Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Even Moa Myklebust

# A robustness evaluation of the latent manifold tuning model

Master's thesis in Industrial Mathematics
Supervisor: Benjamin Adric Dunn
July 2020

NTNU
Norwegian University of
Science and Technology

Even Moa Myklebust

# A robustness evaluation of the latent manifold tuning model

Master's thesis in Industrial Mathematics
Supervisor: Benjamin Adric Dunn
July 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Mathematical Sciences

Norwegian University of
Science and Technology

# Abstract

Recent advances in neural recording techniques give access to simultaneous recordings of increasingly many neurons. Dimensionality reduction techniques can be used to investigate how neurons work together as a system by extracting low-dimensional latent variables from high-dimensional neural data. A doubly nonlinear model for latent variable inference called the latent manifold tuning model was introduced by Anqi Wu and colleagues in 2017. We explicitly state some assumptions that were not mentioned in the original article and investigate how the model's initialization affects its convergence. We evaluate the robustness of the model with regards to different tuning strengths and data lengths and discover an optimal tuning strength that depends on the data length. Finally, we apply the model to neural data by Adrien Peyrache and colleagues, where we use a periodic covariance kernel to infer the head direction of a mouse.

# Sammendrag

Nye teknikker for nevrale opptak setter forskere i stand til å gjøre simultanopptak av aktiviteten til stadig flere nevroner. Teknikker for dimensjonsreduksjon kan brukes til å undersøke hvordan nevroner jobber sammen som et system ved å ekstrahere lavdimensjonale latente variabler fra høydimensjonal nevral data. En dobbelt ikke-lineær modell for inferens av latent variabler, kalt "the latent manifold tuning model", ble introdusert av Anqi Wu og medforfattere i 2017. Vi beskriver eksplisitt noen antakelser som ikke ble beskrevet i den opprinnelige artikkelen, og undersøker hvordan modellens initialisering påvirker dens konvergens. Vi evaluerer modellens robusthet i forhold til ulike responsstyrker (hvor mye aktiviteten til et nevron påvirkes av en variabel) og ulike datalengder, og finner en optimal responsstyrke som avhenger av datalengden. Avslutningsvis anvender vi modellen på nevral data innsamlet av Adrien Peyrache og medforfattere, hvor vi bruker en periodisk kovarians-kjerne til å estimere hoderetningen til en mus.

# Preface

This thesis concludes the course TMA4900 - Master's thesis in industrial mathematics at the Norwegian University of Science and Technology (NTNU), and marks my completion of the study program Physics and Mathematics with specialization in statistics.

I would like to thank my supervisor Benjamin Adric Dunn for introducing me to the exciting worlds of neuroscience and dimensionality reduction, for his unwavering support and for the warm and friendly environment he has created in his research group. To all the members of the group, thank you for lively discussions and inspiring talks, and a special thanks to Ben and Claudia for providing me with valuable feedback in the last stages of the project.

Thanks also to my friends and family for helping me take my mind off work and come back invigorated. The year 2020 will be remembered by many for the coronavirus lockdown, making every social interaction all the more valuable, either online, offline, or on some latent manifold. Finally, thanks to all the fantastic teachers I have had through the years, without whom I would still be grappling with the basics.

Even Moa Myklebust
Trondheim, Norway
July 2020

# Table of Contents


**Abstract** **I**

**Sammendrag** **I**

**Preface** **III**

**Table of Contents** **V**

**1 Introduction** **1**

1.1 Dimensionality reduction . . . . . 1
1.2 Neural tuning . . . . . 2
1.3 Probabilistic methods for dimensionality reduction . . . . . 3
1.4 Our contribution . . . . . 4

**2 Neural activity and mouse head direction data** **5**

2.1 Observed head direction and neural activity . . . . . 5
2.1.1 Neuron tuning . . . . . 7
2.2 Applying principal component analysis . . . . . 8

**3 Theoretical background** **11**

3.1 Parameter estimation . . . . . 11
3.1.1 Maximum likelihood estimate . . . . . 12
3.1.2 Maximum a posteriori estimate . . . . . 12
3.1.3 Heuristics for global optimization . . . . . 13
3.2 Gaussian processes . . . . . 13
3.2.1 Conditional distribution . . . . . 15
3.2.2 Noisy observations . . . . . 16
3.3 Approximate Gaussian processes . . . . . 17
3.4 Principal component analysis . . . . . 18
3.5 Generalized Linear Models . . . . . 19

**4 The latent manifold tuning model** **21**

4.1 The latent manifold tuning model . . . . . 21
4.1.1 Modeling the latent variable . . . . . 21
4.1.2 Modeling the spike counts and tuning curves . . . . . 22
4.2 Inference . . . . . 24

4.2.1 MAP estimate of tuning curves . . . 24
4.2.2 MAP estimate of the latent variable . . . 25
4.2.3 Gradient . . . 28
4.2.4 The iterative MAP procedure . . . 31

**5 Applying the LMT model to simulated and experimental data 33**
5.1 Convergence and pitfalls . . . 33
5.1.1 Flipping . . . 35
5.1.2 Scaling . . . 36
5.1.3 Partly flipped estimates . . . 37
5.1.4 Placement of the inducing grid . . . 38
5.2 Initialization . . . 38
5.2.1 Initial estimate for $\mathbf{F}$ . . . 38
5.2.2 Initial estimate for $\mathbf{X}$ . . . 40
5.3 Robustness evaluation . . . 42
5.3.1 Choosing between final estimates . . . 43
5.4 Application to head direction data . . . 45
5.4.1 Initialization 1: True X and optimal F . . . 47
5.4.2 Initialization 2: True X and estimated F . . . 48
5.4.3 Initialization 3: PCA initialization of X and optimal F . . . 49
5.4.4 Initialization 4: PCA initialization of X and estimated F . . . 50
5.4.5 Initialization 5: Flat initialization of X and estimated F . . . 51
5.4.6 Comparison of different initializations . . . 52

**6 Discussion and further work 53**
6.1 Simulated data . . . 53
6.2 Robustness evaluation . . . 53
6.3 Head direction data . . . 54
6.4 Future work . . . 54
6.5 Conclusion . . . 55

**Bibliography 57**

**Appendices 61**

**A Theorems and derivations 61**
A.1 Matrix calculus . . . 61
A.1.1 Maximizing the fraction of two quadratic forms . . . 61
A.1.2 Matrix inversion lemma . . . 61
A.1.3 Theorem 1.3.22 from Horn and Johnson (1985) . . . 61
A.1.4 Matrix differentiation . . . 62
A.2 Bernoulli spike model . . . 62

**B Python code 63**
B.1 Function library . . . 63
B.2 Application to head direction dataset . . . 74
B.3 Robustness evaluation . . . 88

Chapter 1

# Introduction

In this chapter, we introduce the concepts of dimensionality reduction and neural tuning and contextualize the latent manifold tuning model by relating it to other dimensionality reduction techniques.

## 1.1 Dimensionality reduction

Recent advances in neural recording techniques enable simultaneous recording of increasingly many neurons (Nicolelis et al. (2003), Ahrens et al. (2013) and Steinmetz et al. (2018) are some examples). The number of simultaneously recorded neurons is growing exponentially, and comparisons have been made to Moore's law (Stevenson and Kording, 2011). These growing datasets present new opportunities to researchers, along with new challenges.

A lot of research has focused on understanding individual neurons' behavior in terms of their response to a set of external, measurable covariates (e.g., Mimica (2019)), but simultaneous recordings of many neurons open the field for more exploratory approaches. A time series of the activity of $N$ neurons can be viewed as an $N$-dimensional variable developing through time. In many cases, the population activity of these $N$ neurons can be represented by a lower-dimensional, *latent*, variable. *Dimensionality reduction techniques* (see Cunningham and Byron (2014) for an overview) can be used to understand the behavior of neurons on the population level by extracting a latent variable from the high-dimensional data.

Variables with dimensionality lower than four have the advantage of being easy to visualize, and a lot easier to interpret than high-dimensional variables. In addition, extracting latent variables from a population can provide new insight into what role the population plays in the brain. Some examples of low dimensional variables could be allocentric features of a task, such as the speed, direction, or position of the animal whose brain is recorded. Low-dimensional representations can also relate to the low dimensionality of the task the animal is performing (see Gao and Ganguli (2015)). The usefulness of dimensionality reduction techniques extends beyond neuroscience to any domain concerned with high-dimensional data, like environmental sciences or social networks modeling. The domain of this lower-dimensional variable is often referred to as the *latent manifold*.

## 1.2 Neural tuning

Determining how each neuron is related to the latent variable is essential in any dimensionality reduction technique. Every neuron has an electric potential that can be brought to a sharp increase in potential known as a *spike* for a short amount of time, after which it returns to its resting potential. A visualization of a spike is shown in Figure 1.1.

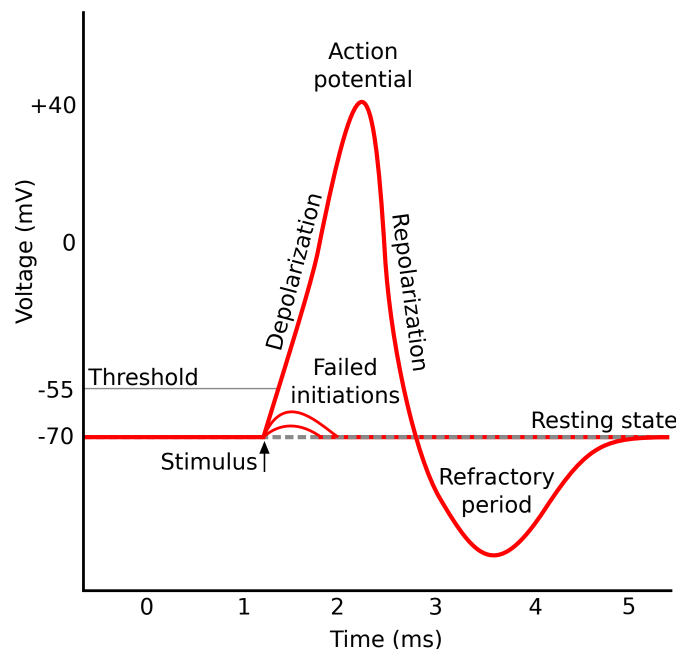


**Figure 1.1:** The red line shows the change in a neuron's action potential associated with a spike. Source: https://commons.wikimedia.org/w/index.php?curid=2241513. License: CC BY-SA 3.0.

Neurons are linked to other neurons, and when a neuron spikes, it passes on an electric pulse to its connected neurons. Depending on the type of connection, a received pulse may either increase or decrease the potential of the receiving neuron, increasing or decreasing the probability of the receiving neuron producing a spike of its own. In this way, signals caused by some internal or external stimuli will be either propagated or inhibited based on the relaying neurons.

A neuron is said to be *tuned to* a variable if the state of the variable alters its tendency to spike. The probability of observing a neuron in an active state can then be linked to the state of the variable by the use of probabilistic models (e.g., Truccolo et al. (2005), Paninski (2004)). A function describing the expected activity of a neuron as a function of the variable is known as a *tuning curve*.

Figure 1.2 shows a tuning curve from the dataset by Peyrache et al. (2015) that will be introduced in Chapter 2. The neuron activity is measured as the number of spikes for a given time bin, and in the figure, the activity is shown as a function of the animal's head direction. The neuron seems to be a clearly tuned to the head direction for a given angular interval.

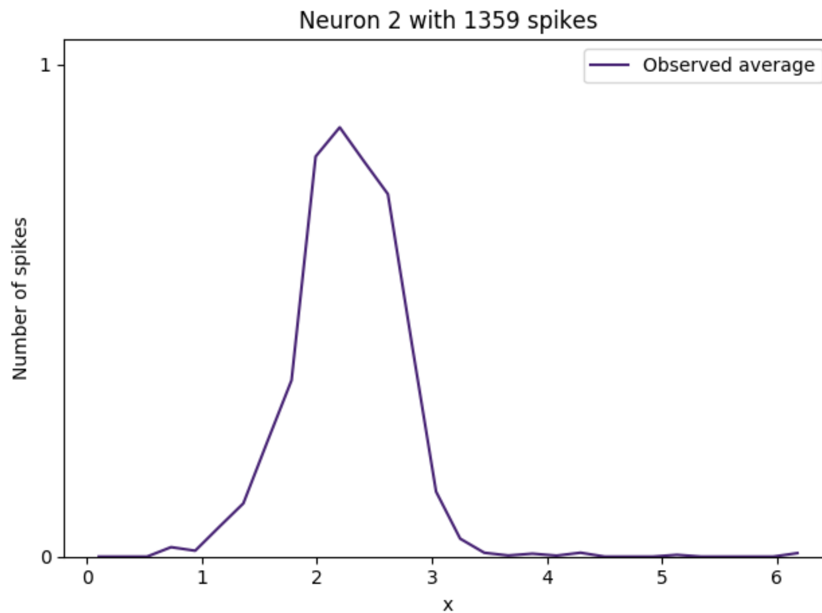


**Figure 1.2:** Observed average number of spikes per time bin with size 25.6 ms for one neuron plotted against the animal's head direction, given as an angle in radians. The neuron is active mainly for head directions between 1 and 3, thus tuned to this interval.

## 1.3 Probabilistic methods for dimensionality reduction

One well-known dimensionality reduction technique is principal component analysis (PCA), which finds a linear mapping from the observed data onto a lower-dimensional space chosen to maximize the variance in the projected data. Examples of more flexible methods are ISOMAP (Tenenbaum et al., 2000) and locally linear embedding (LLE) (Roweis and Saul, 2000). PCA is well-known due to its simplicity but is limited to linear mappings between the latent manifold and the data. Neil D. Lawrence proposed a nonlinear generalization of PCA (Lawrence, 2004), called the *Gaussian process latent variable model* (GPLVM), where the mappings from the latent space to the observed-data space were allowed to be nonlinear by modeling them as Gaussian processes. Gaussian processes (see Rasmussen and Williams (2006) for an introduction) are powerful tools for both regression and classification problems and are highly nonlinear since they work by placing a probability distribution over every continuous function.

If the latent variable is assumed to move somewhat smoothly in time, it too can be modeled as a Gaussian process (see, e.g., Byron et al. (2009)). Furthermore, the spike count in a given time interval can be modeled as a Poisson process (e.g., Macke et al. (2015)). Anqi Wu (Wu et al., 2017) combined these three elements (GP tuning curves, GP latent variable, Poisson observation model) into the *Poisson Gaussian process latent variable model* (P-GPLVM), later renamed to the *latent manifold tuning model* (LMT). An example of its use is Wu et al. (2018), where a latent "odor space" was introduced, and positions in this space, as well as mappings from neural responses, were inferred using the LMT model.

The GPLVM can be generalized even further by extending the shape of the latent manifold to include non-Euclidean manifolds like spheres, tori, or rotation groups of various dimensions. This was recently done by Jensen et al. (2020), who also used cross-validation to perform model selection between different types of manifolds.

## 1.4 Our contribution

In the latent manifold tuning model, estimates of the latent variable and tuning curves must be found using approximate methods. Wu et al. (2017) introduced the *decoupled Laplace approximation*, a computationally efficient method for approximate inference. We use an iterative maximum a priori (MAP) procedure instead, where MAP estimates of the latent variable and tuning curves are updated iteratively.

We also show explicitly how the inference framework is made computationally efficient using approximate Gaussian processes, and provide a free-standing implementation of the LMT model in Python. The convergence properties of the algorithm are discussed through exploring different ways the algorithm can be initialized and its consequent results. We evaluate the robustness of the algorithm with regards to different tuning strengths and data lengths and find that there is an optimal tuning strength that depends on the data length.

Finally, we apply the model to head direction neurons in a dataset by Peyrache et al. (2015) and infer the observed head direction with higher precision than PCA.

Chapter 2

# Neural activity and mouse head direction data

In this chapter, we introduce a dataset by Peyrache et al. (2015), that will be analyzed in Chapter 5. We then discuss the time bin width, a hyperparameter of the model, and look at how consistently these neurons are tuned. Finally, we apply principal component analysis to the dataset to explore the lower-dimensional projections of the data.

## 2.1 Observed head direction and neural activity

Peyrache et al. (2015) studied the brain's mechanisms for head direction monitoring by making simultaneous recordings from the antero-dorsal thalamic nucleus and the post-subiculum parts of the brains of seven mice. We will limit our analysis to one mouse trial called *Mouse28-140313*. The recordings were done using extracellular multi-electrode arrays, and a camera tracked the head direction (measured as the azimuthal angle of the animal's head in a reference frame anchored to the recording room) of the mouse during the experiment. Figure 2.1 shows the head direction during an approximately four-minute interval, observed at 25.6 milliseconds (ms) intervals for a total of 10.000 observations. Observe how the head direction "wraps around" from 0 to $2\pi$ whenever it reaches the border of the domain. We can model the head direction as a 1-dimensional, $2\pi$-periodic variable, and this latent variable is what we will be looking for in the recorded neural data.

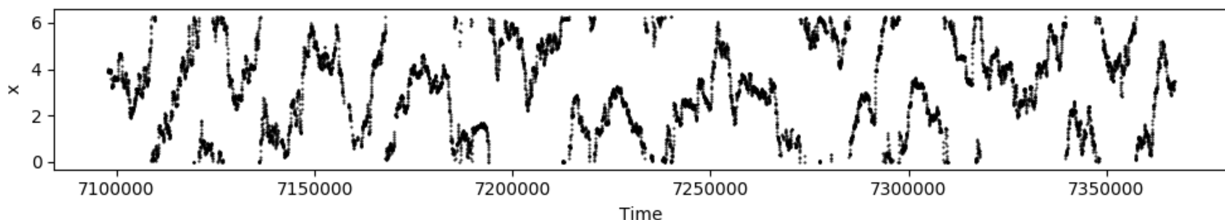


**Figure 2.1:** Observed head direction in radians for the 10.000 bins starting at time 6881305.6 in the dataset Mouse28-140313. Some missing observations have been removed.

Through the multi-electrode array, the time of each spike was recorded for each neuron. By dividing time into bins of a certain *bin width*, we can count the number of spikes in each time bin for every neuron. These *spike counts* can then be compared to the head direction value in that time bin. Alternatively, one can look at the *spike presence* instead of spike count, which is the presence of at least one spike in a particular bin. Figure 2.2 shows a representation of the spike presence.

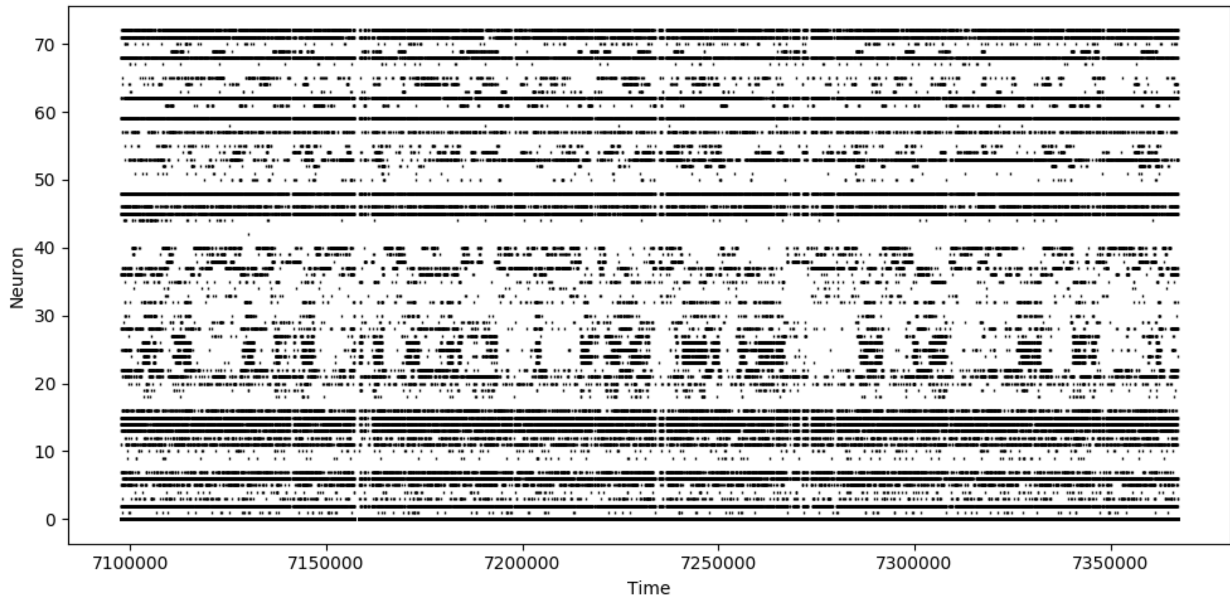


**Figure 2.2:** Spike presence for 73 neurons across 10.000 time bins. The neurons are placed on the $y$-axis, and a black dot means that at least one spike was observed in that time bin for that neuron.

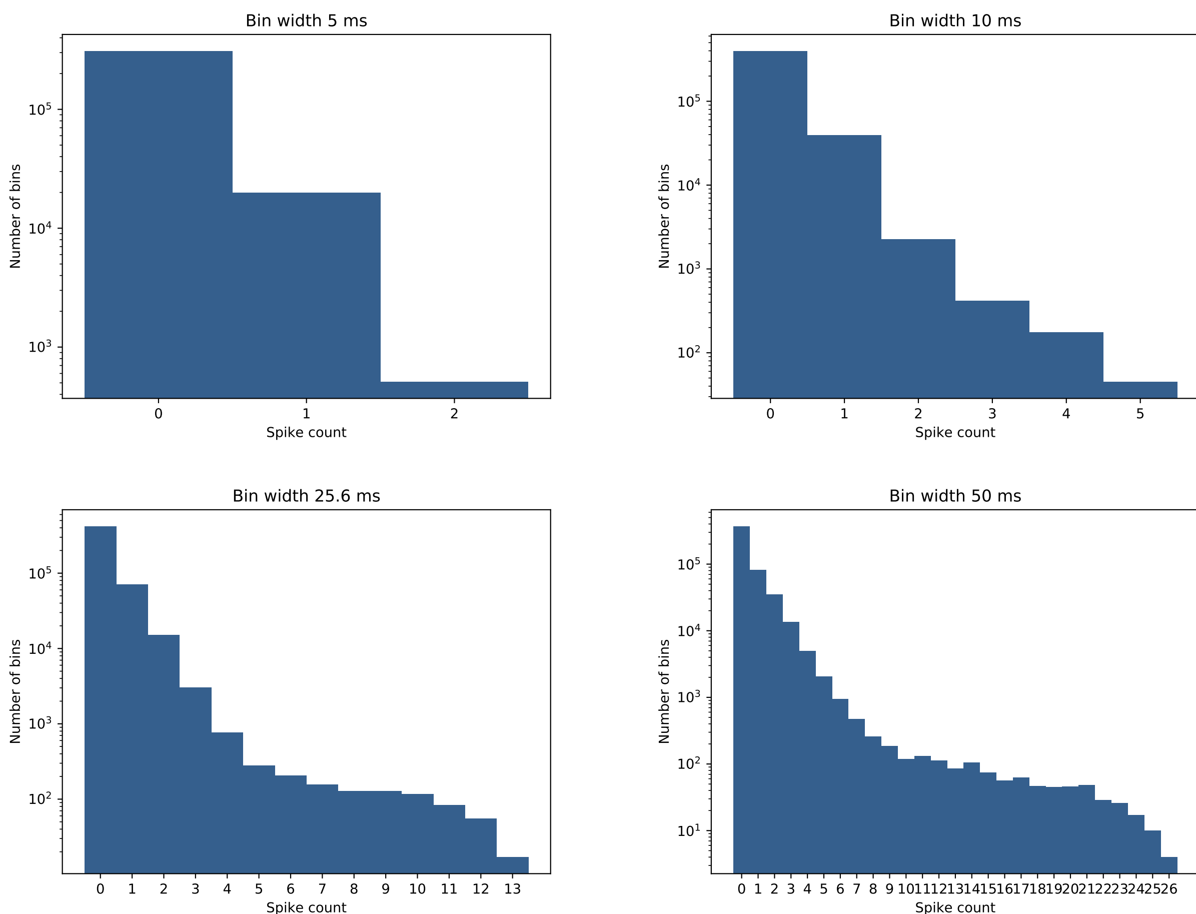


**Figure 2.3:** Log scale distribution of spike counts across all neurons, for 10.000 bins starting at time 6891305.6. Upper left: Bin width 5 ms. Upper right: Bin width 10 ms. Lower left: Bin width 25.6 ms. Lower right: Bin width 50 ms.

In Figure 2.2, the bin width was chosen equal to the intervals between head direction observations, 25.6 ms. The chosen time interval is the same as in Figure 2.1. By interpolating between the observed head direction values, or averaging over them, the width of the time bins can be chosen as smaller or wider than the camera capture rate. The choice of bin width changes the distribution of the spike counts. Figure 2.3 shows histograms in logarithmic scale of the distribution of spike counts for all neurons and four different bin widths and 10.000 bins starting at time 6891305.6.

If the bin width is set too low, the total number of bins needed to cover the observed data increases, and most methods increase in computing time as the number of time bins increases. On the other hand, if the bin size is set wider than the interval between head direction observations, some information may be lost when we average over the observed head direction values. In addition, if the data is handled based on *spike presence* instead of *spike count*, some information is lost whenever there is more than one spike in a bin. Ultimately, the preferred bin width depends on the observed neurons as well as on the model. For a model dealing with spike presence, a smaller bin width may be preferred, while for models that deal with spike counts a tradeoff must be made between the resolution of the observed head direction and the computational complexity of the model.

### 2.1.1 Neuron tuning

Some neurons are more clearly tuned to the head direction than others. By partitioning the domain of $[0, 2\pi]$ into 50 uniformly sized intervals, the average number of spikes per bin can be represented as a function of the head direction. Figure 2.4 shows the observed average number of spikes per bin with size 25.6 ms for two neurons, for the same time period as in figures 2.1 and 2.2.

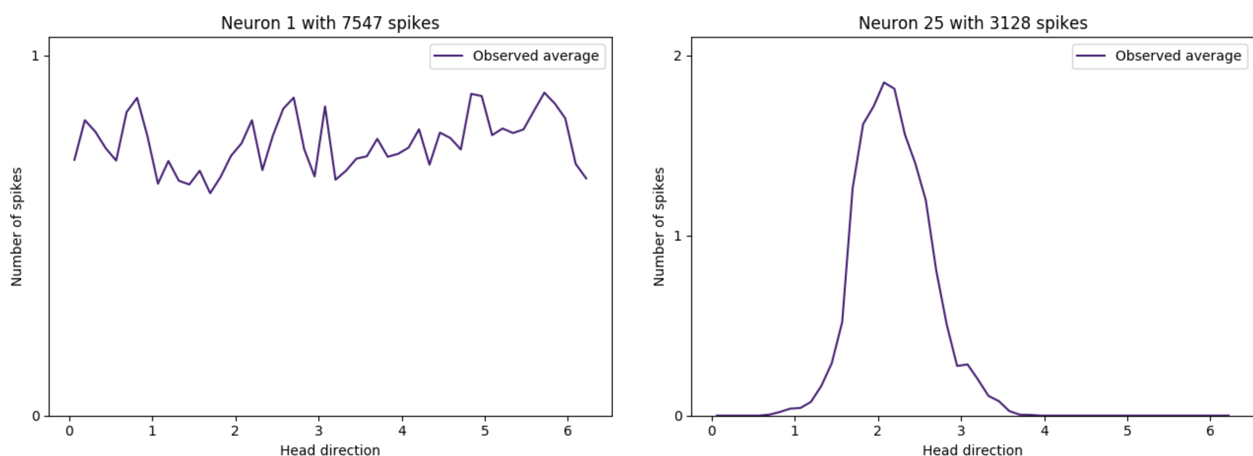


**Figure 2.4:** Observed spike counts per bin, with size 25.6 ms, for two observed neurons. Left: Neuron 1 that does not seem to be tuned to the head direction. Right: Neuron 25 that seems clearly tuned to head direction values between approximately 1.5 and 3.

If a neuron is truly tuned to the head direction, that tuning should be consistent through time. To investigate how consistent the tuning was for these neurons, we looked at two separate time intervals of 10.000 bins, or a little more than four minutes. Figure 2.5 shows the average number of spikes in a bin of size 25.6 ms, as a function of the head direction, for four selected neurons. The tuning of these neurons seems quite robust between the different time periods.

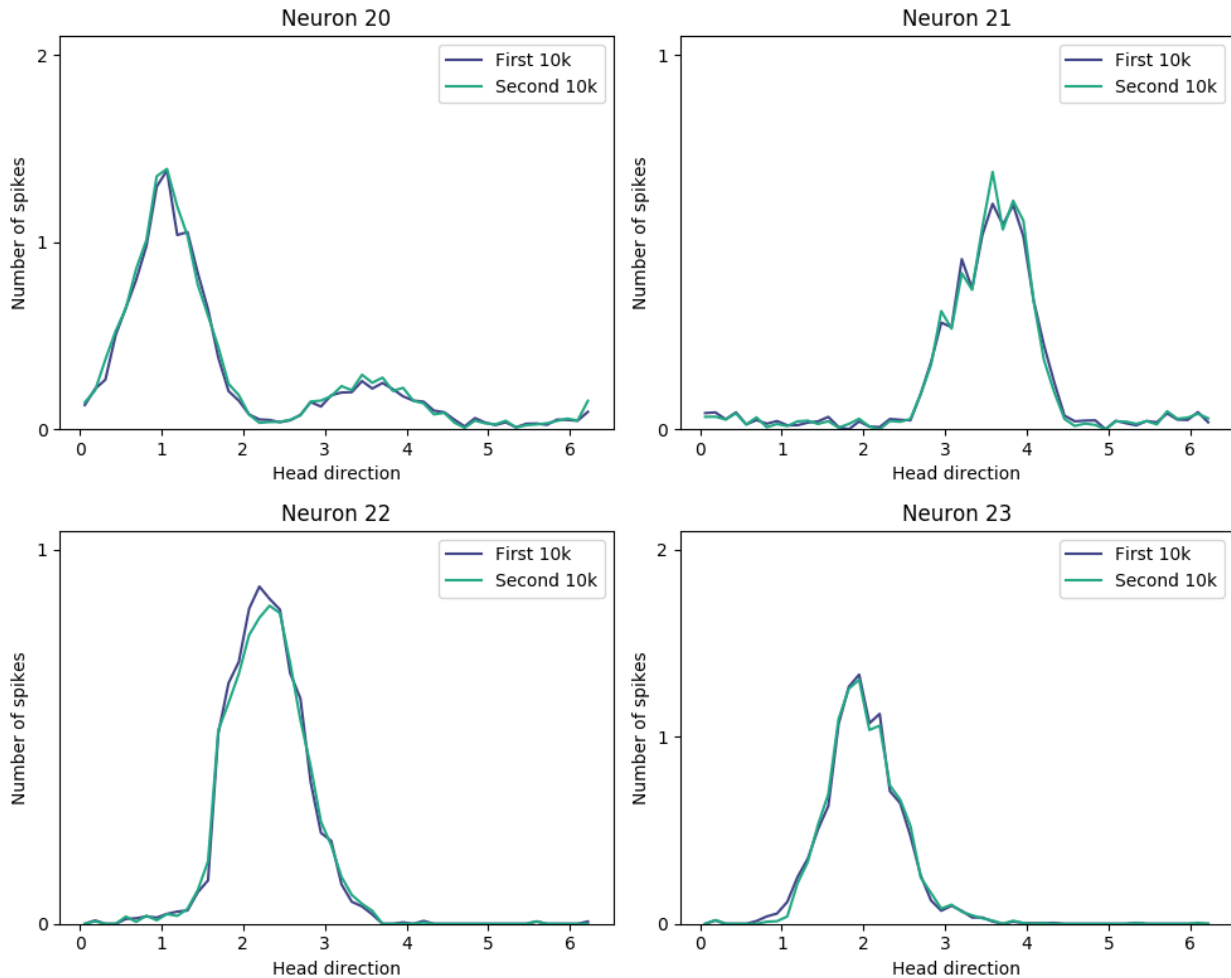


**Figure 2.5:** Comparison of observed spike counts between two non-overlapping time intervals of 10.000 time bins with bin width 25.6 ms.

## 2.2 Applying principal component analysis

Before we introduce the latent manifold tuning model, let us apply principal component analysis (PCA), a well-known dimensionality reduction technique described in section 3.4, to the data to see what we can find. After removing any neurons that produced less than 1000 spikes during the entire trial, 51 neurons remained. The goal of dimensionality reduction is to find a lower-dimensional variable that will explain the behavior of this 51-dimensional variable. We may, for example, choose to look for a two-dimensional latent variable. The observed spike counts were smoothed in time by a Gaussian smoothing kernel with a standard deviation of ten bins, or equivalently 256 ms, and then PCA was applied to the smoothed data. Figure 2.6 shows the value of the two first principal components for all the 85.504 time bins in the observed data.

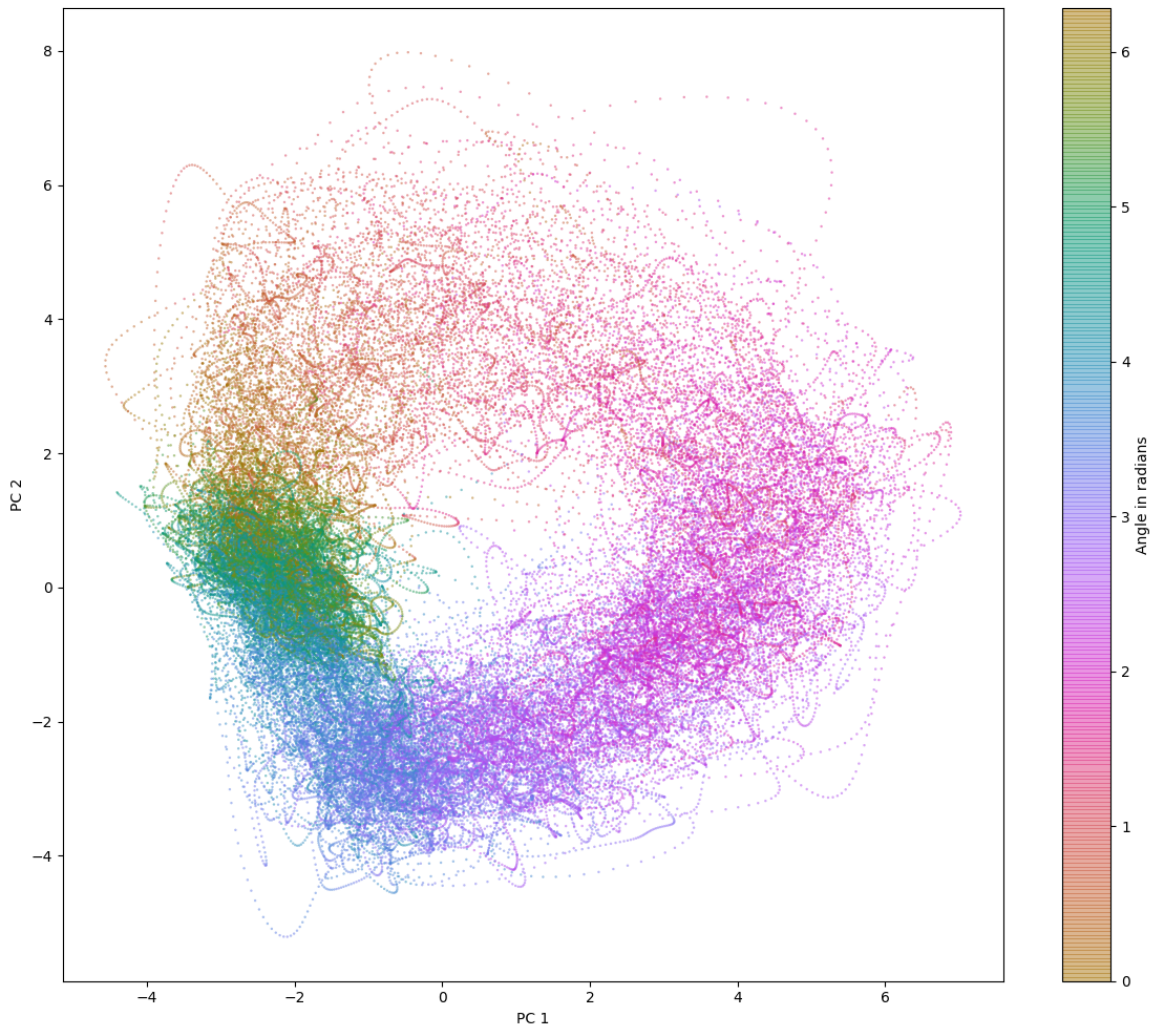


**Figure 2.6:** A projection of the observed neural activity into the space spanned by the two first principal components found by applying PCA to the spike data. The color of each point represents the observed head direction value for that time bin, colored according to the bar to the right.

The visualization suggests that the latent variable lives on a circular manifold, which indeed is the case for head direction. We also see that the angle of the position in the PCA domain corresponds well to the coloring from the observed head direction. This indicates that the PCA has managed to extract the head direction from the spike data to some degree. We note that while in this case, the shape of the latent manifold can be deduced by the dimensionality reduction technique, the correct rotation can not be inferred. We shall see in Chapter 5 that the correct rotation, or offset for nonperiodic variables, has to be found by comparing the estimate with the true latent variable.

In Figure 2.6, every time bin is represented by a point in the space spanned by the two first principal components. The angle between the positive axis of the first principal component and the line from the origin to this point can be calculated using standard geometrical properties. Since the cloud of points appears to live on a circle in PCA space, we can compare these angles with the observed head direction angles to check how well they match up. Figure 2.7 shows the angle of the points in Figure 2.6 compared to the observed head direction. The latent variable is referred to as $\mathbf{X}$, which will be a recurring notation in the following chapters. Even though PCA implies a circular domain, the angle in the PCA space does not correspond very well to the observed head direction. The fit is particularly bad whenever the head direction wraps around from 0 to $2\pi$. An example of this is the bump in the principal component around time bin 750.

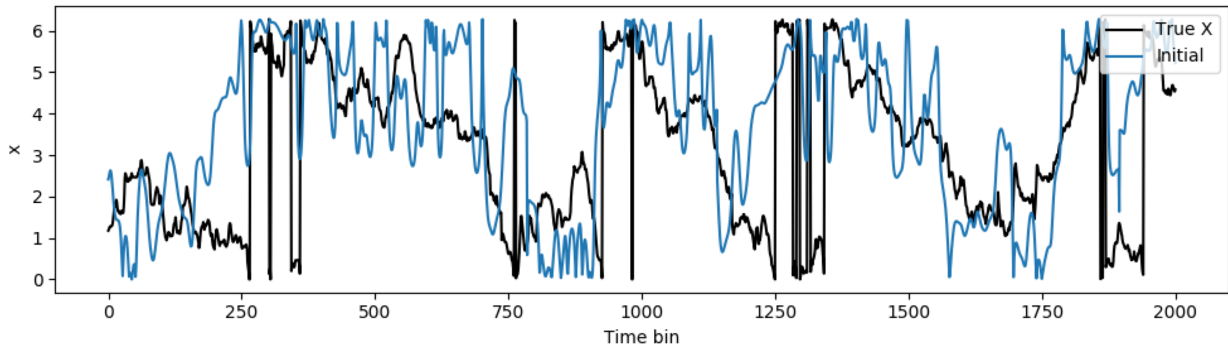


**Figure 2.7:** The angle in Figure 2.6 compared to the observed head direction.

Instead of a two-dimensional variable, we may ask how well a one-dimensional variable can explain the data. Figure 2.8 shows the first principal component compared to the observed head direction. Using the first principal component actually provides an estimate that is slightly better than using the angle from the two-dimensional PCA.

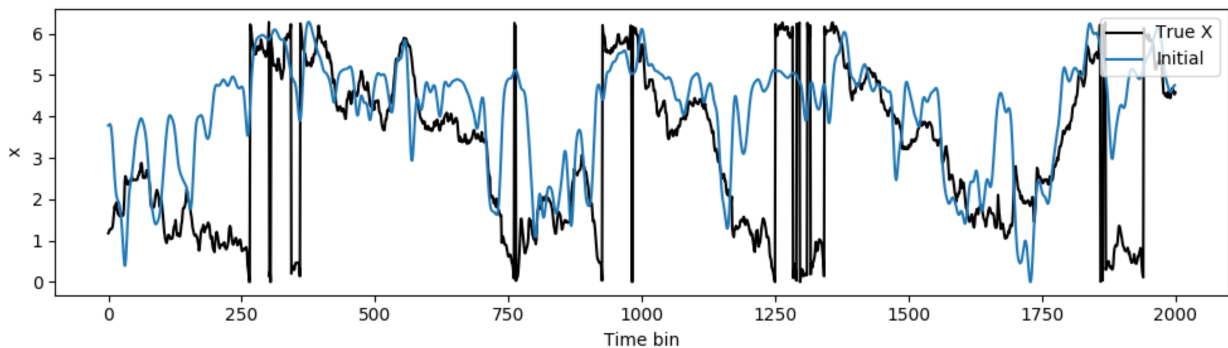


**Figure 2.8:** The first principal component compared to the actual observed head direction for 2000 time bins.

It is helpful to remove inactive neurons and neurons that are not tuned to head direction. This was done before the analysis described in Chapter 5. Of course, this type of screening can only be done if the latent variable of interest is known. Methods for identifying such neural ensembles is an active line of research (e.g., Rybakken et al. (2019), Carrillo-Reid et al. (2016)).

It appears that inferring a latent variable is not trivial even when the dimensionality and the domain of the latent variable are known. Though some of the latent dynamics can be recovered by using PCA, this model is limited due to its linear mappings. We will introduce the theory that is necessary to define the doubly nonlinear latent manifold tuning model in the next chapter.

# Chapter 3

# Theoretical background

We begin this chapter by outlining some concepts in parameter estimation in section 3.1. We then introduce Gaussian processes in section 3.2, which we will use to model both the latent variable and the tuning curves of each neuron in Chapter 4. We show how Gaussian processes can be made computationally efficient by sparse approximations in section 3.3. In section 3.4, we describe principal component analysis (PCA), which will be used to find an initial estimate of the latent variable. Finally, in section 3.5, we include some theory about generalized linear models (GLMs), which will be used to model spike counts.

## 3.1 Parameter estimation

Here we will explain how parameters in probability distributions can be inferred using the likelihood function and the posterior distribution. Let us begin by defining a random sample from a population. A probability distribution of a random variable $X$ either has a probability density function (pdf) if $X$ is continuous or a probability mass function (pmf) if $X$ is discrete. Casella and Berger (2002) use the following definition of a random sample:

"The random variables $X_1, \dots, X_n$ are called *a random sample of size* n *from the population* $f(x)$ if $X_1, \dots, X_n$ are mutually independent random variables and the marginal pdf or pmf of each $X_i$ is the same funciton $f(x)$." - Casella and Berger (2002)

A probability distribution typically depends on one or more parameters $\theta = \{\theta_1, \theta_2, \dots, \theta_k\}$, and we can write the pdf or pmf as $f(x|\theta)$. For example, the normal distribution has two parameters: the mean $\mu$ and variance $\sigma^2$, while the Poisson distribution has one parameter, the rate $\lambda$ which determines both the mean and the variance. In a typical experiment, we have some observations from a process that we model using some probability distribution. Knowledge of the underlying parameters gives the experimenter knowledge about the entire population from which the sample is drawn, but in a typical experiment, the parameters are not known and must be estimated. Casella and Berger (2002) define a *point estimator* as "any function $W(X_1, \dots, X_n))$ of a sample; that is, any statistic is a point estimator." - Casella and Berger (2002)

Obviously, this is a very wide definition, and some estimators are typically better than others. Here, we will describe two commonly used estimators, the *maximum likelihood estimator* (MLE) and *maximum a posteriori estimate* (MAP).

### 3.1.1 Maximum likelihood estimate

For observations $\mathbf{x} = \{x_1, \ldots, x_n\}$ of a random sample $\mathbf{X} = \{X_1, \ldots, X_n\}$ from the population $f(x|\theta)$, the likelihood function is defined as

$$L(\theta|\mathbf{x}) = \prod_{i=1}^{n} f(x_i|\theta) \tag{3.1}$$

and the maximum likelihood estimate (MLE) is defined as

$$\begin{aligned} \hat{\theta}_{\text{MLE}} &= \text{argmax}_\theta L(\theta|\mathbf{x}) \\ &\Leftrightarrow \text{argmax}_\theta \log L(\theta|\mathbf{x}) \end{aligned} \tag{3.2}$$

Usually, the loglikelihood function $\log L$ is optimized instead of $L$ due to numerical stability issues. In a standard linear regression setting, the MLE can be found analytically, but in most cases, an analytical expression is not available. In these cases, gradient-based iterative optimization techniques can be used to find the MLE exactly provided that the loglikelihood function is concave. If the loglikelihood function is not concave, an estimate can still be found using iterative methods, but it is not necessarily the exact MLE estimate.

### 3.1.2 Maximum a posteriori estimate

Another estimator is the Bayesian maximum a posteriori estimator (Casella and Berger (2002), p. 324). In Bayesian statistics, a prior distribution $\pi(\theta)$ is placed over the domain of the parameters. The prior is chosen either to reflect the experimenter's previous knowledge about the parameters or to assume as little as possible about the parameters, a so-called *un-informed prior*. By applying Bayes' rule, a posterior distribution $f(\theta|\mathbf{x})$ of $\theta$ can be obtained.

$$f(\theta|\mathbf{x}) = \frac{f(\mathbf{x}|\theta) \times \pi(\theta)}{f(\mathbf{x})} \tag{3.3}$$

For some combinations of likelihood function $f(\mathbf{x}|\theta)$ and prior $\pi(\theta)$ the resulting posterior belongs to a known probability distribution. In this case, the prior and posterior are called *conjugate* distributions. Using a conjugate prior is very favorable computationally since it provides an analytic expression of the posterior distribution. In the absence of a conjugate prior, finding or approximating the posterior distribution is generally a computationally demanding task. The task of finding posterior distributions can be approached, for example, by sampling-based techniques like Markov Chain Monte Carlo methods (e.g., Geyer (1992) or Chib and Greenberg (1995)), or by approximate methods like Integrated nested Laplace approximation (INLA, Rue et al. (2009)). The posterior distribution can be used to find point estimates of $\theta$ like the MAP estimate, which is defined as:

$$\begin{aligned} \hat{\theta}_{\text{MAP}} &= \text{argmax}_\theta f(\mathbf{x}|\theta) \times \pi(\theta) \\ &= \text{argmax}_\theta \log \Big[ f(\mathbf{x}|\theta) + \pi(\theta) \Big] \end{aligned} \tag{3.4}$$

Note that since $f(\mathbf{x})$ is constant in $\theta$, it is not required to find the MAP estimate. As for the MLE estimate, iterative methods may be used to find the MAP estimate, where the log posterior is maximized instead of the posterior due to numerical stability issues. How good the estimate obtained from the iterative method is, depends on the shape of the likelihood function or posterior function, respectively. If the gradient and Hessian of the posterior can be calculated, these can be used to check for local maxima, and if the log posterior is concave, a global maximum can be verified.

**Credible intervals**

For a chosen $\alpha \in (0,1)$, a $(1-\alpha)$ *credible interval* of a posterior distribution is defined as any $I_\theta = [t_l, t_u]$ such that

$$\int_{t_l}^{t_u} f(\theta|\mathbf{x}) d\theta = 1 - \alpha \tag{3.5}$$

A credible interval is called *equi-tailed* if

$$\int_{\theta_{\text{min}}}^{t_l} f(\theta|\mathbf{x}) d\theta = \int_{t_u}^{\theta_{\text{max}}} f(\theta|\mathbf{x}) d\theta \tag{3.6}$$

where $\theta_{\text{min}}$ and $\theta_{\text{max}}$ indicate the lower and upper boundary of the domain of $\theta$, respectively. A credible interval is said to be a *highest posterior density interval* if

$$f(\theta_1|\mathbf{x}) \geq f(\theta_2|\mathbf{x}) \;\; \forall \;\; \{\theta_1 \in I_\theta, \theta_2 \notin I_\theta\} \tag{3.7}$$

For symmetric posterior distributions, the highest posterior density interval and the equi-tailed credible intervals are equivalent.

### 3.1.3 Heuristics for global optimization

Unfortunately, many posteriors are not concave, providing no guarantee that the obtained MAP estimate will be optimal. As for the MLE, the problem is that gradient-based optimization algorithms can become trapped in suboptimal local maxima from which they cannot escape. To prevent this, several heuristic techniques exist that deal with global optimization. Particle swarm optimization (Kennedy and Eberhart, 1995) is one of several techniques that select several initial positions, then chooses the best estimate among the final estimates after the iteration has converged.

Another technique is *simulated annealing* (Van Laarhoven and Aarts, 1987), a heuristic that takes its name from the process of controlled cooling of a piece of metal. Instead of simply climbing the gradient to find better estimates at every iteration, it introduces a nonzero probability for moving to an estimate in the next iteration that is worse than the current iteration. The probability of making such a move starts high and is then lowered gradually. This creates an initial phase of exploration before the algorithm hopefully settles and converges to the best local maximum. An adjacent idea is that of *graduated optimization* (Hazan et al. (2016), Wu (1996)), in which the objective function is approximated or smoothed to keep the estimate from getting trapped in local maxima. The amount of smoothing, analogous to a temperature, starts at a high value and decreases at every iteration until there is no smoothing, and one is left with the plain objective function.

## 3.2 Gaussian processes

For a comprehensive introduction to Gaussian processes, see Rasmussen and Williams (2006). For material on sparse Gaussian processes, see Bauer et al. (2016) and Quiñonero-Candela and Rasmussen (2005). Formally, a Gaussian process is a set of random variables where any finite collection of those random variables has a joint multivariate normal distribution. If we let $f(x)$ be a real-valued function and let the domain of x be some continuous domain (e.g., space or time), then if the variable $\mathbf{f} = (f(x_1), \ldots, f(x_n))$ follows a multivariate normal distribution *for any collection of* n *locations* $\mathbf{x} = (x_1, \ldots, x_n)$, then we have a Gaussian process. A Gaussian process can be completely specified

by its mean function $m(x)$ and *covariance kernel function* $k(x, x')$, since they determine the mean vector and covariance matrix of the joint normal distribution.

$$\begin{aligned} m(x) &= \mathbb{E}[f(x)] \\ k(x, x') &= \mathbb{E}[(f(x) - m(x))(f(x') - m(x'))] \end{aligned} \tag{3.8}$$

In Bayesian terms, this gives us the following prior over the function values $\mathbf{f}$:

$$\mathbf{f} \sim \mathcal{N}(\mathbf{m}(\mathbf{x}), K_x(\mathbf{x})) \tag{3.9}$$

where $\mathbf{m}(\mathbf{x})$ is the mean vector $\mathbf{m}(\mathbf{x}) = (m(x_1), \ldots, m(x_n))$ and $K_x(\mathbf{x})$ is the covariance matrix containing the covariance function evaluated for all pairs of locations in $\mathbf{x}$. We will use the following equivalent notation:

$$\mathbf{f} \sim \mathcal{GP}(\mathbf{m}(\mathbf{x}), k_x) \tag{3.10}$$

where $k_x$ represents the covariance kernel function. The choice of covariance function and its hyperparameters determines the assumed smoothness of the function and should reflect Tobler's first law of geography: "everything is related to everything else, but near things are more related than distant things" (Tobler, 1970). Some common covariance functions are the Gaussian, Matérn and exponential covariance functions:

$$\begin{aligned} k_{\text{Gauss}}(x, x') &= \sigma \;\; \exp(-||x - x'||_2^2/2\delta^2) \\ k_{\text{Matérn}}(x, x') &= \sigma \;\; (1 + ||x - x'||_2/\delta) \;\; \exp(-||x - x'||_2/\delta) \\ k_{\text{Exponential}}(x, x') &= \sigma \;\; \exp(-||x - x'||_2/\delta) \end{aligned} \tag{3.11}$$

All these three covariance functions have two hyperparameters: The marginal variance $\sigma$ and the length scale $\delta$. Other covariance functions could have a different number of hyperparameters. Observe that the covariance between two function values $f(x)$ and $f(x')$ depends only on the positions $x$ and $x'$, and not on the values $f(x)$ and $f(x')$. We can choose a set of points in the domain of x (for example, an evenly spaced grid) and sample values $f(x)$ from their joint prior multivariate normal distribution. Figure 3.1 shows samples from a Gaussian process with zero mean function and Gaussian covariance function, with different choices for the hyperparameters.

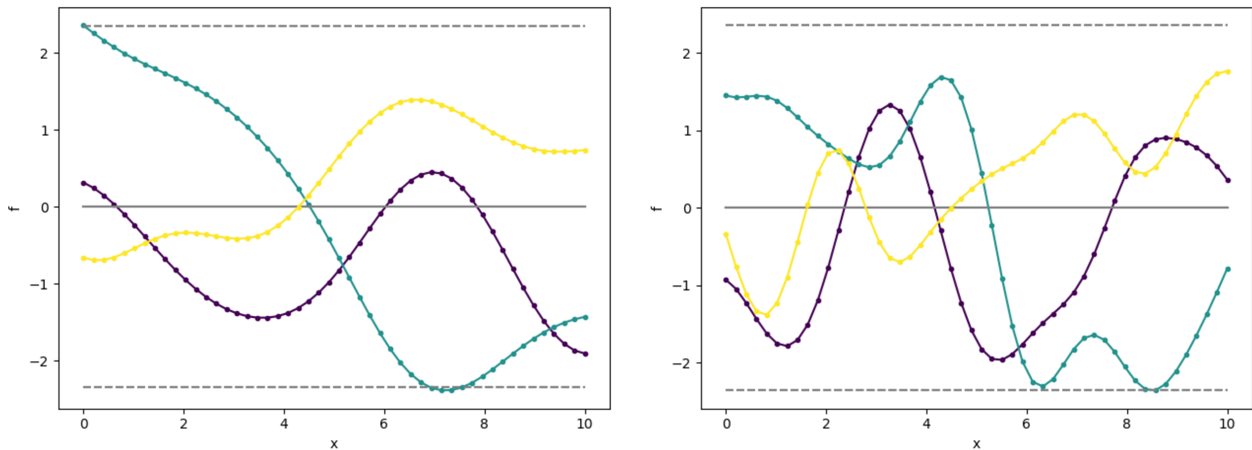


**Figure 3.1:** Four realizations from a Gaussian process prior with Gaussian covariance function, evaluated at a grid of 50 x values with equal distance. Function values at the grid are shown with dots. Left: $\sigma = 1.2, \delta = 2$. Right: $\sigma = 1.2, \delta = 1$. The zero mean is shown as a grey line and the boundaries of the 95 % confidence interval of the prior is shown in dotted grey.

The interpretation of the hyperparameters is as follows: By increasing $\delta$, we increase the smoothness of the function along the x axis. By increasing $\sigma$, we increase the amplitude of their deviation from the mean function. The Gaussian covariance function is smoother than the exponential covariance function for the same choice of parameters. Figure 3.2 shows a comparative plot with samples from a Gaussian process with zero mean function and exponential covariance function, with the same hyperparameters as in Figure 3.1.

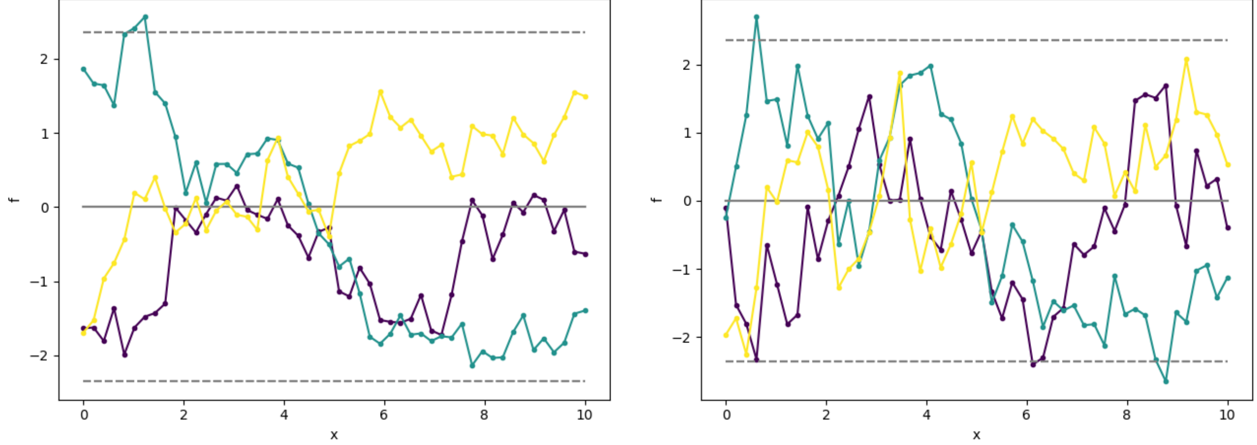


**Figure 3.2:** Four realizations from a Gaussian process prior with exponential covariance function, evaluated at a grid of 50 x values with equal distance. Function values at the grid are shown with dots. Left: $\sigma = 1.2, \delta = 4$. Right: $\sigma = 1.2, \delta = 1$. The zero mean is shown as a grey line and the boundaries of the 95 % confidence interval of the prior is shown in dotted grey.

### 3.2.1 Conditional distribution

Based on our observations, we may want to predict the function values at another set of points. For instance, we may want to predict the tuning curve values on a uniformly spaced grid in the domain of x to visualize the tuning curve. Since the value of $\mathbf{f}$ at any collection of points is jointly normally distributed, the joint distribution $p(\mathbf{f}, \mathbf{f}_{\text{grid}})$ of some observations $\mathbf{f}$ and $p$ values $\mathbf{f}_{\text{grid}}$ on a grid $\mathbf{x}_{\text{grid}} = (x_{\text{grid}_1}, \ldots, x_{\text{grid}_p})$ is

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_{\text{grid}} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K_x & K_{\text{grid},x} \\ K_{x,\text{grid}} & K_{\text{grid}} \end{bmatrix}\right) \tag{3.12}$$

where the entries in the covariance matrices are found by evaluating the covariance kernel at the respective combinations of points:

$$\begin{aligned} K_{x[i,j]} &= k(x_i, x_j) \\ K_{\text{grid},x[i,j]} &= k(x_{\text{grid}_i}, x_j) \\ K_{x,\text{grid}[i,j]} &= k(x_i, x_{\text{grid}_j}) \\ K_{\text{grid}[i,j]} &= k(x_{\text{grid}_i}, x_{\text{grid}_j}) \end{aligned} \tag{3.13}$$

Using well known facts about the multivariate normal distribution, we know that the conditional probability distribution $p(\mathbf{f}_{\text{grid}}|\mathbf{f}) = \int p(\mathbf{f}_{\text{grid}}, \mathbf{f}) d\mathbf{f}$ is also multivariate normal:

$$\begin{aligned} & \mathbf{f}_{\text{grid}}|\mathbf{f} \sim \mathcal{N}(\mu_{\text{grid}}, \Sigma_{\text{grid}}) \\ \text{where} \quad & \mu_{\text{grid}} = K_{x,\text{grid}} K_x^{-1}(\mathbf{f} - \mu(\mathbf{x}_{\text{grid}})) \\ & \Sigma_{\text{grid}} = K_{\text{grid}} - K_{x,\text{grid}} K_x^{-1} K_{\text{grid},x} \end{aligned} \tag{3.14}$$

The conditional mean $\mu_{\text{grid}}$ is the best linear unbiased predictor of the Gaussian process, and is also known as the Kriging estimate.

### 3.2.2 Noisy observations

Often, a noisy version of the actual function values $\mathbf{f}$ is observed. If we assume additive and independent, identically distributed Gaussian noise, $\mathbf{f} = \mathbf{f}_{\text{noiseless}}(\mathbf{x}) + \varepsilon, \varepsilon \sim \mathcal{N}(0, \sigma_\varepsilon^2)$, where $\sigma_\varepsilon^2$ is the noise variance, then the likelihood model is $p(\mathbf{f}|\mathbf{f}_{\text{noiseless}}) \sim \mathcal{N}(\mathbf{f}_{\text{noiseless}}, \sigma_\varepsilon^2 I)$, where $I$ represents the identity matrix of size $N$ and the joint distribution of observations $\mathbf{f}$ and $\mathbf{f}_{\text{grid}}$ is

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_{\text{grid}} \end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} K_x + \sigma_\varepsilon^2 I & K_{\text{grid},x} \\ K_{x,\text{grid}} & K_{\text{grid}} \end{bmatrix}\right) \tag{3.15}$$

The conditional distribution with observation noise is

$$\begin{aligned} & \mathbf{f}_{\text{grid}}|\mathbf{f} \sim \mathcal{N}(\mu_{\text{grid}}, \Sigma_{\text{grid}}) \\ \text{where} \quad & \mu_{\text{grid}} = K_{x,\text{grid}}[K_x + \sigma_\varepsilon^2 I]^{-1}(\mathbf{f} - \mu(\mathbf{x}_{\text{grid}})) \\ & \Sigma_{\text{grid}} = K_{\text{grid}} - K_{x,\text{grid}}[K_x + \sigma_\varepsilon^2 I]^{-1} K_{\text{grid},x} \end{aligned} \tag{3.16}$$

Figure 3.3 shows the posterior mean and 95 % credible interval with and without the assumption of observation noise for the same set of observations, with noise parameter $\sigma_\varepsilon^2$. Since a normal posterior distribution is symmetric, the highest posterior density interval is equal to the equi-tailed credible interval, and this is the credible interval we will use throughout.

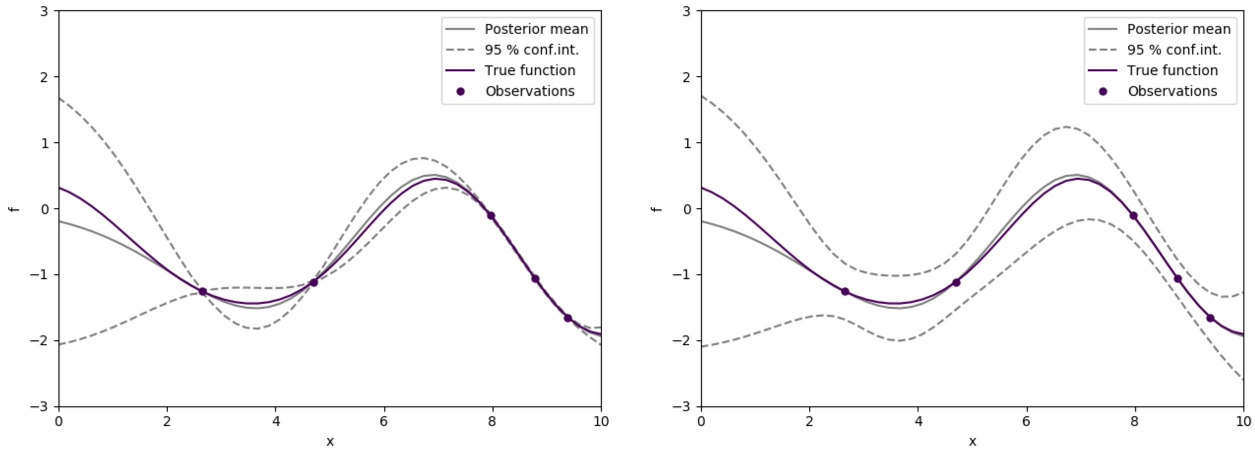


**Figure 3.3:** Posterior mean (grey) and 95 % credible interval (dotted grey) for a fitted Gaussian process with parameters $\sigma = 1.2, \delta = 2$, based on 5 observations. Left: With the assumption of zero noise. Right: With the assumption of independent additive Gaussian noise with $\sigma_\varepsilon^2 = 0.05$. The underlying tuning curve was generated from a Gaussian process with squared exponential covariance function with parameters $\sigma = 1.2, \delta = 2$.

An interpretation of the noise parameter $\sigma_\varepsilon^2$ is that when the noise parameter is increased, we increase the willingness of the estimate to deviate from the observed points in order to make the graph smoother. A higher noise parameter also causes the credible intervals to become wider. In the inference described in Chapter 4, we will borrow ideas from graduated optimization and simulated annealing in an attempt to tackle the nonconvexity of the posterior distribution. We will start with a high noise parameter, then lower its value at every iteration to hopefully decrease the chance of the estimate ending up at some suboptimal local maximum.

In the above, we calculated the posterior using the exact hyperparameters $\delta, \sigma$, and $\sigma_\varepsilon^2$ that were used to generate the observations. In real applications, we cannot know a priori which hyperparameters provide the best fit, and these have to be estimated. Note also that if there are two observations with the same x-value, this will make two columns in $K_x$ linearly dependent, making it non-invertible. However, when noise is assumed in the model, the addition of $\sigma_\varepsilon^2$ on the diagonal makes the two columns different. Therefore, in practice, the linear dependency goes unnoticed.

## 3.3 Approximate Gaussian processes

A drawback of the Gaussian process is that it scales poorly to big datasets. With $N$ observations, the size of the covariance matrix is $N$ by $N$. To calculate the posterior distribution, this matrix must be inverted, an operation of computational complexity $N^3$. This limits the exact treatment of Gaussian processes to a couple of thousands of observations on modern laptops, especially if the covariance matrix must be inverted more than once. Fortunately, several models for sparse, approximate Gaussian processes exist that are much more computationally efficient. A good overview is provided by Quiñonero-Candela and Rasmussen (2005).

A common way to deal with the computational complexity is to introduce a set of $N_{\text{ind}}$ latent variables $\mathbf{u} = (u_1, \ldots, u_{N_{\text{ind}}})$, which are function values in the same sense as $\mathbf{f}$. These are evaluated at positions $(\mathbf{x}_{u_1}, \ldots, \mathbf{x}_{u_{N_{\text{ind}}}})$, referred to as *inducing points*. These inducing points can be chosen as a subset of the observed points $\mathbf{x}$, or as separate points. The optimal placement of these points is in itself an interesting problem, but for convenience, we can choose a uniform placement over the domain of $\mathbf{x}$. According to the properties of a GP, $\mathbf{f}_{\text{grid}}$, $\mathbf{f}$ and $\mathbf{u}$ are all jointly normally distributed. To find the posterior $p(\mathbf{f}_{\text{grid}}|\mathbf{f})$ we must first integrate out the inducing points to find $p(\mathbf{f}, \mathbf{f}_{\text{grid}})$:

$$
\begin{aligned}
p(\mathbf{f}, \mathbf{f}_{\text{grid}}) &= \int p(\mathbf{f}, \mathbf{f}_{\text{grid}}, \mathbf{u}) d\mathbf{u} \\
&= \int p(\mathbf{f}, \mathbf{f}_{\text{grid}}|\mathbf{u}) p(\mathbf{u}) d\mathbf{u}
\end{aligned} \tag{3.17}
$$

where $p(\mathbf{u}) = \mathcal{N}(\mathbf{0}, K_{\mathbf{u},\mathbf{u}})$. One model assumption shared by all the approximations reviewed by Quiñonero-Candela and Rasmussen (2005), is that the *training* values $\mathbf{f}$ and the *test* values $\mathbf{f}_{\text{grid}}$ are conditionally independent given $\mathbf{u}$:

$$
p(\mathbf{f}, \mathbf{f}_{\text{grid}}) \approx q(\mathbf{f}, \mathbf{f}_{\text{grid}}) = \int q(\mathbf{f}|\mathbf{u}) q(\mathbf{f}_{\text{grid}}|\mathbf{u}) p(\mathbf{u}) d\mathbf{u} \tag{3.18}
$$

This gives the following conditional distributions of $\mathbf{f}$ and $\mathbf{f}_{\text{grid}}$, where we use the notation $K_{\mathbf{a},\mathbf{b}}$ to mean the covariance matrix evaluated at the combinations of $x$ values of the vectors $\mathbf{a}$ and $\mathbf{b}$ (which are vectors of function values), and we define $Q_{\mathbf{a},\mathbf{b}} = K_{\mathbf{a},\mathbf{u}} K_{\mathbf{u},\mathbf{u}}^{-1} K_{\mathbf{u},\mathbf{b}}$.

$$
\begin{aligned}
\mathbf{f}|\mathbf{u} &\sim \mathcal{N}(K_{\mathbf{f},\mathbf{u}} K_{\mathbf{u},\mathbf{u}}^{-1} \mathbf{u}, \;\; K_{\mathbf{f},\mathbf{f}} - Q_{\mathbf{u},\mathbf{u}}) \\
\mathbf{f}_{\text{grid}}|\mathbf{u} &\sim \mathcal{N}(K_{\mathbf{f}_{\text{grid}},\mathbf{u}} K_{\mathbf{u},\mathbf{u}}^{-1} \mathbf{u}, \;\; K_{\mathbf{f}_{\text{grid}},\mathbf{f}_{\text{grid}}} - Q_{\mathbf{u},\mathbf{u}})
\end{aligned} \tag{3.19}
$$

As noted by Bauer et al. (2016), "the eigenvalues of $K_{\mathbf{u},\mathbf{u}}$ are not bounded away from zero. Any practical implementation will have to address this to avoid numerical instability." To be able to invert this matrix, we add a small *jitter term* to the diagonal of $K_{\mathbf{u},\mathbf{u}}$. This practice has been described by e.g., Bauer et al. (2016) and Titsias (2009). The addition of the jitter term affects the numerical properties of the model. It should be chosen as small as possible while still making the matrix invertible.

The approximate GP method we will use is referred to as *deterministic training conditional* (DTC) by Quiñonero-Candela and Rasmussen (2005). It assumes a deterministic relation between the inducing points $\mathbf{u}$ and the true tuning curve values $\mathbf{f}$, with the standard likelihood model for observations $\mathbf{f}$ with additive Gaussian noise: $p(\mathbf{f}|\mathbf{f}_{\text{noiseless}}) = \mathcal{N}(\mathbf{0}, \sigma^2 I)$. The relation between $\mathbf{f}_{\text{grid}}$ and $\mathbf{u}$ is kept the same as in equation (3.19). This gives us the following joint distribution:

$$q_{DTC}(\mathbf{f}, \mathbf{f}_{\text{grid}}) = \mathcal{N}\left(\mathbf{0}, \begin{bmatrix} Q_{\mathbf{f},\mathbf{f}} & Q_{\mathbf{f},\mathbf{f}_{\text{grid}}} \\ Q_{\mathbf{f}_{\text{grid}},\mathbf{f}} & K_{\mathbf{f}_{\text{grid}},\mathbf{f}_{\text{grid}}} \end{bmatrix}\right) \tag{3.20}$$

We want to find the posterior distribution on a grid given noisy observations $\mathbf{f}$. If we assume additive Gaussian noise, the posterior distribution of $\mathbf{f}_{\text{grid}}$ can be found easily using conjugacy:

$$\mathbf{f}_{\text{grid}}|\mathbf{f} \sim \mathcal{N}(Q_{\mathbf{f}_{\text{grid}},\mathbf{f}}(Q_{\mathbf{f},\mathbf{f}} + \sigma^2 I)^{-1}\mathbf{f},\ \ [K_{\mathbf{f}_{\text{grid}},\mathbf{f}_{\text{grid}}} - Q_{\mathbf{f}_{\text{grid}},\mathbf{f}}(Q_{\mathbf{f},\mathbf{f}} + \sigma^2 I)^{-1} Q_{\mathbf{f},\mathbf{f}_{\text{grid}}}]) \tag{3.21}$$

In Chapter 4 we will approximate $K_x$ by the sparse approximation $\tilde{K}_x := Q_{\mathbf{f},\mathbf{f}} + \sigma^2 I$. Note that this is an $N$ by $N$ matrix that still needs to be inverted in the model inference. But thanks to the inducing points we can use the matrix inversion lemma (eq. (A.2)) to invert an $N_{\text{ind}}$ by $N_{\text{ind}}$ matrix instead, and herein lies the computational advantage of the approximation.

$$\begin{aligned} (Q_{\mathbf{f},\mathbf{f}} + \sigma^2 I)^{-1} &= (K_{\mathbf{f},\mathbf{u}} K_{\mathbf{u},\mathbf{u}}^{-1} K_{\mathbf{u},\mathbf{f}} + \sigma^2 I)^{-1} \\ &= \sigma^{-2} I - \sigma^{-2} K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1} K_{\mathbf{f},\mathbf{u}}^T \end{aligned} \tag{3.22}$$

Here, $I$ represents the identity matrix of size $N$.

## 3.4 Principal component analysis

The following introduction has been adapted from Härdle and Simar (2007).

Let us introduce principal component analysis by looking at how to find the one-dimensional subspace that best represents a set of $N$ observations of $P$ random variables, $\{\mathbf{x}_1, \ldots, \mathbf{x}_N\}, \mathbf{x}_i \in \mathbb{R}^P$. By the best representation we mean the one dimensional subspace that minimizes

$$\sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{p}_i||^2 \tag{3.23}$$

where $\mathbf{p}_i$ are the projection points of $\mathbf{x}_i$ onto the subspace. Since the observations are projected orthogonally onto the lower dimensional subspace, we have from Pythagoras' theorem that $||\mathbf{x}_i - \mathbf{p}_i||^2 = ||\mathbf{x}_i||^2 - ||\mathbf{p}_i||^2$. Therefore, minimizing $\sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{p}_i||^2$ is equivalent to maximixing

$$\sum_{i=1}^{N} ||\mathbf{p}_i||^2 \tag{3.24}$$

The one-dimensional subspace can be fully described by a unit vector $\mathbf{u}_1 \in \mathbb{R}^P$ that makes up a basis of the subspace, and the coordinate of the projection in the low-dimensional subspace is $\mathbf{p}_i = \mathbf{x}_i^T \mathbf{u}_1$. By gathering the $N$ observations $\mathbf{x}_i$ in an observation matrix $\mathcal{X} \in \mathbb{R}^{N \times P}$, we have that

$$\begin{bmatrix} \mathbf{p}_1 \\ \mathbf{p}_2 \\ \vdots \\ \mathbf{p}_N \end{bmatrix} = \begin{bmatrix} \mathbf{x}_1^T \mathbf{u}_1 \\ \mathbf{x}_2^T \mathbf{u}_1 \\ \vdots \\ \mathbf{x}_N^T \mathbf{u}_1 \end{bmatrix} = \mathcal{X}\mathbf{u}_1 \tag{3.25}$$

Consequently, maximizing $\sum_{i=1}^{N} ||\mathbf{p}_i||^2$ is equivalent to maximizing

$$(\mathcal{X}\mathbf{u}_1)^T \mathcal{X}\mathbf{u}_1 = \mathbf{u}_1^T \mathcal{X}^T \mathcal{X}\mathbf{u}_1 \tag{3.26}$$

The solution to this is given by inserting $\mathcal{X}^T\mathcal{X}$ for A and $I$ for B in eq. (A.1), and we find that the $\mathbf{u}_1$ that maximizes the expression in 3.26 is equal to the largest eigenvalue $\lambda_1$ of $\mathcal{X}^T\mathcal{X}$. This generalizes to higher dimensions in the way that the $D$-dimensional subspace minimizing the sum

$$\sum_{i=1}^{N} ||\mathbf{x}_i - \mathbf{p}_i||^2 \tag{3.27}$$

is the subspace whose basis of unit vectors $\{\mathbf{u}_1, \dots, \mathbf{u}_D\}, \mathbf{u}_i \in \mathbb{R}^P$ are the eigenvectors corresponding to the $D$ largest eigenvalues of $\mathcal{X}^T\mathcal{X}$.

Now, instead of $P$ variables, let the $\{\mathbf{x}_1, \dots, \mathbf{x}_N\}, \mathbf{x}_i \in \mathbb{R}^P$ be observations of a $P$-dimensional random variable $\mathbf{X} \in \mathbb{R}^P$ with mean vector $\boldsymbol{\mu}$ and covariance matrix $\Sigma$. It is often helpful to standardize the observation before applying PCA. $\Sigma$ can be decomposed into $\Sigma = \Gamma\Lambda\Gamma^T$, where $\Lambda$ is the diagonal matrix of eigenvalues sorted in descending order, and $\Gamma$ is a matrix with columns equal to the corresponding eigenvectors of $\Sigma$. The PC transformation is defined as

$$\mathbf{Y} = \Gamma^T(\mathbf{X} - \boldsymbol{\mu}) \tag{3.28}$$

where $\mathbf{Y} = [Y_1, \dots, Y_P]^T$ contains the $P$ principal components, and these are orthogonal linear combinations of the $P$ dimensions of $\mathbf{X}$. When using PCA for dimensionality reduction, the standard approach is to select the first $m$ principal components that correspond to the largest $m$ eigenvalues. However, there is no guarantee that these variables will have the most explanatory power in a given regression setting.

## 3.5 Generalized Linear Models

The framework of generalized linear models (McCullagh and Nelder, 1989) gathers several common probability distributions into a unified framework. The three core components in a generalized linear model (GLM) are (i) a probability distribution belonging to the exponential family (the *random component*), (ii) a linear predictor $\eta = \mathbf{X}\boldsymbol{\beta}$ (the *systematic component*), and (iii) a *link function* $g(\mu_i)$ from the mean $\mu_i = E[y_i]$ to the linear predictor $\eta$. In the linear predictor $\eta$, $\mathbf{X}$ is the design matrix and $\boldsymbol{\beta}$ is the parameter vector.

The exponential family of distribution consists of all pdfs and pmfs $f_{Y_i}$ that can be expressed as

$$f_{Y_i}(y_i; \theta_i, \phi, w_i) = \exp\Big(\frac{y_i\theta_i - b(\theta_i)}{\phi} w_i + c(y_i, \phi, w_i)\Big) \tag{3.29}$$

where $\theta_i$ is called the *natural parameter*, $\phi$ the *dispersion parameter* and $w_i$ the *weight parameter*.

If $g(\eta)$ is chosen such that $g(\mu_i) = \theta_i$, it is called the canonical link function. The link function has a corresponding response function $h(\eta) = g^{-1}(\eta)$. For the Poisson distribution the canonical response function is

$$h(\eta) = \exp(\eta) \tag{3.30}$$

and for the Bernoulli distribution, the canonical response function is

$$h(\eta) = \frac{\exp(\eta)}{1 + \exp(\eta)} \tag{3.31}$$

With the canonical link function, the loglikelihood function becomes concave. This is very useful since it allows us to find maximum likelihood estimates easily using gradient-based optimization methods.

Chapter 4

# The latent manifold tuning model

In this chapter, in section 4.1, we will draw on theory from Chapter 3 to present the building blocks of the latent manifold tuning model as presented by Wu et al. (2017). Then, in section 4.2, we show how MAP estimates of the tuning curves and latent variable can be found, leading us to an iterative MAP procedure for joint inference of tuning curves and latent variable. The convergence properties of this method will be discussed in Chapter 5.

## 4.1 The latent manifold tuning model

We want to infer a low-dimensional latent variable that underlies the spike counts of $N$ neurons indexed by $i = 1, \ldots, N$, with time divided into bins indexed by $t = 1, \ldots, T$ over the period we are interested in. Let the number of spikes of neuron $i$ in bin $t$ be denoted by $y_{i,t}$. Furthermore, let $\mathbf{y}_t \in \mathbb{R}^N$ denote the vector of spike counts for all neurons at time $t$, let $\mathbf{y}_j \in \mathbb{R}^T$ denote the vector of spike counts in all time bins for neuron $j$, and let $\mathbf{Y} \in \mathbb{R}^{N \times T}$ denote the matrix of spike counts for all neurons for all time bins, with rows equal to $\mathbf{y}_j$ and columns equal to $\mathbf{y}_t$. The goal is to construct a model of the latent variable $\mathbf{x}(t)$ and the tuning curves $h_i(\mathbf{x})$ that allow us to infer $\mathbf{x}(t)$ and $\{h_i(\mathbf{x})\}$ given the observed spikes $\mathbf{Y}$.

### 4.1.1 Modeling the latent variable

The latent process is a $P$-dimensional latent variable $\mathbf{x}(t) \in \mathbb{R}^P$ that develops in time. In section 3.2 we introduced Gaussian processes. Each component $x_j(t), j = 1, \ldots, P$ of the latent variable is modeled as an independent Gaussian process in the time domain,

$$x_j(t) \sim \mathcal{GP}(0, k_t) \tag{4.1}$$

with zero mean and temporal covariance function $k_t(t, t')$. We will follow Wu et al and use an exponential covariance function,

$$k_t(t, t') = r \exp(-|t - t'|/l) \tag{4.2}$$

which enforces smoothness in time for the latent variable. This is reasonable for several physical variables, for example head direction or spatial position. Denote by $\mathbf{x}_j$ the vector of length $T$ containing the values $x_j(t)$ evaluated at all time bins. Since it is a Gaussian process it will then follow a normal distribution,

$$\mathbf{x}_j \sim \mathcal{N}(0, K_t), \tag{4.3}$$

where $K_t \in \mathbb{R}^{T\times T}$ is the covariance matrix containing the covariance function evaluated at every combination of time points. Let the vector $\mathbf{x}_t = \mathbf{x}(t)$ denote the value of the latent variable at time $t$, and let the matrix $\mathbf{X} \in \mathbb{R}^{P\times T}$ contain the values of the P-dimensional latent variable for all time bins, such that the rows of $\mathbf{X}$ are equal to $\mathbf{x}_j$.

### 4.1.2 Modeling the spike counts and tuning curves

A tuning curve can map the latent variable to for example a firing rate $\lambda_{i,t} = E[y_{i,t}]$ or a firing probability $P(y_{i,t} = 1)$, depending on the choice of response model for the spike data. We model the number of spikes $y_{i,t}$ of neuron $i$ in time bin $t$ as an inhomogeneous Poisson variable with a firing rate $\lambda_{i,t}$ that depends on the position of the latent variable at time $t$. For each neuron $i$ let the function $h_i(\mathbf{x}(t)) : \mathbb{R}^P \longmapsto \mathbb{R}$ describe a mapping from the latent variable state at time $t$ to the firing rate $\lambda_{i,t}$ of neuron $i$ at time $t$. $h_i(\mathbf{x})$ will be referred to as the tuning curve of neuron $i$:

$$\lambda_{i,t} = h_i(\mathbf{x}(t)) \tag{4.4}$$

A firing rate is constrained to positive values, so instead of inferring $h_i(\mathbf{x})$ directly, it can be practical to infer the log tuning curves $f_i(\mathbf{x}) = \log h_i(\mathbf{x}(t))$ for $i = 1, \ldots, N$ and $t = 1, \ldots, T$ and then find the values of $h_i(\mathbf{x})$ through the link

$$h_i(\mathbf{x}) = \exp(f_i(\mathbf{x})) \tag{4.5}$$

In the view of generalized linear models, the $f_i(\mathbf{x}(t))$ corresponds to the canonical parameter of the Poisson distribution. The spikes can also be modeled using another distribution, e.g., as Bernoulli variables, in which case the canonical link would be

$$h_i(\mathbf{x}) = \frac{\exp(f(\mathbf{x}))}{1 + \exp(f(\mathbf{x}))} \tag{4.6}$$

We will stick to the Poisson distribution in the remainder of the chapter. The corresponding results for a Bernoulli model using spike probability $\pi_{i,t} = P(y_{i,t} = 1)$ can be found in appendix A.2. The likelihood model links the observed spikes to the latent variable through the log tuning curves.

$$y_{i,t} | f_i, \mathbf{x}_t \sim \text{Poiss}(\exp(f_i(\mathbf{x}_t))) \tag{4.7}$$

We follow Wu et al. (2017) in modeling the log tuning curve $f_i(\mathbf{x})$ of neuron $i$ as a Gaussian process over the $P$-dimensional space of the latent variable,

$$f_i(\mathbf{x}) \sim \mathcal{GP}(0, k_x) \tag{4.8}$$

using the squared exponential covariance kernel, which we assume has the same parameters for all $N$ neurons:

$$k_x(\mathbf{x}, \mathbf{x}') = \sigma \exp(-||\mathbf{x} - \mathbf{x}'||_2^2 / 2\delta^2) \tag{4.9}$$

This enforces smoothness in the latent variable space for the tuning curves. The degree of smoothness can be adjusted by the choice of the parameter $\delta$, and the magnitude can be adjusted using the $\sigma$ variable.

Let the vector $\mathbf{f}_i \in \mathbb{R}^T$ contain the estimated value of the log tuning curve $f_{i,t} = f_i(\mathbf{x}_t)$ for all times $t$. With a noisy observation model, the multivariate normal distribution of $\mathbf{f}_i$ conditioned on the latent variable $\mathbf{X}$ is

$$\mathbf{f}_i | \mathbf{X} \sim \mathcal{N}(0, K_x + \sigma_\varepsilon^2 I) \tag{4.10}$$

where $K_x \in \mathbb{R}^{T \times T}$ is the covariance matrix of $\mathbf{f}_i$ containing elements $K_{x\{t,t'\}} = k_x(\mathbf{x}_t, \mathbf{x}_{t'})$ for every pair of latent states $(\mathbf{x}, \mathbf{x}')$ that $\mathbf{x}$ attains between $t = 1$ and $t = T$; $\sigma_\varepsilon^2$ is the noisy parameter, and $I$ is the identity matrix of size $T$.

Strictly speaking, the latent variable definition in equation (4.3) allows the $\mathbf{x}$ value to have the same value for two different time bins. In that case, the multivariate normal distribution in equation (4.10) can not represent a sample from a Gaussian process, since equation (4.10) allows $f_i(x)$ to have different values for the same $\mathbf{x}$ value, even when the noise parameter is set to zero. In this sense, $\mathbf{f}_i$ represents neither samples from a tuning curve or a Gaussian process, since a tuning curve only can have one value for a given $\mathbf{x}$. However, this is just a theoretical distinction and will have no practical implication for the inference.

To get a similar notation to $\mathbf{Y}$ and $\mathbf{X}$, we gather the $\mathbf{f}_i$ vectors as rows in the matrix $\mathbf{F} \in \mathbb{R}^{N \times T}$. Then the rows of $\mathbf{F}$ contain the values of the log tuning curves of a single neuron evaluated at every time bin, and a column $\mathbf{f}_t$ of $\mathbf{F}$ describes the values of the log tuning curves at time $t$ for the entire neuron population $i = 1, \ldots, N$.

## 4.2 Inference

Given the definitions in the previous section, the joint probability distribution of observations $\mathbf{Y}$, tuning curves $\mathbf{F}$, latent variables $\mathbf{X}$ and hyperparameters $\boldsymbol{\theta} = \{\sigma, \delta, r, l\}$ in the LMT model is:

$$\begin{aligned}
p(\mathbf{Y}, \mathbf{F}, \mathbf{X}, \boldsymbol{\theta}) &= p(\mathbf{Y}|\mathbf{F})p(\mathbf{F}|\mathbf{X}, \sigma, \delta)p(\mathbf{X}|r, l) \\
&= \prod_{i=1}^{N}\prod_{t=1}^{T} p(y_{i,t}|f_{i,t}) \prod_{i=1}^{N} p(\mathbf{f}_i|\mathbf{X}, \sigma, \delta) \prod_{j=1}^{P} p(\mathbf{x}_j|r, l) \\
&= \prod_{i=1}^{N}\prod_{t=1}^{T} \text{Poiss}(\exp(f_{i,t})) \prod_{i=1}^{N} \phi(\mathbf{f}_i; 0, K_x) \prod_{j=1}^{P} \phi(\mathbf{x}_j; 0, K_t) \\
&= \prod_{i=1}^{N}\prod_{t=1}^{T} \frac{(\exp(f_{i,t}))^{y_{i,t}}}{y_{i,t}!} \exp(-\exp(f_{i,t})) \\
&\times \frac{1}{(2\pi)^{\frac{NT}{2}}|K_x|^{\frac{N}{2}}} \exp\left( -\frac{1}{2}\sum_{i=1}^{N} \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) \\
&\times \frac{1}{(2\pi)^{\frac{PT}{2}}|K_t|^{\frac{P}{2}}} \exp\left( -\frac{1}{2}\sum_{j=1}^{P} \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right)
\end{aligned} \tag{4.11}$$

Here, $\phi(\mathbf{f}_i; 0, K_x)$ means the pdf of the multivariate normal distribution with mean 0 and covariance matrix $K_x$. From here on, we will not refer to the hyperparameters, in order to simplify the notation. Our goal is to find the maximum a posteriori estimates of $\hat{\mathbf{F}}_{\text{MAP}}$ and $\hat{\mathbf{X}}_{\text{MAP}}$. Wu et al. (2017) introduced the *decoupled Laplace approximation*, an iterative solution in which some initial estimate $\mathbf{X}^0$ is provided, and then at every iteration the estimate is updated based on an approximate posterior distribution of $\mathbf{X}$ obtained by using the Laplace approximation to integrate over the tuning curves $\mathbf{f}_i$. Another option is to use an iterative procedure where some initial $\mathbf{X}^0$ and $\mathbf{F}^0$ are chosen, and then update these estimates at every iteration $k$, first finding $\mathbf{F}^k = \hat{\mathbf{F}}_{\text{MAP}}$ by conditioning on $\mathbf{X}^{k-1}$, and then finding $\mathbf{X}^k = \hat{\mathbf{X}}_{\text{MAP}}$ by conditioning on $\mathbf{F}^k$. This is the approach that we will take. The algorithm is shown at the end of the chapter. In the following sections, we will show how these estimates are calculated.

### 4.2.1 MAP estimate of tuning curves

Each tuning curve $\mathbf{f}_i$ in $\mathbf{F}$ is modeled as independent from the others, so we can estimate them separately. We use Bayes' rule to express the posterior distribution of $\mathbf{f}_i$ given $\mathbf{X}$ and $\mathbf{y}_i$.

$$p(\mathbf{f}_i|\mathbf{y}_i, \mathbf{X}) = \frac{p(\mathbf{y}_i|\mathbf{f}_i)p(\mathbf{f}_i|\mathbf{X})}{p(\mathbf{y}_i|\mathbf{X})} \propto \prod_{t=1}^{T} p(y_{i,t}|f_{i,t}) \times p(\mathbf{f}_i|\mathbf{X}) \tag{4.12}$$

The Gaussian process provides the prior over the $\mathbf{f}_i$, and the Poisson spiking model provides the likelihood. Using the Poisson spiking model, we assume that the spike count $y_{i,t}$ of neuron $i$ in bin $t$ is Poisson distributed with firing rate equal to $\lambda_{i,t}$.

$$y_{i,t}|\lambda_{i,t} \sim \text{Poiss}(\lambda_{i,t}) = \frac{\lambda_{i,t}^{y_{i,t}}}{y_{i,t}!} \exp^{-\lambda_{i,t}} \tag{4.13}$$

with $y_{i,t} \in \{0, 1, 2, \dots\}$. Using the canonical link function $f_{i,t} = \log \lambda_{i,t}$, we can write this pdf as

$$\begin{aligned} p(y_{i,t}|f_{i,t}) &= \exp\Big(y_{i,t}f_{i,t} - \exp(f_{i,t}) - \log(y_{i,t}!)\Big) \\ \implies \log p(y_{i,t}|f_{i,t}) &= y_{i,t}f_{i,t} - \exp(f_{i,t}) - \log(y_{i,t}!) \end{aligned} \tag{4.14}$$

The MAP point estimate $\hat{\mathbf{f}}_i$ of each vector $\mathbf{f}_i$ is found independently by maximizing the log posterior, which using equations 4.10, 4.12 and 4.14, can be written as

$$\begin{aligned} \hat{\mathbf{f}}_i^k &= \text{argmax}_{\mathbf{f}_i} \log\Big(\frac{p(\mathbf{y}_i|\mathbf{f}_i)p(\mathbf{f}_i|\mathbf{X})}{p(\mathbf{y}_i|\mathbf{X})}\Big) \\ &= \text{argmax}_{\mathbf{f}_i} \log\Big(p(\mathbf{y}_i|\mathbf{f}_i)\Big) + \log\Big(p(\mathbf{f}_i|\mathbf{X})\Big) \\ &= \text{argmax}_{\mathbf{f}_i}\Big[\sum_{t=1}^{T}(y_{i,t}f_{i,t} - \exp(f_{i,t})) - \frac{1}{2}\mathbf{f}_i^T K_x^{-1}\mathbf{f}_i\Big] \end{aligned} \tag{4.15}$$

where terms that are constant in $\mathbf{f}_i$ have been omitted. Since the Poisson distribution belongs to the exponential family and we are using the canonical link function, we know that the likelihood function is concave in $f_{i,t}$. Furthermore, we see that the quadratic term $-\frac{1}{2}\mathbf{f}_i^T K_x^{-1}\mathbf{f}_i$ is concave, since $K_x^{-1}$ is a positive semi-definite matrix, meaning that the sum of these two functions is concave. Therefore, finding the MAP estimate of $\mathbf{f}$ is a concave optimization problem that can be solved optimally using any gradient-based optimization method. To improve the speed of the algorithm we can find the gradient and hessian matrix by differentiating the expression explicitly. Let $\Psi(\mathbf{f}_i)$ denote the objective function for this optimization problem:

$$\begin{aligned} \Psi(\mathbf{f}_i) :&= \log p(\mathbf{y}_i|\mathbf{f}_i) + \log p(\mathbf{f}_i|\mathbf{X}) \\ &= \sum_{t=1}^{T}\Big[y_{i,t}f_{i,t} - \exp(f_{i,t}) - \log(y_{i,t}!)\Big] - \frac{1}{2}\mathbf{f}_i^T K_x^{-1}\mathbf{f}_i \end{aligned} \tag{4.16}$$

We calculate the first derivative of $\Psi(\mathbf{f}_i)$. The gradient notation used is $\nabla = [\frac{\partial}{\partial f_{i,1}} \dots \frac{\partial}{\partial f_{i,T}}]^T$.

$$\begin{aligned} \frac{\partial}{\partial f_{i,t}}\Psi(\mathbf{f}_i) &= y_{i,t} - \exp(f_{i,t}) - \sum_{j=1}^{T} f_{i,j}K_{x\{t,j\}}^{-1} \\ \iff \nabla\Psi(\mathbf{f}_i) &= \mathbf{y}_i - \mathbf{e}_i^{\text{Poiss}} - K_x^{-1}\mathbf{f}_i, \end{aligned} \tag{4.17}$$

where we have used the fact that $K_x$ is symmetric. Here the vector $\mathbf{e}_i^{poiss}$ has elements $e_{i,t}^{poiss} = \exp(f_{i,t}), t = 1, \dots, T$. We calculate the second derivative of $\Psi(\mathbf{f}_i)$:

$$\begin{aligned} \frac{\partial^2}{\partial f_{i,t_1}\partial f_{i,t_2}}\Psi(\mathbf{f}_i) &= \begin{cases} -\exp(f_{i,t_1}) - K_{x\{t_1,t_1\}} & \text{for} \quad t_1 = t_2 \\ -K_{x\{t_1,t_2\}} & \text{for} \quad t_1 \neq t_2 \end{cases} \\ \implies \nabla\nabla\Psi(\mathbf{f}_i) &= -I\mathbf{e}_i^{poiss} - K_x^{-1} \end{aligned} \tag{4.18}$$

where $I$ represents the identity matrix of size $T$.

### 4.2.2 MAP estimate of the latent variable

For the posterior distribution of $\mathbf{X}$, the prior is the Gaussian process of $\mathbf{X}$, and the Gaussian processes of the tuning curves $p(\mathbf{f}_i|\mathbf{X})$ take on the role of a likelihood model. Notice that when $\mathbf{X}$ is conditioned

on $\mathbf{F}$, the observed spikes $\mathbf{Y}$ do not enter into the MAP expression for $\mathbf{X}$.

$$\begin{aligned}
p(\mathbf{X}|\mathbf{F}) &= \frac{p(\mathbf{F}|\mathbf{X})p(\mathbf{X})}{p(\mathbf{F})} \\
&= [p(\mathbf{F})]^{-1} \prod_{i=1}^{N} p(\mathbf{f}_i|\mathbf{X}) \prod_{j=1}^{P} p(\mathbf{x}_j) \\
&= [p(\mathbf{F})]^{-1} \times \frac{1}{(2\pi)^{\frac{NT}{2}} |K_x|^{\frac{N}{2}}} \exp\left( -\frac{1}{2} \sum_{i=1}^{N} \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) \\
&\qquad \times \frac{1}{(2\pi)^{\frac{PT}{2}} |K_t|^{\frac{P}{2}}} \exp\left( -\frac{1}{2} \sum_{j=1}^{P} \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right) \\
&= \frac{1}{|K_x|^{\frac{N}{2}}} \exp\left( -\frac{1}{2} \sum_{i=1}^{N} \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) \exp\left( -\frac{1}{2} \sum_{j=1}^{P} \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right) \times C' \\
\Rightarrow \log p(\mathbf{X}|\mathbf{F}) &= -\frac{N}{2} \log |K_x| - \frac{1}{2} \sum_{i=1}^{N} \left( \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) - \frac{1}{2} \sum_{j=1}^{P} \left( \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right) + C
\end{aligned} \tag{4.19}$$

where $C' = [p(\mathbf{F}) \times (2\pi)^{\frac{(N+P)T}{2}} |K_t|^{\frac{P}{2}}]^{-1}$ and $C = \log C'$. The MAP estimate of $\mathbf{X}$ is found by maximizing the log posterior:

$$\hat{\mathbf{X}}_{\text{MAP}}|\mathbf{F} = \text{argmax}_{\mathbf{X}} \left[ -\frac{N}{2} \log |K_x| - \frac{1}{2} \sum_{i=1}^{N} \left( \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) - \frac{1}{2} \sum_{j=1}^{P} \left( \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right) \right] \tag{4.20}$$

and we define the objective function

$$\mathcal{L}(\mathbf{X}) := -\frac{N}{2} \log |K_x| - \frac{1}{2} \sum_{i=1}^{N} \left( \mathbf{f}_i^T K_x^{-1} \mathbf{f}_i \right) - \frac{1}{2} \sum_{j=1}^{P} \left( \mathbf{x}_j^T K_t^{-1} \mathbf{x}_j \right) \tag{4.21}$$

Observe that this function is not concave in $\mathbf{X}$. It is therefore not guaranteed that the estimate of $\mathbf{X}$ found by a gradient-based optimization method will be optimal. Furthermore, in order to address the challenges with the computational complexity of the covariance matrix, we replace the $T$ by $T$ covariance matrix $K_x$ with its sparse deterministic training conditional approximation $\tilde{K}_x$ as described in subsection 3.3.

$$\tilde{K}_x = K_{\mathbf{f},\mathbf{u}} K_{\mathbf{u},\mathbf{u}}^{-1} K_{\mathbf{u},\mathbf{f}} + \sigma^2 I \tag{4.22}$$

where $I$ is the identity matrix of size $T$; $\mathbf{u}$ is the vector of function values at $N_{\text{ind}}$ inducing points uniformly spaced in the range of $\mathbf{x}$ and the covariance matrices $K_{\mathbf{u},\mathbf{u}} \in \mathbb{R}^{N_{\text{ind}} \times N_{\text{ind}}}$, $K_{\mathbf{f},\mathbf{u}} \in \mathbb{R}^{T \times N_{\text{ind}}}$ and $K_{\mathbf{u},\mathbf{f}} = K_{\mathbf{f},\mathbf{u}}^T$ are found as described in section 3.2:

$$\begin{aligned}
K_{\mathbf{u},\mathbf{u}[i,j]} &= k(\mathbf{x}_{u_i}, \mathbf{x}_{u_j}) \\
K_{\mathbf{f},\mathbf{u}[i,j]} &= k(\mathbf{x}_{t_i}, \mathbf{x}_{u_j})
\end{aligned} \tag{4.23}$$

where $\mathbf{x}_{t_i}$ are the estimated states of the latent variables at times $t_i$ and $t_j$, and $\mathbf{x}_{u_j}$ are the inducing points. In the next sections we derive the expression of $\mathcal{L}(\mathbf{X})$ with the sparse approximation $\tilde{K}_x$. Since this part involves some algebra, we break the log posterior of $\mathbf{X}$ into its additive terms and

evaluate the effect of the inducing points approximation on each of them separately. We name the terms in $\mathcal{L}(\mathbf{X})$ as follows:

$$\begin{aligned}\mathcal{L}(\mathbf{X}) &= -\frac{N}{2}\log|\tilde{K}_x| - \frac{1}{2}\sum_{i=1}^{N}\Big(\mathbf{f}_i^T\tilde{K}_x^{-1}\mathbf{f}_i\Big) - \frac{1}{2}\sum_{j=1}^{P}\Big(\mathbf{x}_j^T K_t^{-1}\mathbf{x}_j\Big) \\ &= \text{log determinant term} + \text{quadratic term} + \text{x prior term}\end{aligned} \tag{4.24}$$

**The log determinant term**

With the sparse approximation, the logdeterminant term becomes:

$$\begin{aligned}-\frac{N}{2}\log|\tilde{K}_x| &= -\frac{N}{2}\log|K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}} + \sigma^2 I_T| \\ &= -\frac{N}{2}\log|(K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}\sigma^{-2} + I_T)(\sigma^2 I_T)| \\ &= -\frac{N}{2}\Big(\log|K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}\sigma^{-2} + I_T| + T\log(\sigma^2)\Big) \\ &= -\frac{N}{2}\Big(\log|K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}}\sigma^{-2} + I_{N_{\text{ind}}}| + T\log(\sigma^2)\Big) \\ &= -\frac{N}{2}\Big(\log|K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}}\sigma^{-2} + K_{\mathbf{u},\mathbf{u}}| + \log|K_{\mathbf{u},\mathbf{u}}|^{-1} + T\log(\sigma^2)\Big) \\ &= -\frac{N}{2}\Big(\log|K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}} + \sigma^2 K_{\mathbf{u},\mathbf{u}}| - N_{\text{ind}}\log|\sigma^2| - \log|K_{\mathbf{u},\mathbf{u}}| + T\log(\sigma^2)\Big) \\ &= -\frac{N}{2}\Big(\log|K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}} + \sigma^2 K_{\mathbf{u},\mathbf{u}}| - \log|K_{\mathbf{u},\mathbf{u}}| + (T - N_{\text{ind}})\log(\sigma^2)\Big)\end{aligned} \tag{4.25}$$

In the transition from the third to the fourth line we use Theorem 1.3.22 from Horn and Johnson (1985), listed in appendix A.1.3, stating that the eigenvalues of $K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}$ are equal to the eigenvalues of $K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}}$ together with $T - N$ zeroes. Clearly, $K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}$ is a symmetric matrix. For any symmetric matrix $A$ we have that

$$\text{eig}(cA + I) = c\ \text{eig}(A) + 1 \tag{4.26}$$

where eig$(A)$ means the eigenvalues of $A$. Therefore

$$\text{eig}(K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}\sigma^{-2} + I_T) = \sigma^{-2}\text{eig}(K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}) + 1 \tag{4.27}$$

Together with Theorem 1.3.22 from Horn and Johnson (1985), this means that

$$\begin{aligned}|K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}\sigma^{-2} + I_T| &= \prod_{e_i\in\text{eig}(K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}})}(\sigma^{-2}e_i + 1) \\ &= \prod_{e_i\in\text{eig}(K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}})}(\sigma^{-2}e_i + 1) \\ &= |K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}}\sigma^{-2} + I_{N_{\text{ind}}}|\end{aligned} \tag{4.28}$$

allowing us to make the transition from line three to four.

**The quadratic term**

$$
\begin{aligned}
\text{quadratic term} &= -\frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T \tilde{K}_x^{-1}\mathbf{f}_i\right) \\
&= -\frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T (K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{u},\mathbf{f}}\sigma^{-2} + I_T)^{-1}\mathbf{f}_i\right) \\
&= -\frac{1}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T (\sigma^{-2}I_{N_{\text{ind}}} - \sigma^{-2}K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T)\mathbf{f}_i\right) \\
&= -\frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T\mathbf{f}_i\right) + \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T (K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T)\mathbf{f}_i\right)
\end{aligned}
\tag{4.29}
$$

Here we have applied the Matrix inversion lemma (eq. (A.2)) to make the inversion of the covariance matrix computationally efficient.

Notice that the x prior term in eq. (4.24) is not affected by the inducing points approximation. Thus the objective function with the inducing points approximation is:

$$
\begin{aligned}
\mathcal{L}(\mathbf{X}) = &-\frac{1}{2}\sum_{i=1}^{N}\frac{N}{2}\log|K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}} + \sigma^2 K_{\mathbf{u},\mathbf{u}}| - \frac{N}{2}\log|K_{\mathbf{u},\mathbf{u}}^{-1}| - \frac{N(T-N_{\text{ind}})}{2}\log|\sigma^2| \\
&- \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T\mathbf{f}_i\right) + \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T (K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T)\mathbf{f}_i\right) \\
&- \frac{1}{2}\sum_{j=1}^{P}\left(\mathbf{x}_j^T K_t^{-1}\mathbf{x}_j\right)
\end{aligned}
\tag{4.30}
$$

### 4.2.3 Gradient

Using numerical estimates for the gradient is very slow when the number of bins $T$ is large. Therefore, we want to derive an analytical expression for the gradient of $\mathcal{L}(\mathbf{X})$ for the optimization. Calculations are shown here for a one-dimensional latent variable, denoted by $\mathbf{x} = (x_1, \ldots, x_T)$. For clarity, the notation used for the gradient is

$$
\nabla\mathcal{L}(\mathbf{X}) = \begin{bmatrix} \frac{\partial}{\partial x_1}\mathcal{L}(\mathbf{X}) & \ldots & \frac{\partial}{\partial x_T}\mathcal{L}(\mathbf{X}) \end{bmatrix}^T \tag{4.31}
$$

Again we will treat the three terms of $\mathcal{L}(\mathbf{X})$ separately:

$$
\frac{\partial}{\partial X_t}\mathcal{L}(\mathbf{X}) = \frac{\partial}{\partial X_t}(\text{log determinant term}) + \frac{\partial}{\partial X_t}(\text{quadratic term}) + \frac{\partial}{\partial X_t}(\text{x prior term}) \tag{4.32}
$$

**Gradient of the log determinant term**

We define $B := (K_{\mathbf{u,f}} K_{\mathbf{f,u}} + \sigma^2 K_{\mathbf{u,u}})$.

$$
\begin{aligned}
\frac{\partial}{\partial x_t}(\text{log determinant term}) &= \frac{\partial}{\partial x_t}\left[ -\frac{N}{2}\log|B| + \frac{N}{2}\log|K_{\mathbf{u,u}}| - \frac{N(T-N_{\text{ind}})}{2}\log|\sigma^2| \right] \\
&= -\frac{N}{2}\left(\frac{\partial}{\partial x_t}\log|B|\right) + \frac{N}{2}\frac{\partial}{\partial x_t}\left(\log|K_{\mathbf{u,u}}|\right) \\
&= -\frac{N}{2}\text{trace}\left(B^{-1}\frac{\partial}{\partial x_t}(K_{\mathbf{u,f}}K_{\mathbf{f,u}} + \sigma^2 K_{\mathbf{u,u}})\right) + \frac{N}{2}\text{tr}\left(K_{\mathbf{u,u}}^{-1}\frac{\partial}{\partial x_t}(K_{\mathbf{u,u}})\right) \\
&= -\frac{N}{2}\text{trace}\left(B^{-1}\Big[(\frac{\partial}{\partial x_t}K_{\mathbf{u,f}})K_{\mathbf{f,u}} + K_{\mathbf{u,f}}(\frac{\partial}{\partial x_t}K_{\mathbf{f,u}})\Big]\right) \\
&= -\frac{N}{2}\text{trace}\left(B^{-1}\Big[[K_{\mathbf{u,f}}(\frac{\partial}{\partial x_t}K_{\mathbf{f,u}})]^T + K_{\mathbf{u,f}}(\frac{\partial}{\partial x_t}K_{\mathbf{f,u}}))\Big]\right)
\end{aligned}
\tag{4.33}
$$

To differentiate the log determinant of a matrix, we use equation (A.8). In the transition from line two to three, we use the fact that $K_{\mathbf{u,u}}$ does not depend on x, then in the transition from lines three to four, we use the product rule for derivatives. In the transition from line four to five we used the fact that $K_{\mathbf{f,u}}$ and $K_{\mathbf{u,f}}$ are each other's transpose and that $(AB)^T = B^T A^T$.

$K_{\mathbf{f,u}}$ is a $T$ by $N_{\text{ind}}$ matrix with entries defined by the squared exponential covariance kernel. For any $t \in [1, \ldots, T]$:

$$
K_{\mathbf{f,u}_{[t,j]}} = \sigma \exp(-(x_t - \mathbf{x}_{u_j})^2/(2\delta^2)) \tag{4.34}
$$

So

$$
\frac{\partial}{\partial x_t}K_{\mathbf{f,u}_{[t,j]}} = -(x_t - \mathbf{x}_{u_j})\frac{\sigma}{\delta^2}\exp(-(x_t - \mathbf{x}_{u_j})^2/(2\delta^2)) := f_1(x_t, \mathbf{x}_{u_j}), \tag{4.35}
$$

and

$$
\frac{\partial}{\partial x_t}K_{\mathbf{f,u}} = \begin{bmatrix} \mathbf{0}_{t-1,N_{\text{ind}}} \\ \mathbf{f_1}(x_t, \mathbf{x_{grid}}) \\ \mathbf{0}_{T-t,N_{\text{ind}}} \end{bmatrix} \tag{4.36}
$$

where $\mathbf{f_1}(x_t, \mathbf{x_{grid}}) = \Big[f_1(x_t, \mathbf{x}_{u_1}) \ldots f_1(x_t, \mathbf{x}_{u_{N_{\text{ind}}}})\Big]$ and the zero matrices have the appropriate size such that the nonzero row is placed at index $t$.

**Gradient of the quadratic term**

We compute the gradient of the quadratic term (the two terms in the second row of the right-hand side of eq. (4.30)). We will need the derivative of the $B$ matrix:

$$
\begin{aligned}
\frac{\partial}{\partial x_t}B &= \frac{\partial}{\partial x_t}((K_{\mathbf{u,f}}K_{\mathbf{f,u}} + \sigma^2 K_{\mathbf{u,u}})) \\
&= \frac{\partial}{\partial x_t}(K_{\mathbf{u,f}}K_{\mathbf{f,u}}) \\
&= \Big(\frac{\partial}{\partial x_t}K_{\mathbf{u,f}}\Big)K_{\mathbf{f,u}} + K_{\mathbf{u,f}}\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f,u}}\Big)
\end{aligned}
\tag{4.37}
$$

Eq. 4.37 allows us to develop the gradient of the quadratic term as follows:

$$
\begin{aligned}
\frac{\partial}{\partial x_t}(\text{quadratic term}) &= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Big(K_{\mathbf{f},\mathbf{u}}(\sigma^2K_{\mathbf{u},\mathbf{u}}+K_{\mathbf{f},\mathbf{u}}^TK_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T\Big)\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\frac{\partial}{\partial x_t}\Big(K_{\mathbf{f},\mathbf{u}}B^{-1}K_{\mathbf{u},\mathbf{f}}\Big)\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Bigg[\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}}\Big)B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}\Big(\frac{\partial}{\partial x_t}B^{-1}\Big)K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}B^{-1}\Big(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}}\Big)\Bigg]\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Bigg[\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}}\Big)B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}\Big(-B^{-1}\Big(\frac{\partial}{\partial x_t}B\Big)B^{-1}\Big)K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}B^{-1}\Big(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}}\Big)\Bigg]\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Bigg[\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}}\Big)B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}\Big(-B^{-1}\Big((\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}})K_{\mathbf{f},\mathbf{u}}+K_{\mathbf{u},\mathbf{f}}(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}})\Big)B^{-1}\Big)K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}B^{-1}\Big(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}}\Big)\Bigg]\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Bigg[\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}}\Big)B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad - K_{\mathbf{f},\mathbf{u}}B^{-1}(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}})K_{\mathbf{f},\mathbf{u}}B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad - K_{\mathbf{f},\mathbf{u}}B^{-1}K_{\mathbf{u},\mathbf{f}}(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}})B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}B^{-1}\Big(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}}\Big)\Bigg]\mathbf{f}_i \\
&= \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\mathbf{f}_i^T\Bigg[\Bigg(I_T-K_{\mathbf{f},\mathbf{u}}B^{-1}K_{\mathbf{u},\mathbf{f}}\Bigg)\Big(\frac{\partial}{\partial x_t}K_{\mathbf{f},\mathbf{u}}\Big)B^{-1}K_{\mathbf{u},\mathbf{f}} \\
&\qquad + K_{\mathbf{f},\mathbf{u}}B^{-1}(\frac{\partial}{\partial x_t}K_{\mathbf{u},\mathbf{f}})\Bigg(I_T-K_{\mathbf{f},\mathbf{u}}B^{-1}K_{\mathbf{u},\mathbf{f}}\Bigg)\Bigg]\mathbf{f}_i
\end{aligned}
\tag{4.38}
$$

where in the third line we use the product rule for differentiation: (ABC)' = A'BC + AB'C + ABC', and in the fourth line we use equation (A.6) to differentiate the inverse of $B$. Notice that $B =$

$(K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}} + \sigma^2 K_{\mathbf{u},\mathbf{u}})^{-1}$ is symmetric because $K_{\mathbf{u},\mathbf{f}} = K_{\mathbf{f},\mathbf{u}}^T$ and because a product $AA^T$ is always symmetric.

**Gradient of the x prior term**

Finding the gradient of the x prior term is easy.

$$\begin{aligned}\nabla(\text{x prior term}) &= -\frac{1}{2}\nabla[\mathbf{x}_j^T K_t^{-1}\mathbf{x}_j] \\ &= -\mathbf{x}_j^T K_t^{-1}\end{aligned} \tag{4.39}$$

### 4.2.4 The iterative MAP procedure

By starting with some initial guesses $\mathbf{X}^0$ and $\mathbf{F}^0$, and then updating the estimates iteratively using the maximum a posteriori formulas we have described, one can converge to some final estimates of both $\mathbf{F}$ and $\mathbf{X}$. This setup is described in algorithm 1 and corresponds to Algorithm 1 in Wu et al. (2017) without the decoupled Laplace approximation.

As mentioned in section 3.2, we will start with a high noise term for the Gaussian process tuning curves, $\sigma_\varepsilon^2$, and then lower its value at every iteration by multiplying with a learning rate $lr$, where $0 < lr < 1$. By increasing the assumed noise level in the model, $\mathbf{f}_i$ estimates that would be considered extreme or highly unlikely with a lower noise term, appear more likely. This changes the objective function, allowing the algorithm to explore more in the first iterations. Convergence is declared when the root mean squared error deviation between the estimates $\mathbf{X}^k$ and $\mathbf{X}^{k-1}$ becomes lower than the predetermined tolerance value, or by having reached the predetermined maximum number of iterations.

**Algorithm 1:** Iterative MAP procedure

**Input:** observations $\mathbf{Y}$, initial guesses $\mathbf{X}^0$ and $\mathbf{F}^0$, initial $\sigma_\varepsilon^2$

1 **begin**
2   **while** ***not converged, at iteration*** $k$ **do**
3     **for** *i = 1, . . . , N* **do**
4       $\hat{\mathbf{f}}_i^k = \text{argmax}_{\mathbf{f}_i}\Psi(\mathbf{f}_i)$
5     **end**
6     $\hat{\mathbf{X}}^k = \text{argmax}_{\mathbf{X}}\mathcal{L}(\mathbf{X})$
7     $\sigma_\varepsilon^2 = lr \times \sigma_\varepsilon^2$
8   **end**
9   **return** $\hat{\mathbf{F}}^k$, $\hat{\mathbf{X}}^k$
10 **end**

We will make one more alteration to algorithm 1. As we shall see in Chapter 5, initial estimates of $\mathbf{F}$ and $\mathbf{X}$ can be obtained based on the observed matrix of spikes $\mathbf{Y}$. An explanation of this may be that the initial estimate of $\mathbf{F}$ is better than the initial estimate for $\mathbf{X}$. If the $\mathbf{X}$ estimate is bad, then by updating $\mathbf{F}$ in the first iteration, we are worsening the $\mathbf{F}$ estimate, which has consequences for the remaining iterations. Therefore, we alter algorithm 1 by skipping the $\mathbf{F}$ update in the first iteration. To find the MAP estimates at each step, an efficient approximate conjugate gradient method known as L-BFGS-B (Zhu et al., 1997) is used. The initial estimates for $\mathbf{F}$ and $\mathbf{X}$ will be discussed in greater detail in Chapter 5.

Chapter 5

# Applying the LMT model to simulated and experimental data

In this chapter, we present results from the implementation of the algorithm presented in chapter 4, starting with a discussion on the challenges with convergence, which we address in sections 5.1 and 5.2. In section 5.3, we evaluate the robustness of the model on synthetic data against two fundamental hyperparameters, namely the tuning strength and the data length. Finally, in section 5.4, we use the model to infer head direction in the dataset described in Chapter 2.

## 5.1 Convergence and pitfalls

As mentioned in Chapter 4, the objective function we maximize to find the posterior estimate of the latent variable $\mathbf{X}$ is not a concave function. Therefore, there is no guarantee that the local maximum the algorithm converges to will be “optimal” in any sense of the word. Our definition of the “best” estimate will be the one with the lowest root mean squared error (RMSE) between the final estimate $\hat{\mathbf{X}}$ and the true $\mathbf{X}$, which is only available for synthetic data.

$$\text{RMSE} = \sqrt{\frac{1}{T}\sum_{t=1}^{T} ||\mathbf{x}_t - \hat{\mathbf{x}}_t||_2^2} \tag{5.1}$$

A common technique for dealing with non-concave problems, mentioned in section 3.1.3, is to start at several initial estimates and then pick the best estimate once they all have converged. However, when choosing between these estimates, we cannot use the RMSE value, since we cannot use the true $\mathbf{X}$ to select our estimate. Instead, the log posterior $\mathcal{L}(\mathbf{X})$ can be used to rate different solutions. Unfortunately, there is no guarantee that the estimate with the lowest RMSE will also have the highest $\mathcal{L}(\mathbf{X})$ value. It may even be the case that picking between estimates based on the $\mathcal{L}(\mathbf{X})$ value gives a worse RMSE value on average than just picking the estimate randomly. We will investigate the feasibility of using $\mathcal{L}(\mathbf{X})$ to pick between estimates in section 5.3. First, we will define a simple simulated latent variable to get to know the model.

In this section we will look at a simulated one-dimensional latent variable with values restricted by a minimum and a maximum value: $\mathbf{x} \in [\mathbf{x}_{\text{min}}, \mathbf{x}_{\text{max}}] \subset \mathbb{R}^1$. Every log tuning curve $f_i(\mathbf{x})$ will be defined as a Gaussian bump with its peak positioned randomly between $\mathbf{x}_{\text{min}}$ and $\mathbf{x}_{\text{max}}$. The *tuning strength* of a neuron is defined as its firing rate $h(\mathbf{x}) = \exp(f(\mathbf{x}))$ when $\mathbf{x}$ is positioned exactly where the tuning curve has its peak. We will also refer to the *background firing rate*, which is the firing rate a neuron approaches in the limit as the distance between $\mathbf{x}$ and the peak of its tuning curve increases.

To link this setup to a neuroscience scenario, imagine a rodent in a corridor that is so narrow that movement is mainly one-dimensional except when the animal is turning, and let the recorded neurons be place-cells that are tuned to different locations along the length of the corridor. At the center of the place field, the tuning curve of the neuron will reach the tuning strength value, while outside of the place field, the probability of the neuron firing in each time bin will correspond to the background firing rate.

We choose $\mathbf{x}_{\text{min}} = 0$ and $\mathbf{x}_{\text{max}} = 10$. The "path" of $\mathbf{x}$ is sampled from a generative Gaussian process identical to the prior of $\mathbf{X}$ in chapter 4. A neuron will only produce spikes if the simulated latent variable visits the zone it is tuned to during the simulation. Therefore, to record information about all the neurons, it is desirable to sample a path that covers the entire domain of $\mathbf{X}$. Since the path of $\mathbf{X}$ is sampled randomly from the Gaussian process prior, the minimum and maximum values will vary between simulations. By increasing the variance parameter $\sigma_x$, we can make sure the path covers our chosen domain of $[0, 10]$. To keep the path inside the domain, we "fold" the path back into the domain whenever it moves outside the min or max value. This is an important distinction because this path is no longer strictly sampled from the generative Gaussian prior, and consequently has a slightly different distribution. The hyperparameters in this example were chosen by trial and error, and the jitter term added to the diagonal of $K_{\mathbf{u},\mathbf{u}}$ was set to $10^{-5}$. Figure 5.1 shows an example path with length $T = 1000$ sampled from the prior and kept inside the domain by folding it back whenever it meets the boundary.

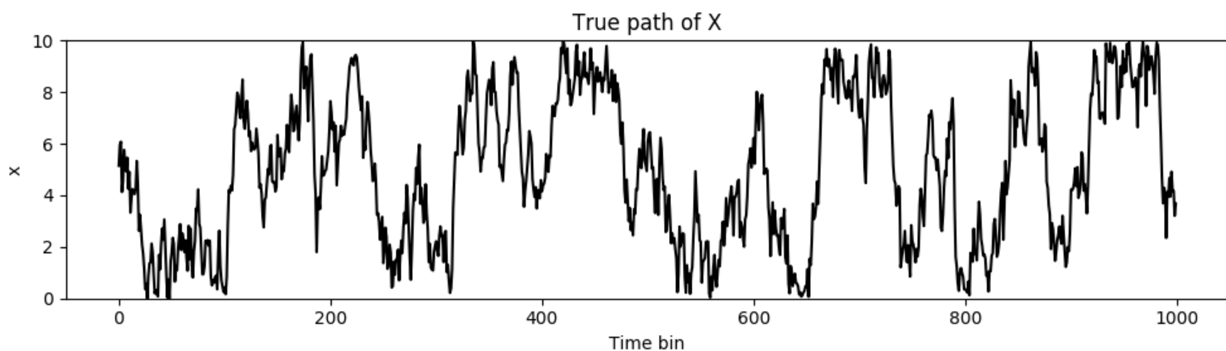


**Figure 5.1:** An example of a generated path for the latent variable, kept between the limits of 0 and 10 by folding it back whenever the path goes outside the domain.

### A comment on neural density and inactive neurons

The background firing rate was set to 0.5 spikes per bin, and the tuning strength set to 4 spikes per bin. There will be a total of 100 neurons, whose tuning peaks are distributed randomly along the domain of $\mathbf{x}$. Ideally, we would want the number of neuron peaks per unit length in the domain of $\mathbf{x}$ (the *neural density*) to be constant for the estimate of $\mathbf{X}$ to be equally good at different regions in the domain. However, consider the tuning curves shown in Figure 5.2, which represents a realization of the tuning curve definition just described. Here, the neural density is lower in the regions close to the edges of the domain. To achieve a constant neural density, we should either add some neurons to each side with peaks positioned outside the domain and tails entering the domain or impose periodic boundary conditions, which we do not want to do because it is harder to infer a periodic variable, as we shall see in section 5.4. Unfortunately, experiments have shown that if neurons are added with too few observed spikes, the quality of the $\mathbf{X}$ estimate is worsened. Therefore, we choose to not add any neurons that are partly outside, and as a consequence of this, we should expect the inference of $\mathbf{X}$ to be less precise near the edges of the domain.

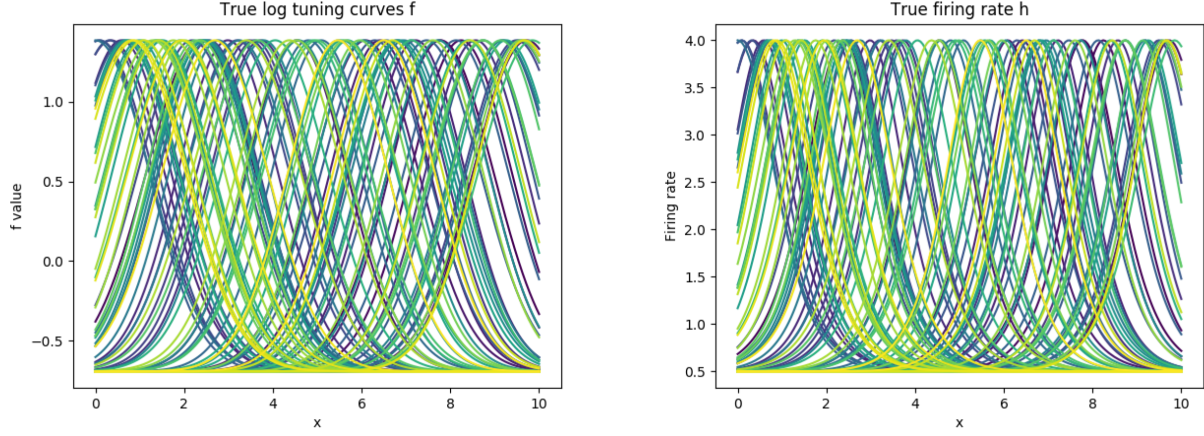


**Figure 5.2:** Left: Simulated log tuning curves $\mathbf{f}_i(\mathbf{x})$ with peaks distributed randomly across the domain of $\mathbf{x}$. Right: Corresponding firing rates $\mathbf{h}_i(\mathbf{x}) = \exp(\mathbf{f}_i(\mathbf{x}))$ with background firing rate is 0.5 and tuning strength 4.

### 5.1.1 Flipping

We will now describe some types of local maxima that the algorithm may converge to. Since the log posterior of $\mathbf{X}$ is not concave, the starting point of X will determine how good the final estimate is. First, because $K_{\mathbf{f},\mathbf{u}}$ only depends on on $\mathbf{X}$ through the squared distances $(\mathbf{x}_t - \mathbf{x}_{u_j})^2$, we see that $\mathcal{L}(\mathbf{X})$ is an even function:

$$\begin{aligned}
\mathcal{L}(-\mathbf{X}) &= -\frac{1}{2}\sum_{i=1}^{N}\frac{N}{2}\log|K_{\mathbf{u},\mathbf{f}}K_{\mathbf{f},\mathbf{u}} + \sigma^2 K_{\mathbf{u},\mathbf{u}}| - \frac{N}{2}\log|K_{\mathbf{u},\mathbf{u}}^{-1}| - \frac{N(T-N_{\text{ind}})}{2}\log|\sigma^2| \\
&- \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T\mathbf{f}_i\right) + \frac{\sigma^{-2}}{2}\sum_{i=1}^{N}\left(\mathbf{f}_i^T(K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T)\mathbf{f}_i\right) \\
&- \frac{1}{2}\sum_{j=1}^{P}\left((-\mathbf{x}_j^T)K_t^{-1}(-\mathbf{x}_j)\right)\Big) \\
&= \mathcal{L}(\mathbf{X})
\end{aligned} \tag{5.2}$$

Since $\mathcal{L}(\mathbf{X})$ is even, any local maximum $\hat{\mathbf{X}}$ will be repeated for $-\hat{\mathbf{X}}$. In addition to this, the covariance matrices can not distinguish between $\hat{\mathbf{X}}$ or $\hat{\mathbf{X}}+c$, where $c$ is some constant offset, due to the isotropic covariance kernel. This means that we are just as likely to converge to an estimate that is upside down as not, and also that the offset $c$ can not be inferred. Figure 5.3 shows how different random initial positions may lead to estimates that are either upside down or correctly aligned. In this case, the offset has been found by comparing with the true $\mathbf{X}$. Observe also how the estimate is less accurate near the boundaries of the domain due to the lower neural density there.

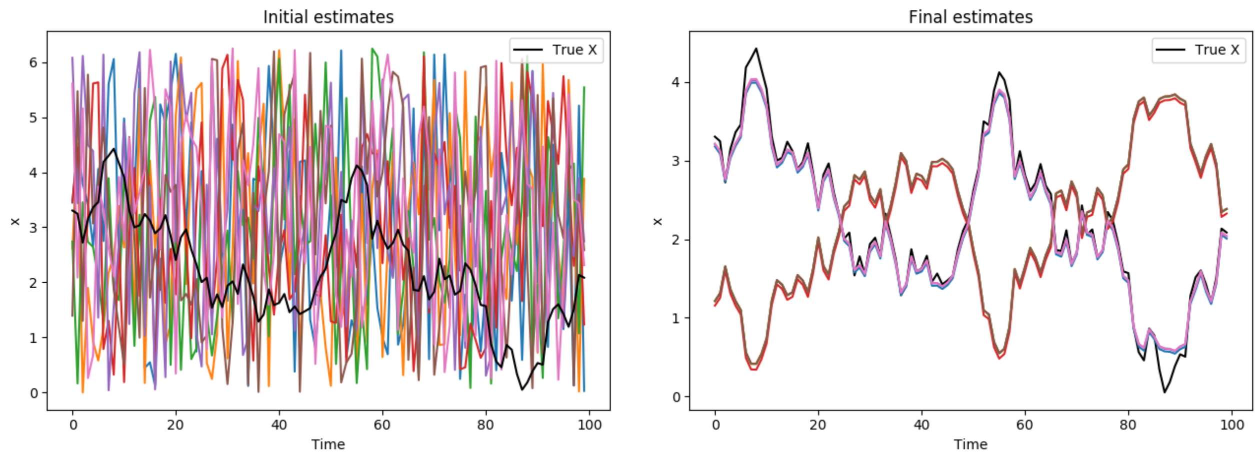


**Figure 5.3:** Left: 7 random initial estimates, where the value at every time point $\mathbf{x}_t$ is sampled independently from a uniform distribution in the range from 0 to the upper limit $2\pi$ (not implying periodicity in this example). Right: Final estimates.

### 5.1.2 Scaling

Analytically, the $\mathcal{L}(\mathbf{X})$ function should be able to distinguish between differently scaled versions of $\mathbf{X}$. A scaled-up estimate of $\mathbf{X}$ means that the off-diagonal entries in $K_x$ will, on average, have lower values, since the $\mathbf{x}$ values are further apart. Still, to be able to find the correct scaling of $\mathbf{X}$, the correct hyperparameters for the covariance kernel must be known precisely, and there would still be no guarantee that the algorithm would converge to a correctly scaled estimate due to the non-concavity of $\mathcal{L}(\mathbf{X})$.

Figure 5.4 shows an example where the estimate converges to an estimate of $\mathbf{X}$ that has the right shape, but the wrong scaling, and Figure 5.5 shows the same estimate after rescaling.

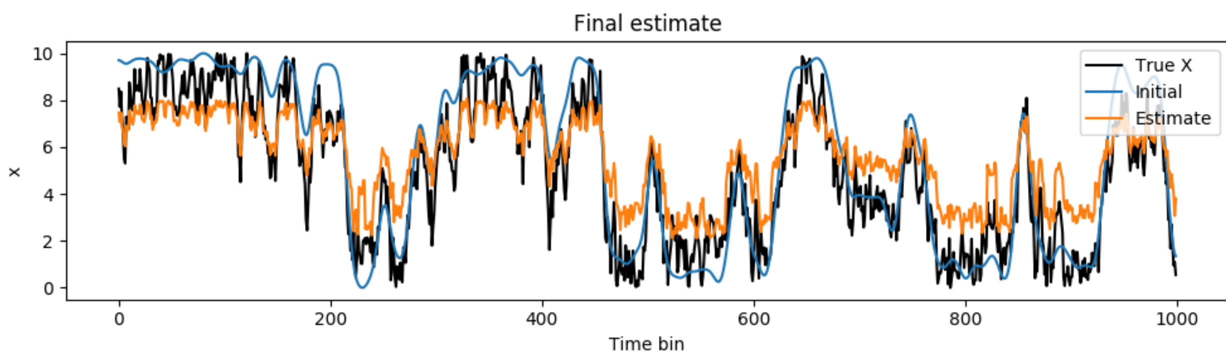


**Figure 5.4:** An example where the final estimate is wrongly scaled. The correct scaling can be found by comparing to the true latent variable.

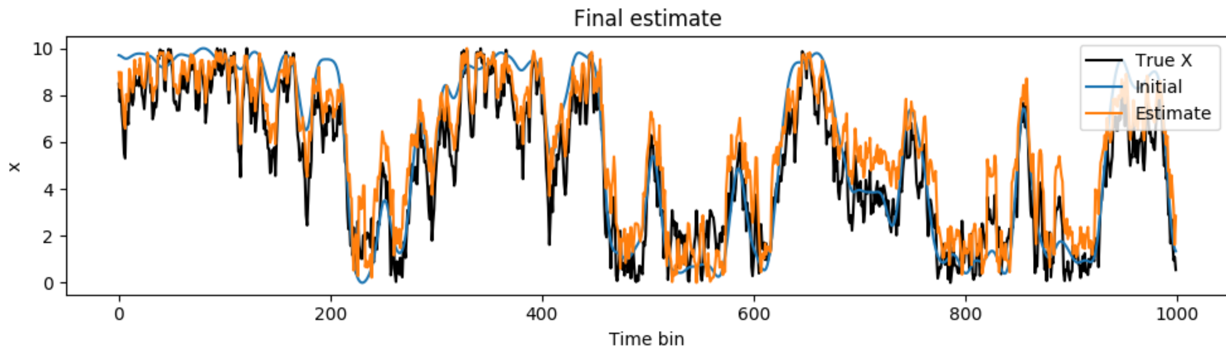


**Figure 5.5:** The same experiment as in Figure 5.4 except the estimate has been rescaled after convergence to fit the range of the true latent variable.

In this test case, the neural density is lower near the boundaries of the domain, making it harder to infer the latent variable near the border. Still, in the supervised learning case where there is a known latent variable to compare the estimate with, the flipping, offset, and scaling can be determined by comparing to the true **X**. In an exploratory setting where the latent variable was unknown, the shape and dynamics of the inferred latent variable could suggest variables to which the population might be responding. Those variables could be used to adjust the flipping, offset, and scaling of the estimate. In the remainder of the chapter, we will find the correct rotation, offset, and scaling for the estimate by comparing the final estimate with the true path. This is done before the RMSE value is calculated.

### 5.1.3 Partly flipped estimates

In addition to the issues described above, the estimate may converge to a local maximum where the estimate is partly upside down, which is more problematic since this can not be handled by flipping, scaling, or adding an offset. In Figure 5.6, the number of initial estimates has been increased from 7 to 20, and estimates that ended up upside down have been flipped by comparing to the true path. However, one of the random initial starts has converged to an estimate that appears to be partly upside down (light blue). It appears like the estimate is correctly aligned for all values above 3, but upside down for values below 3. Partial flipping is a problem that can not be avoided entirely.

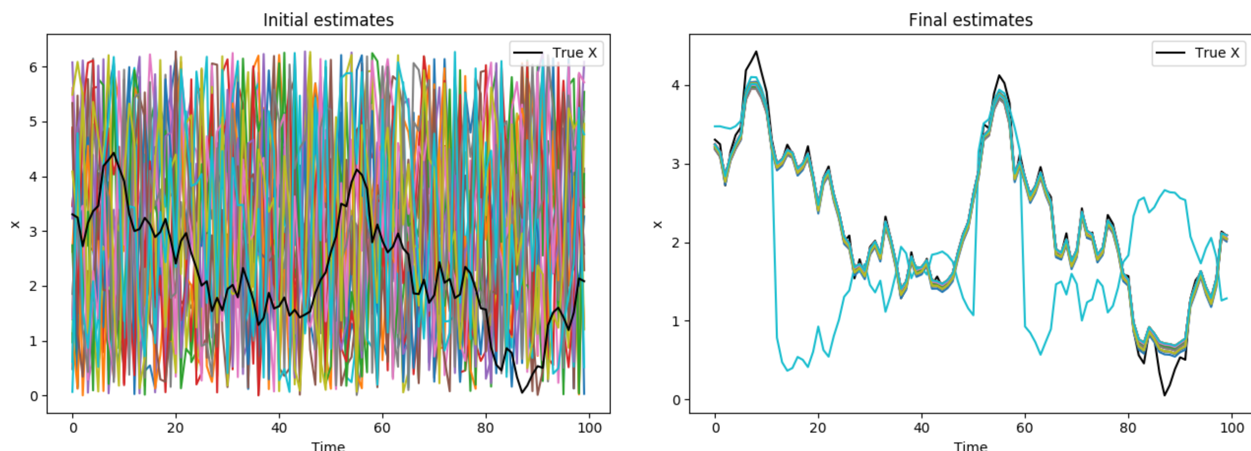


**Figure 5.6:** Left: 20 random initial estimates for **X**. Right: Final estimates corrected for flipping.

### 5.1.4 Placement of the inducing grid

Finding the optimal position for and number of the inducing points is an interesting problem by itself (see Titsias (2009) for a variational inference approach). For simplicity, we choose a uniformly spaced grid. Choosing the range of the inducing grid requires knowledge of the domain and range of $\mathbf{X}$. If the range of $\mathbf{X}$ is unknown, one approach is to start with a wide range for the inducing points and then set the range equal to the range of the $\mathbf{X}$ estimate at every iteration as it changes. In this chapter, the range of the inducing grid was set equal to the range of the domain, $[0, 10]$.

## 5.2 Initialization

### 5.2.1 Initial estimate for F

Since the log posterior of $\mathbf{F}$ is a concave function, finding an initial estimate of $\mathbf{F}$ would be irrelevant if the true $\mathbf{X}$ were known, since the algorithm would converge to the global maximum no matter the initialization. However, if we start with an unfortunate estimate for $\mathbf{F}$, we may converge to a suboptimal local maximum for $\mathbf{X}$, which will, in turn, affect the estimate of $\mathbf{F}$. The observed spike matrix $\mathbf{Y}$ can be used to find a good starting point for $\mathbf{F}$. With a Poisson likelihood model, $f_{i,t} = \log \lambda_{i,t} = \log E[y_{i,t}]$. Based on this, a suggestion for the initial $\mathbf{F}$ could be

$$\mathbf{F}_{\text{initial}} = \log(\mathbf{Y} + \epsilon) \tag{5.3}$$

where some $\epsilon < 0$ is introduced to be able to take the logarithm when $y_{i,t} = 0$. Another candidate is the Anscombe transform (Anscombe, 1948), that has been used to transform Poisson distributed data to approximately normally distributed:

$$\mathbf{F}_{\text{initial}} = 2\sqrt{\mathbf{Y} + \frac{3}{8}} \tag{5.4}$$

However, after trying to find a transformation of the $\mathbf{Y}$ matrix that made the distribution of the transformed values look like the true synthetic log firing rates, the following initialization was found to produced the best results:

$$\mathbf{F}_{\text{initial}} = \sqrt{\mathbf{Y}} - \frac{\max(\sqrt{\mathbf{Y}})}{2} \tag{5.5}$$

The usefulness of the square root transformation for Poisson data before modeling it with a Gaussian process has been described before, e.g., by Byron et al. (2009). In addition to this, the transformation in equation (5.5) moves the mean of the data closer to zero. Figures 5.7 and 5.8 shows a heat map of the true $\mathbf{F}$ compared to the initial $\mathbf{F}_{\text{initial}}$ for 100 simulated neurons.

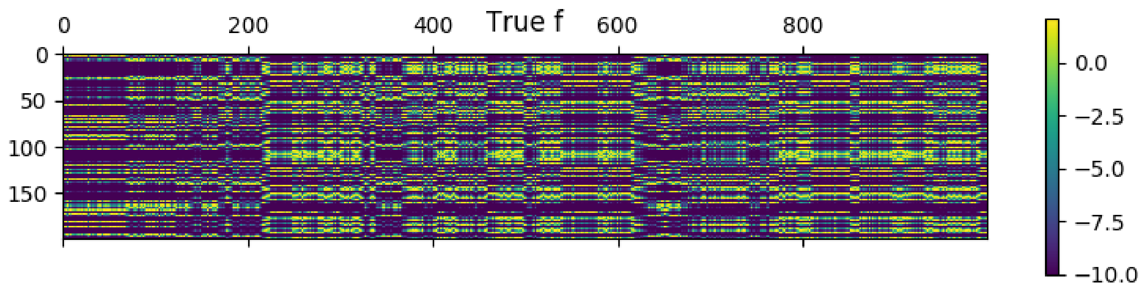


**Figure 5.7:** True $\mathbf{F}$ values. The y axis shows the index of the neurons, and the x axis shows the index of the timebins.

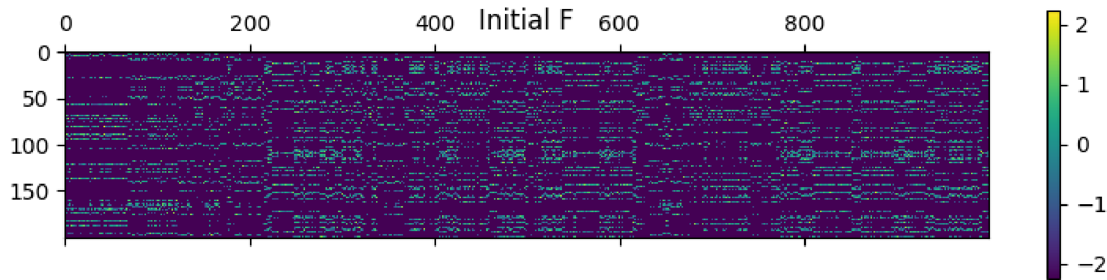


**Figure 5.8:** Initial $\mathbf{F}$ with the square root initialization (eq. (5.5)). The y axis shows the index of the neurons, and the x axis shows the index of the timebins.

For visualization, it may be better to find the posterior mean of the tuning curve on a uniform grid in the domain of $\mathbf{X}$, given the true path of $\mathbf{X}$. Figure 5.9 shows the mean of the posterior distribution of $\mathbf{f}_i(\mathbf{X})$ on the grid given the square root initialization. Figure 5.10 shows the same posterior mean as in eq. (3.14) given the logarithm initialization.

Both initializations seem to capture the location of the peak, but the square root initialization is better in terms of finding the width of the bump, while the log is better at capturing the height of the peak. Exactly how this choice of initial $\mathbf{F}$ affects the $\mathcal{L}(\mathbf{X})$ function is not obvious, but it is clear that larger values in $\mathbf{F}$ make the quadratic term more important relative to the logdet and xprior terms. Another explanation is that more pronounced tuning curves like the ones in Figure 5.10 cause local maxima in $\mathcal{L}(\mathbf{X})$ to become more pronounced, limiting the region that the $\mathbf{X}$ estimate is able to explore in the first iteration.

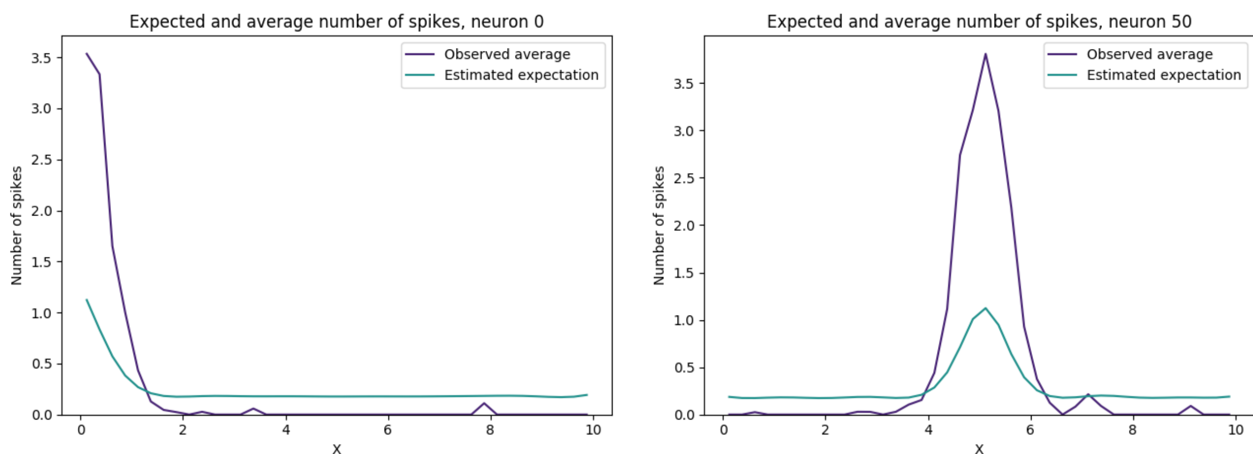


**Figure 5.9:** The posterior mean of the tuning curve on a grid of points, obtained using the square root initialization.

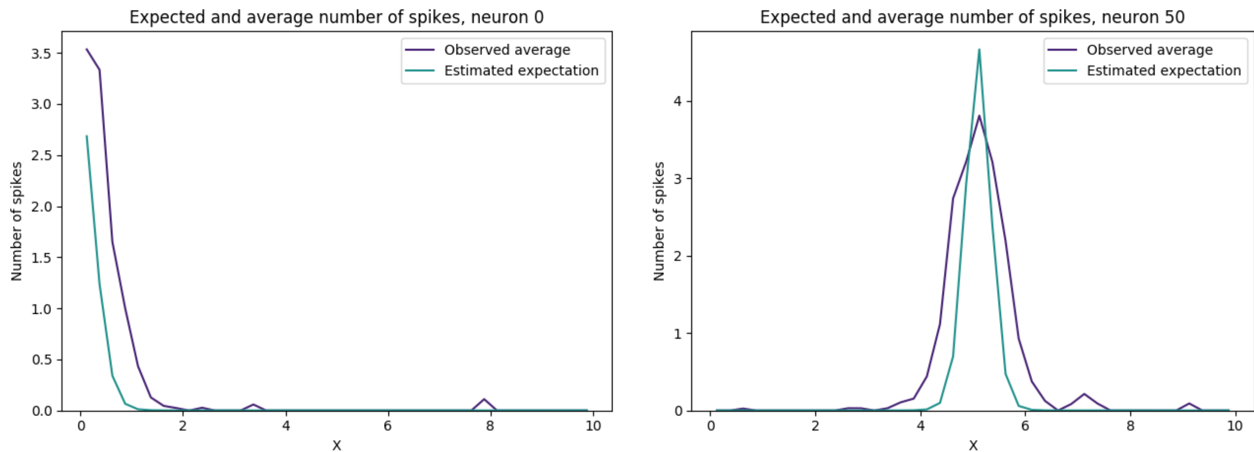


**Figure 5.10:** The posterior mean of the tuning curve on a grid of points, obtained using the logarithm initialization.

Note that in order to find a good posterior estimate of $\mathbf{F}$, it is essential that the final $\mathbf{X}$ estimate is reasonable. To illustrate this, we provide the initial estimate for $\mathbf{F}$ described in eq. (5.5), but instead of using the true path to find the posterior of the tuning curves, we provide it with random noise as the estimate of $\mathbf{X}$. Figure 5.11 shows that the posterior is useless in this case. The lesson is that the posterior estimate of the tuning curve depends heavily on the estimate of $\mathbf{X}$.

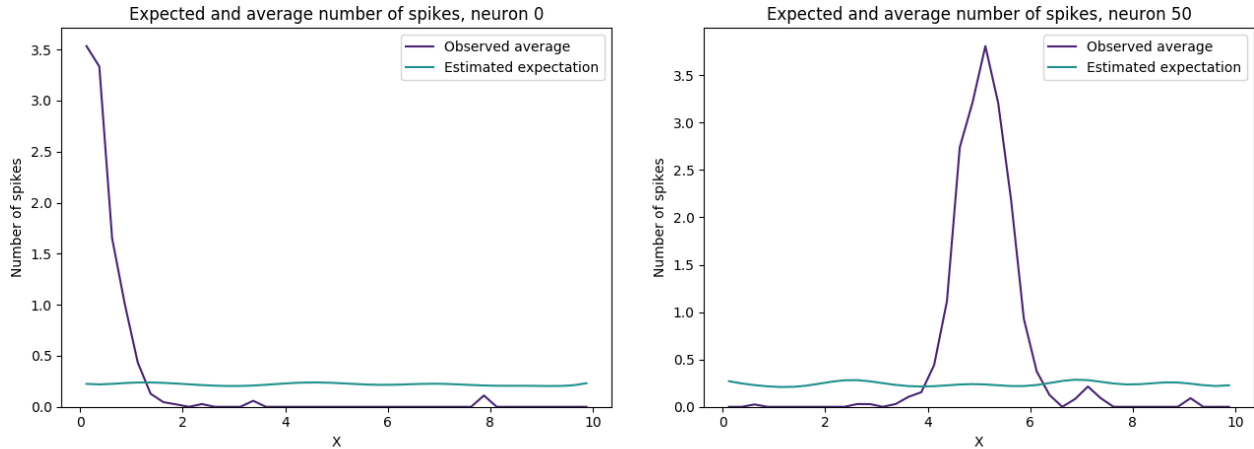


**Figure 5.11:** Posterior estimate of $\mathbf{f}_i(\mathbf{X})$ based on an estimate of $\mathbf{X}$ that is just randm noise.

### 5.2.2 Initial estimate for X

The initial estimate of $\mathbf{X}$ is important since $\mathcal{L}(\mathbf{X})$ is not a concave function. Several candidates were examined, including flat lines, random paths generated from the generative model, and principal component analysis applied to the spike matrix $\mathbf{Y}$. First, the observed spike matrix was smoothed with a Gaussian filter independently per neuron, then PCA was applied, and the PCA estimate was rescaled to match the domain of $\mathbf{X}$. An example PCA initialization with standard deviation 15 in the Gaussian filter is shown together with the true path of $\mathbf{X}$ in Figure 5.12.

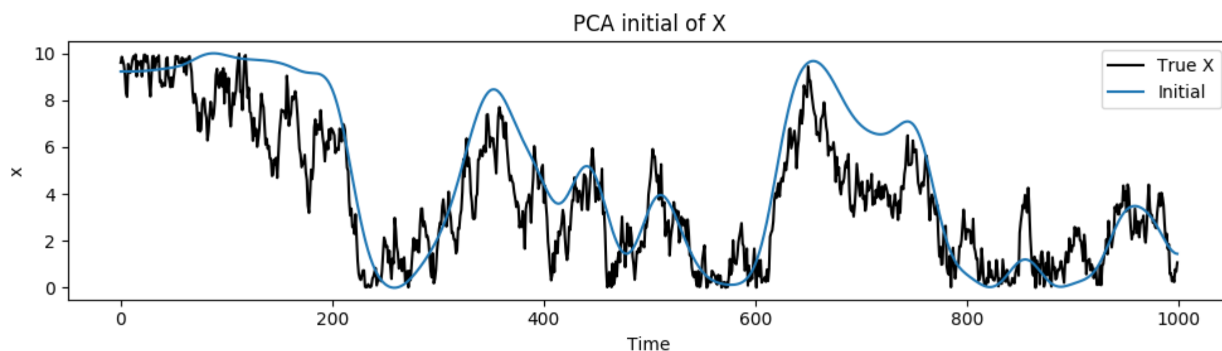


**Figure 5.12:** The first principal component shown together with the true simulated path.

Figures 5.13 and 5.14 show an example of how the performance of the algorithm varies with the initialization, with all other parameters kept equal. In this case, the estimate of $\mathbf{X}$ based on the PCA initialization has an RMSE value of 0.222, while the $\mathbf{X}$ estimate found by starting with a flat line has an RMSE value of 2.831. However, in other cases, the flat initialization provided a better estimate than the PCA initialization, so there seems to be no initialization that is best in general. The point here is to visualize the possible impact of the initialization. For this simulated data, the final estimate is impressively accurate. However, for recorded neural data, the neurons may be tuned to other things in addition to the head direction, making it harder to infer the head direction. In the next section, we will evaluate how the tuning strength and data length influence the quality of the estimate.

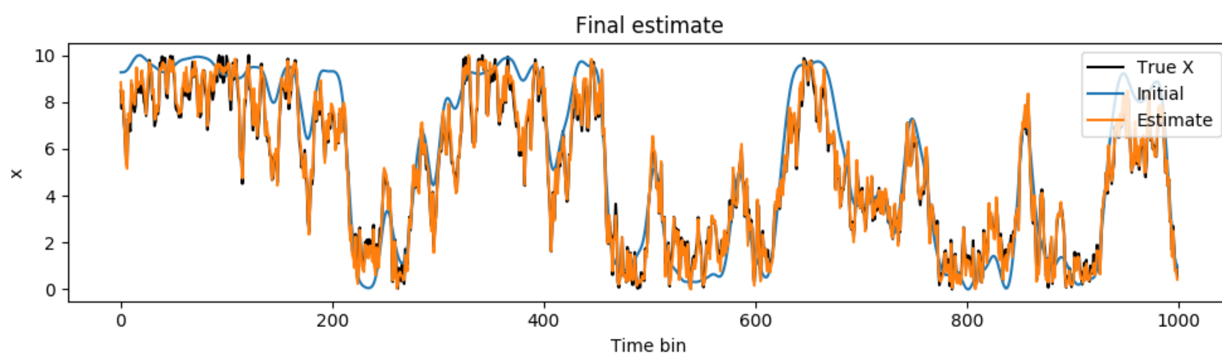


**Figure 5.13:** Final estimate for a PCA initialization of $\mathbf{X}$.

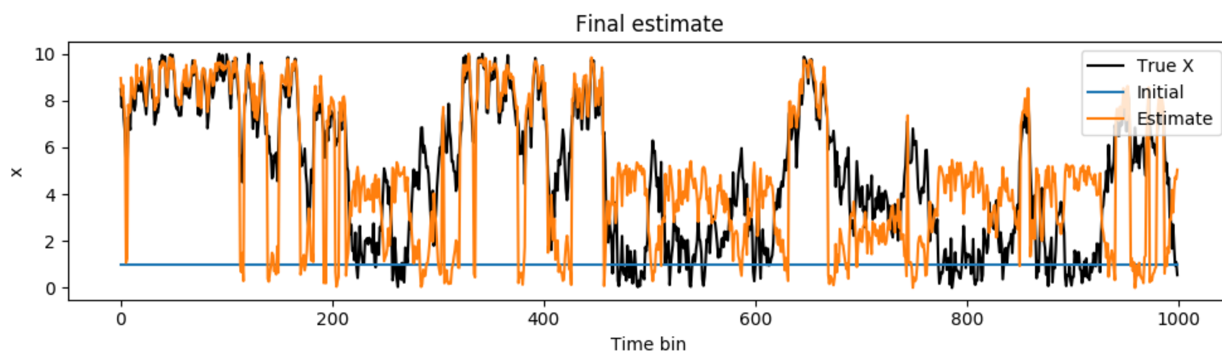


**Figure 5.14:** Final estimate for a flat initialization of $\mathbf{X}$.

## 5.3 Robustness evaluation

We will take a closer look at two of the many factors that influence the latent variable inference: the tuning strength $\lambda$ and the number of observed time bins $T$. The tuning strength determines how much the firing rate is affected by the position of the latent variable. Intuitively, one would think that a higher tuning strength should improve the inference since it should be easier to detect more substantial changes in the firing rate. Also, one would expect that observing more time bins should lead to a better estimate since more data is available.

Recall that the tuning curves are defined as Gaussian bumps and that the tuning strength is defined as the firing rate at the peak of the bump. The background noise level will be the firing rate in the limit where the distance between $\mathbf{x}$ and the peak increases towards infinity. The loss function we will use to describe how well the inference works is the root mean squared error (RMSE) between the estimated $\mathbf{X}$ and the true path of $\mathbf{X}$, defined in eq. (5.1). We set the background tuning strength to 0.5, and choose an array of 21 tuning strengths:

$$\lambda = \begin{bmatrix} 0.51 & 0.6 & 0.7 & 0.8 & 0.9 & 1.0 & 1.25 \\ 1.5 & 1.75 & 2.0 & 2.25 & 2.5 & 3.0 & 3.5 \\ 4.0 & 4.5 & 5.5 & 6.5 & 7.5 & 8.5 & 9.5 \end{bmatrix} \tag{5.6}$$

We select six different $T$ values, $T = (200, 500, 1000, 2000, 3000, 5000)$, and for each of these we construct 20 simulated paths for $\mathbf{X}$ of length $T$. Then, for every combination of $T$ and tuning strength $\lambda$, we initialize the estimate of $\mathbf{X}$ using PCA with a smoothing filter standard deviation of 5 and let the algorithm converge independently for every path in the array of 20 test cases. We can then calculate the average RMSE across the 20 paths, for all combinations of $\lambda$ and $T$. Figure 5.15 shows the average RMSE values plotted as a function of the tuning strength for different $T$ values.

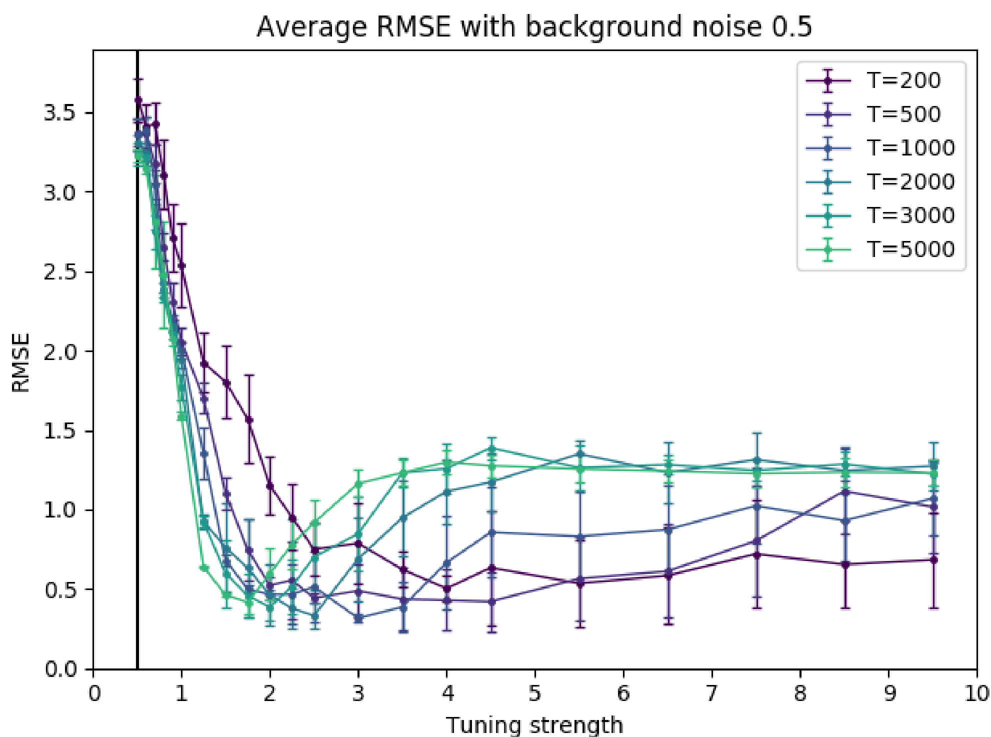


**Figure 5.15:** Mean RMSE across 20 individual random paths for each $T$ value, with 95 % confidence intervals calculated from a t-distribution with 19 degrees of freedom. The background firing rate of 0.5 is shown with a black vertical line.

Surprisingly, for every value of $T$, there appears to be an optimal tuning strength beyond which increasing the tuning strength worsens the RMSE value. We see that for very low tuning strengths, the RMSE value is high because the neurons are effectively not tuned to the latent variable. As the tuning strength increases towards the optimal tuning strength, the RMSE value lowers as the tuning becomes more pronounced, allowing the algorithm to infer the latent variable. The curve drops faster for higher $T$ values since the inference performs better when more data is available. Beyond the optimal point, however, the RMSE increases as the tuning strength increases.

Similar observations have been made by Davidovich et al. (2020) in the reconstruction of the hidden node problem. We have established that there exist suboptimal local maxima the algorithm may converge to. An explanation for the existence of an optimal tuning strength may be that the local maxima become more pronounced as the tuning strength increases. Combined with the fact that a higher number of time bins provides more opportunities for such local maxima to occur, this may explain why the optimal tuning strength occurs at a lower value for higher $T$ values.

### 5.3.1 Choosing between final estimates

For every individual iteration in Figure 5.15, the initial estimate was found by applying PCA to the observed spike matrix $\mathbf{Y}$ after smoothing it with a Gaussian filter with standard deviation 5. By varying the smoothing width in the Gaussian filter, several reasonable initial estimates with varying degrees of smoothness can be obtained. By re-running all the iterations of Figure 5.15 with standard deviations in the smoothing filter equal to 3, 5, and 10, we find the RMSE values shown in Figure 5.16. From this simulation, it seems like a width of 5 is preferred for tuning strengths lower than 3, while for higher tuning strengths, a width of 3 may be marginally better.

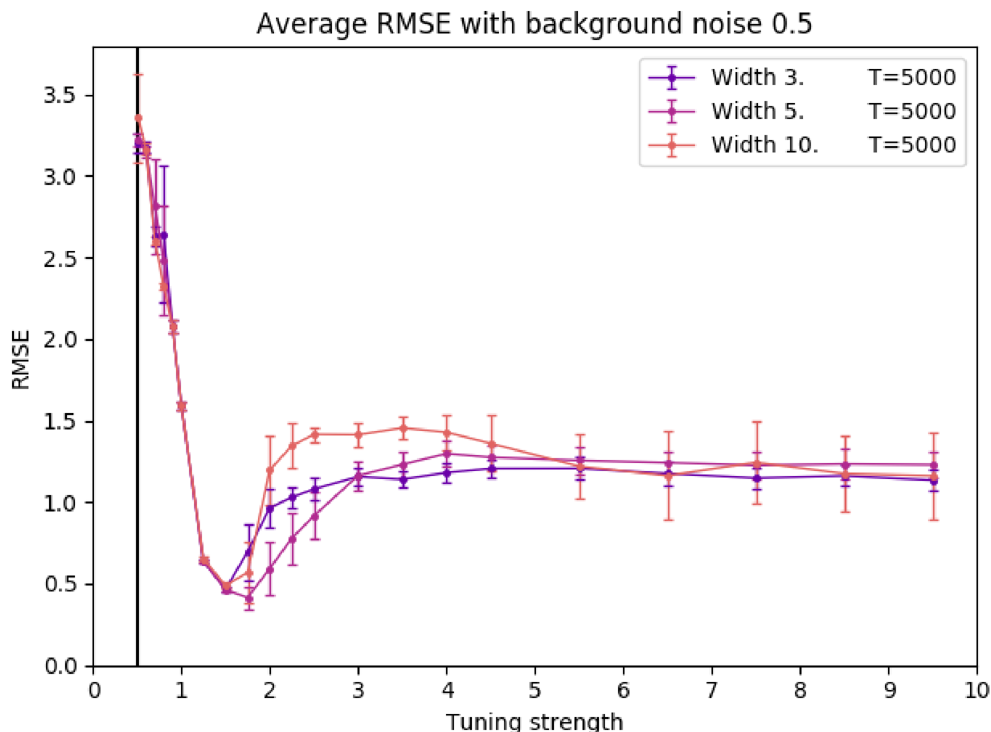


**Figure 5.16:** For each tuning strength $\lambda$, 20 random paths of length $T = 5000$ were generated. For each path, three different initializations were generated by using three different widths for the smoothing filter applied to $\mathbf{Y}$ before applying PCA. This plot shows the average RMSE value for each tuning strength for the three standard deviations. 95 % confidence intervals are shown as bars.

We mentioned in the introduction to section 5.1 that if we wish to start an ensemble of differently initialized iterations for a specific problem, then after convergence we must choose the best estimate among them by comparing their $\mathcal{L}(\mathbf{X})$ scores, since the true $\mathbf{X}$ can not be used for comparison when it comes to real data. To test whether the $\mathcal{L}(\mathbf{X})$ metric can differentiate between better and worse estimates, we calculate the average RMSE value that results from using the $\mathcal{L}(\mathbf{X})$ value to pick between estimates. Figure 5.17 shows these values compared to the RMSE values that result from only using a smoothing filter width of 5. For reference, we include the RMSE values where the final estimate has been picked based on comparing their actual RMSE values, which represents the optimal choice.

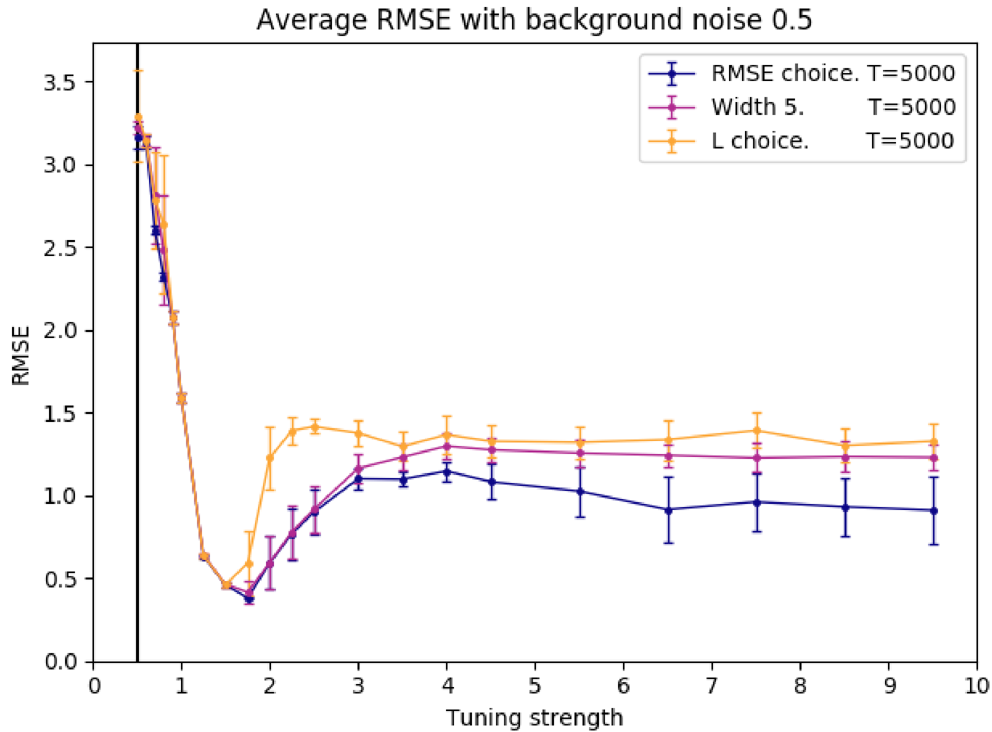


**Figure 5.17:** Comparison of average RMSE values resulting from consistently picking the estimate with smoothing filter width 5; the estimate with the highest $\mathcal{L}(\mathbf{X})$ value; or the estimate with the lowest RMSE value, respectively. The plot shows the mean RMSE values over the 20 paths. 95 % confidence intervals are shown as bars.

It seems like using the $\mathcal{L}(\mathbf{X})$ value to pick between estimates is worse than just sticking with one estimate based on a smoothing filter width of 5. The results for other time lengths than $T = 5000$ were similar. For the remainder of the chapter, we will stick to a smoothing filter standard deviation of 5.

## 5.4 Application to head direction data

We applied the model to the head direction cell data described in Chapter 2, recorded by Peyrache et al. (2015). We will assume that head direction is the only variable driving the recorded population. To evaluate the final estimates, we use the RMSE described in equation 5.1, where we let the observed head direction take the role of the “true” $\mathbf{X}$. Since it is known that head direction is a latent variable driving these neurons, it makes sense to look for a latent variable with a one-dimensional, $2\pi$-periodic domain. Compared to the nonperiodic example from earlier, the only change in the model is treating the domain of $\mathbf{X}$ as periodic with period $2\pi$. This is reflected in the entries of the covaraince matrix $K_{\mathbf{u},\mathbf{u}}$. For periodic data, the jitter term added to the diagonal of $K_{\mathbf{u},\mathbf{u}}$ had to be increased to $10^{-3}$ for the posterior covariance matrix of the tuning curves to still be positive definite. We must also expect that the head direction behaves periodically by wrapping around the border. An interval of 5000 time bins with bin width 25.6 ms was chosen for the analysis. The observed head direction for this time interval is shown in Figure 5.18.

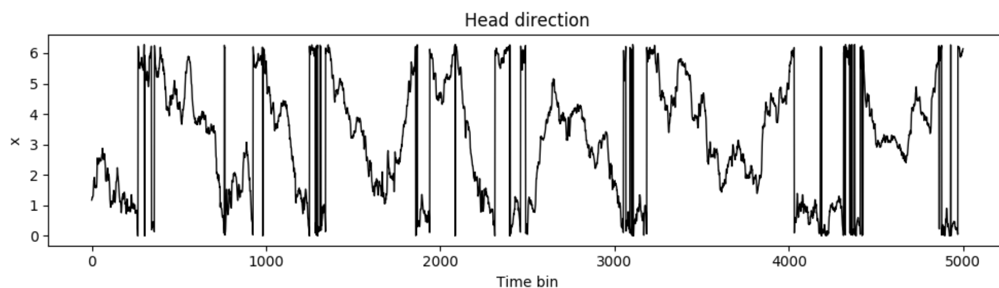


**Figure 5.18:** Observed head direction.

For this period, 16 neurons appeared to be active and tuned to the head direction. The tuning of these neurons for the selected interval is shown in Figure 5.19.

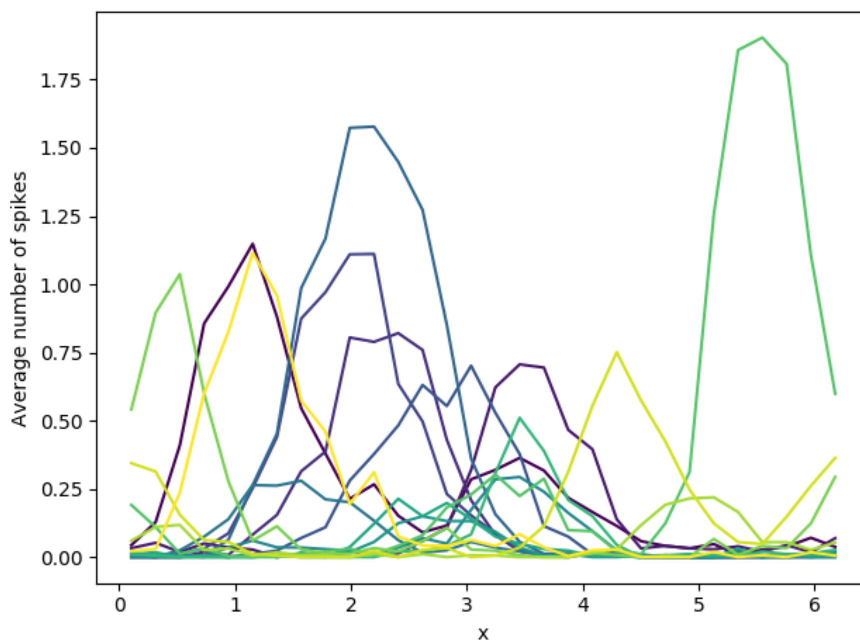


**Figure 5.19:** Observed firing rates (in number of spikes per bin) for the 16 neurons. The bin size is 25.6 ms.

From the observed firing rates, we see that there are neurons that are tuned to every specific bit of the domain, but 16 is considerably less than the 100 simulated neurons from the simulated examples. Table 5.1 shows five different ways in which the algorithm can be initialized. The following pages show the resulting final estimates of $\mathbf{X}$ and $\mathbf{f}_i(\mathbf{X})$ for these initializations. A comparison of their RMSE values and $\mathcal{L}(\mathbf{X})$ values follows at the end of the chapter. As a reference point, we can initialize the algorithm at the true values of $\mathbf{X}$ and see if the estimated path deviates from the true one. We do not have access to any true $\mathbf{F}$ values, only the observed spike matrix $\mathbf{Y}$, but the MAP estimate of $\mathbf{F}$ conditioned on the true values of $\mathbf{X}$ is a very good estimate. This will be referred to as the "optimal estimate" of $\mathbf{F}$.

| Initialization | True $\mathbf{X}$ | Optimal $\mathbf{F}$ |
|---|---|---|
| (1) True $\mathbf{X}$ and optimal $\mathbf{F}$ | Yes | Yes |
| (2) True $\mathbf{X}$ and estimated $\mathbf{F}$ | Yes | - |
| (3) PCA initialization of $\mathbf{X}$ and optimal $\mathbf{F}$ | - | Yes |
| (4) PCA initialization of $\mathbf{X}$ and estimated $\mathbf{F}$ | - | - |
| (5) Flat initialization of $\mathbf{X}$ and estimated $\mathbf{F}$ | - | - |

**Table 5.1:** Comparing the different initializations.

### 5.4.1 Initialization 1: True X and optimal F

Figure 5.20 shows the final estimate obtained when using the true $\mathbf{X}$ and optimal $\mathbf{F}$ as initial estimates. The RMSE of this estimate of $\mathbf{X}$ is 0.520. For comparison, the RMSE of an entirely random estimate is 2.569. Figure 5.21 shows the inferred tuning curves. The estimate stays in place when initialized with these values for $\mathbf{F}$ and $\mathbf{X}$, and the position of the tuning curves are reconstructed well. The shapes of the tuning curves are reconstructed fairly well except for neuron 2, which appears too narrow.

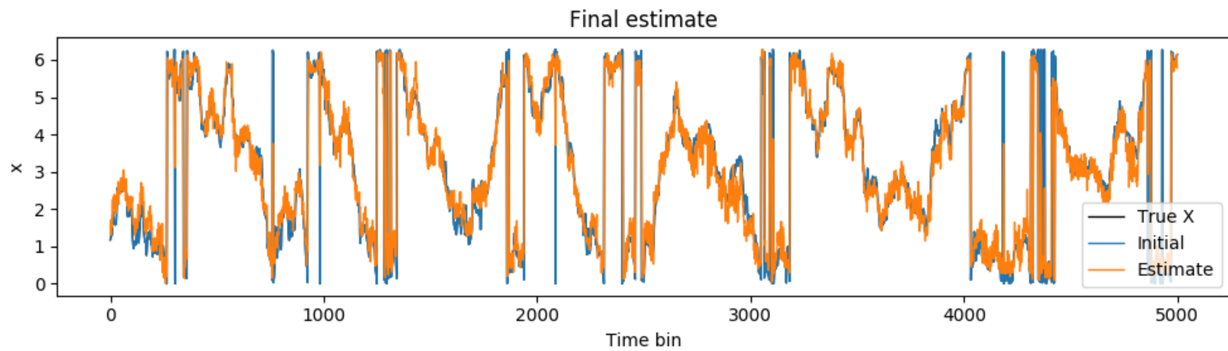


**Figure 5.20:** Final estimate of $\mathbf{X}$ compared to initialization and true path.

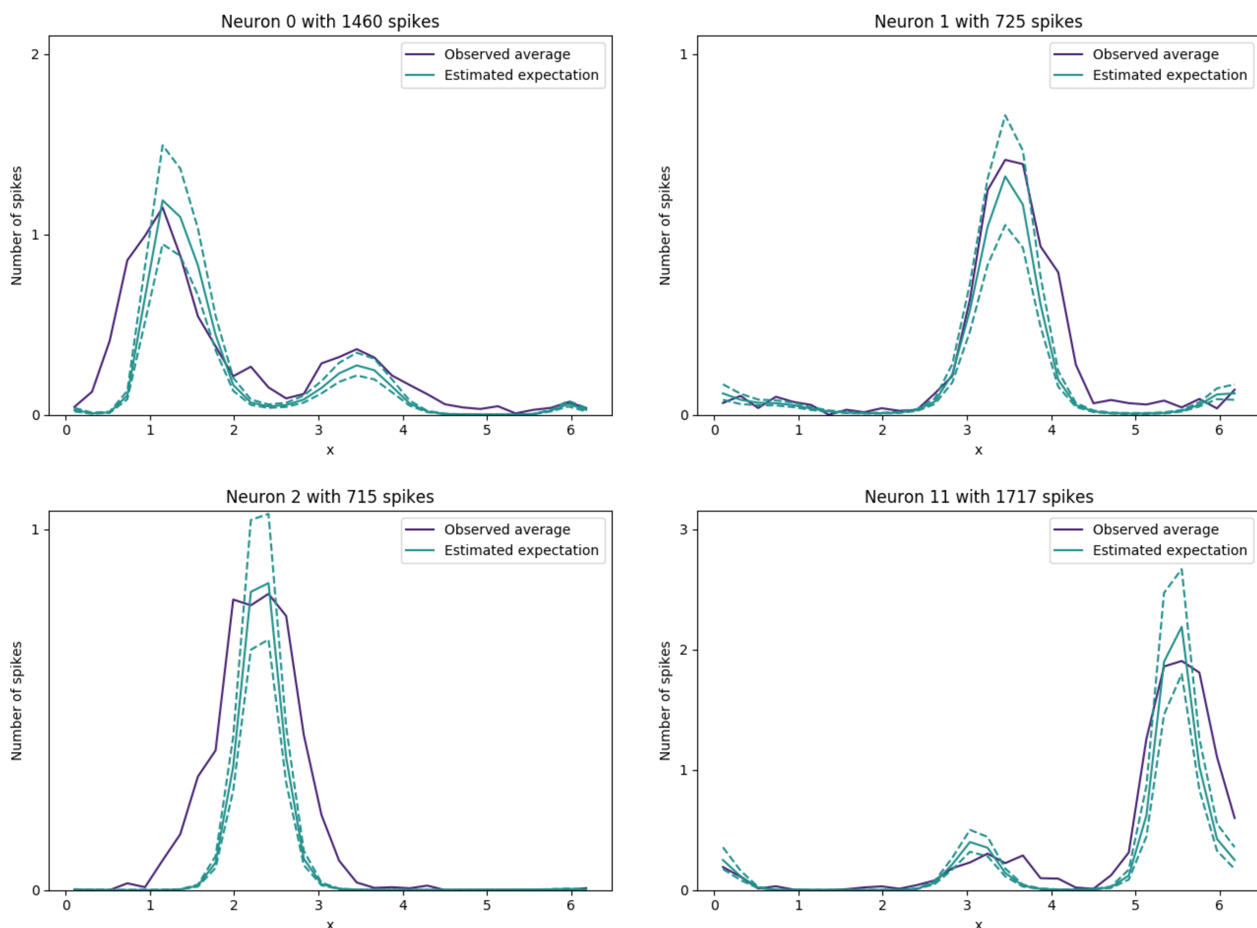


**Figure 5.21:** Posterior estimates of $\mathbf{f}_i(\mathbf{X})$ based on final estimates of $\mathbf{F}$ and $\mathbf{X}$. 95 % credible intervals are shown with dotted lines.

### 5.4.2 Initialization 2: True $\mathbf{X}$ and estimated $\mathbf{F}$

Figure 5.22 shows the final estimate when starting at the true $\mathbf{X}$ and the initial estimate of $\mathbf{F}$ estimate described in equation (5.5). The RMSE of this estimate is 1.499. The estimate of $\mathbf{X}$ appears shifted away from the optimal starting point. Since the first update of $\mathbf{F}$ is skipped, the $\mathbf{X}$ estimate is shifted away from the truth to match the provided $\mathbf{F}$ estimate in the first iteration. The estimated tuning curves in Figure 5.23 are shifted due to the shifting of $\mathbf{X}$. Here, the initial $\mathbf{F}$ estimate in equation (5.3) may have been better, as this estimate was closer in shape to the true tuning curves.

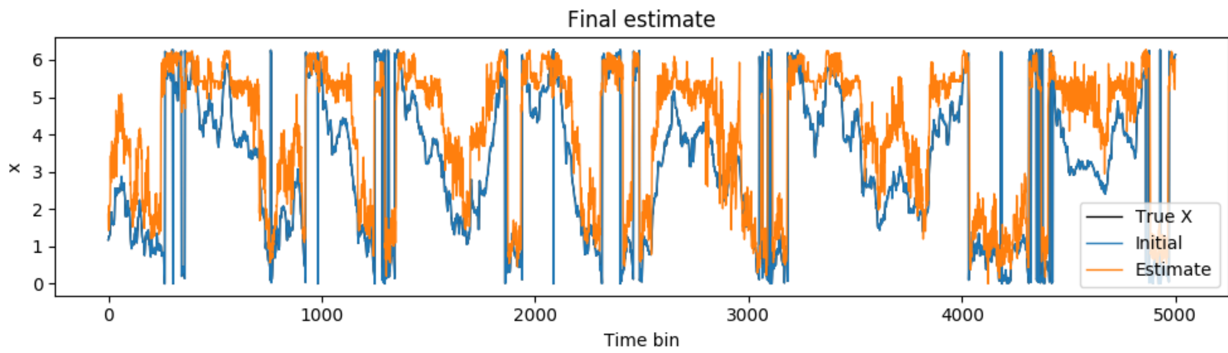


**Figure 5.22:** Final estimate of $\mathbf{X}$ compared to initialization and true path.

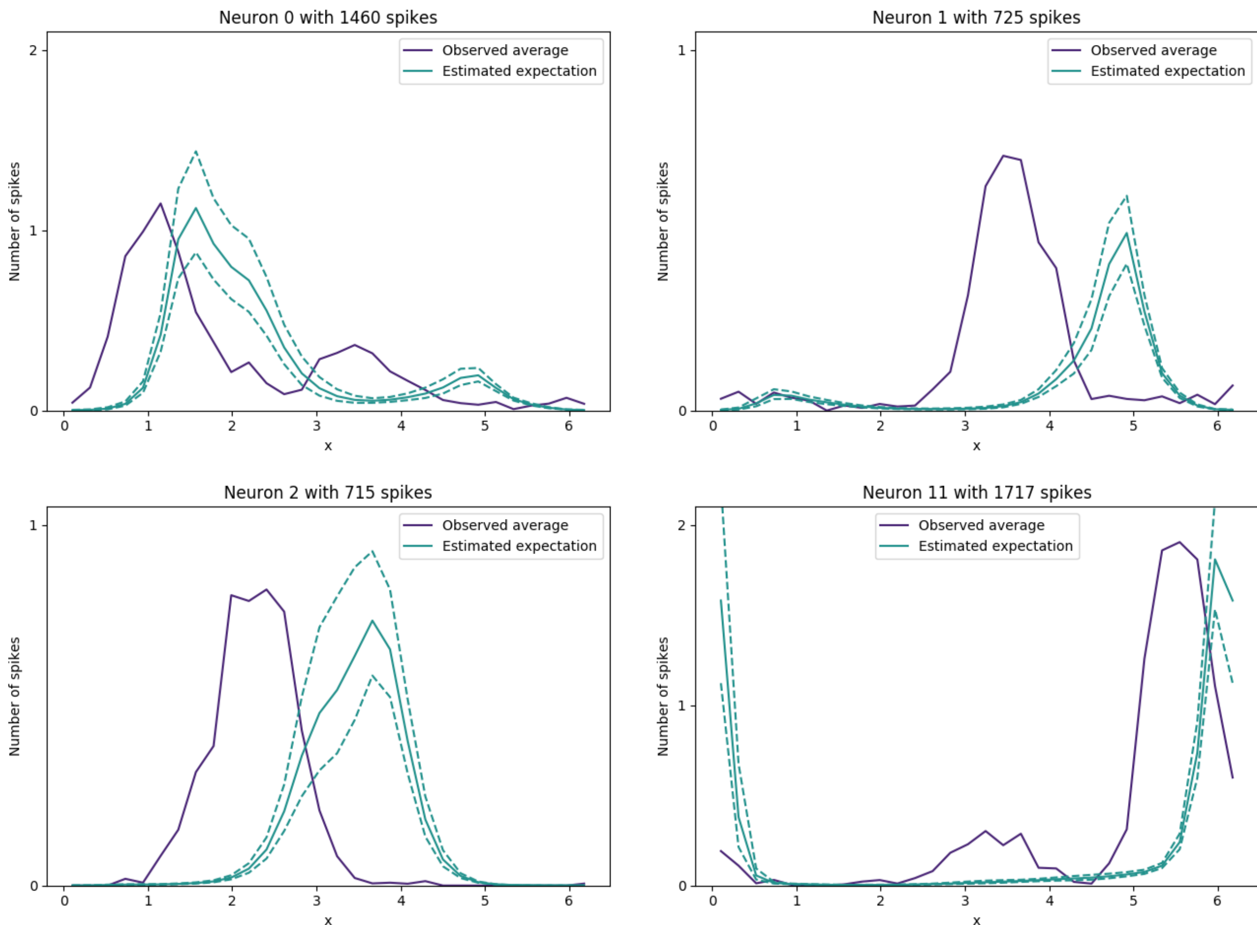


**Figure 5.23:** Posterior estimates of $\mathbf{f}_i(\mathbf{X})$ based on final estimates of $\mathbf{F}$ and $\mathbf{X}$. 95 % credible intervals are shown with dotted lines.

### 5.4.3 Initialization 3: PCA initialization of X and optimal F

Figure 5.24 shows the final estimate obtained when the initial $\mathbf{X}$ is found using PCA and the initial $\mathbf{F}$ is the MAP estimate of $\mathbf{F}$ conditioned on the true $\mathbf{X}$. The RMSE of this estimate is 1.857, the worst so far after the random estimate. The PCA initialization does not capture points where $\mathbf{X}$ wraps around from $2\pi$ to zero, like after time bin 4000. Since the model is unable to correct the PCA's biggest mistakes, this is reflected in the final estimate. The tuning curves in Figure 5.25 are affected by the misplaced $\mathbf{X}$ estimate, which makes the tuning curves appear misplaced and with several peaks.

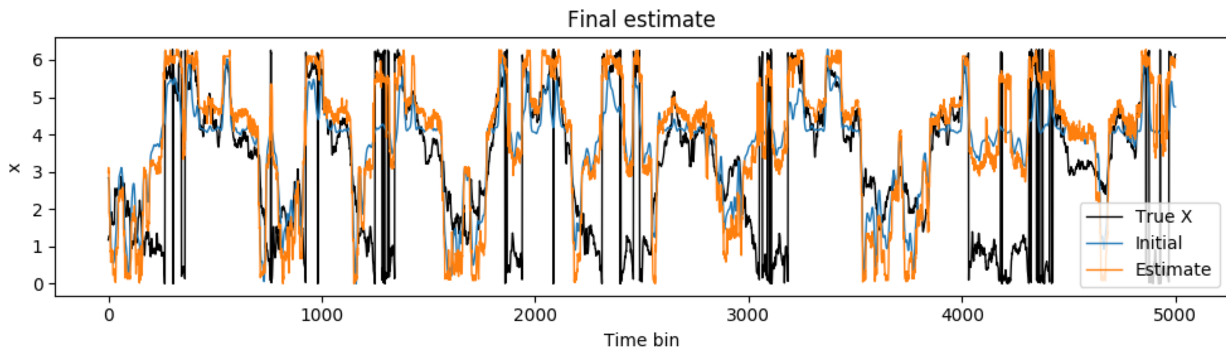


**Figure 5.24:** Final estimate of $\mathbf{X}$ compared to initialization and true path.

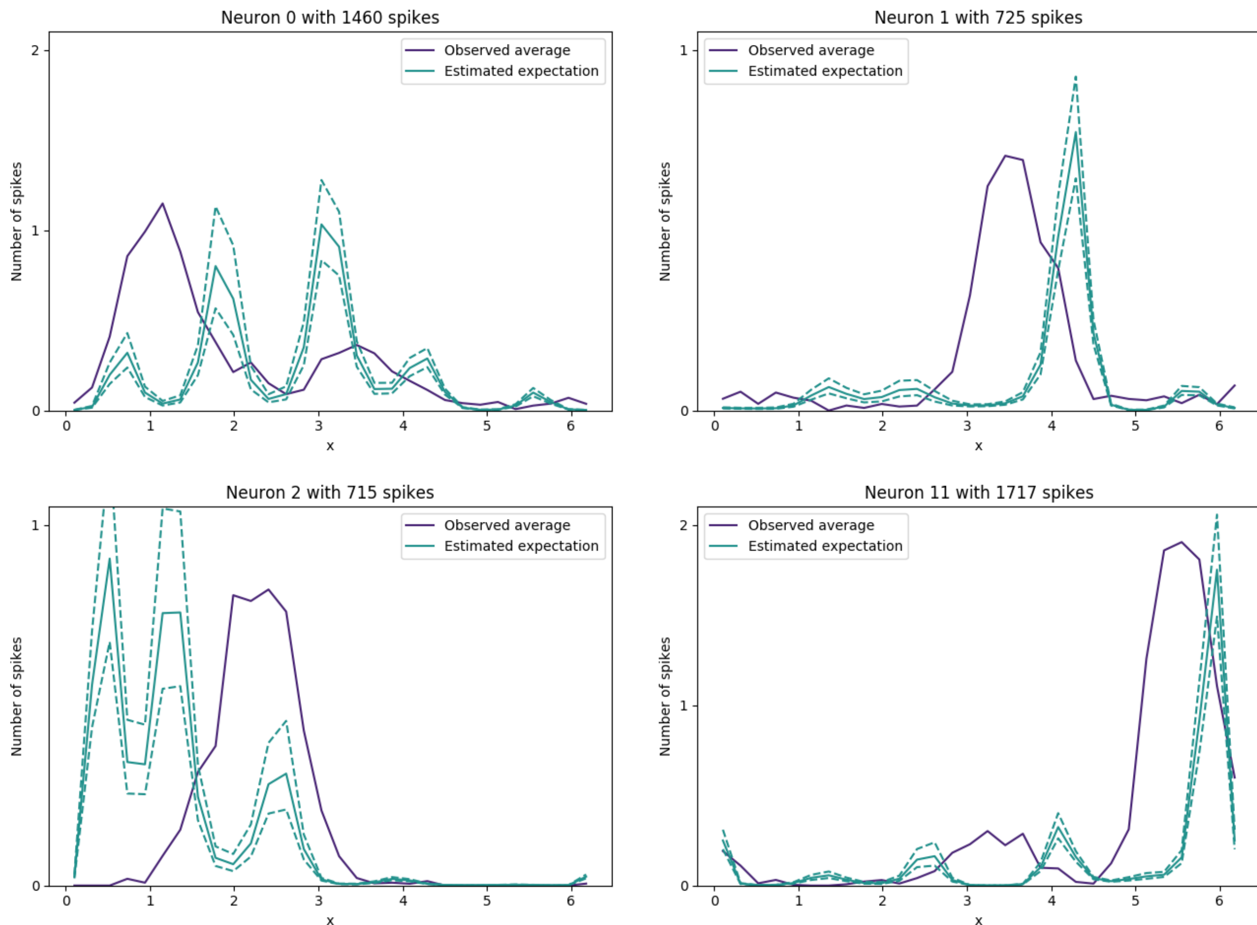


**Figure 5.25:** Posterior estimates of $\mathbf{f}_i(\mathbf{X})$ based on final estimates of $\mathbf{F}$ and $\mathbf{X}$. 95 % credible intervals are shown with dotted lines.

### 5.4.4 Initialization 4: PCA initialization of X and estimated F

Figure 5.26 shows the final estimate obtained when PCA is used to find the initial estimate of $\mathbf{X}$ and $\mathbf{F}$ is initialized as described in equation (5.5). The RMSE of this estimate is 1.541. As in the previous example, when not initialized at the true $\mathbf{X}$ values, the model does not capture the points where the latent variable wraps around the border of the domain very well. This is reflected in the tuning curve estimates in Figure 5.27, which appear misplaced and with several peaks.

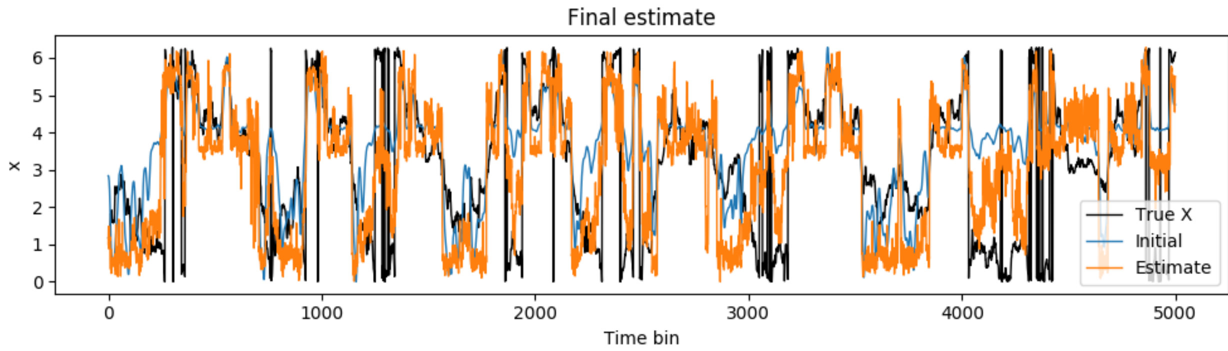


**Figure 5.26:** Final estimate of $\mathbf{X}$ compared to initialization and true path.

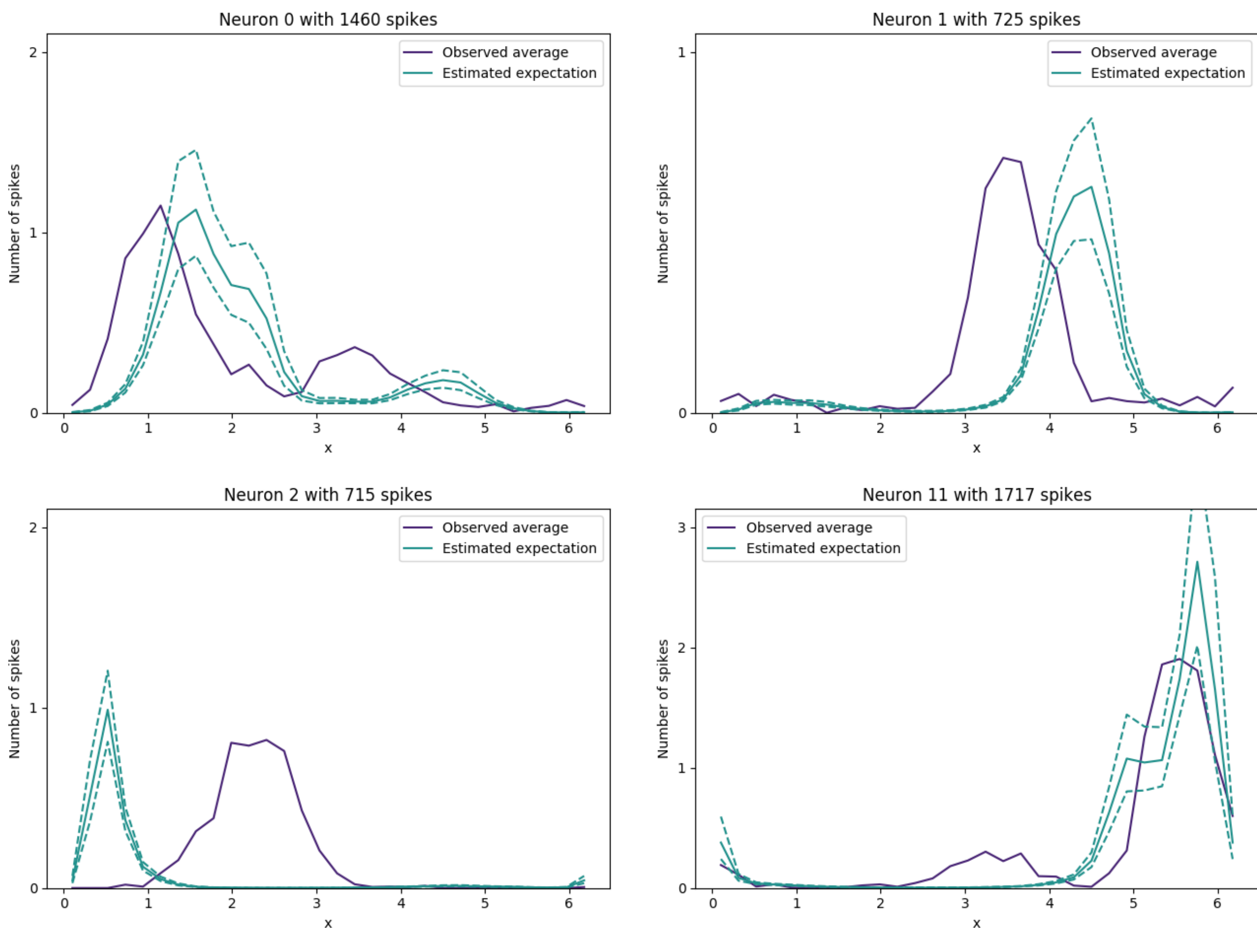


**Figure 5.27:** Posterior estimates of $\mathbf{f}_i(\mathbf{X})$ based on final estimates of $\mathbf{F}$ and $\mathbf{X}$. 95 % credible intervals are shown with dotted lines.

### 5.4.5 Initialization 5: Flat initialization of X and estimated F

Figure 5.28 shows the final estimate when $\mathbf{X}$ is initialized as a flat line and $\mathbf{F}$ is initialized as described in equation (5.5) The RMSE of this estimate is 1.537. Interestingly, this RMSE value is slightly better than the one found by using PCA to initialize $\mathbf{X}$. Figure 5.29 shows the inferred tuning curves.

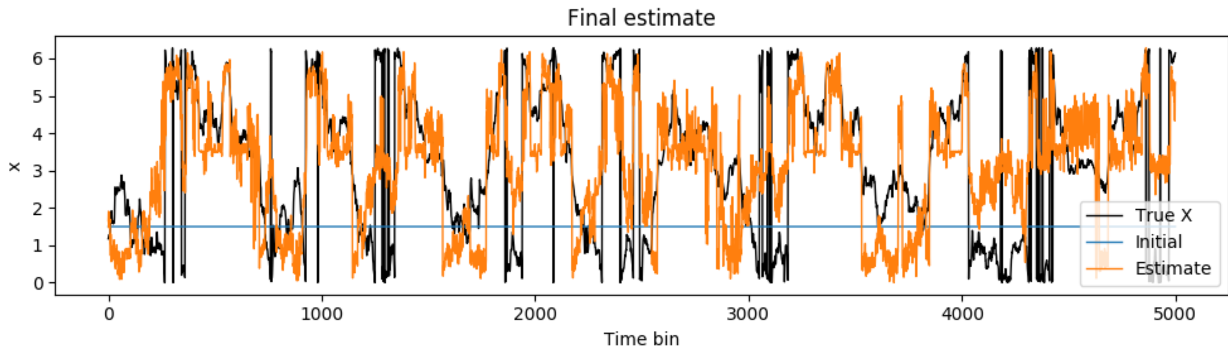


**Figure 5.28:** Final estimate of $\mathbf{X}$ compared to initialization and true path.

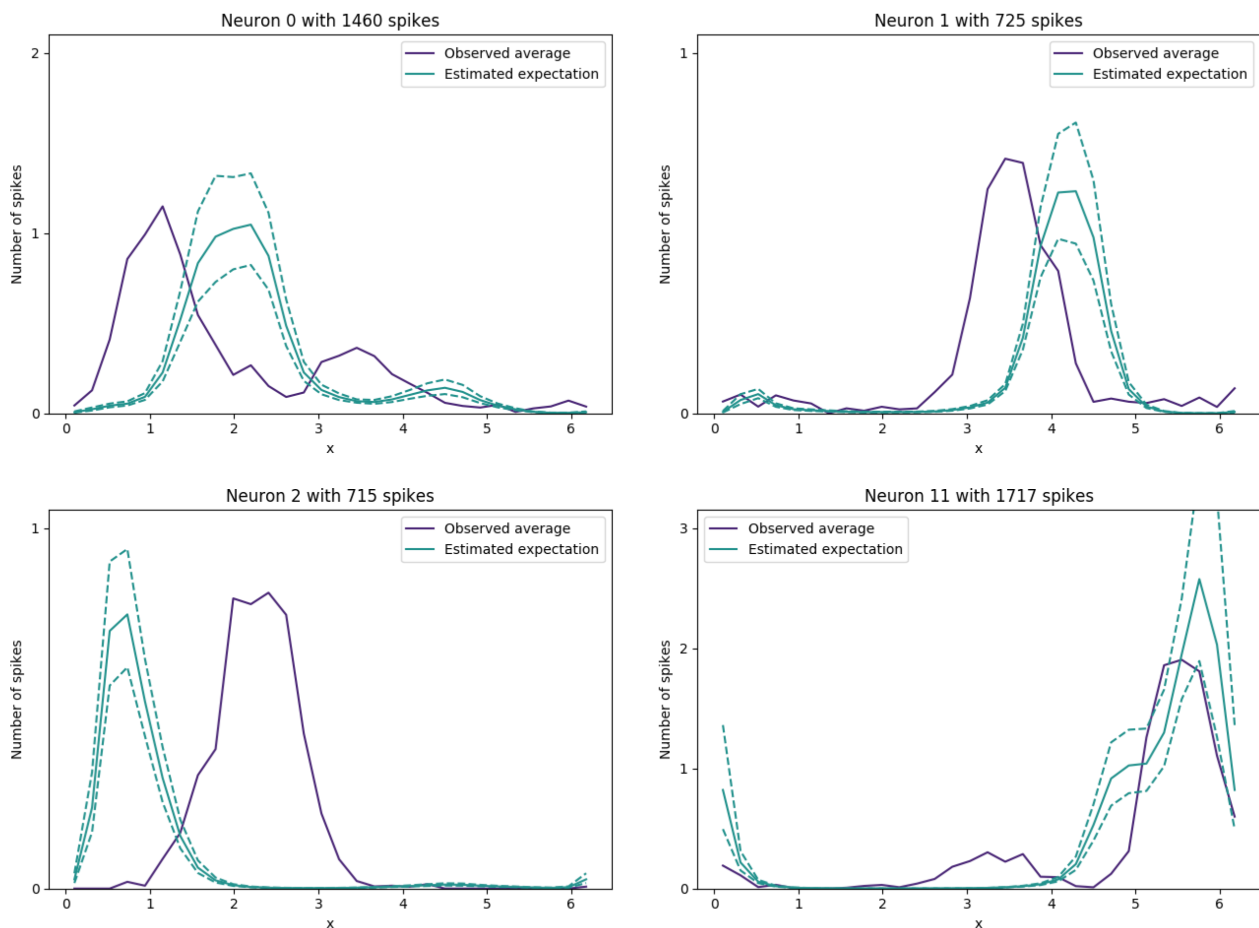


**Figure 5.29:** Posterior estimates of $\mathbf{f}_i(\mathbf{X})$ based on final estimates of $\mathbf{F}$ and $\mathbf{X}$. 95 % credible intervals are shown with dotted lines.

### 5.4.6 Comparison of different initializations

The RMSE and $\mathcal{L}(\mathbf{X})$ values for all the different initializations are listed in order of descending RMSE value in table 5.2. The RMSE value for a PCA estimate of $\mathbf{X}$ with standard deviation 4 in the Gaussian smoothing filter and a random estimate are also included. Notice that the estimate with the highest $\mathcal{L}(\mathbf{X})$ value does not have the lowest RMSE value. The estimate found by initializing at the true values of $\mathbf{X}$ and the optimal estimate of $\mathbf{F}$ has the lowest RMSE value. The flat initialization of $\mathbf{X}$ is about equally good as the PCA initialization for this data.

By comparing the RMSE values of initializations (3) and (4), we see that when $\mathbf{X}$ is initialized using PCA, it is better to use the initialization described in equation (5.5) than using the MAP estimate of $\mathbf{F}$ based on the true $\mathbf{X}$ value. This may be because for the initialization in equation (5.5), the values in $\mathbf{F}$ are lower and have less variance. This might make local maxima in $\mathcal{L}(\mathbf{X})$ less pronounced, allowing more values for $\mathbf{X}$ to be considered in the first iteration, as hypothesized in section 5.2.

Lastly, in figures 5.26 and 5.28, the $\mathbf{X}$ estimate appears to be upside down for $\mathbf{X}$ values below 3. This is reflected in the estimated tuning curves in figures 5.27 and 5.29, where the inferred tuning curves of neurons 0 and 2 would be improved by mirroring the inferred tuning curve values between 0 and 3 about the middle point $\mathbf{x} = 1.5$.

| Initialization | RMSE value | $\mathcal{L}(\mathbf{X})$ value |
|---|---|---|
| (1) True $\mathbf{X}$ and optimal $\mathbf{F}$ | 0.520 | -62593 |
| (2) True $\mathbf{X}$ and estimated $\mathbf{F}$ | 1.499 | -63402 |
| (5) Flat initialization of $\mathbf{X}$ and estimated $\mathbf{F}$ | 1.537 | -63945 |
| (4) PCA initialization of $\mathbf{X}$ and estimated $\mathbf{F}$ | 1.541 | -64139 |
| Just PCA estimate of $\mathbf{X}$ from $\mathbf{Y}$ | 1.727 | -74525 |
| (3) PCA initialization of $\mathbf{X}$ and optimal $\mathbf{F}$ | 1.857 | -62271 |
| Random estimate | 2.569 | -156869 |

**Table 5.2:** Comparison of RMSE values for different initializations and $\mathcal{L}(\mathbf{X})$ value.

Chapter 6

# Discussion and further work

In this chapter, we summarize and interpret the results from Chapter 5, and suggest topics for further research.

## 6.1 Simulated data

In section 5.1, we described how an affine transformation is needed to find the correct flipping, scaling, and offset compared to the true latent variable. In addition, the LMT is exposed to suboptimal local maxima where the estimate of $\mathbf{X}$ may, for example, be partly upside down. A partly upside down estimate was observed for simulated data in Figure 5.3, and for head direction data in figures 5.26 and 5.28.

In section 5.2, we described how the LMT model depends on the initial estimates of $\mathbf{F}$ and $\mathbf{X}$, and showed how the observed spike data $\mathbf{Y}$ and principal component analysis can be used to find initial estimates for both. We suspect that initializing the estimate of $\mathbf{F}$ with a "less confident" estimate like in equation (5.9) makes the local maxima of $\mathcal{L}(\mathbf{X})$ less pronounced, allowing the algorithm to explore a wider range of $\mathbf{X}$ values in the first iteration. Our attempts to determine whether an informed estimate of $\mathbf{X}$ is better than just a flat initialization have been inconclusive.

## 6.2 Robustness evaluation

In the robustness evaluation in section 5.3, we observed that with our implementation of the LMT there seems to be an optimal tuning strength value. The value of this optimal tuning strength depends on the data length $T$, and the optimal tuning strength value is lower for higher $T$. For tuning strengths lower than the optimal value, the RMSE values in Figure 5.15 are lower for longer data. This is in line with our hypothesis. However, for tuning strengths that are higher than the optimal value, the RMSE value is higher for longer data. This is unexpected. As mentioned in section 5.3, we suspect that the local maxima become more pronounced as the tuning strength increases, trapping the estimate of $\mathbf{X}$ in the iterative algorithm. We also suspect that in longer data, there will be more local maxima. This may explain why the optimal tuning strength occurs at a lower value for higher $T$ values.

As mentioned in section 5.3, similar observations have been made by Davidovich et al. (2020) in the reconstruction of the hidden node problem. Like them, we used a Poisson likelihood to model the spike counts. Further work should investigate the effect of exchanging this with another model like the Bernoulli distribution, to see if the presence of an optimal tuning strength is an artifact of the Poisson likelihood model.

Knowledge of the optimal tuning strength could be used to select an optimal bin width since the tuning strength is defined in terms of the expected number of spikes per bin. However, another factor influencing the choice of optimal bin width is the latent variable's smoothness in time. Increasing the bin size would mean averaging over observations of the latent variable, meaning that some precision will be lost.

Figure 5.16 showed that the $\mathcal{L}(\mathbf{X})$ function can not be used to pick between different estimates. This could be investigated further by averaging the different estimates of $\mathbf{F}$ before computing the $\mathcal{L}(\mathbf{X})$ values, since the estimate of $\mathbf{F}$ influences the $\mathcal{L}(\mathbf{X})$ value.

We acknowledge that some or all of these results may be due to our implementation and choices of hyperparameters. For example, in our robustness evaluation, the tuning curves were defined as Gaussian bumps with a certain width. Changing the width of these bumps, or selecting another tuning curve shape could lead to other results. This is just one example of the many choices that have been made in the modeling.

## 6.3 Head direction data

In section 5.4, we applied the model to head direction neurons recorded by Peyrache et al. (2015). In contrast with the simulated data, these neurons may be tuned to other things in addition to the head direction, even though we performed a screening beforehand by selecting only neurons that were tuned to head direction. Furthermore, real neurons have autoregressive properties, meaning that a neuron is more likely to spike if it has spiked in the near past. There is also connectivity between neurons, and neither of these properties are included in the model. Therefore, we should not expect these estimates to be as good as for simulated latent variables. Nevertheless, we observed that the estimate stayed in place when initialized by the true $\mathbf{X}$ value and an estimate of $\mathbf{F}$ based on the true $\mathbf{X}$ value. This indicates that the neurons are strongly tuned to head direction.

As mentioned for the simulated data, we were unable to conclude whether an informed initialization of $\mathbf{X}$, like PCA, is better than a flat initial estimate in general. Figures 5.26 and 5.28 showed an example where the PCA initialization and flat initialization led to similar RMSE values, with the flat initialization being slightly better.

## 6.4 Future work

If the head direction neurons were tuned to more than one variable, the LMT might infer some combination of these variables instead of just inferring one. The other variables could be modeled as well, provided that their dimensionality and domain could be determined. It is expected that inferring several variables would be more challenging than inferring just one, but we would like to investigate this further.

We have used an iterative MAP procedure to infer the latent variable instead of the decoupled Laplace approximation introduced by Wu et al. (2017). It would be interesting to investigate how this algorithm compares with the decoupled Laplace approximation in terms of accuracy and computational complexity.

In all our data analysis, the hyperparameters $\boldsymbol{\theta} = \{\sigma, \delta, r, l\}$ were set through an extensive process of trial and error, and the noise parameter $\sigma_\varepsilon^2$ was adjusted at every iteration until convergence, as has been described. Ideally, the estimates of these hyperparameters should be found by optimization rather than trial and error. In a supervised setting, prior knowledge of the tuning and the latent variable may be used to select good initial estimates. In an unsupervised setting, it is our belief that the observation matrix $\mathbf{Y}$ can be used to find reasonable initial estimates, as was done for $\mathbf{F}$ and $\mathbf{X}$.

In our robustness evaluation, the number of simulated neurons was set to 100. It would be interesting to investigate how the RMSE value and the ideal tuning strength would vary with the number of observed neurons.

## 6.5 Conclusion

We showed how an iterative MAP procedure can be used instead of the decoupled Laplace approximation to infer the head direction from a neural recording, with a lower RMSE value than PCA. We have contributed to the Latent Manifold Model (LMT) as described by Wu et al. (2017) by highlighting some convergence issues that users of the method should be aware of. Care should be taken to select good initial estimates for $\mathbf{F}$ and $\mathbf{X}$, and we have described an initialization of $\mathbf{F}$ that worked well in our application.

We made some details in the implementation explicit by showing how the deterministic training conditional inducing points approximation can be used to implement a computationally efficient gradient-based optimization, and motivated the practice of lowering the noise term in a way reminiscent of simulated annealing and graduated optimization, which was not done by Wu et al. (2017).

Furthermore, we evaluated the feasibility of using the $\mathcal{L}(\mathbf{X})$ function to pick between different estimates and shown that this function is not helpful for this use. In addition, we evaluated the robustness of the algorithm with regards to different tuning strengths and data lengths and found that there is an optimal tuning strength that depends on the data length. Whether this is an artifact of our specific implementation or caused by the use of the Poisson distribution to model the neural activity should be the topic of future research.

# Bibliography


Ahrens, M.B., Orger, M.B., Robson, D.N., Li, J.M., Keller, P.J., 2013. Whole-brain functional imaging at cellular resolution using light-sheet microscopy. Nature methods 10, 413.

Anscombe, F.J., 1948. The transformation of poisson, binomial and negative-binomial data. Biometrika 35, 246–254.

Bauer, M., van der Wilk, M., Rasmussen, C.E., 2016. Understanding probabilistic sparse gaussian process approximations, in: Advances in neural information processing systems, pp. 1533–1541.

Byron, M.Y., Cunningham, J.P., Santhanam, G., Ryu, S.I., Shenoy, K.V., Sahani, M., 2009. Gaussian-process factor analysis for low-dimensional single-trial analysis of neural population activity, in: Advances in neural information processing systems, pp. 1881–1888.

Carrillo-Reid, L., Yang, W., Bando, Y., Peterka, D.S., Yuste, R., 2016. Imprinting and recalling cortical ensembles. Science 353, 691–694.

Casella, G., Berger, R.L., 2002. Statistical inference. volume 2. Duxbury Pacific Grove, CA.

Chib, S., Greenberg, E., 1995. Understanding the metropolis-hastings algorithm. The american statistician 49, 327–335.

Cunningham, J.P., Byron, M.Y., 2014. Dimensionality reduction for large-scale neural recordings. Nature neuroscience 17, 1500.

Davidovich, I., Dunn, B.A., Hertz, J., Roudi, Y., 2020. Mean field theory inference and learning in networks with stochastic natural exponential family neurons.

Gao, P., Ganguli, S., 2015. On simplicity and complexity in the brave new world of large-scale neuroscience. Current opinion in neurobiology 32, 148–155.

Geyer, C.J., 1992. Practical markov chain monte carlo. Statistical science , 473–483.

Hanche-Olsen, H., 1997. The derivative of a determinant. Personal note available at `http://www.math.ntnu.no/~hanche/notes/diffdet/diffdet.pdf`.

Härdle, W., Simar, L., 2007. Applied multivariate statistical analysis. Springer.

Hazan, E., Levy, K.Y., Shalev-Shwartz, S., 2016. On graduated optimization for stochastic non-convex problems, in: International conference on machine learning, pp. 1833–1841.

Horn, R., Johnson, C., 1985. Matrix analysis. Cambridge University, Cambridge, UK , p. 65.

Jensen, K.T., Kao, T.C., Tripodi, M., Hennequin, G., 2020. Manifold gplvms for discovering non-euclidean latent structure in neural data. arXiv preprint arXiv:2006.07429 .

Kennedy, J., Eberhart, R., 1995. Particle swarm optimization, in: Proceedings of ICNN'95-International Conference on Neural Networks, IEEE. pp. 1942–1948.

Lawrence, N.D., 2004. Gaussian process latent variable models for visualisation of high dimensional data, in: Advances in neural information processing systems, pp. 329–336.

Macke, J.H., Buesing, L., Sahani, M., Chen, Z., 2015. Estimating state and parameters in state space models of spike trains. Advanced state space methods for neural and clinical data 137.

Magnus, J.R., Neudecker, H., 1988. Matrix differential calculus with applications in statistics and econometrics. John Wiley & Sons.

McCullagh, P., Nelder, J.A., 1989. Generalized linear models, volume 37. Monographs on statistics and applied probability .

Mimica, B., 2019. Neural coding of behavior in the rodent associative cortices. PhD thesis, NTNU.

Nicolelis, M.A., Dimitrov, D., Carmena, J.M., Crist, R., Lehew, G., Kralik, J.D., Wise, S.P., 2003. Chronic, multisite, multielectrode recordings in macaque monkeys. Proceedings of the National Academy of Sciences 100, 11041–11046.

Paninski, L., 2004. Maximum likelihood estimation of cascade point-process neural encoding models. Network: Computation in Neural Systems 15, 243–262.

Peyrache, A., Lacroix, M.M., Petersen, P.C., Buzsáki, G., 2015. Internally organized mechanisms of the head direction sense. Nature neuroscience 18, 569.

Quiñonero-Candela, J., Rasmussen, C.E., 2005. A unifying view of sparse approximate gaussian process regression. Journal of Machine Learning Research 6, 1939–1959.

Rasmussen, C., Williams, C., 2006. Gaussian Processes for Machine Learning. MIT press.

Roweis, S.T., Saul, L.K., 2000. Nonlinear dimensionality reduction by locally linear embedding. Science 290, 2323–2326.

Rue, H., Martino, S., Chopin, N., 2009. Approximate bayesian inference for latent gaussian models by using integrated nested laplace approximations. Journal of the royal statistical society: Series b (statistical methodology) 71, 319–392.

Rybakken, E., Baas, N., Dunn, B., 2019. Decoding of neural data using cohomological feature extraction. Neural computation 31, 68–93.

Steinmetz, N.A., Koch, C., Harris, K.D., Carandini, M., 2018. Challenges and opportunities for large-scale electrophysiology with neuropixels probes. Current opinion in neurobiology 50, 92–100.

Stevenson, I.H., Kording, K.P., 2011. How advances in neural recording affect data analysis. Nature neuroscience 14, 139.

Tenenbaum, J.B., De Silva, V., Langford, J.C., 2000. A global geometric framework for nonlinear dimensionality reduction. Science 290, 2319–2323.

Teukolsky, S.A., Flannery, B.P., Press, W., Vetterling, W., 1992. Numerical recipes in c. SMR 693, 59–70.

Titsias, M.K., 2009. Variational model selection for sparse gaussian process regression. Report, University of Manchester, UK .

Tobler, W.R., 1970. A computer movie simulating urban growth in the detroit region. Economic geography 46, 234–240.

Truccolo, W., Eden, U.T., Fellows, M.R., Donoghue, J.P., Brown, E.N., 2005. A point process framework for relating neural spiking activity to spiking history, neural ensemble, and extrinsic covariate effects. Journal of neurophysiology 93, 1074–1089.

Van Laarhoven, P.J., Aarts, E.H., 1987. Simulated annealing: Theory and applications, Springer, pp. 7–15.

Wu, A., Pashkovski, S., Datta, S.R., Pillow, J.W., 2018. Learning a latent manifold of odor representations from neural responses in piriform cortex, in: Advances in Neural Information Processing Systems, pp. 5378–5388.

Wu, A., Roy, N.A., Keeley, S., Pillow, J.W., 2017. Gaussian process based nonlinear latent structure discovery in multivariate spike train data, in: Advances in neural information processing systems, pp. 3496–3505.

Wu, Z., 1996. The effective energy transformation scheme as a special continuation approach to global optimization with application to molecular conformation. SIAM Journal on Optimization 6, 748–768.

Zhu, C., Byrd, R.H., Lu, P., Nocedal, J., 1997. Algorithm 778: L-bfgs-b: Fortran subroutines for large-scale bound-constrained optimization. ACM Transactions on Mathematical Software (TOMS) 23, 550–560.

Appendix A

# Theorems and derivations

## A.1 Matrix calculus

### A.1.1 Maximizing the fraction of two quadratic forms

The following theorem from Härdle and Simar (2007) is used to find the optimal lower dimensional manifold in principal component analysis (section 3.4). If two matrices A and B are symmetric and B is positive definite, then the maximum of $\frac{x^T Ax}{x^T Bx}$ is given by the largest eigenvalue of $B^{-1}A$. More generally,

$$\max_x \frac{x^T Ax}{x^T Bx} = \lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_p = \min_x \frac{x^T Ax}{x^T Bx} \tag{A.1}$$

where $\lambda_1, ..., \lambda_p$ are the eigenvalues of $B^{-1}A$.

### A.1.2 Matrix inversion lemma

The matrix inversion lemma (Teukolsky et al., 1992) provides a computational shortcut when calculating the inverse of an $n$ by $n$ matrix of the following form:

$$(UWV^T + Z)^{-1} = Z^{-1} - Z^{-1}U(W^{-1} + V^T Z^{-1}U)^{-1}V^T Z^{-1} \tag{A.2}$$

If $Z = \sigma^2 I$, we get:

$$\begin{aligned}(UWV^T + \sigma^2 I)^{-1} &= \sigma^{-2}I - \sigma^{-2}U(W^{-1} + V^T\sigma^{-2}IU)^{-1}V^T\sigma^{-2}I \\ &= \sigma^{-2}I - \sigma^{-2}U(\sigma^2 W^{-1} + V^T U)^{-1}V^T\end{aligned} \tag{A.3}$$

This can be used to efficiently invert the approximate covariance matrix of a Gaussian process, where $U = V = K_{\mathbf{f},\mathbf{u}}$ and $W = K_{\mathbf{u},\mathbf{u}}^{-1}$:

$$(K_{\mathbf{f},\mathbf{u}}K_{\mathbf{u},\mathbf{u}}^{-1}K_{\mathbf{f},\mathbf{u}}^T + \sigma^2 I)^{-1} = \sigma^{-2}I - \sigma^{-2}K_{\mathbf{f},\mathbf{u}}(\sigma^2 K_{\mathbf{u},\mathbf{u}} + K_{\mathbf{f},\mathbf{u}}^T K_{\mathbf{f},\mathbf{u}})^{-1}K_{\mathbf{f},\mathbf{u}}^T \tag{A.4}$$

### A.1.3 Theorem 1.3.22 from Horn and Johnson (1985)

Let $p_{AB}(t)$ be the characteristic polynomial $p_{AB}(t) = |tI - AB|$, where $|\cdot|$ is the determinant. Theorem 1.3.22 in "Matrix Analysis" by Horn and Johnson (1985) states the following for two matrices $A$ and $B$, where the notation has been changed slightly to match the notation of this thesis:

"Suppose that $A \in \mathbb{R}^{m\times n}$, $B \in \mathbb{R}^{n\times m}$ with $m \leq n$. Then the $n$ eigenvalues of $BA$ are the $m$ eigenvalues of $AB$ together with $n - m$ zeroes; that is, $p_{BA}(t) = t^{n-m}p_{AB}(t)$. If $m = n$ and at least one of $A$ or $B$ is nonsingular, than $AB$ and $BA$ are similar."

### A.1.4 Matrix differentiation

Differentiating a matrix with regards to a scalar is done elementwise for the matrix.

$$\frac{\partial}{\partial x}\{A\}_{ij} = \{\frac{\partial A_{i,j}}{\partial x}\}_{ij} \tag{A.5}$$

Differentiating the inverse of a matrix is done like this:

$$\frac{\partial A^{-1}}{\partial x} = -A^{-1}\frac{\partial A}{\partial x}A^{-1} \tag{A.6}$$

Differentiating the determinant of a matrix is done using Jacobi's formula. Though this is a classical result, Magnus and Neudecker (1988) is often cited as a reference. The formula states:

$$\frac{d}{dx}\det A(x) = \text{trace}\left(\text{adj}(A(x))\frac{dA(x)}{dx}\right) \tag{A.7}$$

where adj($a(x)$) is the adjugate, or classical adjoint, of $A$. As noted by Hanche-Olsen (1997), the following holds when $A$ is invertible:

$$\frac{d}{dx}\log\det A(x) = \text{trace}\left(A(x)^{-1}\frac{dA(x)}{dx}\right) \tag{A.8}$$

## A.2 Bernoulli spike model

Let $\pi_{i,t} = \frac{\exp(f_{i,t})}{1+\exp(f_{i,t})} = P(y_{i,t} = 1)$ be the spike probability. The Bernoulli likelihood is

$$p(y_{i,t}|\pi_{i,t}) = \pi_{i,t}^{y_{i,t}}(1-\pi_{i,t})^{1-y_{i,t}} \tag{A.9}$$

or equivalently, on the form of the exponential family,

$$\begin{aligned} p(y_{i,t}|f_{i,t}) &= \exp\Big(y_{i,t}f_{i,t} - \log(1+\exp(f_{i,t}))\Big) \\ \Longrightarrow \log p(y_{i,t}|f_{i,t}) &= y_{i,t}f_{i,t} - \log(1+\exp(f_{i,t})) \end{aligned} \tag{A.10}$$

This means that the unnormalized log posterior of $\mathbf{f}_i$ (the objective function) becomes

$$\begin{aligned} \Psi(\mathbf{f}_i) &:= \log p(\mathbf{y}_i|\mathbf{f}_i) + \log p(\mathbf{f}_i|\mathbf{X}) \\ &= \sum_{t=1}^{T} y_{i,t}f_{i,t} - \log(1+\exp(f_{i,t})) - \frac{1}{2}\mathbf{f}_i^T K_x^{-1}\mathbf{f}_i - \log\left((2\pi)^{\frac{T}{2}}|K_x|^{\frac{1}{2}}\right) \end{aligned} \tag{A.11}$$

The first derivative of $\Psi(\mathbf{f}_i)$ is:

$$\begin{aligned} \frac{\partial}{\partial f_{i,t}}\Psi(\mathbf{f}_i) =& y_{i,t} - \frac{\exp(f_{i,t})}{1+\exp(f_{i,t})} - \sum_{j=1}^{T} f_{i,j}K^{-1}_{x\{t,j\}} \\ \Longleftrightarrow \nabla\Psi(\mathbf{f}_i) =& \mathbf{y}_i - \mathbf{e}_i^{bern} - K_x^{-1}\mathbf{f}_i, \end{aligned} \tag{A.12}$$

where the vector $\mathbf{e}_i^{bern}$ has elements $e_{i,t}^{bern} = \frac{\exp(f_{i,t})}{1+\exp(f_{i,t})}, t = 1, \ldots, T$. The second derivative of $\Psi(\mathbf{f}_i)$ is

$$\begin{aligned} \frac{\partial^2}{\partial f_{i,t_1}\partial f_{i,t_2}}\Psi(\mathbf{f}_i) &= \begin{cases} \frac{\exp(f_{i,t_1})}{(1+\exp(f_{i,t_1}))^2} - K_{x\{t_1,t_1\}} & \text{for} \quad t_1 = t_2 \\ -K_{x\{t_1,t_2\}} & \text{for} \quad t_1 \neq t_2 \end{cases} \\ \Longrightarrow \nabla\nabla\Psi(\mathbf{f}_i) &= -I\tilde{\mathbf{e}}_i^{bern} - K_x^{-1} \end{aligned} \tag{A.13}$$

where $I$ represents the identity matrix of size $T$ and the vector $\tilde{\mathbf{e}}_i^{bern}$ has elements $\tilde{e}_{i,t}^{bern} = \frac{\exp(f_{i,t})}{(1+\exp(f_{i,t}))^2}$.

Appendix B

# Python code

The following Python code shows the implementation in three parts: a shared function library, the application to the head direction dataset, and the robustness evaluation. The entire repository is available at `https://github.com/evenmm`.

## B.1 Function library

The function library is shared between the application to head direction and the robustness evaluation.

**function_library.py**

```
1 from scipy import *
2 import scipy.io
3 import scipy.ndimage
4 import numpy as np
5 import scipy.optimize as spoptim
6 import numpy.random
7 import matplotlib
8 #matplotlib.use('Agg') # When running on cluster, plots cannot be shown and this
      must be used
9 import matplotlib.pyplot as plt
10 import time
11 import sys
12 plt.rc('image', cmap='viridis')
13 from scipy import optimize
14 numpy.random.seed(13)
15 from multiprocessing import Pool
16 from sklearn.decomposition import PCA
17
18 # Peyrache data with head direction neurons
19 if sys.argv[0] == "em-algorithm-peyrache-data.py":
20     from parameter_file_peyrache import *
21
22 # Robustness evaluation
23 elif sys.argv[0] == "cluster-parallel-robustness-evaluation.py":
24     from parameter_file_robustness import *
25
26 # Example plotting
27 elif sys.argv[0] == "example_plotting.py":
28     from parameter_file_exampleplotting import *
```

```
29
30 #####################
31 # Covariance kernels #
32 #####################
33
34 def squared_exponential_covariance(xvector1, xvector2, sigma, delta):
35     if COVARIANCE_KERNEL_KX == "nonperiodic":
36         distancesquared = scipy.spatial.distance.cdist(xvector1, xvector2, '
    sqeuclidean')
37     if COVARIANCE_KERNEL_KX == "periodic":
38         # This handles paths that stretches across anywhere as though the domain
     is truly periodic
39         # First put every time point between 0 and 2pi
40         xvector1 = xvector1 % (2*np.pi)
41         xvector2 = xvector2 % (2*np.pi)
42         # Then take care of periodicity
43         distancesquared_1 = scipy.spatial.distance.cdist(xvector1, xvector2, '
    sqeuclidean')
44         distancesquared_2 = scipy.spatial.distance.cdist(xvector1+2*np.pi,
    xvector2, 'sqeuclidean')
45         distancesquared_3 = scipy.spatial.distance.cdist(xvector1-2*np.pi,
    xvector2, 'sqeuclidean')
46         min_1 = np.minimum(distancesquared_1, distancesquared_2)
47         distancesquared = np.minimum(min_1, distancesquared_3)
48     return sigma * exp(-distancesquared/(2*delta**2))
49
50 def exponential_covariance(tvector1, tvector2, sigma, delta):
51     absolutedistance = scipy.spatial.distance.cdist(tvector1, tvector2, '
    euclidean')
52     return sigma * exp(-absolutedistance/delta)
53
54 #######################
55 # Covariance matrices  #
56 #######################
57 K_t = exponential_covariance(np.linspace(1,T,T).reshape((T,1)),np.linspace(1,T,T
    ).reshape((T,1)), sigma_x, delta_x)
58 K_t_inverse = np.linalg.inv(K_t)
59
60 ########################
61 ## Likelihood functions #
62 ########################
63
64 # NEGATIVE Loglikelihood, gradient and Hessian. minimize to maximize. Equation
    (4.17)++
65 def f_loglikelihood_bernoulli(f_i, sigma_n, y_i, K_xg_prev, K_gg): # Psi
66     likelihoodterm = sum( np.multiply(y_i, f_i) - np.log(1+np.exp(f_i))) #
    Corrected 16.03 from sum( np.multiply(y_i, (f_i - np.log(1+np.exp(f_i)))) +
    np.multiply((1-y_i), np.log(1- np.divide(np.exp(f_i), 1 + np.exp(f_i)))))
67     priorterm_1 = -0.5*sigma_n**-2 * np.dot(f_i.T, f_i)
68     fT_k = np.dot(f_i, K_xg_prev)
69     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
70     priorterm_2 = 0.5*sigma_n**-2 * np.dot(np.dot(fT_k, smallinverse), fT_k.T)
71     return - (likelihoodterm + priorterm_1 + priorterm_2)
72 def f_jacobian_bernoulli(f_i, sigma_n, y_i, K_xg_prev, K_gg):
73     yf_term = y_i - np.divide(np.exp(f_i), 1 + np.exp(f_i))
74     priorterm_1 = -sigma_n**-2 * f_i
75     kTf = np.dot(K_xg_prev.T, f_i)
```

```
76     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
77     priorterm_2 = sigma_n**-2 * np.dot(K_xg_prev, np.dot(smallinverse, kTf))
78     f_derivative = yf_term + priorterm_1 + priorterm_2
79     return - f_derivative
80 def f_hessian_bernoulli(f_i, sigma_n, y_i, K_xg_prev, K_gg):
81     e_tilde = np.divide(np.exp(f_i), (1 + np.exp(f_i))**2)
82     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
83     f_hessian = - np.diag(e_tilde) - sigma_n**-2 * np.identity(T) + sigma_n**-2
    * np.dot(K_xg_prev, np.dot(smallinverse, K_xg_prev.T))
84     return - f_hessian
85 
86 # NEGATIVE Loglikelihood, gradient and Hessian. minimize to maximize.
87 def f_loglikelihood_poisson(f_i, sigma_n, y_i, K_xg_prev, K_gg):
88     likelihoodterm = sum( np.multiply(y_i, f_i) - np.exp(f_i))
89     priorterm_1 = -0.5*sigma_n**-2 * np.dot(f_i.T, f_i)
90     fT_k = np.dot(f_i, K_xg_prev)
91     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
92     priorterm_2 = 0.5*sigma_n**-2 * np.dot(np.dot(fT_k, smallinverse), fT_k.T)
93     return - (likelihoodterm + priorterm_1 + priorterm_2)
94 
95 def f_jacobian_poisson(f_i, sigma_n, y_i, K_xg_prev, K_gg):
96     yf_term = y_i - np.exp(f_i)
97     priorterm_1 = -sigma_n**-2 * f_i
98     kTf = np.dot(K_xg_prev.T, f_i)
99     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
100     priorterm_2 = sigma_n**-2 * np.dot(K_xg_prev, np.dot(smallinverse, kTf))
101     f_derivative = yf_term + priorterm_1 + priorterm_2
102     return - f_derivative
103 def f_hessian_poisson(f_i, sigma_n, y_i, K_xg_prev, K_gg):
104     e_poiss = np.exp(f_i)
105     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_xg_prev.T,
    K_xg_prev))
106     f_hessian = - np.diag(e_poiss) - sigma_n**-2*np.identity(T) + sigma_n**-2 *
    np.dot(K_xg_prev, np.dot(smallinverse, K_xg_prev.T))
107     return - f_hessian
108 
109 # L function
110 def x_posterior_no_la(X_estimate, sigma_n, F_estimate, K_gg, x_grid_induce):
111     start = time.time()
112     K_xg = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
113     K_gx = K_xg.T
114     stop = time.time()
115     if SPEEDCHECK:
116         print("Speedcheck of L function:")
117         print("Making Kxg            :", stop-start)
118 
119     start = time.time()
120     #Kx_inducing = np.matmul(np.matmul(K_xg, K_gg_inverse), K_gx) + sigma_n**2
121     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_gx, K_xg))
122     # Kx_inducing_inverse = sigma_n**-2*np.identity(T) - sigma_n**-2 * np.matmul
    (np.matmul(K_xg, smallinverse), K_gx)
123     tempmatrix = np.matmul(np.matmul(K_xg, smallinverse), K_gx)
124     stop = time.time()
```

```
125     if SPEEDCHECK:
126         print("Making small/tempmatrx:", stop-start)
127 
128     # yf_term ##########
129     ####################
130     #start = time.time()
131     #if LIKELIHOOD_MODEL == "bernoulli": # equation 4.26
132     #    yf_term = sum(np.multiply(y_spikes, F_estimate) - np.log(1 + np.exp(
        F_estimate)))
133     #elif LIKELIHOOD_MODEL == "poisson": # equation 4.43
134     #    yf_term = sum(np.multiply(y_spikes, F_estimate) - np.exp(F_estimate))
135     #stop = time.time()
136     #if SPEEDCHECK:
137     #    print("yf term               :", stop-start)
138 
139     # f prior term #####
140     ####################
141     start = time.time()
142     f_prior_term_1 = sigma_n**-2 * np.trace(np.matmul(F_estimate, F_estimate.T))
143     fK = np.matmul(F_estimate, tempmatrix)
144     fKf = np.matmul(fK, F_estimate.T)
145     f_prior_term_2 = - sigma_n**-2 * np.trace(fKf)
146 
147     f_prior_term = - 0.5 * (f_prior_term_1 + f_prior_term_2)
148     stop = time.time()
149     if SPEEDCHECK:
150         print("f prior term          :", stop-start)
151 
152     # logdet term ######
153     ####################
154     #logdet_term = - 0.5 * N * np.log(np.linalg.det(Kx_inducing))
155     # smallinverse = np.linalg.inv(np.matmul(K_gx, K_xg) + K_gg*sigma_n**2)
156 
157     start = time.time()
158     logDetS1 = np.log(np.linalg.det(np.matmul(K_gx, K_xg) + K_gg*sigma_n**2)) -
        np.log(np.linalg.det(K_gg)) + (T-N_inducing_points) * np.log(sigma_n**2)
159     logdet_term = - 0.5 * N * logDetS1
160     stop = time.time()
161     if SPEEDCHECK:
162         print("logdet term           :", stop-start)
163 
164     # x prior term #####
165     ####################
166     start = time.time()
167     xTKt = np.dot(X_estimate.T, K_t_inverse) # Inversion trick for this too? No.
         If we don't do Fourier then we are limited by this.
168     x_prior_term = - 0.5 * np.dot(xTKt, X_estimate)
169     stop = time.time()
170     if SPEEDCHECK:
171         print("X prior term          :", stop-start)
172         print("logdet_term", logdet_term)
173         print("f_prior_term", f_prior_term)
174         print("x_prior_term", x_prior_term)
175     posterior_loglikelihood = logdet_term + f_prior_term + x_prior_term #+
        yf_term
176     return - posterior_loglikelihood
177 
178 # Gradient of L
```

```
179 def x_jacobian_no_la(X_estimate, sigma_n, F_estimate, K_gg, x_grid_induce):
180     ####################
181     # Initial matrices #
182     ####################
183     start = time.time()
184     K_xg = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
185     K_gx = K_xg.T
186     stop = time.time()
187     if SPEEDCHECK:
188         print("\nSpeedcheck of x_jacobian function:")
189         print("Making Kxg             :", stop-start)
190 
191     start = time.time()
192     B_matrix = np.matmul(K_gx, K_xg) + (sigma_n**2) * K_gg
193     B_matrix_inverse = np.linalg.inv(B_matrix)
194     stop = time.time()
195     if SPEEDCHECK:
196         print("Making B and B inverse:", stop-start)
197 
198     start = time.time()
199     #Kx_inducing = np.matmul(np.matmul(K_xg, K_gg_inverse), K_gx) + sigma_n**2
200     #smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_gx, K_xg))
201     # Kx_inducing_inverse = sigma_n**-2*np.identity(T) - sigma_n**-2 * np.matmul
    (np.matmul(K_xg, smallinverse), K_gx)
202     stop = time.time()
203     if SPEEDCHECK:
204         print("Making small/tempmatrx:", stop-start)
205 
206     ####################
207     # logdet term ######
208     ####################
209     start = time.time()
210 
211     ## Evaluate the derivative of K_xg. Row t of this matrix holds the nonzero
    row of the matrix d/dx_t K_xg
212     d_Kxg = scipy.spatial.distance.cdist(X_estimate.reshape((T,1)),x_grid_induce
    .reshape((N_inducing_points,1)), lambda u, v: -(u-v)*np.exp(-(u-v)**2/(2*
    delta_f_fit**2)))
213     d_Kxg = d_Kxg*sigma_f_fit*(delta_f_fit**-2)
214 
215     ## Reshape K_gx and K_xg to speed up matrix multiplication
216     K_g_column_tensor = K_gx.T.reshape((T, N_inducing_points, 1)) # Tensor with
    T depth containing single columns of length N_ind
217     d_Kx_row_tensor = d_Kxg.reshape((T, 1, N_inducing_points)) # Tensor with T
    depth containing single rows of length N_ind
218 
219     # Matrix multiply K_gx and d(K_xg)
220     product_Kgx_dKxg = np.matmul(K_g_column_tensor, d_Kx_row_tensor) # 1000 by
    30 by 30
221 
222     # Sum with transpose
223     trans_sum_K_dK = product_Kgx_dKxg + np.transpose(product_Kgx_dKxg, axes
    =(0,2,1))
224 
225     # Create B^-1 copies for vectorial matrix multiplication
226     B_inv_tensor = np.repeat([B_matrix_inverse],T,axis=0)
227 
```

```
    # Then tensor multiply B^-1 with all the different trans_sum_K_dK
    big_tensor = np.matmul(B_inv_tensor, trans_sum_K_dK)

    # Take trace of each individually
    trace_array = np.trace(big_tensor, axis1=1, axis2=2)

    # Multiply by - N/2
    logdet_gradient = - N/2 * trace_array

    stop = time.time()
    if SPEEDCHECK:
        print("logdet term            :", stop-start)

    ####################
    # f prior term ##### (speeded up 10x)
    ####################
    start = time.time()
    fMf = np.zeros((T,N,N))

    ## New hot take:
    # Elementwise in the sum, priority on things with dim T, AND things that don't need to be vectorized *first*.
    # Wrap things in from the sides to sandwich the tensor.
    f_Kx = np.matmul(F_estimate, K_xg)
    f_Kx_Binv = np.matmul(f_Kx, B_matrix_inverse)
    #Binv_Kg_f = np.transpose(f_Kx_Binv)

    #d_Kg_column_tensor = np.transpose(d_Kx_row_tensor, axes=(0,2,1))

    # partial derivatives need tensorization
    # f_dKx = np.matmul(F_estimate, d_Kxg)
    f_column_tensor = F_estimate.T.reshape((T, N, 1))
    f_dKx_tensor = np.matmul(f_column_tensor, d_Kx_row_tensor) # (N x N_inducing) matrices
    dKg_f_tensor = np.transpose(f_dKx_tensor, axes=(0,2,1))

    f_Kx_Binv_copy_tensor = np.repeat([f_Kx_Binv], T, axis=0)
    Binv_Kg_f_copy_tensor = np.transpose(f_Kx_Binv_copy_tensor, axes=(0,2,1)) # repeat([Binv_Kg_f], T, axis=0)

    ## A: f dKx Binv Kgx f
    fMf += np.matmul(f_dKx_tensor, Binv_Kg_f_copy_tensor)

    ## C: - f Kx Binv Kg dKx Binv Kg f
    Kg_dKx_tensor = np.matmul(K_g_column_tensor, d_Kx_row_tensor)
    f_Kx_Binv_Kg_dKx_tensor = np.matmul(f_Kx_Binv_copy_tensor, Kg_dKx_tensor)
    fMf -= np.matmul(f_Kx_Binv_Kg_dKx_tensor, Binv_Kg_f_copy_tensor)

    ## B: - f Kx Binv dKg Kx Binv Kg f
    dKg_Kx_tensor = np.transpose(Kg_dKx_tensor, axes=(0,2,1))
    f_Kx_Binv_dKg_Kx_tensor = np.matmul(f_Kx_Binv_copy_tensor, dKg_Kx_tensor)
    fMf -= np.matmul(f_Kx_Binv_dKg_Kx_tensor, Binv_Kg_f_copy_tensor)

    ## D: f Kx Binv dKg f
    fMf += np.matmul(f_Kx_Binv_copy_tensor, dKg_f_tensor)

    ## Trace for each matrix in the tensor
    fMfsum = np.trace(fMf, axis1=1, axis2=2)
```

```
283     f_prior_gradient = sigma_n**(-2) / 2 * fMfsum
284 
285     stop = time.time()
286     if SPEEDCHECK:
287         print("f prior term          :", stop-start)
288 
289     ####################
290     # x prior term #####
291     ####################
292     start = time.time()
293     x_prior_gradient = (-1) * np.dot(X_estimate.T, K_t_inverse)
294     stop = time.time()
295     if SPEEDCHECK:
296         print("X prior term          :", stop-start)
297     ####################
298     x_gradient = logdet_gradient + f_prior_gradient + x_prior_gradient
299     return - x_gradient
300 
301 def just_fprior_term(X_estimate):
302     K_xg = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
303     K_gx = K_xg.T
304     #Kx_inducing = np.matmul(np.matmul(K_xg, K_gg_inverse), K_gx) + sigma_n**2
305     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_gx, K_xg))
306     # Kx_inducing_inverse = sigma_n**-2*np.identity(T) - sigma_n**-2 * np.matmul
    (np.matmul(K_xg, smallinverse), K_gx)
307     tempmatrix = np.matmul(np.matmul(K_xg, smallinverse), K_gx)
308 
309     # f prior term #####
310     ####################
311     f_prior_term_1 = sigma_n**-2 * np.trace(np.matmul(F_estimate, F_estimate.T))
312     fK = np.matmul(F_estimate, tempmatrix)
313     fKf = np.matmul(fK, F_estimate.T)
314     f_prior_term_2 = - sigma_n**-2 * np.trace(fKf)
315 
316     f_prior_term = - 0.5 * (f_prior_term_1 + f_prior_term_2)
317 
318     posterior_loglikelihood = f_prior_term #+ logdet_term #+ x_prior_term
319     return - posterior_loglikelihood
320 
321 ############################################################
322 ##### Posterior inference of tuning curves on a grid ######
323 ############################################################
324 
325 def posterior_f_inference(X_estimate, F_estimate, sigma_n, y_spikes, path,
    x_grid_for_plotting, bins_for_plotting, peak_f_offset, baseline_f_value,
    binsize):
326     #X_estimate = np.copy(path)
327     #print("Setting X_estimate = path for posterior F")
328 
329     if N_inducing_points == N_plotgridpoints:
330         ##################################################
331         # Find posterior prediction of log tuning curve #
332         ##################################################
333 
334         # Inducing points (g refers to inducing points. Originally u did.)
335         x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points)
```

```
336
337         # K_xg = K_fu
338         K_xg = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
339         K_gx = K_xg.T
340
341         # K_gg = K_uu and stands for inducing points
342         K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((
    N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)),
    sigma_f_fit, delta_f_fit)
343         # Adding tiny jitter term to diagonal of K_gg (not the same as sigma_n
    that we're adding to the diagonal of K_xgK_gg^-1K_gx later on)
344         K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points)
345         K_gg_inverse = np.linalg.inv(K_gg)
346
347         # Plot K_gg inverse
348         fig, ax = plt.subplots()
349         kxmat = ax.matshow(K_gg_inverse, cmap=plt.cm.Blues)
350         fig.colorbar(kxmat, ax=ax)
351         plt.title("K_gg_inverse")
352         plt.tight_layout()
353         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-
    K_gg_inverse.png")
354
355         # Infer mean on the grid
356         smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_gx, K_xg))
357         Q_xx_plus_sigma_inverse = sigma_n**-2 * np.identity(T) - sigma_n**-2 *
    np.matmul(np.matmul(K_xg, smallinverse), K_gx)
358         Kxx_times_F = np.matmul(Q_xx_plus_sigma_inverse, F_estimate.T)
359         #mu_posterior = np.matmul(Q_plotgrid_x, Kxx_times_F) # Here we have Kx
    crossover. Check what happens if swapped with Q = KKK
360         mu_posterior = np.matmul(K_gx, Kxx_times_F)
361
362         # Calculate standard deviations
363         #sigma_posterior = K_plotgrid_plotgrid - np.matmul(Q_plotgrid_x, np.
    matmul(Q_xx_plus_sigma_inverse, Q_x_plotgrid))
364         sigma_posterior = K_gg - np.matmul(K_gx, np.matmul(
    Q_xx_plus_sigma_inverse, K_xg))
365
366     else:
367         # If the number of plotgridpoints is different from inducing points, we
    do this:
368         ## A new grid is introduced here for plotting
369
370         ##################################################
371         # Find posterior prediction of log tuning curve #
372         ##################################################
373
374         # Inducing points (g efers to inducing points. Originally u did.)
375         x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points)
376
377         # K_xg = K_fu
378         K_xg = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
379         K_gx = K_xg.T
380
381         # K_gg = K_uu and stands for inducing points
```

```
382     K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
383     # Adding tiny jitter term to diagonal of K_gg (not the same as sigma_n that we're adding to the diagonal of K_xgK_gg^-1K_gx later on)
384     K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points)
385     K_gg_inverse = np.linalg.inv(K_gg)
386 
387     # Plot K_gg inverse
388     fig, ax = plt.subplots()
389     kxmat = ax.matshow(K_gg_inverse, cmap=plt.cm.Blues)
390     fig.colorbar(kxmat, ax=ax)
391     plt.title("K_gg_inverse")
392     plt.tight_layout()
393     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-K_gg_inverse.png")
394 
395     # Connect x to plotgrid through inducing points
396     K_g_plotgrid = squared_exponential_covariance(x_grid_induce.reshape((N_inducing_points,1)),x_grid_for_plotting.reshape((N_plotgridpoints,1)), sigma_f_fit, delta_f_fit)
397     K_plotgrid_g = K_g_plotgrid.T
398 
399     # Plot K_g_plotgrid
400     fig, ax = plt.subplots()
401     kx_cross_mat = ax.matshow(K_g_plotgrid, cmap=plt.cm.Blues)
402     fig.colorbar(kx_cross_mat, ax=ax)
403     plt.title("K_g_plotgrid")
404     plt.tight_layout()
405     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-K_g_plotgrid.png")
406     print("Making spatial covariance matrice: Kx grid")
407 
408     K_plotgrid_plotgrid = squared_exponential_covariance(x_grid_for_plotting.reshape((N_plotgridpoints,1)),x_grid_for_plotting.reshape((N_plotgridpoints,1)), sigma_f_fit, delta_f_fit)
409 
410     # Plot K_plotgrid_plotgrid
411     fig, ax = plt.subplots()
412     kxmat = ax.matshow(K_plotgrid_plotgrid, cmap=plt.cm.Blues)
413     fig.colorbar(kxmat, ax=ax)
414     plt.title("K_plotgrid_plotgrid")
415     plt.tight_layout()
416     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-K_plotgrid_plotgrid.png")
417 
418     Q_plotgrid_x = np.matmul(np.matmul(K_plotgrid_g, K_gg_inverse), K_gx)
419     Q_x_plotgrid = Q_plotgrid_x.T
420 
421     # Infer mean on the grid
422     smallinverse = np.linalg.inv(K_gg*sigma_n**2 + np.matmul(K_gx, K_xg))
423     Q_xx_plus_sigma_inverse = sigma_n**-2 * np.identity(T) - sigma_n**-2 * np.matmul(np.matmul(K_xg, smallinverse), K_gx)
424     Kxx_times_F = np.matmul(Q_xx_plus_sigma_inverse, F_estimate.T)
425     mu_posterior = np.matmul(Q_plotgrid_x, Kxx_times_F) # Here we have Kx crossover. Check what happens if swapped with Q = KKK
426 
427     # Calculate standard deviations
```

```
        sigma_posterior = K_plotgrid_plotgrid - np.matmul(Q_plotgrid_x, np.matmul(Q_xx_plus_sigma_inverse, Q_x_plotgrid))
        #########################################
        ### End of special treatment for different n.o. plotgridpoints ###
        ##################################################################

    # Plot posterior covariance matrix
    fig, ax = plt.subplots()
    sigma_posteriormat = ax.matshow(sigma_posterior, cmap=plt.cm.Blues)
    fig.colorbar(sigma_posteriormat, ax=ax)
    plt.title("Posterior covariance matrix")
    plt.tight_layout()
    plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-sigma_posterior.png")

    ###############################################
    # Plot tuning curve with confidence intervals #
    ###############################################
    standard_deviation = [np.sqrt(np.diag(sigma_posterior))]
    print("posterior marginal standard deviation:\n",standard_deviation[0])
    standard_deviation = np.repeat(standard_deviation, N, axis=0)
    upper_confidence_limit = mu_posterior + 1.96*standard_deviation.T
    lower_confidence_limit = mu_posterior - 1.96*standard_deviation.T

    if LIKELIHOOD_MODEL == "bernoulli":
        h_estimate = np.divide( np.exp(mu_posterior), (1 + np.exp(mu_posterior)))
        h_upper_confidence_limit = np.exp(upper_confidence_limit) / (1 + np.exp(upper_confidence_limit))
        h_lower_confidence_limit = np.exp(lower_confidence_limit) / (1 + np.exp(lower_confidence_limit))
    if LIKELIHOOD_MODEL == "poisson":
        h_estimate = np.exp(mu_posterior)
        h_upper_confidence_limit = np.exp(upper_confidence_limit)
        h_lower_confidence_limit = np.exp(lower_confidence_limit)

    mu_posterior = mu_posterior.T
    h_estimate = h_estimate.T
    h_upper_confidence_limit = h_upper_confidence_limit.T
    h_lower_confidence_limit = h_lower_confidence_limit.T

    ## Find true rate on plotgrid
    #if len(peak_lambda_array) > 1:
    #    print("NBNB! Take care which peak_lambda posterior F are found for!!!")
    #peak_lambda_global = peak_lambda_array[-1]
    #peak_f_offset = np.log(peak_lambda_global) - baseline_f_value

    ## ONLY FOR SIMULATED DATA THAT HAS A BUMPFUNCTION
    #true_plot_f = np.zeros((N, N_plotgridpoints))
    #for i in range(N):
    #    for t in range(N_plotgridpoints):
    #        true_plot_f[i,t] = bumptuningfunction(x_grid_for_plotting[t], i, peak_f_offset)
    #true_expectation = np.exp(true_plot_f) #poisson

    ## Find observed firing rate
    observed_mean_spikes_in_bins = zeros((N, N_plotgridpoints))
    for i in range(N):
```

```
480         for x in range(N_plotgridpoints):
481             timesinbin = (path>bins_for_plotting[x])*(path<bins_for_plotting[x
    +1])
482             if(sum(timesinbin)>0):
483                 observed_mean_spikes_in_bins[i,x] = mean( y_spikes[i, timesinbin
    ] )
484             elif i==0:
485                 print("No observations of X between",bins_for_plotting[x],"and",
    bins_for_plotting[x+1],".")
486     for i in range(N):
487         max_firing_rate_per_bin = math.ceil(max(1, 1.05*max(
    observed_mean_spikes_in_bins[i,:]), 1.05*max(h_estimate[i,:])))
488         max_firing_rate_per_second = int(max_firing_rate_per_bin / binsize)
489         plt.figure()
490         plt.plot(x_grid_for_plotting, observed_mean_spikes_in_bins[i,:], color=
    plt.cm.viridis(0.1), label="Observed average")
491         #plt.plot(x_grid_for_plotting, true_expectation[i,:], color=plt.cm.
    viridis(0.3), label="True expectation")
492         plt.plot(x_grid_for_plotting, h_estimate[i,:], color=plt.cm.viridis(0.5)
    , label="Estimated expectation")
493         plt.plot(x_grid_for_plotting, h_lower_confidence_limit[i,:], "--", color
    =plt.cm.viridis(0.5))
494         plt.plot(x_grid_for_plotting, h_upper_confidence_limit[i,:], "--", color
    =plt.cm.viridis(0.5))
495         #plt.plot(x_grid_for_plotting, mu_posterior[i,:], color=plt.cm.viridis
    (0.5))
496         #plt.title("Expected and average number of spikes, neuron "+str(i)) #
    spikes
497         plt.title("Neuron "+str(i)+" with "+str(int(sum(y_spikes[i,:])))+"
    spikes")
498         plt.yticks(range(0,1+max_firing_rate_per_bin))
499         plt.ylim(ymin=0., ymax=max(1, 1.05*max_firing_rate_per_bin))
500         #plt.yticks([0, max(1, 1.05*max(observed_mean_spikes_in_bins[i,:]),
    1.05*max(h_estimate[i,:]))])
501         plt.xlabel("x")
502         plt.ylabel("Number of spikes")
503         plt.legend()
504         plt.tight_layout()
505         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-
    tuning-"+str(i)+".png")
506 
507     # Plot observed tuning for all neurons together
508     colors = [plt.cm.viridis(t) for t in np.linspace(0, 1, N)]
509     plt.figure()
510     for i in range(N):
511         plt.plot(x_grid_for_plotting, observed_mean_spikes_in_bins[i,:], color=
    colors[i])
512     #    plt.plot(x_grid_for_plotting, h_estimate[neuron[i,j],:], color=plt.cm.
    viridis(0.5))
513         plt.xlabel("x")
514         plt.ylabel("Average number of spikes")
515     plt.tight_layout()
516     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-posterior-f-infrence-tuning-
    collected.png")
517     #plt.show()
```

**Listing B.1:** function_library.py

## B.2 Application to head direction dataset

The required function call is:

```
python em-algorithm.py Mouse12-120806_stuff_simple_awakedata.mat
```

**parameter_file_peyrache.py**

```python
from scipy import *
import scipy.io
import scipy.ndimage
import numpy as np
import scipy.optimize as spoptim
import numpy.random
import matplotlib
#matplotlib.use('Agg') # When running on cluster, plots cannot be shown and this
    must be used
import matplotlib.pyplot as plt
import time
import sys
plt.rc('image', cmap='viridis')
from scipy import optimize
numpy.random.seed(13)
from multiprocessing import Pool
from sklearn.decomposition import PCA

################################################
# Parameters for inference, not for generating #
################################################
T = 5000 #5000 #2000 # Max time 85504

N = 16 # Total of 73 neurons
       # Time offset 70400:   cutoff_spike_number 10:19   50:16  100:16   200:14
       # Time offset 0: downsample 2:  cutoff_spike_number 30-50:16   100:15
       #                downsample 1:  cutoff_spike_number 30:16   50:15
print("N =", N, "but take care that it must be changed manually if neuron
   screening settings are changed")

N_iterations = 50
global_initial_sigma_n = 2.5
sigma_n = np.copy(global_initial_sigma_n) # Assumed variance of observations for
    the GP that is fitted. 10e-5
lr = 0.99 # Learning rate by which we multiply sigma_n at every iteration

# Parameters for data loading    #
downsampling_factor = 1 #supreme: 2
offset = 0 #3700 #0 #70400 # 0 is good and wraps around a lot #64460 (not so
   good) #68170 (getting stuck lower in middle) # 70400 (supreme)

RECONVERGE_IF_FLIPPED = False
KEEP_PATH_BETWEEN_ZERO_AND_TWO_PI = True
INFER_F_POSTERIORS = True
GRADIENT_FLAG = True # Set True to use analytic gradient
USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION = False
USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE = True
USE_ONLY_OFFSET_AFTER_CONVERGENCE = False
TOLERANCE = 1e-5
```

```
46 X_initialization = "pca" #"true" "true_noisy" "ones" "pca" "randomrandom" "flat"
       "flatrandom" "randomprior" "linspace" "supreme"
47 smoothingwindow_for_PCA = 4
48 PCA_TYPE = "1d" #"angle" "1d"
49 USE_ENTIRE_DATA_LENGTH_FOR_PCA_INITIALIZATION = False
50 LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE = False # If False, they stay
      at (min_inducing_point, max_inducing_point)
51 FLIP_AFTER_SOME_ITERATION = False
52 FLIP_AFTER_HOW_MANY = 1
53 NOISE_REGULARIZATION = False
54 SMOOTHING_REGULARIZATION = False
55 GIVEN_TRUE_F = False
56 SPEEDCHECK = False
57 OPTIMIZE_HYPERPARAMETERS = False
58 PLOTTING = True
59 LIKELIHOOD_MODEL = "poisson" # "bernoulli" "poisson"
60 COVARIANCE_KERNEL_KX = "periodic" # "periodic" "nonperiodic"
61 PLOT_GRADIENT_CHECK = False
62 N_inducing_points = 30 # Number of inducing points. Wu uses 25 in 1D and 10 per
      dim in 2D
63 N_plotgridpoints = 30 # Number of grid points for plotting f posterior only
64 sigma_f_fit = 8 # Variance for the tuning curve GP that is fitted. 8
65 delta_f_fit = 0.5 # Scale for the tuning curve GP that is fitted. 0.3
66 min_inducing_point = 0
67 max_inducing_point = 2*np.pi
68 # For inference:
69 sigma_x = 5 # Variance of X for K_t
70 delta_x = 50 # Scale of X for K_t
71 jitter_term = 1e-3 #Nonperiodic 1e-5
72 cutoff_spike_number = 30 # How many spikes a neuron must produce in chosen time
      interval for us to include it
73
74 if (COVARIANCE_KERNEL_KX == "periodic") and (downsampling_factor != 1):
75     sys.exit("Don't downsample when there is data that wraps around!")
76 print("-- using Peyrache parameter file --")
```

**Listing B.2:** parameter_file_peyrache.py

**em-algorithm-peyrache-data.py**

```
1 from scipy import *
2 import scipy.io
3 import scipy.ndimage
4 import numpy as np
5 import scipy.optimize as spoptim
6 import numpy.random
7 import matplotlib
8 #matplotlib.use('Agg') # When running on cluster, plots cannot be shown and this
       must be used
9 import matplotlib.pyplot as plt
10 import time
11 import sys
12 plt.rc('image', cmap='viridis')
13 from scipy import optimize
14 numpy.random.seed(13)
15 from multiprocessing import Pool
16 from sklearn.decomposition import PCA
17 #from parameter_file_peyrache import * # where all the parameters are set (Not
      needed because importing in function library)
```

```
18 from function_library import * # loglikelihoods, gradients, covariance functions
   , tuning curve definitions, posterior tuning curve inference
19
20 ##### Inferring actual HD in Peyrache data #####
21
22 ## History:
23 ## Formerly known as em-algorithm-peyrache-data.py
24 ## Made before parallel-robustness-evaluation.py
25 ## 16.06: Incorporate changes from parallel, apply to Peyrache data
26
27 print("Likelihood model:",LIKELIHOOD_MODEL)
28 print("Covariance kernel for Kx:", COVARIANCE_KERNEL_KX)
29 print("Using gradient?", GRADIENT_FLAG)
30 print("Noise regulation:",NOISE_REGULARIZATION)
31 print("Initial sigma_n:", sigma_n)
32 print("Learning rate:", lr)
33 print("T:", T, "\n")
34 print("PCA smoothingwidth:", smoothingwindow_for_PCA)
35 if FLIP_AFTER_SOME_ITERATION:
36     print("NBBBB!!! We're flipping the estimate after the second iteration in
   line 600.")
37 print("Offset:", offset)
38 print("Downsampling factor:", downsampling_factor)
39 #######################################
40 ## Loading data                      ##
41 #######################################
42 ## 1) Load data variables
43 name = sys.argv[1] #'Mouse28-140313_stuff_BS0030_awakedata.mat'
44 mat = scipy.io.loadmat(name)
45 headangle = ravel(array(mat['headangle'])) # Observed head direction
46 cellspikes = array(mat['cellspikes']) # Observed spike time points
47 cellnames = array(mat['cellnames']) # Alphanumeric identifiers for cells
48 trackingtimes = ravel(array(mat['trackingtimes'])) # Time stamps of head
   direction observations
49 path = headangle
50 T_maximum = len(path)
51 #print("T_maximum", T_maximum)
52 if offset + T*downsampling_factor > T_maximum:
53     sys.exit("Combination of offset, downsampling and T places the end of path
   outside T_maximum. Choose lower T, offset or downsampling factor.")
54
55 ## 1) Remove headangles where the headangle value is NaN
56 # Spikes for Nan values are removed in step 2)
57 #print("How many NaN elements in path:", sum(np.isnan(path)))
58 whiches = np.isnan(path)
59 path = path[~whiches]
60
61 ## 1.5) Make path continuous where it moves from 0 to 2pi
62 if not KEEP_PATH_BETWEEN_ZERO_AND_TWO_PI:
63     for t in range(1,len(path)):
64         if (path[t] - path[t-1]) < - np.pi:
65             path[t:] += 2*np.pi
66         if (path[t] - path[t-1]) > np.pi:
67             path[t:] -= 2*np.pi
68
69 ## 2) Since spikes are recorded as time points, we must make a matrix with
   counts 0,1,2,3,4
70 # Here we also remove spikes that happen at NaN headangles, and then we
```

```
    downsample the spike matrix by summing over bins
71 starttime = min(trackingtimes)
72 tracking_interval = mean(trackingtimes[1:]-trackingtimes[:(-1)])
73 #print("Observation frequency for path, and binsize for initial sampling:",
    tracking_interval)
74 binsize = tracking_interval
75 nbins = len(trackingtimes)
76 #print("Number of bins for entire interval:", nbins)
77 print("Putting spikes in bins and making a matrix of it...")
78 binnedspikes = zeros((len(cellnames), nbins))
79 for i in range(len(cellnames)):
80     spikes = ravel((cellspikes[0])[i])
81     for j in range(len(spikes)):
82         # note 1ms binning means that number of ms from start is the correct
    index
83         timebin = int(floor(  (spikes[j] - starttime)/float(binsize)  ))
84         if(timebin>nbins-1 or timebin<0): # check if outside bounds of the awake
     time
85             continue
86         binnedspikes[i,timebin] += 1 # add a spike to the thing
87 
88 # Now remove spikes for NaN path values
89 binnedspikes = binnedspikes[:,~whiches]
90 # Copy entire spike data for PCA analysis before downsampling
91 entire_y_spikes = np.copy(binnedspikes)
92 # And downsample
93 binsize = downsampling_factor * tracking_interval
94 nbins = len(trackingtimes) // downsampling_factor
95 print("Bin size after downsampling: {:.2f}".format(binsize))
96 print("Number of bins for entire interval:", nbins)
97 print("Downsampling binned spikes...")
98 downsampled_binnedspikes = np.zeros((len(cellnames), nbins))
99 for i in range(len(cellnames)):
100     for j in range(nbins):
101         downsampled_binnedspikes[i,j] = sum(binnedspikes[i,downsampling_factor*j
    :downsampling_factor*(j+1)])
102 binnedspikes = downsampled_binnedspikes
103 
104 if LIKELIHOOD_MODEL == "bernoulli":
105     binnedspikes = (binnedspikes>0)*1
106 
107 ## 3) Select an interval of time and deal with downsampling
108 # We need to downsample the observed head direction when we tamper with the
    binsize (Here we chop off the end of the observations)
109 downsampled_path = np.zeros(len(path) // downsampling_factor)
110 for i in range(len(path) // downsampling_factor):
111     downsampled_path[i] = mean(path[downsampling_factor*i:downsampling_factor*(i
    +1)])
112 path = downsampled_path
113 # Then do downsampled offset
114 downsampled_offset = offset // downsampling_factor
115 path = path[downsampled_offset:downsampled_offset+T]
116 binnedspikes = binnedspikes[:,downsampled_offset:downsampled_offset+T]
117 
118 ## plot head direction for the selected interval
119 if PLOTTING:
120     if T > 100:
121         plt.figure(figsize=(10,3))
```

```
122     else:
123         plt.figure()
124     plt.plot(path, color="black", label='True X', linewidth=1) #plt.plot(path,
        '.', color='black', markersize=1.) # trackingtimes as x optional
125     #plt.plot(trackingtimes, path, '.', color='black', markersize=1.) #
        trackingtimes as x optional
126     #plt.plot(trackingtimes-trackingtimes[0], path, '.', color='black',
        markersize=1.) # trackingtimes as x optional
127     plt.xlabel("Time bin")
128     plt.ylabel("x")
129     plt.title("Head direction")
130     #plt.yticks([0,3.14,6.28])
131     plt.tight_layout()
132     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-path.png")
133
134 ## 5) Remove neurons that are not actually tuned to head direction
135 # 70400
136 active_and_strongly_tuned_from_70400_to_74400 =
        [20,21,22,23,25,26,27,29,31,33,34,35,36,37,38,39,45,53,63,68] #33 has few
        spikes
137 active_and_slightly_tuned_from_70400_to_74400 = [70,61,58,56,52,47,44,24,5,4]
138 barely_active_maybe_tuned_from_70400_to_74400 = [69,64,62,60,28,18,17,3,2]
139 active_but_not_tuned_from_70400_to_74400 = [71,67,66,15,14,13,12,11,10,1]
140 # 0
141 active_and_strongly_tuned_from_0_to_4000 =
        [20,21,22,23,24,25,26,27,29,31,35,36,37,38,39,68]  # 29 has only 97 spikes
142 active_and_slightly_tuned_from_0_to_4000 = [17,18,19,28,34,44] #34 is quite good
         just a bit all over with 139 spikes
143 active_and_maybe_tuned_from_0_to_4000 = [4,5,6,12,13,61,67,69]
144 active_but_not_tuned_from_0_to_4000 = [1,10,11,14,15,43,45,47,58,70,71]
145 # On the entire range of time, these neurons are tuned to head direction:
146 #neuronsthataretunedtoheaddirection = [   17,18,
        20,21,22,23,24,25,26,27,28,29,   31,32,34,35,36,37,38,39,68] # from my
        analysis and no spike cutoff
147 #
        [16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,            38, 47] ##
        froms tc-inference, after removing those with too few spikers
148 #neuronsthataretunedtoheaddirection =
        [17,18,19,20,21,22,23,24,25,26,27,29,31,34,35,36,38,39,68] # for presentation
149 #neuronsthataretunedtoheaddirection = [i for i in range(len(cellnames))] # all
        of them
150 #neuronsthataretunedtoheaddirection =
        [17,18,19,20,21,22,23,24,25,26,27,28,29,30,31,32,34,35,36,37,38,39,47,68] #
        best of both worlds?
151 sgood = np.zeros(len(cellnames))<1
152 #for i in range(len(cellnames)): # Threshold value of 1000
153 #    if sum(entire_y_spikes[i,:]) < 1000:
154 #        sgood[i] = False
155 for i in range(len(cellnames)):
156     #print("Neuron", i, "has" sum(binnedspikes[i,:]), "spikes in chosen interval
        ")
157     if offset == 0:
158         if ((i not in active_and_strongly_tuned_from_0_to_4000) or (sum(
        binnedspikes[i,:]) < cutoff_spike_number)):
159             sgood[i] = False
160     elif offset == 70400:
161         if ((i not in active_and_strongly_tuned_from_70400_to_74400) or (sum(
        binnedspikes[i,:]) < cutoff_spike_number)):
```

```
162             sgood[i] = False
163     else:
164         if ((i not in active_and_strongly_tuned_from_70400_to_74400) or (sum(
    binnedspikes[i,:]) < cutoff_spike_number)):
165             sgood[i] = False
166 binnedspikes = binnedspikes[sgood,:]
167 cellnames = cellnames[sgood]
168 print("Cutoff value:", cutoff_spike_number)
169 print("len(cellnames) after removing less active neurons:",len(cellnames))
170 
171 # Plot binned spikes for selected neurons in the selected interval (Bernoulli
    style since they are binned)
172 bernoullispikes = (binnedspikes>0)*1
173 if PLOTTING:
174     plt.figure(figsize=(5,4))
175     for i in range(len(cellnames)):
176         plt.plot(bernoullispikes[i,:]*(i+1), '|', color='black', markersize=2.)
177         plt.ylabel("neuron")
178         plt.xlabel("Time bin")
179     plt.ylim(ymin=0.5)
180     plt.yticks(range(1,len(cellnames)+1))
181     #plt.yticks([9*i+1 for i in range(0,9)])
182     plt.tight_layout()
183     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-binnedspikes.png",
    format="png")
184 ## 6) Change names to fit the rest of the code
185 #N = len(cellnames) #51 with cutoff at 1000 spikes
186 #print("N:",N)
187 if len(cellnames) != N:
188     sys.exit("N must be set equal to " + str(len(cellnames)) + " in
    parameter_file_peyrache")
189 y_spikes = binnedspikes
190 if PLOTTING:
191     plt.figure()
192     plt.title("Total number of spikes in bin")
193     plt.xlabel("Time bin")
194     plt.plot(sum(y_spikes, axis=0))
195     plt.tight_layout()
196     #plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-spikesum.png",
    format="png")
197 print("mean(y_spikes)",mean(y_spikes))
198 print("mean(y_spikes>0)",mean(y_spikes[y_spikes>0]))
199 # Spike distribution evaluation
200 spike_count = np.ndarray.flatten(binnedspikes)
201 #print("This is wrong: Portion of bins with more than one spike:", sum(
    spike_count>1)/T)
202 #print("This is wrong: Portion of nonzero bins with more than one:", sum(
    spike_count>1) / sum(spike_count>0))
203 # Remove zero entries:
204 #spike_count = spike_count[spike_count>0]
205 if PLOTTING:
206     plt.figure()
207     plt.hist(spike_count, bins=np.arange(0,int(max(spike_count))+1)-0.5, log=
    True, color=plt.cm.viridis(0.3))
208     plt.ylabel("Number of bins")
209     plt.xlabel("Spike count")
210     plt.title("Spike histogram")
211     plt.xticks(range(0,int(max(spike_count)),1))
```

```
212     plt.tight_layout()
213     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-spike-histogram-log
    .png")
214
215 # Plot y spikes
216 if PLOTTING:
217     fig, ax = plt.subplots(figsize=(8,1))
218     foo_mat = ax.matshow(y_spikes) #cmap=plt.cm.Blues
219     fig.colorbar(foo_mat, ax=ax)
220     plt.title("y spikes")
221     plt.tight_layout()
222     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-y-spikes.png")
223
224 # Inducing points based on a predetermined range
225 x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points) #np.linspace(min(path), max(path), N_inducing_points)
226 print("Min and max of path:", min(path), max(path))
227
228 K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((
    N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)),
    sigma_f_fit, delta_f_fit)
229 if PLOTTING:
230     fig, ax = plt.subplots()
231     foo_mat = ax.matshow(K_gg_plain, cmap=plt.cm.Blues)
232     fig.colorbar(foo_mat, ax=ax)
233     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-kgg.png")
234 K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points)
235
236 # Grid for plotting
237 bins_for_plotting = np.linspace(0, 2*np.pi, num=N_plotgridpoints + 1)
238 x_grid_for_plotting = 0.5*(bins_for_plotting[:(-1)]+bins_for_plotting[1:])
239
240 #####################
241 # Initialize X and F #
242 #####################
243 # PCA initialization:
244 if USE_ENTIRE_DATA_LENGTH_FOR_PCA_INITIALIZATION:
245     celldata = zeros(shape(entire_y_spikes))
246     for i in range(N):
247     #    celldata[i,:] = entire_y_spikes[i,:] # not good
248         celldata[i,:] = scipy.ndimage.filters.gaussian_filter1d(entire_y_spikes[
    i,:], smoothingwindow_for_PCA) # smooth
249         #celldata[i,:] = (celldata[i,:]-mean(celldata[i,:]))/std(celldata[i,:])
                 # standardization requires at least one spike
250 else:
251     celldata = zeros(shape(y_spikes))
252     for i in range(N):
253     #    celldata[i,:] = y_spikes[i,:] # not good
254         celldata[i,:] = scipy.ndimage.filters.gaussian_filter1d(y_spikes[i,:],
    smoothingwindow_for_PCA) # smooth
255         #celldata[i,:] = (celldata[i,:]-mean(celldata[i,:]))/std(celldata[i,:])
                 # standardization requires at least one spike
256 X_pca_result_2comp = PCA(n_components=2, svd_solver='full').fit_transform(
    transpose(celldata))
257 X_pca_result_1comp = PCA(n_components=1, svd_solver='full').fit_transform(
    transpose(celldata))
258 pca_radii = np.sqrt(X_pca_result_2comp[:,0]**2 + X_pca_result_2comp[:,1]**2)
259 pca_angles = np.arccos(X_pca_result_2comp[:,0]/pca_radii)
```

```
260 if PCA_TYPE == "angle":
261     X_pca_initial = pca_angles
262 elif PCA_TYPE == "1d":
263     X_pca_initial = np.zeros(T)
264     for i in range(T):
265         X_pca_initial[i] = X_pca_result_1comp[i][0]
266 if USE_ENTIRE_DATA_LENGTH_FOR_PCA_INITIALIZATION:
267     # Crop PCA initialization to selected time interval
268     X_pca_initial = X_pca_initial[downsampled_offset:downsampled_offset+T]
269 # Scale PCA initialization to fit domain:
270 X_pca_initial -= min(X_pca_initial)
271 X_pca_initial /= max(X_pca_initial)
272 X_pca_initial *= 2*np.pi
273 X_pca_initial += 0
274 # Flip PCA initialization correctly by comparing to true path
275 X_pca_initial_flipped = 2*mean(X_pca_initial) - X_pca_initial
276 X_pca_initial_rmse = np.sqrt(sum((X_pca_initial-path)**2) / T)
277 X_pca_initial_flipped_rmse = np.sqrt(sum((X_pca_initial_flipped-path)**2) / T)
278 if X_pca_initial_flipped_rmse < X_pca_initial_rmse:
279     X_pca_initial = X_pca_initial_flipped
280     X_pca_initial -= min(X_pca_initial)
281     X_pca_initial /= max(X_pca_initial)
282     X_pca_initial *= 2*np.pi
283     X_pca_initial += 0
284 # Plot PCA initialization
285 if T > 100:
286     plt.figure(figsize=(10,3))
287 else:
288     plt.figure()
289 plt.xlabel("Time bin")
290 plt.ylabel("x")
291 plt.title("PCA initial of X")
292 plt.plot(path, color="black", label='True X', linewidth=1)
293 #plt.plot(linspace(offset, offset+T, T), path, color="black", label='True X',
        linewidth=1)
294 #plt.plot(linspace(offset, offset+T, T), X_pca_initial, label="Initial",
        linewidth=1)
295 plt.plot(X_pca_initial, label="Initial", linewidth=1)
296 plt.legend(loc="upper right")
297 plt.tight_layout()
298 plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-PCA-initial.png")
299
300
301 # Initialize X
302 np.random.seed(0)
303 if X_initialization == "true":
304     X_initial = np.copy(path)
305 if X_initialization == "true_noisy":
306     X_initial = np.copy(path) + np.pi/4*np.sin(np.linspace(0,10*np.pi,T))
307     upper_domain_limit = 2*np.pi
308     lower_domain_limit = 0
309     #X_initial = np.copy(path) + 1*np.random.multivariate_normal(np.zeros(T),
        K_t) #2*np.random.random(T) - 1
310     X_initial -= lower_domain_limit # bring X_initial to 0
311     modulo_two_pi_values = X_initial // (upper_domain_limit)
312     oddmodulos = (modulo_two_pi_values % 2).astype(bool)
313     evenmodulos = np.invert(oddmodulos)
314     # Even modulos: Adjust for being outside
```

```
    X_initial[evenmodulos] -= upper_domain_limit*modulo_two_pi_values[
   evenmodulos]
    # Odd modulos: Adjust for being outside and flip for continuity
    X_initial[oddmodulos] -= upper_domain_limit*(modulo_two_pi_values[oddmodulos
   ])
    differences = upper_domain_limit - X_initial[oddmodulos]
    X_initial[oddmodulos] = differences
    X_initial += lower_domain_limit # bring X_initial back to min value for
   tuning
if X_initialization == "ones":
    X_initial = np.ones(T)
if X_initialization == "pca":
    X_initial = X_pca_initial
if X_initialization == "randomrandom":
    X_initial = (max_inducing_point - min_inducing_point)*np.random.random(T)
if X_initialization == "randomprior":
    X_initial = (max_inducing_point - min_inducing_point)*np.random.
   multivariate_normal(np.zeros(T), K_t)
if X_initialization == "linspace":
    X_initial = np.linspace(min_inducing_point, max_inducing_point, T)
if X_initialization == "supreme":
    X_initial = np.load("X_estimate_supreme.npy")
if X_initialization == "flatrandom":
    X_initial = 1.5*np.ones(T) + 0.2*np.random.random(T)
if X_initialization == "flat":
    X_initial = 1.5*np.ones(T)

initial_rmse = np.sqrt(sum((X_initial-path)**2) / T)
print("Initial RMSE:", initial_rmse)
X_estimate = np.copy(X_initial)

if PLOTTING:
    if T > 100:
        plt.figure(figsize=(10,3))
    else:
        plt.figure()
    plt.title("Initial X")
    plt.xlabel("Time bin")
    plt.ylabel("x")
    plt.plot(path, color="black", label='True X', linewidth=1)
    plt.plot(X_initial, label='Initial', linewidth=1)
    #plt.legend(loc="upper right")
    #plt.ylim((0, 2*np.pi))
    plt.tight_layout()
    plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-initial.png")

# Initialize F
# Anscombe transform
#F_initial = 2*np.sqrt(y_spikes + 3/8)
F_initial = np.sqrt(y_spikes) - np.amax(np.sqrt(y_spikes))/2 #np.log(y_spikes +
   0.0008)
F_estimate = np.copy(F_initial)
if X_initialization == "supreme":
    print("Initializing F supremely too")
    F_initial = np.load("F_estimate_supreme.npy")
F_estimate = np.copy(F_initial)

if GIVEN_TRUE_F:
```

```
368     # Initialize F at the values given path:
369     print("Setting f hat to the estimates given the true path")
370     temp_X_estimate = np.copy(X_estimate)
371     X_estimate = np.copy(path)
372     K_xg_prev = squared_exponential_covariance(X_estimate.reshape((T,1)),
        x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
373     if LIKELIHOOD_MODEL == "bernoulli":
374         for i in range(N):
375             y_i = y_spikes[i]
376             optimization_result = optimize.minimize(fun=
        f_loglikelihood_bernoulli, x0=F_estimate[i], jac=f_jacobian_bernoulli, args=(
        sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
        #hess=f_hessian_bernoulli,
377             F_estimate[i] = optimization_result.x
378     elif LIKELIHOOD_MODEL == "poisson":
379         for i in range(N):
380             y_i = y_spikes[i]
381             optimization_result = optimize.minimize(fun=f_loglikelihood_poisson,
         x0=F_estimate[i], jac=f_jacobian_poisson, args=(sigma_n, y_i, K_xg_prev,
        K_gg), method = 'L-BFGS-B', options={'disp':False}) #hess=f_hessian_poisson,
382             F_estimate[i] = optimization_result.x
383     true_f = np.copy(F_estimate)
384     ## Plot F estimate
385     if PLOTTING:
386         fig, ax = plt.subplots(figsize=(10,1))
387         foo_mat = ax.matshow(F_estimate) #cmap=plt.cm.Blues
388         fig.colorbar(foo_mat, ax=ax)
389         plt.title("F given path")
390         plt.tight_layout()
391         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-F-optimal.png")
392     X_estimate = temp_X_estimate
393 
394 ## Plot initial f
395 if PLOTTING:
396     fig, ax = plt.subplots(figsize=(8,1))
397     foo_mat = ax.matshow(F_initial) #cmap=plt.cm.Blues
398     fig.colorbar(foo_mat, ax=ax)
399     plt.title("Initial f")
400     plt.tight_layout()
401     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-initial-f.png")
402 
403 if PLOTTING:
404     if T > 100:
405         plt.figure(figsize=(10,3))
406     else:
407         plt.figure()
408     plt.title("X estimate")
409     plt.xlabel("Time bin")
410     plt.ylabel("x")
411     plt.plot(path, color="black", label='True X', linewidth=1)
412     plt.plot(X_initial, label='Initial', linewidth=1)
413     #plt.legend(loc="upper right")
414     #plt.ylim((0, 2*np.pi))
415     plt.tight_layout()
416     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-estimate.png")
417 
418 collected_estimates = np.zeros((N_iterations, T))
419 prev_X_estimate = np.Inf
```

```
420 startalgorithmtime = time.time()
421 ### EM algorithm: Find f given X, then X given f.
422 for iteration in range(N_iterations):
423     if iteration > 0:
424         sigma_n = sigma_n * lr  # decrease the noise variance with a learning
    rate
425         if LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE:
426             x_grid_induce = np.linspace(min(X_estimate), max(X_estimate),
    N_inducing_points) # Change position of grid to position of estimate
427     K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points)
428     K_xg_prev = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
429     # Find F estimate only if we're not at the first iteration
430     if iteration == 0:
431         print("L value of initial estimate", x_posterior_no_la(X_estimate,
    sigma_n, F_estimate, K_gg, x_grid_induce))
432     if iteration > 0:
433         if LIKELIHOOD_MODEL == "bernoulli":
434             for i in range(N):
435                 y_i = y_spikes[i]
436                 optimization_result = optimize.minimize(fun=
    f_loglikelihood_bernoulli, x0=F_estimate[i], jac=f_jacobian_bernoulli, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_bernoulli,
437                 F_estimate[i] = optimization_result.x
438         elif LIKELIHOOD_MODEL == "poisson":
439             for i in range(N):
440                 y_i = y_spikes[i]
441                 optimization_result = optimize.minimize(fun=
    f_loglikelihood_poisson, x0=F_estimate[i], jac=f_jacobian_poisson, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_poisson,
442                 F_estimate[i] = optimization_result.x
443     # Find next X estimate, that can be outside (0,2pi)
444     if NOISE_REGULARIZATION:
445         X_estimate += 2*np.random.multivariate_normal(np.zeros(T), K_t) - 1
446     if SMOOTHING_REGULARIZATION and iteration < (N_iterations-1) :
447         X_estimate = scipy.ndimage.filters.gaussian_filter1d(X_estimate, 4)
448     if GRADIENT_FLAG:
449         optimization_result = optimize.minimize(fun=x_posterior_no_la, x0=
    X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-BFGS
    -B", jac=x_jacobian_no_la, options = {'disp':False})
450     else:
451         optimization_result = optimize.minimize(fun=x_posterior_no_la, x0=
    X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-BFGS
    -B", options = {'disp':False})
452     X_estimate = optimization_result.x
453     if (iteration == (FLIP_AFTER_HOW_MANY - 1)) and FLIP_AFTER_SOME_ITERATION:
454         # Flipping estimate after iteration 1 has been plotted
455         X_estimate = 2*mean(X_estimate) - X_estimate
456     if USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION:
457         X_estimate -= min(X_estimate) #set offset of min to 0
458         X_estimate /= max(X_estimate) #scale length to 1
459         X_estimate *= (max(path)-min(path)) #scale length to length of path
460         X_estimate += min(path) #set offset to offset of path
461     if PLOTTING:
462         plt.plot(X_estimate, label='Estimate', linewidth=1)
463         #plt.ylim((min_neural_tuning_X, max_neural_tuning_X))
```

```
        plt.tight_layout()
        plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-estimate.png"
    )
    if np.linalg.norm(X_estimate - prev_X_estimate) < TOLERANCE:
        break
    prev_X_estimate = X_estimate
    #np.save("X_estimate", X_estimate)
print("Time used:", time.time()-startalgorithmtime)
if USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE:
    X_estimate -= min(X_estimate) #set offset of min to 0
    X_estimate /= max(X_estimate) #scale length to 1
    X_estimate *= (max(path)-min(path)) #scale length to length of path
    X_estimate += min(path) #set offset to offset of path
if USE_ONLY_OFFSET_AFTER_CONVERGENCE:
    X_estimate -= np.mean(X_estimate)
    X_estimate += np.mean(path)
# Flipped
X_flipped = - X_estimate + 2*mean(X_estimate)
# Rootmeansquarederror for X
X_rmse = np.sqrt(sum((X_estimate-path)**2) / T)
X_flipped_rmse = np.sqrt(sum((X_flipped-path)**2) / T)
##### Check if flipped and maybe iterate again with flipped estimate
if X_flipped_rmse < X_rmse and RECONVERGE_IF_FLIPPED:
    #print("RMSE for X:", X_rmse)
    #print("RMSE for X flipped:", X_flipped_rmse)
    print("Re-iterating because of flip")
    x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points) #np.linspace(min(path), max(path), N_inducing_points)
    K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((
    N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)),
    sigma_f_fit, delta_f_fit)
    X_initial_2 = np.copy(X_flipped)
    X_estimate = np.copy(X_flipped)
    F_estimate = np.copy(F_initial)
    if GIVEN_TRUE_F:
        F_estimate = np.copy(true_f)
    if PLOTTING:
        if T > 100:
            plt.figure(figsize=(10,3))
        else:
            plt.figure()
        #plt.title("After flipping") # as we go
        plt.xlabel("Time bin")
        plt.ylabel("x")
        plt.plot(path, color="black", label='True X', linewidth=1)
        plt.plot(X_initial_2, label='Initial', linewidth=1)
        #plt.ylim((min_neural_tuning_X, max_neural_tuning_X))
        plt.tight_layout()
        plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-flipped.png")
    ##############################
    # EM after flipped           #
    ##############################
    prev_X_estimate = np.Inf
    sigma_n = np.copy(global_initial_sigma_n)
    for iteration in range(N_iterations):
        if iteration > 0:
            sigma_n = sigma_n * lr  # decrease the noise variance with a
    learning rate
```

```
517            if LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE:
518                x_grid_induce = np.linspace(min(X_estimate), max(X_estimate),
    N_inducing_points) # Change position of grid to position of estimate
519        K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points)
520        K_xg_prev = squared_exponential_covariance(X_estimate.reshape((T,1)),
    x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
521        # Find F estimate only if we're not at the first iteration
522        if iteration > 0:
523            if LIKELIHOOD_MODEL == "bernoulli":
524                for i in range(N):
525                    y_i = y_spikes[i]
526                    optimization_result = optimize.minimize(fun=
    f_loglikelihood_bernoulli, x0=F_estimate[i], jac=f_jacobian_bernoulli, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_bernoulli,
527                    F_estimate[i] = optimization_result.x
528            elif LIKELIHOOD_MODEL == "poisson":
529                for i in range(N):
530                    y_i = y_spikes[i]
531                    optimization_result = optimize.minimize(fun=
    f_loglikelihood_poisson, x0=F_estimate[i], jac=f_jacobian_poisson, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_poisson,
532                    F_estimate[i] = optimization_result.x
533        # Find next X estimate, that can be outside (0,2pi)
534        if NOISE_REGULARIZATION:
535            X_estimate += 2*np.random.multivariate_normal(np.zeros(T), K_t) - 1
536        if SMOOTHING_REGULARIZATION and iteration < (N_iterations-1) :
537            X_estimate = scipy.ndimage.filters.gaussian_filter1d(X_estimate, 4)
538        if GRADIENT_FLAG:
539            optimization_result = optimize.minimize(fun=x_posterior_no_la, x0=
    X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-BFGS
    -B", jac=x_jacobian_no_la, options = {'disp':False})
540        else:
541            optimization_result = optimize.minimize(fun=x_posterior_no_la, x0=
    X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-BFGS
    -B", options = {'disp':False})
542        X_estimate = optimization_result.x
543        if (iteration == (FLIP_AFTER_HOW_MANY - 1)) and
    FLIP_AFTER_SOME_ITERATION:
544            # Flipping estimate after iteration 1 has been plotted
545            X_estimate = 2*mean(X_estimate) - X_estimate
546        if USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION:
547            X_estimate -= min(X_estimate) #set offset of min to 0
548            X_estimate /= max(X_estimate) #scale length to 1
549            X_estimate *= (max(path)-min(path)) #scale length to length of path
550            X_estimate += min(path) #set offset to offset of path
551        if PLOTTING:
552            plt.plot(X_estimate, label='Estimate (after flip)', linewidth=1)
553            #plt.ylim((min_neural_tuning_X, max_neural_tuning_X))
554            plt.tight_layout()
555            plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-flipped.
    png")
556        if np.linalg.norm(X_estimate - prev_X_estimate) < TOLERANCE:
557            #print("Seed", seeds[seedindex], "Iterations after flip:", iteration
    +1, "Change in X smaller than TOL")
558            break
559        #if iteration == N_iterations-1:
```

```
560         #    print("Seed", seeds[seedindex], "Iterations after flip:", iteration
    +1, "N_iterations reached")
561         prev_X_estimate = X_estimate
562     if USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE:
563         X_estimate -= min(X_estimate) #set offset of min to 0
564         X_estimate /= max(X_estimate) #scale length to 1
565         X_estimate *= (max(path)-min(path)) #scale length to length of path
566         X_estimate += min(path) #set offset to offset of path
567     if USE_ONLY_OFFSET_AFTER_CONVERGENCE:
568         X_estimate -= np.mean(X_estimate)
569         X_estimate += np.mean(path)
570     # Rootmeansquarederror for X
571     X_rmse = np.sqrt(sum((X_estimate-path)**2) / T)
572 ######################
573 #### Handle rotation #
574 ######################
575 SStot = sum((path - mean(path))**2)
576 SSdev = sum((X_estimate-path)**2)
577 Rsquared = 1 - SSdev / SStot
578 print("R squared value of X estimate:", Rsquared)
579 print("RMSE value of X estimate:", X_rmse)
580 print("L value of final estimate:", x_posterior_no_la(X_estimate, sigma_n,
    F_estimate, K_gg, x_grid_induce))
581 if PLOTTING:
582     if T > 100:
583         plt.figure(figsize=(10,3))
584     else:
585         plt.figure()
586     plt.title("Final estimate") # as we go
587     plt.xlabel("Time bin")
588     plt.ylabel("x")
589     plt.plot(path, color="black", label='True X', linewidth=1)
590     plt.plot(X_initial, label='Initial', linewidth=1)
591     plt.plot(X_estimate, label='Estimate', linewidth=1)
592     plt.legend(loc="lower right")
593     #plt.ylim((min_neural_tuning_X, max_neural_tuning_X))
594     plt.tight_layout()
595     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-peyrache-X-final-RMSE-" +
    str(X_rmse) + "-L-" + str(x_posterior_no_la(X_estimate, sigma_n, F_estimate,
    K_gg, x_grid_induce)) + ".png")
596
597 #################################################
598 # Find posterior prediction of log tuning curve #
599 #################################################
600 if INFER_F_POSTERIORS:
601     print("Sigma_n:", sigma_n)
602     bins_for_plotting = np.linspace(0, 2*np.pi, num=N_plotgridpoints + 1)
603     x_grid_for_plotting = 0.5*(bins_for_plotting[:(-1)]+bins_for_plotting[1:])
604     #posterior_f_inference(X_estimate, F_estimate, sigma_n, y_spikes, path,
    x_grid_for_plotting, bins_for_plotting, peak_f_offset, baseline_f_value,
    binsize)
605     posterior_f_inference(X_estimate, F_estimate, sigma_n, y_spikes, path,
    x_grid_for_plotting, bins_for_plotting, 0.1, 0.1, binsize) # the two latest
    are only for simulated
```

**Listing B.3:** em-algorithm-peyrache-data.py

## B.3 Robustness evaluation

To find the average RMSE value for the given $T$ value with the tuning strength corresponding to index `<index>` in the tuning difference array, the required function call is:

```
python cluster-parallel-robustness-evaluation.py <index>
```

**parameter_file_robustness.py**

```
1 from scipy import *
2 import scipy.io
3 import scipy.ndimage
4 import numpy as np
5 import scipy.optimize as spoptim
6 import numpy.random
7 import matplotlib
8 #matplotlib.use('Agg') # When running on cluster, plots cannot be shown and this
    must be used
9 import matplotlib.pyplot as plt
10 import time
11 import sys
12 plt.rc('image', cmap='viridis')
13 from scipy import optimize
14 numpy.random.seed(13)
15 from multiprocessing import Pool
16 from sklearn.decomposition import PCA
17
18 ############################
19 # Parameters for inference #
20 ############################
21 T = 1000 # change only after previous job is definitely RUNNING on cluster
22 N_iterations = 20
23
24 global_initial_sigma_n = 2.5 # Assumed variance of observations for the GP that
    is fitted. 10e-5
25 lr = 0.95 # 0.99 # Learning rate by which we multiply sigma_n at every iteration
26
27 RECONVERGE_IF_FLIPPED = False
28 INFER_F_POSTERIORS = False
29 GRADIENT_FLAG = True # Set True to use analytic gradient
30 USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION = False
31 USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE = True
32 TOLERANCE = 1e-6
33 X_initialization = "pca" #"true" "true_noisy" "ones" "pca" "randomrandom" "flat"
     "flatrandom" "randomprior" "linspace" "supreme"
34 # Using ensemble of PCA values
35 ensemble_smoothingwidths = [3,5,10]
36 LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE = False
37 FLIP_AFTER_SOME_ITERATION = False
38 FLIP_AFTER_HOW_MANY = 1
39 NOISE_REGULARIZATION = False
40 SMOOTHING_REGULARIZATION = False
41 GIVEN_TRUE_F = False
42 SPEEDCHECK = False
43 OPTIMIZE_HYPERPARAMETERS = False
44 PLOTTING = False
45 LIKELIHOOD_MODEL = "poisson" # "bernoulli" "poisson"
```

```
46 COVARIANCE_KERNEL_KX = "nonperiodic" # "periodic" "nonperiodic"
47 TUNINGCURVE_DEFINITION = "bumps" # "triangles" "bumps"
48 UNIFORM_BUMPS = False
49 PLOT_GRADIENT_CHECK = False
50 N_inducing_points = 30 # Number of inducing points. Wu uses 25 in 1D and 10 per
    dim in 2D
51 N_plotgridpoints = 40 # Number of grid points for plotting f posterior only
52 tuning_width_delta = 1.2 # 0.1
53 # Peak lambda should not be defined as less than baseline h value
54 baseline_lambda_value = 0.5
55 baseline_f_value = np.log(baseline_lambda_value)
56 tuning_difference_array =
    [0.01,0.1,0.2,0.3,0.4,0.5,0.75,1,1.25,1.5,1.75,2,2.5,3,3.5,4,5,6,7,8,9] #
    choose index 14 for a good example
57 peak_lambda_array = [baseline_lambda_value + tuning_difference_array[i] for i in
     range(len(tuning_difference_array))]
58 seeds = [5,6,7,8,9] #range(5) #[11]
    #[0,2,3,4,5,6,8,9,11,12,16,17,18,19,21,22,25,26,28,29] # chosen only so that
    they cover the entire domain of X for T>=200 and sigma_x=40
59 NUMBER_OF_SEEDS = len(seeds)
60 sigma_f_fit = 2 #8 # Variance for the tuning curve GP that is fitted. 8
61 delta_f_fit = 0.83 # sqrt(0.7) # Scale for the tuning curve GP that is fitted.
    0.3
62 # Define max and min of neural tuning
63 lower_domain_limit = 0
64 upper_domain_limit = 10
65 how_many_added_neurons_outside_factor = 0.0 # Just makes it worse. If you must,
    use 0.1
66 min_neural_tuning_X = lower_domain_limit - how_many_added_neurons_outside_factor
    *(upper_domain_limit - lower_domain_limit)
67 max_neural_tuning_X = upper_domain_limit + how_many_added_neurons_outside_factor
    *(upper_domain_limit - lower_domain_limit)
68 min_inducing_point = lower_domain_limit
69 max_inducing_point = upper_domain_limit
70 # Neural density:
71 N = int((1+2*how_many_added_neurons_outside_factor)*100) # 100 with peaks in
    tuning area and 40 with tails coming in from each side
72 # For inference:
73 sigma_x = 40 # Variance of X for inference matrix K_t
74 delta_x = 100 # Scale of X for inference matrix K_t
75 # Generative parameters for X path:
76 KEEP_PATH_INSIDE_DOMAIN_BY_FOLDING = True # Stop path from going outside defined
     domain with neurons
77 SCALE_UP_PATH_TO_COVER_DOMAIN = False # If True, the generated path is scaled up
     after being generated
78 sigma_x_generate_path = 40 # Variance for path generation. Set high enough so
    the path reaches max and min of tuning area
79 delta_x_generate_path = 100 # Scale for path generation.
80 jitter_term = 1e-5
81
82 print("-- using Robustness evaluation parameter file --")
```

**Listing B.4:** parameter_file_robustness.py

**cluster-parallel-robustness-evaluation.py**

```
1 from scipy import *
2 import scipy.io
3 import scipy.ndimage
```

```
import numpy as np
import scipy.optimize as spoptim
import numpy.random
import matplotlib
#matplotlib.use('Agg') # When running on cluster, plots cannot be shown and this
    must be used
import matplotlib.pyplot as plt
import time
import sys
plt.rc('image', cmap='viridis')
from scipy import optimize
numpy.random.seed(13)
from multiprocessing import Pool
from sklearn.decomposition import PCA
#from parameter_file_robustness import * # where all the parameters are set (Not
    needed because importing in function library)
from function_library import * # loglikelihoods, gradients, covariance functions
   , tuning curve definitions, posterior tuning curve inference

###########################################
##### Cluster - Robustness evaluation #####
###########################################

## Set T and background noise level
## Array of 21 lambda peak strengths is done in parallel using job-array
## For each lambda peak strength: Run 20 seeds sequentially
## For each seed, the best RMSE is taken from an ensemble of 3-5 initializations
    with different wmoothingwindow in the PCA (run sequentially)

## History:
## Branched off from em-algorithm on 11.05.2020
## and from robust-sim-data on 28.05.2020
## then from robust-efficient-script on 30.05.2020
## then from parallel-robustness-evaluation.py on 18.06.2020

######################################
## Data generation                  ##
######################################
K_t_generate = exponential_covariance(np.linspace(1,T,T).reshape((T,1)),np.
   linspace(1,T,T).reshape((T,1)), sigma_x_generate_path, delta_x_generate_path)

############################
# Tuning curve definitions #
############################

if UNIFORM_BUMPS:
    # Uniform positioning and width:'
    bumplocations = [min_neural_tuning_X + (i+0.5)/N*(max_neural_tuning_X -
   min_neural_tuning_X) for i in range(N)]
    bump_delta_distances = tuning_width_delta * np.ones(N)
else:
    # Random placement and width:
    bumplocations = min_neural_tuning_X + (max_neural_tuning_X -
   min_neural_tuning_X) * np.random.random(N)
    bump_delta_distances = tuning_width_delta + tuning_width_delta/4*np.random.
   random(N)

def bumptuningfunction(x, i, peak_f_offset):
```

```
54      x1 = x
55      x2 = bumplocations[i]
56      delta_bumptuning = bump_delta_distances[i]
57      if COVARIANCE_KERNEL_KX == "periodic":
58          distancesquared = min([(x1-x2)**2, (x1+2*pi-x2)**2, (x1-2*pi-x2)**2])
59      elif COVARIANCE_KERNEL_KX == "nonperiodic":
60          distancesquared = (x1-x2)**2
61      return baseline_f_value + peak_f_offset * exp(-distancesquared/(2*
    delta_bumptuning))
62
63  def offset_function(offset_for_estimate, X_estimate, sigma_n, F_estimate, K_gg,
    x_grid_induce):
64      offset_estimate = X_estimate + offset_for_estimate
65      return x_posterior_no_la(offset_estimate, sigma_n, F_estimate, K_gg,
    x_grid_induce)
66
67  def scaling_function(scaling_factor, X_estimate, sigma_n, F_estimate, K_gg,
    x_grid_induce):
68      scaled_estimate = scaling_factor*X_estimate
69      return x_posterior_no_la(scaled_estimate, sigma_n, F_estimate, K_gg,
    x_grid_induce)
70
71  def scale_and_offset_function(scale_offset, X_estimate, sigma_n, F_estimate,
    K_gg, x_grid_induce):
72      scaled_estimate = scale_offset[0] * X_estimate + scale_offset[1]
73      return x_posterior_no_la(scaled_estimate, sigma_n, F_estimate, K_gg,
    x_grid_induce)
74      #return just_fprior_term(scaled_estimate)
75
76  ######################################
77  ## RMSE function                    ##
78  ######################################
79  def find_rmse_for_this_lambda_this_seed(seedindex):
80      global lower_domain_limit
81      global upper_domain_limit
82      starttime = time.time()
83      #print("Seed", seeds[seedindex], "started.")
84      peak_f_offset = np.log(peak_lambda_global) - baseline_f_value
85      np.random.seed(seeds[seedindex])
86      # Generate path
87      path = (upper_domain_limit-lower_domain_limit)/2 + numpy.random.
    multivariate_normal(np.zeros(T), K_t_generate)
88      #path = np.linspace(lower_domain_limit, upper_domain_limit, T)
89      if KEEP_PATH_INSIDE_DOMAIN_BY_FOLDING:
90          # Use boolean masks to keep X within min and max of tuning
91          path -= lower_domain_limit # bring path to 0
92          modulo_two_pi_values = path // (upper_domain_limit)
93          oddmodulos = (modulo_two_pi_values % 2).astype(bool)
94          evenmodulos = np.invert(oddmodulos)
95          # Even modulos: Adjust for being outside
96          path[evenmodulos] -= upper_domain_limit*modulo_two_pi_values[evenmodulos
    ]
97          # Odd modulos: Adjust for being outside and flip for continuity
98          path[oddmodulos] -= upper_domain_limit*(modulo_two_pi_values[oddmodulos
    ])
99          differences = upper_domain_limit - path[oddmodulos]
100         path[oddmodulos] = differences
101         path += lower_domain_limit # bring path back to min value for tuning
```

```
102     if SCALE_UP_PATH_TO_COVER_DOMAIN:
103         # scale to cover the domain:
104         path -= min(path)
105         path /= max(path)
106         path *= (upper_domain_limit-lower_domain_limit)
107         path += lower_domain_limit
108     if PLOTTING:
109         ## plot path
110         if T > 100:
111             plt.figure(figsize=(10,3))
112         else:
113             plt.figure()
114         plt.plot(path, color="black", label='True X')
115         #plt.plot(path, '.', color='black', markersize=1.) # trackingtimes as x
    optional
116         #plt.plot(trackingtimes-trackingtimes[0], path, '.', color='black',
    markersize=1.) # trackingtimes as x optional
117         plt.xlabel("Time bin")
118         plt.ylabel("x")
119         plt.title("True path of X")
120         plt.ylim((lower_domain_limit, upper_domain_limit))
121         #plt.title("Simulated path of X")
122         #plt.yticks([-15,-10,-5,0,5,10,15])
123         plt.tight_layout()
124         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-T-" +
    str(T)  + "-seed-" + str(seeds[seedindex]) + "-path.png")
125     ## Generate spike data. True tuning curves are defined here
126     if TUNINGCURVE_DEFINITION == "triangles":
127         tuningwidth = 1 # width of tuning (in radians)
128         biasterm = -2 # Average H outside tuningwidth -4
129         tuningcovariatestrength = np.linspace(0.5*tuningwidth,10.*tuningwidth, N
    ) # H value at centre of tuningwidth 6*tuningwidth
130         neuronpeak = [min_neural_tuning_X + (i+0.5)/N*(max_neural_tuning_X -
    min_neural_tuning_X) for i in range(N)]
131         true_f = np.zeros((N, T))
132         y_spikes = np.zeros((N, T))
133         for i in range(N):
134             for t in range(T):
135                 if COVARIANCE_KERNEL_KX == "periodic":
136                     distancefrompeaktopathpoint = min([ abs(neuronpeak[i]+2.*pi-
    path[t]),  abs(neuronpeak[i]-path[t]),  abs(neuronpeak[i]-2.*pi-path[t]) ])
137                 elif COVARIANCE_KERNEL_KX == "nonperiodic":
138                     distancefrompeaktopathpoint = abs(neuronpeak[i]-path[t])
139                 Ht = biasterm
140                 if(distancefrompeaktopathpoint < tuningwidth):
141                     Ht = biasterm + tuningcovariatestrength[i] * (1-
    distancefrompeaktopathpoint/tuningwidth)
142                 true_f[i,t] = Ht
143                 # Spiking
144                 if LIKELIHOOD_MODEL == "bernoulli":
145                     spike_probability = exp(Ht)/(1.+exp(Ht))
146                     y_spikes[i,t] = 1.0*(rand()<spike_probability)
147                     # If you want to remove randomness: y_spikes[i,t] =
    spike_probability
148                 elif LIKELIHOOD_MODEL == "poisson":
149                     spike_rate = exp(Ht)
150                     y_spikes[i,t] = np.random.poisson(spike_rate)
151                     # If you want to remove randomness: y_spikes[i,t] =
```

```
    spike_rate
152     elif TUNINGCURVE_DEFINITION == "bumps":
153         true_f = np.zeros((N, T))
154         y_spikes = np.zeros((N, T))
155         for i in range(N):
156             for t in range(T):
157                 true_f[i,t] = bumptuningfunction(path[t], i, peak_f_offset)
158                 if LIKELIHOOD_MODEL == "bernoulli":
159                     spike_probability = exp(true_f[i,t])/(1.+exp(true_f[i,t]))
160                     y_spikes[i,t] = 1.0*(rand()<spike_probability)
161                 elif LIKELIHOOD_MODEL == "poisson":
162                     spike_rate = exp(true_f[i,t])
163                     y_spikes[i,t] = np.random.poisson(spike_rate)
164     if PLOTTING:
165         ## Plot true f in time
166         plt.figure()
167         color_idx = np.linspace(0, 1, N)
168         plt.title("True log tuning curves f")
169         plt.xlabel("x")
170         plt.ylabel("f value")
171         x_space_grid = np.linspace(lower_domain_limit, upper_domain_limit, T)
172         for i in range(N):
173             plt.plot(x_space_grid, true_f[i], linestyle='-', color=plt.cm.
    viridis(color_idx[i]))
174         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-true-f.
    png")
175     if PLOTTING:
176         ## Plot firing rate h in time
177         plt.figure()
178         color_idx = np.linspace(0, 1, N)
179         plt.title("True firing rate h")
180         plt.xlabel("x")
181         plt.ylabel("Firing rate")
182         x_space_grid = np.linspace(lower_domain_limit, upper_domain_limit, T)
183         for i in range(N):
184             plt.plot(x_space_grid, np.exp(true_f[i]), linestyle='-', color=plt.
    cm.viridis(color_idx[i]))
185         plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-true-h.
    png")
186     ###############################
187     # Covariance matrix Kgg_plain #
188     ###############################
189     # Inducing points based on a predetermined range
190     x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points) #np.linspace(min(path), max(path), N_inducing_points)
191     #print("Min and max of path:", min(path), max(path))
192     #print("Min and max of grid:", min(x_grid_induce), max(x_grid_induce))
193     K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((
    N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)),
    sigma_f_fit, delta_f_fit)
194     ######################
195     # Initialize X and F #
196     ######################
197     # Here the PCA ensemble comes into play:
198     ensemble_array_L_value = np.zeros(len(ensemble_smoothingwidths))
199     ensemble_array_X_rmse = np.zeros(len(ensemble_smoothingwidths))
200     ensemble_array_X_estimate = np.zeros((len(ensemble_smoothingwidths), T))
201     ensemble_array_F_estimate = np.zeros((len(ensemble_smoothingwidths), N, T))
```

```
202    ensemble_array_y_spikes = np.zeros((len(ensemble_smoothingwidths), N, T))
203    ensemble_array_path = np.zeros((len(ensemble_smoothingwidths), T))
204    for smoothingwindow_index in range(len(ensemble_smoothingwidths)):
205        smoothingwindow_for_PCA = ensemble_smoothingwidths[smoothingwindow_index]
206        # PCA initialization:
207        celldata = zeros(shape(y_spikes))
208        for i in range(N):
209            celldata[i,:] = scipy.ndimage.filters.gaussian_filter1d(y_spikes[i,:], smoothingwindow_for_PCA) # smooth
210            #celldata[i,:] = (celldata[i,:]-mean(celldata[i,:]))/std(celldata[i,:])                 # standardization requires at least one spike
211        X_pca_result = PCA(n_components=1, svd_solver='full').fit_transform(transpose(celldata))
212        X_pca_initial = np.zeros(T)
213        for i in range(T):
214            X_pca_initial[i] = X_pca_result[i]
215        # Scale PCA initialization to fit domain:
216        X_pca_initial -= min(X_pca_initial)
217        X_pca_initial /= max(X_pca_initial)
218        X_pca_initial *= (upper_domain_limit-lower_domain_limit)
219        X_pca_initial += lower_domain_limit
220        # Flip PCA initialization correctly by comparing to true path
221        X_pca_initial_flipped = 2*mean(X_pca_initial) - X_pca_initial
222        X_pca_initial_rmse = np.sqrt(sum((X_pca_initial-path)**2) / T)
223        X_pca_initial_flipped_rmse = np.sqrt(sum((X_pca_initial_flipped-path)**2) / T)
224        if X_pca_initial_flipped_rmse < X_pca_initial_rmse:
225            X_pca_initial = X_pca_initial_flipped
226            # Scale PCA initialization to fit domain:
227            X_pca_initial -= min(X_pca_initial)
228            X_pca_initial /= max(X_pca_initial)
229            X_pca_initial *= (upper_domain_limit-lower_domain_limit)
230            X_pca_initial += lower_domain_limit
231        if PLOTTING:
232            # Plot PCA initialization
233            if T > 100:
234                plt.figure(figsize=(10,3))
235            else:
236                plt.figure()
237            plt.xlabel("Time bin")
238            plt.ylabel("x")
239            plt.title("PCA initial of X")
240            plt.plot(path, color="black", label='True X')
241            plt.plot(X_pca_initial, label="Initial")
242            plt.legend(loc="upper right")
243            plt.tight_layout()
244            plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-T-" + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + "-PCA-initial.png")
245        # Initialize X
246        np.random.seed(0)
247        if X_initialization == "true":
248            X_initial = np.copy(path)
249        if X_initialization == "true_noisy":
250            X_initial = np.copy(path) + np.pi/4*np.sin(np.linspace(0,10*np.pi,T))
```

```
251             upper_domain_limit = 2*np.pi
252             lower_domain_limit = 0
253             #X_initial = np.copy(path) + 1*np.random.multivariate_normal(np.
    zeros(T), K_t) #2*np.random.random(T) - 1
254             X_initial -= lower_domain_limit # bring X_initial to 0
255             modulo_two_pi_values = X_initial // (upper_domain_limit)
256             oddmodulos = (modulo_two_pi_values % 2).astype(bool)
257             evenmodulos = np.invert(oddmodulos)
258             # Even modulos: Adjust for being outside
259             X_initial[evenmodulos] -= upper_domain_limit*modulo_two_pi_values[
    evenmodulos]
260             # Odd modulos: Adjust for being outside and flip for continuity
261             X_initial[oddmodulos] -= upper_domain_limit*(modulo_two_pi_values[
    oddmodulos])
262             differences = upper_domain_limit - X_initial[oddmodulos]
263             X_initial[oddmodulos] = differences
264             X_initial += lower_domain_limit # bring X_initial back to min value
    for tuning
265         if X_initialization == "ones":
266             X_initial = np.ones(T)
267         if X_initialization == "pca":
268             X_initial = X_pca_initial
269         if X_initialization == "randomrandom":
270             X_initial = (max_inducing_point - min_inducing_point)*np.random.
    random(T)
271         if X_initialization == "randomprior":
272             X_initial = (max_inducing_point - min_inducing_point)*np.random.
    multivariate_normal(np.zeros(T), K_t)
273         if X_initialization == "linspace":
274             X_initial = np.linspace(min_inducing_point, max_inducing_point, T)
275         if X_initialization == "supreme":
276             X_initial = np.load("X_estimate_supreme.npy")
277         if X_initialization == "flatrandom":
278             X_initial = 1.5*np.ones(T) + 0.2*np.random.random(T)
279         if X_initialization == "flat":
280             X_initial = 1.5*np.ones(T)
281         initial_rmse = np.sqrt(sum((X_initial-path)**2) / T)
282         print("Initial RMSE", initial_rmse)
283         X_estimate = np.copy(X_initial)
284         # Initialize F
285         F_initial = np.sqrt(y_spikes) - np.amax(np.sqrt(y_spikes))/2 #np.log(
    y_spikes + 0.0008)
286         F_estimate = np.copy(F_initial)
287         if GIVEN_TRUE_F:
288             F_estimate = true_f
289         if PLOTTING:
290             if T > 100:
291                 plt.figure(figsize=(10,3))
292             else:
293                 plt.figure()
294             #plt.title("Path of X")
295             plt.title("X estimate")
296             plt.xlabel("Time bin")
297             plt.ylabel("x")
298             plt.plot(path, color="black", label='True X')
299             plt.plot(X_initial, label='Initial')
300             #plt.legend(loc="upper right")
301             #plt.ylim((lower_domain_limit, upper_domain_limit))
```

```
302             plt.tight_layout()
303             plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-T-"
    + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(
    baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + ".png")
304         if PLOT_GRADIENT_CHECK:
305             sigma_n = np.copy(global_initial_sigma_n)
306             # Adding tiny jitter term to diagonal of K_gg (not the same as
    sigma_n that we're adding to the diagonal of K_xgK_gg^-1K_gx later on)
307             K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points) ##
    K_gg = K_gg_plain + sigma_n*np.identity(N_inducing_points)
308             X_gradient = x_jacobian_no_la(X_estimate, sigma_n, F_estimate, K_gg,
     x_grid_induce)
309             if T > 100:
310                 plt.figure(figsize=(10,3))
311             else:
312                 plt.figure()
313             plt.xlabel("Time bin")
314             plt.ylabel("x")
315             plt.title("Gradient at initial X")
316             plt.plot(path, color="black", label='True X')
317             plt.plot(X_initial, label="Initial")
318             #plt.plot(X_gradient, label="Gradient")
319             plt.plot(X_estimate + 2*X_gradient/max(X_gradient), label="Gradient
    plus offset")
320             plt.legend(loc="upper right")
321             plt.tight_layout()
322             plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-T-"
    + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(
    baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + "-Gradient.png")
323             exit()
324             """
325             print("Testing gradient...")
326             #X_estimate = np.copy(path)
327             #F_estimate = true_f
328             print("Gradient difference using check_grad:",scipy.optimize.
    check_grad(func=x_posterior_no_la, grad=x_jacobian_no_la, x0=path, args=(
    sigma_n, F_estimate, K_gg, x_grid_induce)))
329 
330             #optim_gradient = optimization_result.jac
331             print("Epsilon:", np.sqrt(np.finfo(float).eps))
332             optim_gradient1 = scipy.optimize.approx_fprime(xk=X_estimate, f=
    x_posterior_no_la, epsilon=1*np.sqrt(np.finfo(float).eps), args=(sigma_n,
    F_estimate, K_gg, x_grid_induce))
333             optim_gradient2 = scipy.optimize.approx_fprime(xk=X_estimate, f=
    x_posterior_no_la, epsilon=x_posterior_no_la, 1e-4, args=(sigma_n, F_estimate
    , K_gg, x_grid_induce))
334             optim_gradient3 = scipy.optimize.approx_fprime(xk=X_estimate, f=
    x_posterior_no_la, epsilon=x_posterior_no_la, 1e-2, args=(sigma_n, F_estimate
    , K_gg, x_grid_induce))
335             optim_gradient4 = scipy.optimize.approx_fprime(xk=X_estimate, f=
    x_posterior_no_la, epsilon=x_posterior_no_la, 1e-2, args=(sigma_n, F_estimate
    , K_gg, x_grid_induce))
336             calculated_gradient = x_jacobian_no_la(X_estimate, sigma_n,
    F_estimate, K_gg, x_grid_induce)
337             difference_approx_fprime_1 = optim_gradient1 - calculated_gradient
338             difference_approx_fprime_2 = optim_gradient2 - calculated_gradient
339             difference_approx_fprime_3 = optim_gradient3 - calculated_gradient
340             difference_approx_fprime_4 = optim_gradient4 - calculated_gradient
```

```
341            difference_norm1 = np.linalg.norm(difference_approx_fprime_1)
342            difference_norm2 = np.linalg.norm(difference_approx_fprime_2)
343            difference_norm3 = np.linalg.norm(difference_approx_fprime_3)
344            difference_norm4 = np.linalg.norm(difference_approx_fprime_4)
345            print("Gradient difference using approx f prime, epsilon 1e-8:",
    difference_norm1)
346            print("Gradient difference using approx f prime, epsilon 1e-4:",
    difference_norm2)
347            print("Gradient difference using approx f prime, epsilon 1e-2:",
    difference_norm3)
348            print("Gradient difference using approx f prime, epsilon 1e-2:",
    difference_norm4)
349            plt.figure()
350            plt.title("Gradient compared to numerical gradient")
351            plt.plot(calculated_gradient, label="Analytic")
352            #plt.plot(optim_gradient1, label="Numerical 1")
353            plt.plot(optim_gradient2, label="Numerical 2")
354            plt.plot(optim_gradient3, label="Numerical 3")
355            plt.plot(optim_gradient4, label="Numerical 4")
356            plt.legend()
357            plt.figure()
358            #plt.plot(difference_approx_fprime_1, label="difference 1")
359            plt.plot(difference_approx_fprime_2, label="difference 2")
360            plt.plot(difference_approx_fprime_3, label="difference 3")
361            plt.plot(difference_approx_fprime_4, label="difference 4")
362            plt.legend()
363            plt.show()
364            exit()
365            """
366        #############################
367        # Iterate with EM algorithm #
368        #############################
369        prev_X_estimate = np.Inf
370        sigma_n = np.copy(global_initial_sigma_n)
371        startalgorithmtime = time.time()
372        for iteration in range(N_iterations):
373            if iteration > 0:
374                sigma_n = sigma_n * lr  # decrease the noise variance with a
    learning rate
375                if LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE:
376                    x_grid_induce = np.linspace(min(X_estimate), max(X_estimate)
    , N_inducing_points) # Change position of grid to position of estimate
377            # Adding tiny jitter term to diagonal of K_gg (not the same as
    sigma_n that we're adding to the diagonal of K_xgK_gg^-1K_gx later on)
378            K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points) ##
    K_gg = K_gg_plain + sigma_n*np.identity(N_inducing_points)
379            K_xg_prev = squared_exponential_covariance(X_estimate.reshape((T,1))
    ,x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
380            # Find F estimate only if we're not at the first iteration
381            if iteration == 0:
382                print("L value of initial estimate", x_posterior_no_la(
    X_estimate, sigma_n, F_estimate, K_gg, x_grid_induce))
383            if iteration > 0:
384                if LIKELIHOOD_MODEL == "bernoulli":
385                    for i in range(N):
386                        y_i = y_spikes[i]
387                        optimization_result = optimize.minimize(fun=
    f_loglikelihood_bernoulli, x0=F_estimate[i], jac=f_jacobian_bernoulli, args=(
```

```
sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
#hess=f_hessian_bernoulli,
388                     F_estimate[i] = optimization_result.x
389             elif LIKELIHOOD_MODEL == "poisson":
390                 for i in range(N):
391                     y_i = y_spikes[i]
392                     optimization_result = optimize.minimize(fun=
f_loglikelihood_poisson, x0=F_estimate[i], jac=f_jacobian_poisson, args=(
sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
#hess=f_hessian_poisson,
393                     F_estimate[i] = optimization_result.x
394         # Find next X estimate, that can be outside (0,2pi)
395         if NOISE_REGULARIZATION:
396             X_estimate += 2*np.random.multivariate_normal(np.zeros(T),
K_t_generate) - 1
397         if SMOOTHING_REGULARIZATION and iteration < (N_iterations-1) :
398             X_estimate = scipy.ndimage.filters.gaussian_filter1d(X_estimate,
 4)
399         if GRADIENT_FLAG:
400             optimization_result = optimize.minimize(fun=x_posterior_no_la,
x0=X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-
BFGS-B", jac=x_jacobian_no_la, options = {'disp':False})
401         else:
402             optimization_result = optimize.minimize(fun=x_posterior_no_la,
x0=X_estimate, args=(sigma_n, F_estimate, K_gg, x_grid_induce), method = "L-
BFGS-B", options = {'disp':False})
403         X_estimate = optimization_result.x
404         if (iteration == (FLIP_AFTER_HOW_MANY - 1)) and
FLIP_AFTER_SOME_ITERATION:
405             # Flipping estimate after iteration 1 has been plotted
406             X_estimate = 2*mean(X_estimate) - X_estimate
407         if USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION:
408             X_estimate -= min(X_estimate) #set offset of min to 0
409             X_estimate /= max(X_estimate) #scale length to 1
410             X_estimate *= (max(path)-min(path)) #scale length to length of
path
411             X_estimate += min(path) #set offset to offset of path
412         if PLOTTING:
413             plt.plot(X_estimate, label='Estimate')
414             #plt.ylim((lower_domain_limit, upper_domain_limit))
415             plt.tight_layout()
416             plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-
T-" + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(
baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + ".png")
417         if np.linalg.norm(X_estimate - prev_X_estimate) < TOLERANCE:
418             #print("Seed", seeds[seedindex], "Iterations:", iteration+1, "
Change in X smaller than TOL")
419             break
420         #if iteration == N_iterations-1:
421         #    print("Seed", seeds[seedindex], "Iterations:", iteration+1, "
N_iterations reached")
422         prev_X_estimate = X_estimate
423     if USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE:
424         X_estimate -= min(X_estimate) #set offset of min to 0
425         X_estimate /= max(X_estimate) #scale length to 1
426         X_estimate *= (max(path)-min(path)) #scale length to length of path
427         X_estimate += min(path) #set offset to offset of path
428     # Flipped
```

```
429         X_flipped = - X_estimate + 2*mean(X_estimate)
430         # Rootmeansquarederror for X
431         X_rmse = np.sqrt(sum((X_estimate-path)**2) / T)
432         X_flipped_rmse = np.sqrt(sum((X_flipped-path)**2) / T)
433         ##### Check if flipped and maybe iterate again with flipped estimate
434         if X_flipped_rmse < X_rmse and RECONVERGE_IF_FLIPPED:
435             #print("RMSE for X:", X_rmse)
436             #print("RMSE for X flipped:", X_flipped_rmse)
437             #print("Re-iterating because of flip")
438             x_grid_induce = np.linspace(min_inducing_point, max_inducing_point,
    N_inducing_points) #np.linspace(min(path), max(path), N_inducing_points)
439             K_gg_plain = squared_exponential_covariance(x_grid_induce.reshape((
    N_inducing_points,1)),x_grid_induce.reshape((N_inducing_points,1)),
    sigma_f_fit, delta_f_fit)
440             X_initial_2 = np.copy(X_flipped)
441             X_estimate = np.copy(X_flipped)
442             F_estimate = np.copy(F_initial)
443             if GIVEN_TRUE_F:
444                 F_estimate = true_f
445             if PLOTTING:
446                 if T > 100:
447                     plt.figure(figsize=(10,3))
448                 else:
449                     plt.figure()
450                 #plt.title("After flipping") # as we go
451                 plt.xlabel("Time bin")
452                 plt.ylabel("x")
453                 plt.plot(path, color="black", label='True X')
454                 plt.plot(X_initial_2, label='Initial')
455                 #plt.ylim((lower_domain_limit, upper_domain_limit))
456                 plt.tight_layout()
457                 plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-
    T-" + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(
    baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + "-flipped.png")
458             #############################
459             # EM after flipped          #
460             #############################
461             prev_X_estimate = np.Inf
462             sigma_n = np.copy(global_initial_sigma_n)
463             for iteration in range(N_iterations):
464                 if iteration > 0:
465                     sigma_n = sigma_n * lr  # decrease the noise variance with a
     learning rate
466                     if LET_INDUCING_POINTS_CHANGE_PLACE_WITH_X_ESTIMATE:
467                         x_grid_induce = np.linspace(min(X_estimate), max(
    X_estimate), N_inducing_points) # Change position of grid to position of
    estimate
468                 # Adding tiny jitter term to diagonal of K_gg (not the same as
    sigma_n that we're adding to the diagonal of K_xgK_gg^-1K_gx later on)
469                 K_gg = K_gg_plain + jitter_term*np.identity(N_inducing_points) #
    #K_gg = K_gg_plain + sigma_n*np.identity(N_inducing_points)
470                 K_xg_prev = squared_exponential_covariance(X_estimate.reshape((T
    ,1)),x_grid_induce.reshape((N_inducing_points,1)), sigma_f_fit, delta_f_fit)
471                 # Find F estimate only if we're not at the first iteration
472                 if iteration > 0:
473                     if LIKELIHOOD_MODEL == "bernoulli":
474                         for i in range(N):
475                             y_i = y_spikes[i]
```

```
476                             optimization_result = optimize.minimize(fun=
    f_loglikelihood_bernoulli, x0=F_estimate[i], jac=f_jacobian_bernoulli, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_bernoulli,
477                             F_estimate[i] = optimization_result.x
478                     elif LIKELIHOOD_MODEL == "poisson":
479                         for i in range(N):
480                             y_i = y_spikes[i]
481                             optimization_result = optimize.minimize(fun=
    f_loglikelihood_poisson, x0=F_estimate[i], jac=f_jacobian_poisson, args=(
    sigma_n, y_i, K_xg_prev, K_gg), method = 'L-BFGS-B', options={'disp':False})
    #hess=f_hessian_poisson,
482                             F_estimate[i] = optimization_result.x
483                 # Find next X estimate, that can be outside (0,2pi)
484                 if NOISE_REGULARIZATION:
485                     X_estimate += 2*np.random.multivariate_normal(np.zeros(T),
    K_t_generate) - 1
486                 if SMOOTHING_REGULARIZATION and iteration < (N_iterations-1) :
487                     X_estimate = scipy.ndimage.filters.gaussian_filter1d(
    X_estimate, 4)
488                 if GRADIENT_FLAG:
489                     optimization_result = optimize.minimize(fun=
    x_posterior_no_la, x0=X_estimate, args=(sigma_n, F_estimate, K_gg,
    x_grid_induce), method = "L-BFGS-B", jac=x_jacobian_no_la, options = {'disp':
    False})
490                 else:
491                     optimization_result = optimize.minimize(fun=
    x_posterior_no_la, x0=X_estimate, args=(sigma_n, F_estimate, K_gg,
    x_grid_induce), method = "L-BFGS-B", options = {'disp':False})
492                 X_estimate = optimization_result.x
493                 if (iteration == (FLIP_AFTER_HOW_MANY - 1)) and
    FLIP_AFTER_SOME_ITERATION:
494                     # Flipping estimate after iteration 1 has been plotted
495                     X_estimate = 2*mean(X_estimate) - X_estimate
496                 if USE_OFFSET_AND_SCALING_AT_EVERY_ITERATION:
497                     X_estimate -= min(X_estimate) #set offset of min to 0
498                     X_estimate /= max(X_estimate) #scale length to 1
499                     X_estimate *= (max(path)-min(path)) #scale length to length
    of path
500                     X_estimate += min(path) #set offset to offset of path
501                 if PLOTTING:
502                     plt.plot(X_estimate, label='Estimate (after flip)')
503                     #plt.ylim((lower_domain_limit, upper_domain_limit))
504                     plt.tight_layout()
505                     plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-
    eval-T-" + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" +
    str(baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + "-flipped.png
    ")
506                 if np.linalg.norm(X_estimate - prev_X_estimate) < TOLERANCE:
507                     #print("Seed", seeds[seedindex], "Iterations after flip:",
    iteration+1, "Change in X smaller than TOL")
508                     break
509                 #if iteration == N_iterations-1:
510                 #    print("Seed", seeds[seedindex], "Iterations after flip:",
    iteration+1, "N_iterations reached")
511                 prev_X_estimate = X_estimate
512             if USE_OFFSET_AND_SCALING_AFTER_CONVERGENCE:
513                 X_estimate -= min(X_estimate) #set offset of min to 0
```

```
514                 X_estimate /= max(X_estimate) #scale length to 1
515                 X_estimate *= (max(path)-min(path)) #scale length to length of
    path
516                 X_estimate += min(path) #set offset to offset of path
517                 # Check if flipped is better even after flipped convergence:
518                 X_flipped = - X_estimate + 2*mean(X_estimate)
519                 # Rootmeansquarederror for X
520                 X_rmse = np.sqrt(sum((X_estimate-path)**2) / T)
521                 X_flipped_rmse = np.sqrt(sum((X_flipped-path)**2) / T)
522                 ##### Check if flipped and maybe iterate again with flipped
    estimate
523                 if X_flipped_rmse < X_rmse:
524                     X_estimate = X_flipped
525             # Rootmeansquarederror for X
526             X_rmse = np.sqrt(sum((X_estimate-path)**2) / T)
527         #print("Seed", seeds[seedindex], "smoothingwindow",
    smoothingwindow_for_PCA, "finished. RMSE for X:", X_rmse)
528         #SStot = sum((path - mean(path))**2)
529         #SSdev = sum((X_estimate-path)**2)
530         #Rsquared = 1 - SSdev / SStot
531         #Rsquared_values[seed] = Rsquared
532         #print("R squared value of X estimate:", Rsquared, "\n")
533         #####
534         # Rootmeansquarederror for F
535         #if LIKELIHOOD_MODEL == "bernoulli":
536         #    h_estimate = np.divide( np.exp(F_estimate), (1 + np.exp(F_estimate)
    ))
537         #if LIKELIHOOD_MODEL == "poisson":
538         #    h_estimate = np.exp(F_estimate)
539         #F_rmse = np.sqrt(sum((h_estimate-true_f)**2) / (T*N))
540         if PLOTTING:
541             if T > 100:
542                 plt.figure(figsize=(10,3))
543             else:
544                 plt.figure()
545             plt.title("Final estimate") # as we go
546             plt.xlabel("Time bin")
547             plt.ylabel("x")
548             plt.plot(path, color="black", label='True X')
549             plt.plot(X_initial, label='Initial')
550             plt.plot(X_estimate, label='Estimate')
551             plt.legend(loc="upper right")
552             #plt.ylim((lower_domain_limit, upper_domain_limit))
553             plt.tight_layout()
554             plt.savefig(time.strftime("./plots/%Y-%m-%d")+"-robustness-eval-T-"
    + str(T) + "-lambda-" + str(peak_lambda_global) + "-background-" + str(
    baseline_lambda_value) + "-seed-" + str(seeds[seedindex]) + "-final-L-" + str
    (x_posterior_no_la(X_estimate, sigma_n, F_estimate, K_gg, x_grid_induce)) + "
    .png")
555         ensemble_array_X_rmse[smoothingwindow_index] = X_rmse
556         ensemble_array_L_value[smoothingwindow_index] = x_posterior_no_la(
    X_estimate, sigma_n, F_estimate, K_gg, x_grid_induce)
557         ensemble_array_X_estimate[smoothingwindow_index] = X_estimate
558         ensemble_array_F_estimate[smoothingwindow_index] = F_estimate
559         ensemble_array_y_spikes[smoothingwindow_index] = y_spikes
560         ensemble_array_path[smoothingwindow_index] = np.copy(path)
561         # End of loop for one smoothingwidth
562     # Three smoothingwidths done: Find best X estimate based on L value or RMSE
```

```
    score across
563     final_rmse = ensemble_array_X_rmse[0] # when only one window
564     print("Final RMSE for tuning width 5", final_rmse)
565     index_of_smoothing_with_best_RMSE = np.argmin(ensemble_array_X_rmse)
566     best_X_rmse_based_on_RMSE = ensemble_array_X_rmse[
    index_of_smoothing_with_best_RMSE]
567     index_of_smoothing_with_best_L = np.argmin(ensemble_array_L_value)
568     best_X_rmse_based_on_L = ensemble_array_X_rmse[
    index_of_smoothing_with_best_L]
569     rmse_for_smoothingwidth_3 = ensemble_array_X_rmse[0]
570     rmse_for_smoothingwidth_5 = ensemble_array_X_rmse[1]
571     rmse_for_smoothingwidth_10 = ensemble_array_X_rmse[2]
572     X_estimate = ensemble_array_X_estimate[index_of_smoothing_with_best_L]
573     F_estimate = ensemble_array_F_estimate[index_of_smoothing_with_best_L]
574     y_spikes = ensemble_array_y_spikes[index_of_smoothing_with_best_L]
575     path = ensemble_array_path[index_of_smoothing_with_best_L]
576     endtime = time.time()
577     print("\nSeed", seeds[seedindex])
578     print("Time use:", endtime - starttime)
579     print("Time use without overhead", time.time()-startalgorithmtime)
580     print("RMSEs   :", ensemble_array_X_rmse, "Best smoothing window:          ",
     ensemble_smoothingwidths[index_of_smoothing_with_best_RMSE], "Best RMSE:",
    best_X_rmse_based_on_RMSE)
581     print("L values:", ensemble_array_L_value, "Best smoothing window:",
    ensemble_smoothingwidths[index_of_smoothing_with_best_L], "Best RMSE:",
    best_X_rmse_based_on_L)
582     print("
    Smoothingwidth 3  RMSE:", rmse_for_smoothingwidth_3)
583     print("
    Smoothingwidth 5  RMSE:", rmse_for_smoothingwidth_5)
584     print("
    Smoothingwidth 10 RMSE:", rmse_for_smoothingwidth_10)
585     return [best_X_rmse_based_on_RMSE, best_X_rmse_based_on_L,
    rmse_for_smoothingwidth_3, rmse_for_smoothingwidth_5,
    rmse_for_smoothingwidth_10, X_estimate, F_estimate, y_spikes, path] #
    Returning X, F estimates based on L value since that is the best we can do
    unsupervised
586
587 if __name__ == "__main__":
588     # The job index is the lambda index
589     # Seeds are done sequentially and hope we don't choke on them. Then one job
    requires 4 OMP_THREADS
590     # For each seed we do the pca ensemble sequantially too, and we let the
    numpy do its parallellization thing
591     lambda_index = int(sys.argv[1])
592     # The other version:
593     # The index in the job array is interpreted as a two-dimensional list with
    Cols equal to the number of seeds and Rows equal to the number of lambdas
594     #n_cols = len(seeds)
595     #n_rows = len(peak_lambda_array)
596     #lambda_index = int( int(sys.argv[1]) // n_cols )
597     #seedindex = int( int(sys.argv[1]) % n_cols )
598
599     print("Likelihood model:",LIKELIHOOD_MODEL)
600     print("Covariance kernel for Kx:", COVARIANCE_KERNEL_KX)
601     print("Using gradient?", GRADIENT_FLAG)
602     print("Noise regulation:",NOISE_REGULARIZATION)
603     print("Tuning curve definition:", TUNINGCURVE_DEFINITION)
```

```
604     print("Uniform bumps:", UNIFORM_BUMPS)
605     print("Plotting:", PLOTTING)
606     print("Infer F posteriors:", INFER_F_POSTERIORS)
607     print("Initial sigma_n:", global_initial_sigma_n)
608     print("Learning rate:", lr)
609     print("T:", T)
610     print("N:", N)
611     print("Smoothingwidths:", ensemble_smoothingwidths)
612     print("Number of seeds we average over:", NUMBER_OF_SEEDS)
613     if FLIP_AFTER_SOME_ITERATION:
614         print("NBBBB!!! We're flipping the estimate in line 600.")
615     print("\n")
616 
617     global peak_lambda_global
618     peak_lambda_global = peak_lambda_array[lambda_index]
619 
620     print("Lambda", peak_lambda_global, "started!")
621     seed_rmse_array_based_on_RMSE = np.zeros(len(seeds))
622     seed_rmse_array_based_on_L = np.zeros(len(seeds))
623     seed_rmse_array_for_smoothingwidth_3 = np.zeros(len(seeds))
624     seed_rmse_array_for_smoothingwidth_5 = np.zeros(len(seeds))
625     seed_rmse_array_for_smoothingwidth_10 = np.zeros(len(seeds))
626     X_array = np.zeros((len(seeds), T))
627     F_array = np.zeros((len(seeds), N, T))
628     Y_array = np.zeros((len(seeds), N, T))
629     path_array = np.zeros((len(seeds), T))
630 
631     for i in range(len(seeds)):
632         result_array = find_rmse_for_this_lambda_this_seed(i) # i = seedindex
633         seed_rmse_array_based_on_RMSE[i] = result_array[0]
634         seed_rmse_array_based_on_L[i] = result_array[1]
635         seed_rmse_array_for_smoothingwidth_3[i] = result_array[2]
636         seed_rmse_array_for_smoothingwidth_5[i] = result_array[3]
637         seed_rmse_array_for_smoothingwidth_10[i] = result_array[4]
638         X_array[i] = result_array[5]
639         F_array[i] = result_array[6]
640         Y_array[i] = result_array[7]
641         path_array[i] = result_array[8]
642 
643     # Using RMSE to choose best final X:
644     np.save("m_s_arrays/RMSE-m-base-" + str(baseline_lambda_value) + "-T-" + str
    (T) + "-lambda-index-" + str(lambda_index), np.mean(
    seed_rmse_array_based_on_RMSE))
645     np.save("m_s_arrays/RMSE-s-base-" + str(baseline_lambda_value) + "-T-" + str
    (T) + "-lambda-index-" + str(lambda_index), sum((
    seed_rmse_array_based_on_RMSE - np.mean(seed_rmse_array_based_on_RMSE))**2))
646     # Using L to choose best final X:
647     np.save("m_s_arrays/L-m-base-" + str(baseline_lambda_value) + "-T-" + str(T)
     + "-lambda-index-" + str(lambda_index), np.mean(seed_rmse_array_based_on_L))
648     np.save("m_s_arrays/L-s-base-" + str(baseline_lambda_value) + "-T-" + str(T)
     + "-lambda-index-" + str(lambda_index), sum((seed_rmse_array_based_on_L - np
    .mean(seed_rmse_array_based_on_L))**2))
649     # Sticking with smoothingwidth 3:
650     np.save("m_s_arrays/3-m-base-" + str(baseline_lambda_value) + "-T-" + str(T)
     + "-lambda-index-" + str(lambda_index), np.mean(
    seed_rmse_array_for_smoothingwidth_3))
651     np.save("m_s_arrays/3-s-base-" + str(baseline_lambda_value) + "-T-" + str(T)
     + "-lambda-index-" + str(lambda_index), sum((
```

```
seed_rmse_array_for_smoothingwidth_3 - np.mean(
seed_rmse_array_for_smoothingwidth_3))**2))
652     # Sticking with smoothingwidth 5:
653     np.save("m_s_arrays/5-m-base-" + str(baseline_lambda_value) + "-T-" + str(T)
 + "-lambda-index-" + str(lambda_index), np.mean(
seed_rmse_array_for_smoothingwidth_5))
654     np.save("m_s_arrays/5-s-base-" + str(baseline_lambda_value) + "-T-" + str(T)
 + "-lambda-index-" + str(lambda_index), sum((
seed_rmse_array_for_smoothingwidth_5 - np.mean(
seed_rmse_array_for_smoothingwidth_5))**2))
655     # Sticking with smoothingwidth 10:
656     np.save("m_s_arrays/10-m-base-" + str(baseline_lambda_value) + "-T-" + str(T
) + "-lambda-index-" + str(lambda_index), np.mean(
seed_rmse_array_for_smoothingwidth_10))
657     np.save("m_s_arrays/10-s-base-" + str(baseline_lambda_value) + "-T-" + str(T
) + "-lambda-index-" + str(lambda_index), sum((
seed_rmse_array_for_smoothingwidth_10 - np.mean(
seed_rmse_array_for_smoothingwidth_10))**2))
658
659     print("\n")
660     print("Lambda strength:", peak_lambda_global)
661     print("RMSE for X (chosen by RMSE    ) averaged across seeds:", np.mean(
seed_rmse_array_based_on_RMSE))
662     print("Sum of squared errors in the RMSE:", sum((
seed_rmse_array_based_on_RMSE - np.mean(seed_rmse_array_based_on_RMSE))**2))
663     print("RMSE for X (chosen by L value) averaged across seeds:", np.mean(
seed_rmse_array_based_on_L))
664     print("Sum of squared errors in the RMSE:", sum((seed_rmse_array_based_on_L
- np.mean(seed_rmse_array_based_on_L))**2))
665     print("RMSE for X (smoothing width 3) averaged across seeds:", np.mean(
seed_rmse_array_for_smoothingwidth_3))
666     print("Sum of squared errors in the RMSE:", sum((
seed_rmse_array_for_smoothingwidth_3 - np.mean(
seed_rmse_array_for_smoothingwidth_3))**2))
667     print("RMSE for X (smoothing width 5) averaged across seeds:", np.mean(
seed_rmse_array_for_smoothingwidth_5))
668     print("Sum of squared errors in the RMSE:", sum((
seed_rmse_array_for_smoothingwidth_5 - np.mean(
seed_rmse_array_for_smoothingwidth_5))**2))
669     print("RMSE for X (smoothing width 10) averaged across seeds:", np.mean(
seed_rmse_array_for_smoothingwidth_10))
670     print("Sum of squared errors in the RMSE:", sum((
seed_rmse_array_for_smoothingwidth_10 - np.mean(
seed_rmse_array_for_smoothingwidth_10))**2))
671     print("\n")
672     # Finished all seeds for this lambda
673
674     if INFER_F_POSTERIORS:
675         # Grid for plotting
676         bins_for_plotting = np.linspace(lower_domain_limit, upper_domain_limit,
num=N_plotgridpoints + 1)
677         x_grid_for_plotting = 0.5*(bins_for_plotting[:(-1)]+bins_for_plotting
[1:])
678         peak_lambda_global = peak_lambda_array[-1]
679         print("Peak lambda:", peak_lambda_global)
680         peak_f_offset = np.log(peak_lambda_global) - baseline_f_value
681         #posterior_f_inference(X_estimate, F_estimate, sigma_n, y_spikes, path,
x_grid_for_plotting, bins_for_plotting, peak_f_offset, baseline_f_value,
```

```
    binsize)
682         posterior_f_inference(X_array[0], F_array[0], 1, Y_array[0], path_array
    [0], x_grid_for_plotting, bins_for_plotting, peak_f_offset, baseline_f_value,
     1000) # Bin size has no physical meaning for synthetic data
```

**Listing B.5:** cluster-parallel-robustness-evaluation.py

Even Moa Myklebust

A robustness evaluation of the latent manifold tuning model

NTNU
Norwegian University of
Science and Technology