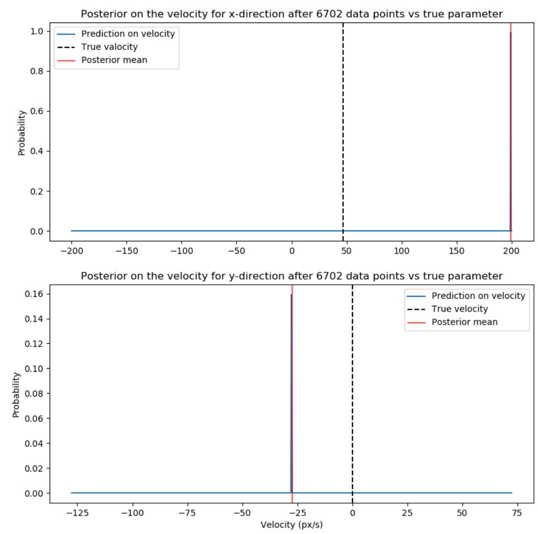

Figure 7: The comparison of the predicted trajectory to the ground truth

Confidence Interval For the first segment, the 95% confidence bounds for the velocity estimate are shown in eq. (3.2)

$$\text{CI for x-direction: } 199.205 \leq 199.205 \leq 199.205 \tag{3.1}$$

$$\text{CI for y-direction: } -27.626 \leq -27.626 \leq 27.626 \tag{3.2}$$

3.1.2 50^{th} Partition

Visualization of the prediction The visualization for interpreting what this prediction implies is helpful to get context on the problem that is trying to be solved. The collection of events are shown in 8. The red line in fig 8 represents the prediction made regarding the trajectory of the events over time, while also being a concise representation of the motion parameters.

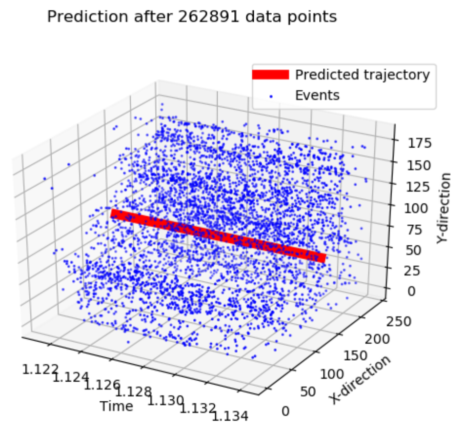

Figure 8: Events generated in the first segment

Figure 9 displays a figure for which the frame aligns with the predicted motion.

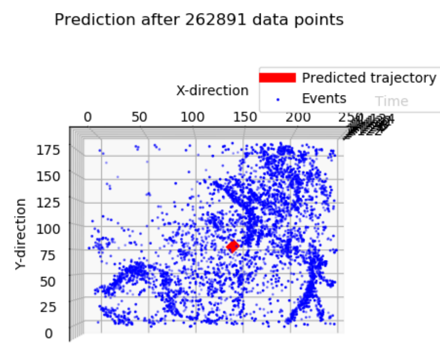

Figure 9: Events aligned according to the prediction made by the algorithm, shown in a 3D representation

A more informative can be seen with figure 10, where the predicted trajectory are shown in 2D plots, for each direction in x and y.

Ground Truth Comparison The comparison of the predicted velocity from the motion parameters, is compared to that of the ground truth data. As the ground truth data does not provide information regarding the initial position, only the comparison between the ground truth velocity can be done.

Figure 11 displays the predicted velocity against the true velocity at the first segment.

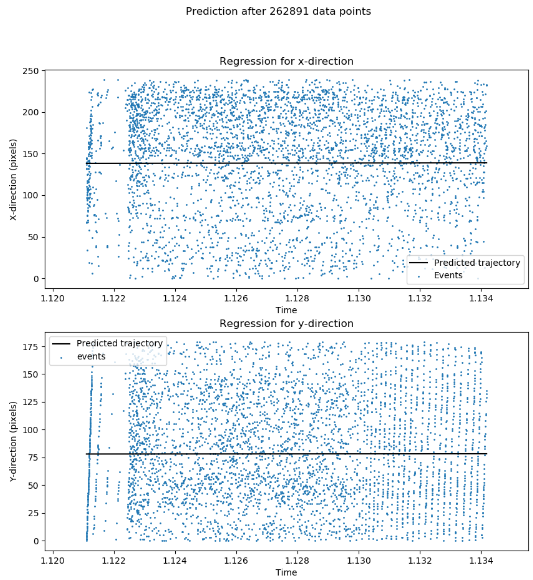

Figure 10: Events aligned according to the prediction made by the algorithm, shown in each dimension of x and y

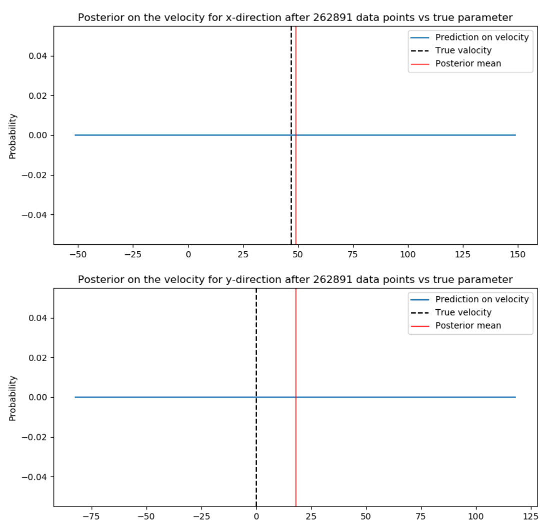

Figure 11: The comparison of the predicted trajectory to the ground truth

Confidence Interval A method to quantify the confidence in the predictions is to use confidence intervals. For the 50^{th} segment, the 95% confidence bounds for the velocity estimate are shown in eq. (3.4)

$$\text{CI for x-direction: } 48.816 \leq 48.816 \leq 48.816 \quad (3.3)$$

$$\text{CI for y-direction: } 17.802 \leq 17.801 \leq 17.801 \quad (3.4)$$

3.1.3 100^{th} Partition

Visualization of the prediction The visualization for interpreting what this prediction implies is helpful to get context on the problem that is trying to be solved. The collection of events are shown in 12. The red line in fig 12 represents the prediction made regarding the trajectory of the events over time, while also being a concise representation of the motion parameters.

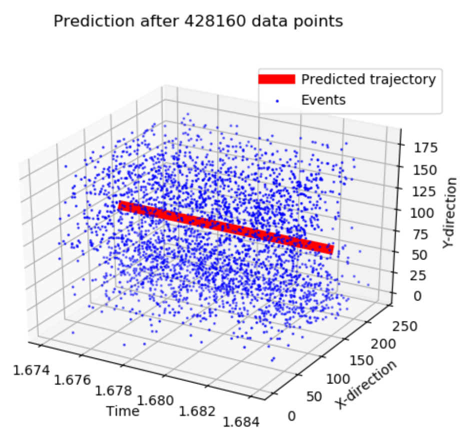

Figure 12: Events generated in the first segment

Figure 13 displays a figure for which the frame aligns with the predicted motion.

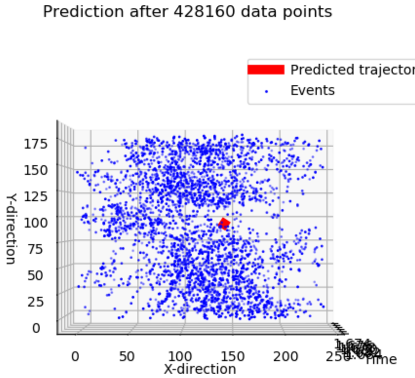

Figure 13: Events aligned according to the prediction made by the algorithm, shown in a 3D representation

A more informative can be seen with figure 14, where the predicted trajectory are shown in 2D plots, for each direction in x and y.

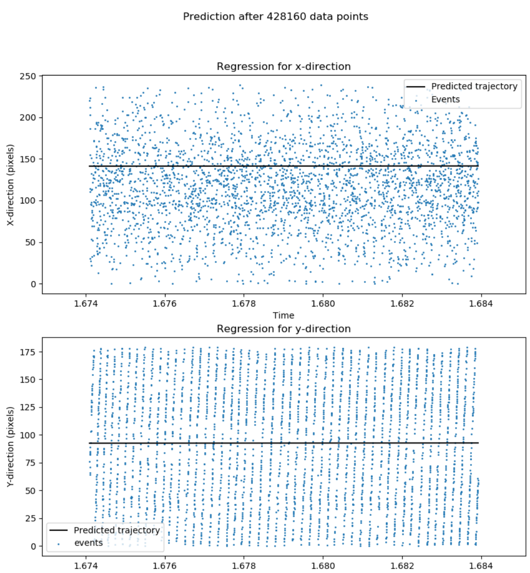

Figure 14: Events aligned according to the prediction made by the algorithm, shown in each dimension of x and y

Ground Truth Comparison The comparison of the predicted velocity from the motion parameters, is compared to that of the ground truth data. As the ground truth data does not provide information regarding the initial position, only the comparison between the ground truth velocity can be done.

Figure 15 displays the predicted velocity against the true velocity at the first segment.

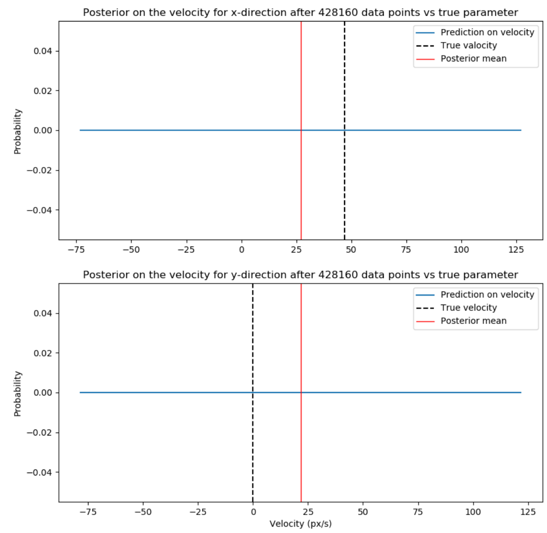

Figure 15: The comparison of the predicted trajectory to the ground truth

Confidence Interval A method to quantify the confidence in the predictions is to use confidence intervals. For the 100^{th} segment, the 95% confidence bounds for the velocity estimate are shown in eq. (3.6)

$$\text{CI for x-direction: } 26.929 \leq 26.929 \leq 26.929 \quad (3.5)$$

$$\text{CI for y-direction: } 21.619 \leq 21.619 \leq 21.619 \quad (3.6)$$

3.1.4 Overall Performance

In order to evaluate the overall performance of the parameters that were learned, the posteriors of the parameters at each partition were plotted over time to observe convergence characteristics, shown with fig. 16.

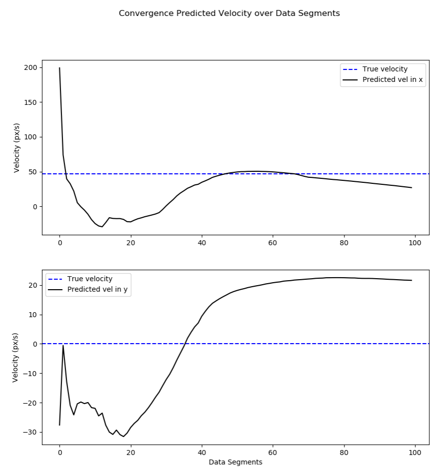

Figure 16: Estimations of velocity over each segment plotted against the true velocity in pixels/s

4 Discussion

4.1 Computation Evaluation

The most drastic performance issue that arose from this method came from the computational complexity. Despite using the Woodbury Matrix Identity to reduce computational complexity with inverting large matrices, or pre-processing matrices with known structures to leverage computational processing, inverting Σ_Θ, which was a $\mathbb{R}^{7500\times7500}$ matrix slowed the computation of each segment to 3 seconds each. This would cause drastic issues in online computations, as it would not be capable of processing data at a high enough speed to be viable in practice.

4.2 Evaluation of Inferred Parameters

4.2.1 Performance on Partitioned Data

The first partition displays the necessity to have informed priors. It can be seen from both the visuals with fig. 6 that the trajectory does not agree with the trend of the data, along with the comparison to the ground truth value, shown with 7.

Although this impacted the prediction for the first partition, this was quickly corrected for in preceding partitions. Using informed priors that utilize the posterior distribution from the previous inference step prevented such issues from occurring for future partitions.

This is especially evident in the 50^{th} and 100^{th} partition estimates, where the estimates were closer to the actual trend of the data and also closer to the ground truth data.

Reduction on the covariance of the estimations A noticeable effect throughout the inference steps was the rapid drop in the uncertainty associated with the estimations. This was apparent in the 95% CI of the estimations, there the lower bound and upper bound were the same value up to the $10^{-4}th$ decimal. A driving reason for this occurs from the fact that the abundance of data for the computation. The only method in which the uncertainty for the estimates to drop is in eq. (2.25).

The implication of the low uncertainty, yet inaccurate prediction at certain intervals point to the necessity to re-evaluate the inference model. Although over time the predictions agreed with the ground truth data, improvements to provide more interpretable and reliable predictions must be developed.

4.2.2 Performance on the entire dataset

Figure 16 represents the convergence of the motion parameters across the estimates. It can be seen that especially for the velocity estimate in the x-direction, which is the direction where there was the linear motion, converges after 40 partitions. It is also relevant to note that the steady state error associated state estimations on the parameters account for ¡ 20 pixels/s.

5 Conclusion

The learning method that utilized the linear-Gaussian method proved to be effective for offline computation. As the event camera provides an abundant amount of data over short periods of time, a probabilistic method in order to estimate the motion parameters is successful in reducing the uncertainty with the estimation, along with providing an understandable interpretation of the data.

The downside of the method is the computational complexity and the interaction with the structure of the data. There must be assumptions made about the linearity of the measurement model in order to implement the method described in this project. If this assumption is not made, the idea to take the average of the measurements to compute the likelihood cannot be done, and a more computational expensive and difficult to implement method must be taken.

The main task highlighted by the work for this project was to understand the data that is being worked with and model the solution to work best with the data. This became prevalant when working with the abundant and highly unstructured data that the event camera output. Developing methods to segment the data and take advantage of the underlying structure of the data not only lead to interpretable outputs regarding the predictions; but also, provide computational ease for the method as well.

References

[1] G. Gallego, H. Rebecq, and D. Scaramuzza, "A unifying contrast maximization framework for event cameras, with applications to motion, depth, and optical flow estimation," *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition*, 2018.

[2] T. Stoffregen and L. Kleeman, "Event Cameras, Contrast Maximization and Reward Functions: An Analysis," in *2019 IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR).* Long Beach, CA, USA: IEEE, Jun. 2019, pp. 12 292–12 300. [Online]. Available: https://ieeexplore.ieee.org/document/8954356/

[3] G. Gallego, T. Delbruck, G. M. Orchard, C. Bartolozzi, B. Taba, A. Censi, S. Leutenegger, A. Davison, J. Conradt, K. Daniilidis, and D. Scaramuzza, "Event-based Vision: A Survey," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, pp. 1–1, 2020. [Online]. Available: https://ieeexplore.ieee.org/document/9138762/

[4] P. Lichtsteiner, C. Posch, and T. Delbruck, "A 128×128 120 db 15 μs latency asynchronous temporal contrast vision sensor," *IEEE Journal of Solid-State Circuits*, vol. 43, no. 2, p. 566–576, 2008.

[5] E. Mueggler, H. Rebecq, G. Gallego, T. Delbruck, and D. Scaramuzza, "The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam," *The International Journal of Robotics Research*, vol. 36, no. 2, p. 142–149, 2017.

[6] ——, "The event-camera dataset and simulator: Event-based data for pose estimation, visual odometry, and slam," *The International Journal of Robotics Research*, vol. 36, no. 2, p. 142–149, 2017.

1 Appendix

1.1 Prior and Posterior Distributions at Partitions

1.1.1 First Partition

Prior Distribution for the first segment was selected as the standard Gaussian $\mathcal{N}(, I_{2\times 2})$.

The prior on the motion parameters are shown with figures 1 and 2.

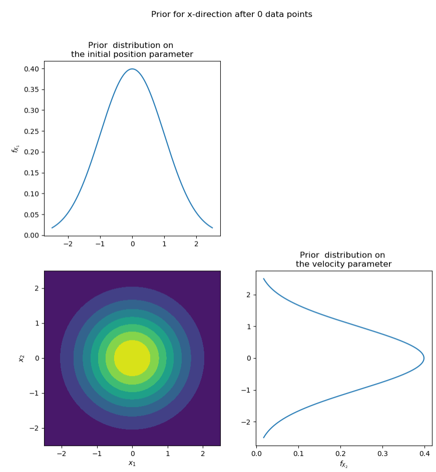

Figure 1: Prior distribution on the motion parameters in the x-direction at the first segment of the data

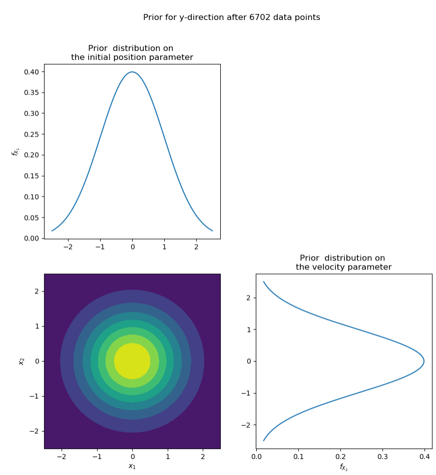

Figure 2: Prior distribution on the motion parameters in the y-direction at the first segment of the data

Posterior Distribution for the first segment is shown in figures 3 and 4.

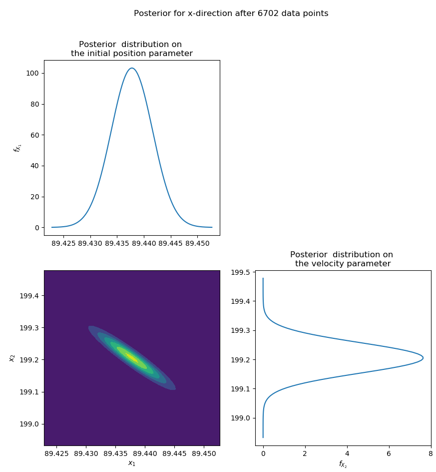

Figure 3: Prior distribution on the motion parameters in the x-direction at the first segment of the data

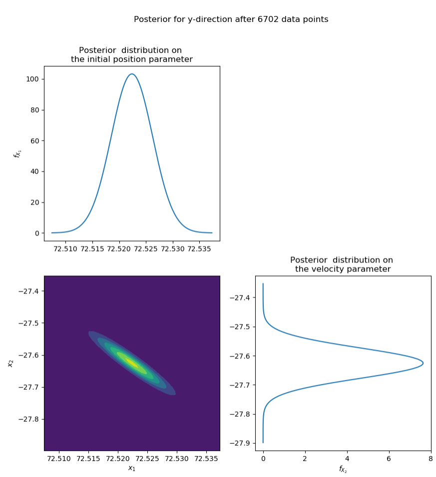

Figure 4: Prior distribution on the motion parameters in the y-direction at the first segment of the data

1.1.2 50^{th} Partition

Prior Distribution for the first segment was selected as the standard Gaussian $\mathcal{N}(, I_{2\times 2})$. The prior on the motion parameters are shown with figures 5 and 6.

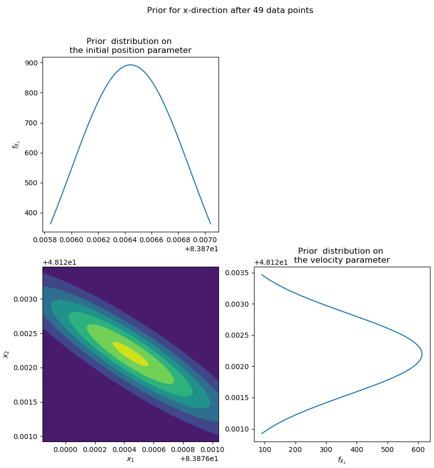

Figure 5: Prior distribution on the motion parameters in the x-direction at the 50^{th} segment of the data

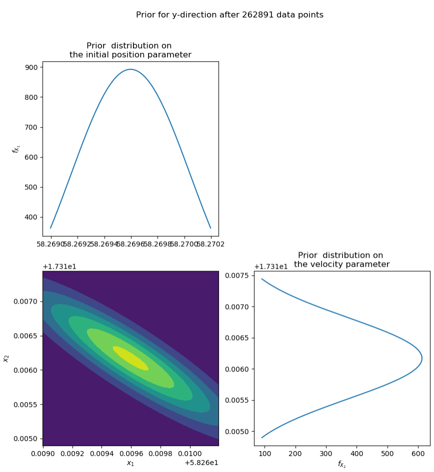

Figure 6: Prior distribution on the motion parameters in the y-direction at the 50^{th} segment of the data

Posterior Distribution for the first segment is shown in figures **??** and **??**.

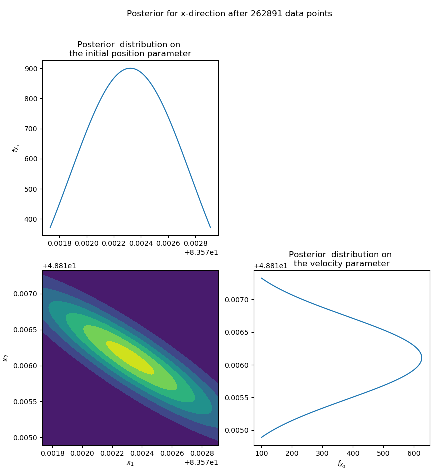

Figure 7: Prior distribution on the motion parameters in the x-direction at the 50^{th} segment of the data

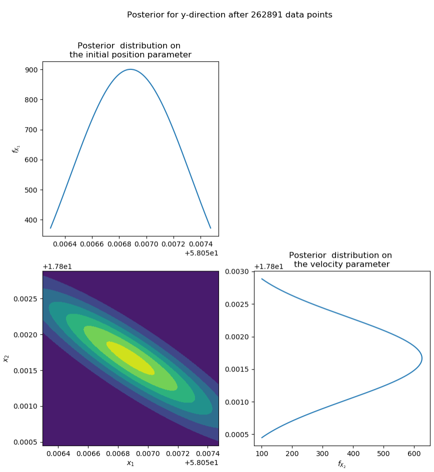

Figure 8: Prior distribution on the motion parameters in the y-direction at the 50^{th} segment of the data

1.1.3 100^{th} Partition

Prior Distribution for the first segment was selected as the standard Gaussian $\mathcal{N}(, I_{2\times 2})$.

The prior on the motion parameters are shown with figures 9 and 10.

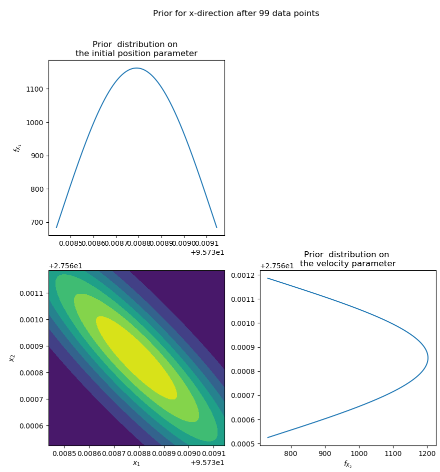

Figure 9: Prior distribution on the motion parameters in the x-direction at the 100^{th} segment of the data

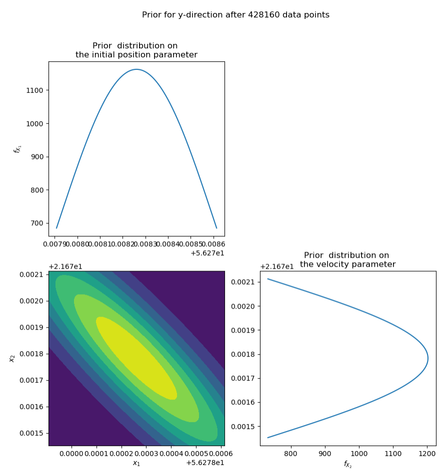

Figure 10: Prior distribution on the motion parameters in the y-direction at the 100^{th} segment of the data

Posterior Distribution for the first segment is shown in figures 11 and 12.

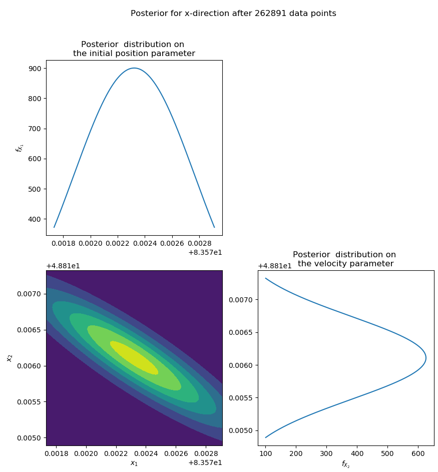

Figure 11: Prior distribution on the motion parameters in the x-direction at the 100^{th} segment of the data

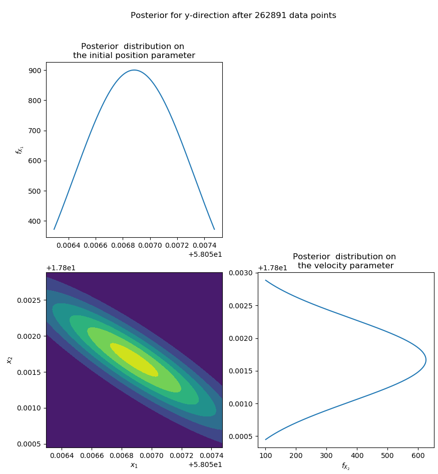

Figure 12: Prior distribution on the motion parameters in the y-direction at the 100^{th} segment of the data

1.2 Code

1.2.1 Linear Gaussian Regression

```
import glob
from operator import mul
import os
import sys

import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import numpy as np
import scipy.stats as stats
import target_distributions as td
from event_plotter import EventPlotter
from target_distributions import evaluate_gaussian
from plotter import plot_bivariate_gauss

```

```
from tqdm import tqdm


def compute_gaussian_predicted_marginal(mu, Sigma, A, Gamma):
    """Compute the marginal distribution of Y where

    Y = A theta + xi,   xi ~ N(0, Gamma)

    with theta distributed according to a Gaussian with mean mu and
    covariance Sigma

    theta is a vector of size (d)

    Inputs
    ------
    mu   : (d, ) array, the mean of theta
    Sigma: (d, d) array, the covariance of sigma
    A    : (n, d) linear model
    Gamma: (n, n) array, the covariance of xi

    Returns
    -------
    (mean, covariance, ASigma)
    """
    ASigma = np.dot(A, Sigma)
    cov = np.dot(ASigma, A.T) + Gamma
    return np.dot(A, mu), cov, ASigma


def linear_gaussian_inverse_problem(
    prior_mean, prior_cov, A, Gamma, data, inverted_noise_var=False
):
    """Compute the posterior of a linear-Gaussian inverse problem

    Inputs
    ------
    prior_mean: (d, ) prior mean
    prior_cov : (d, d) prior Covariance
    A         : (M, d) linear model
    Gamma     : (M, M) noise model
    data      : (M, ) data

    Outputs
    -------
    (post_mean, post_covcov): posterior mean and posterior covariance
    """
    # Check if all entries for gamma are equal
    inverted = False
    if np.all(np.diag(Gamma) == Gamma[0, 0]):
        inverted = True
        Gamma = np.eye(Gamma.shape[0]) * 1 / Gamma[0, 0]

    # Compute Marginal
    mean_pred, cov_pred, ASigma = compute_gaussian_predicted_marginal(
        prior_mean, prior_cov, A, Gamma
    )

    if inverted:
        print(np.linalg.inv(cov_pred))
        first_expression = A.T @ Gamma @ A + np.linalg.inv(cov_pred)
        second_expression = A.T @ Gamma @ data + np.linalg.inv(cov_pred) @ prior_mean
        post_mean = np.linalg.solve(first_expression, second_expression)
        post_cov = np.linalg.inv(first_expression)
    else:
        print(A.T @ np.linalg.solve(Gamma, A))
        post_mean = np.linalg.solve(
```

```
            A.T @ np.linalg.solve(Gamma, A) + np.linalg.inv(prior_cov),
            A.T @ np.linalg.solve(Gamma, data) + np.linalg.solve(prior_cov, prior_mean),
        )
        post_cov = np.linalg.inv(
            A.T @ np.linalg.solve(Gamma, A) + np.linalg.inv(prior_cov)
        )
    return post_mean, post_cov


def linear_regression(
    xtrain, ytrain, xpredict, prior_mean, prior_cov, noise_var, inverted_noise_var=False
):
    """Perform linear regression

    Inputs
    ------
    xtrain    : (N, d) training inputs
    ytrain    : (N) training outputs
    xpredict  : (M, d) an array of points where predictions are required (If none, then is
        ignored)
    prior_mean: (d+1) prior mean
    prior_cov : (d+1, d+1) prior covariance
    noise_var : (N, ) noise variance of each individual output

    Outputs
    -------
    post_mean : posterior mean of the parameters
    post_cov  : posterior covariance of the parameters
    pred_mean : mean prediction at the predict points (optional)
    pred_cov  : covariance of the prediction (optional)

    Notes
    -----
    Predictions assumes no noise: just y = x^T theta
    x
    """
    # Perform inference
    N, d = xtrain.shape
    A = np.ones((N, d + 1))
    A[:, 1:] = xtrain
    # Gamma = np.diag(noise_var)
    post_mean, post_cov = linear_gaussian_inverse_problem(
        prior_mean,
        prior_cov,
        A,
        noise_var,
        ytrain,
        inverted_noise_var=inverted_noise_var,
    )

    # Perform prediction: which is just computing the marginal with a different

    if xpredict is not None:
        Npred, dpred = xpredict.shape
        assert d == dpred, "prediction and testing features must be the same"
        Apred = np.ones((Npred, d + 1))
        Apred[:, 1:] = xpredict
        pred_mean, pred_cov, _ = compute_gaussian_predicted_marginal(
            post_mean, post_cov, Apred, 0
        )
        return post_mean, post_cov, pred_mean, pred_cov
    else:
        return post_mean, post_cov


def store_events_in_matrix(filename):
```

```
    """
    Store the events in a matrix.

    Inputs
    ========
    events: numpy array
    num_of_repeat: numpy array
    """
    EVENT_FILE = (
        f"{sys.path[0]}/../data/new_slider_close/slider_close/split_data/{filename}"
    )
    events = np.genfromtxt(EVENT_FILE, delimiter=" ")
    t = events[:, 0]
    x = events[:, 1]
    y = events[:, 2]
    p = events[:, 3]

    uniquew, inverse = np.unique(t, return_inverse=True)
    counts = np.bincount(inverse)
    max_count = np.max(counts)

    uniques_x = np.zeros(shape=(uniquew.shape[0], max_count + 3))
    uniques_y = np.zeros(shape=(uniquew.shape[0], max_count + 3))

    uniques_x[:] = np.nan
    uniques_y[:] = np.nan

    uniques_x[:, 0] = uniquew
    uniques_y[:, 0] = uniquew
    uniques_x[:, 1] = counts
    uniques_y[:, 1] = counts

    # Store the corresponding values from x and y, vectorized
    # uniques_x[:, 2:] = np.repeat(x, counts)
    # uniques_y[:, 2:] = np.repeat(y, counts)

    for i in range(uniquew.shape[0]):
        num_repeat = int(uniques_x[i, 1])
        uniques_x[i, 3 : 3 + num_repeat] = x[inverse == i]
        uniques_y[i, 3 : 3 + num_repeat] = y[inverse == i]
        mean_of_x = np.mean(uniques_x[i, 3 : 3 + num_repeat], axis=0)
        mean_of_y = np.mean(uniques_y[i, 3 : 3 + num_repeat], axis=0)
        uniques_x[i, 2] = mean_of_x
        uniques_y[i, 2] = mean_of_y

    return uniques_x, uniques_y


def compute_likelihood(theta, data, noise_var):
    """
    Compute the likelihood of the data given the parameters theta
    """
    noise_mat = np.eye(data.shape[0]) * noise_var
    evaluate_gaussian(theta, data, noise_var)


def likelihood_mean(theta, time):
    """
    Compute the mean of the likelihood function
    """
    return theta[0] + theta[1] * time


def likelihood_cov(theta, time):
    """
    Compute the covariance of the likelihood function
```

```python

    theta: (d, d)
    time (N, )
    """
    # Return array of size (N, d, d)


def compute_confidence_interval(mu, cov):
    """
    Compute the confidence interval of the parameters theta
    """
    # Compute the likelihood
    # Compute the posterior
    # Compute the confidence interval
    # return theta_low, theta_high

    # sample from a multivariate gaussian 10E5 times
    samples = mu + np.sqrt(cov) * np.random.randn(int(10e5))
    evaluations = np.where(np.abs(samples - mu) < 1e-6, 1, 0)

    z = 1.96
    std_err = np.std(evaluations) / np.sqrt(10e5)

    lower_bound = mu - z * std_err
    upper_bound = mu + z * std_err
    return lower_bound, upper_bound


def main():
    intrinsic_matrix = np.genfromtxt(
        f"{sys.path[0]}/intrinsic_params.txt", delimiter=" "
    )

    files = os.listdir(f"{sys.path[0]}/../data/slider_far/split_data")
    sorted_files = sorted(files)

    # Define data parameters
    num_data = 7500

    # Load the true parameters that are given from the data in homogenous coordinates
    true = np.array([0.14, 0, 1])
    true_pixels = intrinsic_matrix @ true

    # Define prior parameters
    # X-direction
    prior_mean_x = np.array([0, 0]).reshape(2, 1)
    prior_cov_x = np.array([[1, 0], [0, 1]])

    # Y-direction
    prior_mean_y = np.array([0, 0]).reshape(2, 1)
    prior_cov_y = np.array([[1, 0], [0, 1]])

    # Array to store the results
    x_means = np.zeros(shape=(len(sorted_files), 2))
    x_covs = np.zeros(shape=(len(sorted_files), 2, 2))
    x_space = np.linspace(0, x_means.shape[0] - 1, x_means.shape[0])

    # fig, axs = plt.subplots(2, 1, figsize=(10, 10))
    truex = true_pixels[0]
    truey = true_pixels[1]

    y_means = np.zeros(shape=(len(sorted_files), 2))
    y_covs = np.zeros(shape=(len(sorted_files), 2, 2))

    # Array to store the velocity
    velocities = np.zeros(shape=(len(sorted_files), 2))
```

```
    num_data = 0

    for file_idx, filename in tqdm(enumerate(sorted_files)):

        # Predicting in thex-direction
        # Load the data
        x_values, y_values = store_events_in_matrix(filename)

        training_time = x_values[:, 0].reshape(x_values.shape[0], 1)
        training_px = x_values[:, 2].reshape(x_values.shape[0], 1)

        xpredictions = np.linspace(-10, 10, 1000)[:, np.newaxis]

        noise_var = np.eye(training_px.shape[0]) * 0.01

        (
            post_mean_x,
            post_cov_x,
            post_pred_pred_mean_x,
            post_pred_pred_cov_x,
        ) = linear_regression(
            training_time,
            training_px,
            xpredictions,
            prior_mean_x,
            prior_cov_x,
            noise_var,
        )

        # Get the mean and covariance of the prediction
        mean_x = likelihood_mean(post_mean_x, training_time)

        # Predicting in the y-direction
        training_py = y_values[:, 2].reshape(y_values.shape[0], 1)

        ypredictions = np.linspace(-10, 10, 1000)[:, np.newaxis]

        (
            post_mean_y,
            post_cov_y,
            post_pred_pred_mean_y,
            post_pred_pred_cov_y,
        ) = linear_regression(
            training_time,
            training_py,
            ypredictions,
            prior_mean_y,
            prior_cov_y,
            noise_var,
        )

        # Get the mean and covariance of the prediction
        mean_y = likelihood_mean(post_mean_y, training_time)
        cov_y = likelihood_cov(post_cov_y, training_time)

        # Store the results
        x_means[file_idx] = post_mean_x.reshape(2,)
        x_covs[file_idx] = post_cov_x

        y_means[file_idx] = post_mean_y.reshape(2,)
        y_covs[file_idx] = post_cov_y

        # Compute the velocity
        pixels_x = mean_x[-1] - mean_x[0]
        pixels_y = mean_y[-1] - mean_y[0]
```

```
        pixels = np.array([pixels_x[0], pixels_y[0], 1])
        camera_coordiantes = np.linalg.solve(intrinsic_matrix, pixels)
        homogeneous_coordinates = camera_coordiantes / camera_coordiantes[-1]
        x_vel = homogeneous_coordinates[0]
        y_vel = homogeneous_coordinates[1]

        velocities[file_idx, :] = np.array([x_vel, y_vel]).reshape(2,)

        # Plot the bivariate distribution of the prior

        if file_idx == 0:
            prior_offset_range_x = np.linspace(
                prior_mean_x[0][0] - 2.5, prior_mean_x[0][0] + 2.5, 1000
            )
            prior_slope_range_x = np.linspace(
                prior_mean_x[1][0] - 2.5, prior_mean_x[1][0] + 2.5, 1000
            )
            prior_offset_range_y = np.linspace(
                prior_mean_y[0][0] - 2.5, prior_mean_y[0][0] + 2.5, 1000
            )
            prior_slope_range_y = np.linspace(
                prior_mean_y[1][0] - 2.5, prior_mean_y[1][0] + 2.5, 1000
            )

            offset_range_x = np.linspace(
                post_mean_x[0][0] - 1000 * post_cov_x[0, 0],
                post_mean_x[0][0] + 1000 * post_cov_x[0, 0],
                1000,
            )
            slope_range_x = np.linspace(
                post_mean_x[1][0] - 100 * post_cov_x[1, 1],
                post_mean_x[1][0] + 100 * post_cov_x[1, 1],
                100,
            )
            offset_range_y = np.linspace(
                post_mean_y[0][0] - 1000 * post_cov_y[0, 0],
                post_mean_y[0][0] + 1000 * post_cov_y[0, 0],
                1000,
            )
            slope_range_y = np.linspace(
                post_mean_y[1][0] - 100 * post_cov_y[1, 1],
                post_mean_y[1][0] + 100 * post_cov_y[1, 1],
                100,
            )

        else:

            prior_offset_range_x = np.linspace(
                prior_mean_x[0][0] - 3000 * prior_cov_x[0, 0],
                prior_mean_x[0][0] + 3000 * prior_cov_x[0, 0],
                3000,
            )
            prior_slope_range_x = np.linspace(
                prior_mean_x[1][0] - 3000 * prior_cov_x[1, 1],
                prior_mean_x[1][0] + 3000 * prior_cov_x[1, 1],
                3000,
            )
            prior_offset_range_y = np.linspace(
                prior_mean_y[0][0] - 3000 * prior_cov_y[0, 0],
                prior_mean_y[0][0] + 3000 * prior_cov_y[0, 0],
                3000,
            )
            prior_slope_range_y = np.linspace(
                prior_mean_y[1][0] - 3000 * prior_cov_y[1, 1],
                prior_mean_y[1][0] + 3000 * prior_cov_y[1, 1],
                3000,
```

```python
            )

            # Plot the bivariate distribution of the posterior
            offset_range_x = np.linspace(
                post_mean_x[0][0] - 3000 * post_cov_x[0, 0],
                post_mean_x[0][0] + 3000 * post_cov_x[0, 0],
                3000,
            )
            slope_range_x = np.linspace(
                post_mean_x[1][0] - 3000 * post_cov_x[1, 1],
                post_mean_x[1][0] + 3000 * post_cov_x[1, 1],
                3000,
            )
            offset_range_y = np.linspace(
                post_mean_y[0][0] - 3000 * post_cov_y[0, 0],
                post_mean_y[0][0] + 3000 * post_cov_y[0, 0],
                3000,
            )
            slope_range_y = np.linspace(
                post_mean_y[1][0] - 3000 * post_cov_y[1, 1],
                post_mean_y[1][0] + 3000 * post_cov_y[1, 1],
                3000,
            )

        num_data += x_values.shape[0]

        FIG_DIR = f"{sys.path[0]}/../Report/figures/"

        if file_idx in [0, 49, 99]:

            print("Compute Confidence Intervals:")

            lower_bound_x, upper_bound_x = compute_confidence_interval(
                post_mean_x[1][0], post_cov_x[1, 1]
            )
            lower_bound_y, upper_bound_y = compute_confidence_interval(
                post_mean_y[1][0], post_cov_y[1, 1]
            )

            print(
                f"X-direction: {lower_bound_x} <= {post_mean_x[1][0]} <= {upper_bound_x}"
            )
            print(
                f"Y-direction: {lower_bound_y} <= {post_mean_y[1][0]} <= {upper_bound_y}"
            )

            # Plot the bivariate distribution of the prior
            figp, axp = plot_bivariate_gauss(
                prior_offset_range_x,
                prior_slope_range_x,
                prior_mean_x.reshape(2,),
                prior_cov_x,
                ax_title="Prior",
            )
            figp.suptitle(f"Prior for x-direction after {file_idx} data points")
            plt.savefig(f"{FIG_DIR}/prior_x_{file_idx}.png")
            plt.close()

            figp, axp = plot_bivariate_gauss(
                prior_offset_range_y,
                prior_slope_range_y,
                prior_mean_y.reshape(2,),
                prior_cov_y,
                ax_title="Prior",
            )
            figp.suptitle(f"Prior for y-direction after {num_data} data points")
```

```python
            plt.savefig(f"{FIG_DIR}/prior_y_{file_idx}.png")
            plt.close()

            # Plot the bivariate distribution of the posterior
            figp, axp = plot_bivariate_gauss(
                offset_range_x,
                slope_range_x,
                post_mean_x.reshape(2,),
                post_cov_x,
                ax_title="Posterior",
            )
            figp.suptitle(f"Posterior for x-direction after {num_data} data points")
            plt.savefig(f"{FIG_DIR}/posterior_x_{file_idx}.png")
            plt.close()

            figp, axp = plot_bivariate_gauss(
                offset_range_y,
                slope_range_y,
                post_mean_y.reshape(2,),
                post_cov_y,
                ax_title="Posterior",
            )
            figp.suptitle(f"Posterior for y-direction after {num_data} data points")
            plt.savefig(f"{FIG_DIR}/posterior_y_{file_idx}.png")
            plt.close()

            # Generate 3D plot of the data
            fig = plt.figure(figsize=(10, 10))
            fig.suptitle(f"Prediction after {num_data} data points")

            ax = fig.add_subplot(211)
            ax.scatter(training_time, training_px, s=1, label="Events")
            ax.plot(training_time, mean_x, c="k", label="Predicted trajectory")
            ax.set_title(r"Regression for x-direction")
            ax.set_xlabel("Time")
            ax.set_ylabel("X-direction (pixels)")
            ax.legend()
            ax = fig.add_subplot(212)
            ax.scatter(training_time, training_py, s=1, label="events")
            ax.plot(training_time, mean_y, c="k", label="Predicted trajectory")
            ax.set_title(r"Regression for y-direction")
            ax.set_xlabel("Time")
            ax.set_ylabel("Y-direction (pixels)")
            ax.legend()
            plt.savefig(f"{FIG_DIR}/axes_prediction_{file_idx}.png")
            plt.close()
            fig = plt.figure()
            fig.suptitle(f"Prediction after {num_data} data points")
            ax = fig.add_subplot(111, projection="3d")
            ax.scatter(
                training_time,
                training_px,
                training_py,
                c="b",
                marker="o",
                s=1,
                label="Events",
            )
            ax.set_xlabel("Time")
            ax.set_ylabel("X-direction")
            ax.set_zlabel("Y-direction")
            x_dir = (post_mean_x[0][0] + post_mean_x[1][0] * training_time).reshape(
                training_time.shape[0],
            )
            y_dir = (post_mean_y[0][0] + post_mean_y[1][0] * training_time).reshape(
                training_time.shape[0],
```

```
            )

            ax.plot(
                training_time,
                x_dir,
                y_dir,
                c="r",
                linewidth=7.0,
                label="Predicted trajectory",
            )
            ax.legend()
            plt.savefig(f"{FIG_DIR}/3d_prediction_{file_idx}.png")
            # plt.show()
            plt.close()

            evaluate_gaussian = lambda x, mean, cov: np.exp(
                -0.5 * (x - mean) ** 2 / cov
            )

            fig, axs = plt.subplots(2, 1, figsize=(10, 10))
            truex = true_pixels[0]
            truey = true_pixels[1]
            if abs(post_mean_x[1][0] - truex) > 100:
                slope_range_x = np.linspace(-200, 200, 1000)
            else:
                slope_range_x = np.linspace(
                    post_mean_x[1][0] - 100, post_mean_x[1][0] + 100, 1000
                )
            if abs(post_mean_y[1][0] - truey) > 100:
                slope_range_y = np.linspace(-200, 200, 1000)
            else:
                slope_range_y = np.linspace(
                    post_mean_y[1][0] - 100, post_mean_y[1][0] + 100, 1000
                )
            # slope_range_y = np.linspace(-100, 100, 1000)
            axs[0].plot(
                slope_range_x,
                evaluate_gaussian(slope_range_x, post_mean_x[1][0], post_cov_x[1, 1]),
                label="Prediction on velocity",
            )
            axs[1].plot(
                slope_range_y,
                evaluate_gaussian(slope_range_y, post_mean_y[1][0], post_cov_y[1, 1]),
                label="Prediction on velocity",
            )
            # Draw a vertical like at the true x
            axs[0].axvline(truex, c="k", linestyle="--", label="True valocity")
            axs[0].axvline(
                post_mean_x[1][0], c="r", linewidth=1.0, label="Posterior mean"
            )
            axs[1].axvline(truey, c="k", linestyle="--", label="True velocity")
            axs[1].axvline(
                post_mean_y[1][0], c="r", linewidth=1.0, label="Posterior mean"
            )
            axs[0].set_title(
                f"Posterior on the velocity for x-direction after {num_data} data points vs true parameter"
            )
            axs[1].set_title(
                f"Posterior on the velocity for y-direction after {num_data} data points vs true parameter"
            )
            axs[1].set_xlabel("Velocity (px/s)")
            axs[0].set_ylabel("Probability")
            axs[1].set_ylabel("Probability")
            axs[0].legend()
```

```
            axs[1].legend()
            plt.savefig(f"{FIG_DIR}/posterior_slope_{file_idx}_vs_true.png")
            plt.close()
        prior_mean_x = post_mean_x
        prior_cov_x = post_cov_x

        prior_mean_y = post_mean_y
        prior_cov_y = post_cov_y

    fig, axs = plt.subplots(2, 1, figsize=(10, 10))
    axs[0].axhline(truex, c="b", linestyle="--", label="True velocity")
    axs[1].axhline(truey, c="b", linestyle="--", label="True velocity")
    axs[0].plot(x_space, x_means[:, 1], c="k", label="Predicted vel in x")
    axs[1].plot(x_space, y_means[:, 1], c="k", label="Predicted vel in y")
    axs[0].set_ylabel("Velocity (px/s)")
    axs[1].set_xlabel("Data Segments")
    axs[1].set_ylabel("Velocity (px/s)")
    fig.suptitle("Convergence Predicted Velocity over Data Segments")
    axs[0].legend()
    axs[1].legend()
    plt.savefig(f"{FIG_DIR}/predicted_velocity_over_data_segments.png")
    plt.show()


if __name__ == "__main__":

    main()
```

1.2.2 Plotting and Toy Problems

```
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

class EventPlotter:
    '''
    Class for plotting any event that is input
    '''

    def __init__(self, t, x, y, pol = None):
        self.x = x
        self.y = y
        self.time = t
        if pol is not None:
            self.pol = pol
            self.pol_exists = True
        else:
            self.pol_exists = False


    def plot_events(self, t_start=0, t_end=-1):
        '''
        Plot the events
        '''
        fig = plt.figure()
        ax = fig.add_subplot(111, projection='3d')
        ax.scatter(self.time[t_start:t_end], self.x[t_start:t_end], self.y[t_start:t_end])
        ax.set_xlabel('Time')
        ax.set_ylabel('X')
        ax.set_zlabel('Y')
        plt.show()


    def plot_x_axis(self, t_start=0, t_end=-1):
        '''
        Plot the events only on the x-axis
        '''
        fig = plt.figure()
```

```python
        ax = fig.add_subplot(111)
        ax.scatter(self.time[t_start:t_end], self.x[t_start:t_end], s=1)
        ax.set_xlabel('X')
        ax.set_ylabel('Time')
        plt.show()

    def plot_y_axis(self, t_start=0, t_end=-1):
        '''
        Plot the events only on the y-axis
        '''
        fig = plt.figure()
        ax = fig.add_subplot(111)
        ax.scatter(self.time[t_start:t_end], self.y[t_start:t_end], s=1)
        ax.set_xlabel('Y')
        ax.set_ylabel('Time')
        plt.show()
```

```python
import numpy as np
import matplotlib.pyplot as plt

def normpdf(x, mean, cov):
    n, N = x.shape
    preexp = 1.0 / (2.0 * np.pi)**(n/2) / np.linalg.det(cov)**0.5
    diff = x - np.tile(mean[:, np.newaxis], (1, N))
    sol = np.linalg.solve(cov, diff)
    inexp = np.einsum("ij,ij->j",diff, sol)
    out = preexp * np.exp(-0.5 * inexp)
    return out

def eval_normpdf_on_grid(x, y, mean, cov):
    XX, YY = np.meshgrid(x,y)
    pts = np.stack((XX.reshape(-1), YY.reshape(-1)),axis=0)
    evals = normpdf(pts, mean, cov).reshape(XX.shape)
    return XX, YY, evals

def plot_bivariate_gauss(x, y, mean, cov, ax_title=None):
    std1 = cov[0,0]**0.5
    std2 = cov[1,1]**0.5
    mean1 = mean[0]
    mean2 = mean[1]
    XX, YY, evals = eval_normpdf_on_grid(x, y, mean, cov)
    fig, axis = plt.subplots(2,2, figsize=(10,10))
    axis[0,0].plot(x, normpdf(x[np.newaxis,:], np.array([mean1]), np.array([[std1**2]])))
    axis[0,0].set_ylabel(r'$f_{X_1}$')
    axis[0,0].set_title(f'{ax_title} distribution on \nthe initial position parameter')
    axis[1,1].plot(normpdf(y[np.newaxis,:], np.array([mean2]), np.array([[std2**2]])),y)
    axis[1,1].set_xlabel(r'$f_{X_2}$')
    axis[1,1].set_title(f'{ax_title} distribution on \nthe velocity parameter')
    axis[1,0].contourf(XX, YY, evals)
    axis[1,0].set_xlabel(r'$x_1$')
    axis[1,0].set_ylabel(r'$x_2$')
    axis[0,1].set_visible(False)
    return fig, axis
```

```python
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import math


class ToyEvent:
    """
    Class for generating toy problems.
    """

    def __init__(self, timespan, dt, x_parameter, y_parameter, noise, shape='dot', num_pts
        =1, offtimed=False):
```

```
        '''
        Initialize the toy problem class
        '''
        self.timespan = timespan
        self.dt_ = dt
        self.dt = dt if type(dt) == int else int(dt)
        self.t = int(math.ceil(timespan/dt))

        self.tmp_x = []
        self.tmp_y = []

        self.data_dim = 10

        self.x_parameter = np.ones(shape=(num_pts*self.data_dim)) * x_parameter
        self.y_parameter = y_parameter
        self.shape = shape
        self.offtimed = offtimed
        self.noise = noise
        self.num_pts = num_pts
        tmp_time = np.arange(0, timespan, dt).reshape(self.t, 1)
        self.time = tmp_time


        # Stack if there are multiple objects to track
        for i in range((num_pts*self.data_dim)-1):
            self.time = np.hstack((self.time, tmp_time))
        self.initial_points_x = np.ones(shape=(
            self.t, self.num_pts*self.data_dim)) * 0 #@ (5*np.random.randn(self.num_pts*self.data_dim, self.num_pts*self.data_dim))
        self.initial_points_y = np.ones(shape=(
            self.t, self.num_pts * self.data_dim)) * 0 #@ (5*np.random.randn(self.num_pts*self.data_dim, self.num_pts*self.data_dim))
        self.x = np.zeros(shape=(self.t, self.num_pts * self.data_dim))
        self.y = np.zeros(shape=(self.t, self.num_pts * self.data_dim))

        # print(self.x.shape)

        self.generate_toy_problem()

    def generate_toy_problem(self):
        """
        Generates a toy problem.
        :return:
        """
        if self.shape == 'dot' and self.offtimed == False:
            return self.generate_dot_problem()
        elif self.shape == 'dot' and self.offtimed:
            return self.generate_offtimed_dot_problem()
        elif self.shape == 'circle':
            return self.generate_circle_problem()
        else:
            raise ValueError('Shape not recognized.')

    def generate_dot_problem(self):
        '''
        Generate a multiple dot with noise
        '''
        x = self.x_parameter * self.time
        x_noise = self.noise * np.random.randn(self.t, self.num_pts * self.data_dim)
        self.x = x + x_noise + self.initial_points_x

        y = self.y_parameter * self.time
        y_noise = self.noise * np.random.randn(self.t, self.num_pts * self.data_dim)
        self.y = y + y_noise + self.initial_points_y

    def get_number_of_repeating_points(self, arr):
```

```
        '''
        Returns the number of repeating points in an np.array
        '''
        return len(np.unique(arr, axis=0))

    def get_nearest_value(self, arr1, arr2):
        '''
        Returns the nearest value in arr2 to the value in arr1
        '''
        # return (np.abs(arr1 - arr2)).argmin(axis=1)
        return np.array([np.argmin(np.abs(arr2 - x)) for x in arr1])

    def generate_offtimed_dot_problem(self):
        '''
        Generate a multiple dot with noise
        '''
        time = 0
        events = np.array([[0, 0, time]])
        self.tmp_x = self.x
        self.tmp_y = self.y
        while time < self.timespan:
            num_events = int(5 + 100*np.random.randn())
            num_events = -1*num_events if num_events < 0 else num_events
            for i in range(num_events):
                x = np.mean(self.x_parameter) * time + self.noise * np.random.randn()
                y = np.mean(self.y_parameter) * time + self.noise * np.random.randn()
                events = np.vstack((events, [x, y, time]))
                # add x to tmp_x with the corresponding time index
                time_idx = np.where(self.time == time)
                # print(time)
                # self.tmp_x[np.where(self.time == time)] += x
                # self.tmp_y[np.where(self.time == time)] += y
            time += self.dt_

        self.x = events[:, 0].reshape(events[:, 0].shape[0], 1)
        self.y = events[:, 1].reshape(events[:, 1].shape[0], 1)
        self.time = events[:, 2].reshape(events[:, 2].shape[0], 1)


    def ravel_events(self):
        '''
        Ravels the events into a single array
        '''
        if self.offtimed == False: raise ValueError('Not implemented')
        # find the indices of the repeating values in the time array
```