UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Probabilistic Numerics in Tractography using Diffusion MRI Data

Bachelor Project

Emil Petersen
[**empe@di.ku.dk**]

Victor Nordam Suadicani
[**nordam@di.ku.dk**]

2018–06–13

Supervisors: Aasa Feragen, Anton Mallasto

Abstract

Tractography is the problem of estimating nerve tracts in the brain based on diffusion magnetic resonance imaging (MRI) data. Probabilistic methods are useful in this venture, due to the uncertain nature of the MRI data. The complexity of the problem also makes it difficult to solve analytically, making numerical methods a practical necessity. Unlike standard numerical approximations, probabilistic numerics explicitly incorporates uncertainty by returning a probability over the estimates.

In this thesis, we study probabilistic numerical methods applied to the problem of tractography. First, the theory required to understand the probabilistic numerical methods is introduced. This includes the concept of Gaussian processes in Section 2 and the Kalman filter algorithm in Section 3. Using these concepts, we study probabilistic numerical methods applied to initial value problems in Section 4.

In Section 5, we explain how tractography can be viewed as an initial value problem. We then apply the probabilistic numerical methods on the problem of tractography and present our results. In Section 6 we discuss our results and conclude on the effectiveness of the methods applied to this problem. While the methods are not perfect, we did get promising results and managed to find some tracts inside the brain.

Acknowledgements

Data were provided [in part] by the Human Connectome Project, WU-Minn Consortium (Principal Investigators: David Van Essen and Kamil Ugurbil; 1U54MH091657) funded by the 16 NIH Institutes and Centers that support the NIH Blueprint for Neuroscience Research; and by the McDonnell Center for Systems Neuroscience at Washington University.

Contents

1
Introduction

Tractography is the problem of estimating nerve tracts in the brain based on diffusion MRI data. Diffusion MRI is a method for scanning the human brain, which measures the diffusion of water. Water diffuses along the nerve fibers. These nerve fibers make up what we refer to as brain *tracts* — pathways of fibers inside the brain. The idea with tractography is to use the diffusion MRI data to map these tracts, so that we may learn something about the structure and connectivity of the brain.

However, the measurement of diffusion MRI is a noisy process, which makes the data noisy. A good model of the tracts should arguably take this uncertainty into account. This is what makes the probabilistic methods useful; this approach explicitly incorporates uncertainty into its result. In contrast to a deterministic model, a probabilistic model allows you to measure the uncertainty in the result, by giving samples as the result, or even an entire distribution to sample from.

Additionally, the diffusion MRI data is complicated, which makes it difficult to analyze. It is difficult, if not entirely impossible, to analytically solve the problem of mapping the tracts. Therefore, numerical estimation methods must suffice.

The combination of probabilistic and numerical methods leads to the method of probabilistic numerics. We apply this method to the problem of tractography, since it is well suited to handle the probabilistic and numerical challenges. This would allow us to use numerical estimation to handle the complexity of the diffusion MRI data, while still being able to quantify the uncertainty in our result.

We use *Gaussian processes* as our probabilistic model. The Gaussian Process is a stochastic process, which can be described as a distribution over functions. Section 2 deals with the theory of Gaussian processes and attempts to provide the explanations needed in order to understand its relevance to the project.

When estimating the tracts, the tracts are constructed sequentially, as each point estimate is informed by the previous location. This online process is executed using a *Kalman filter*. The Kalman filter is an algorithm used for estimating the state of a dynamical system. It works on some prior knowledge of the current state of the system, expressed as a prior distribution; a prediction step is then made using this prior knowledge. Usually, a noisy measurement of the value is then observed, and the final estimate is updated based on the prediction and the observation. This three-fold process of predicting, observing and updating the estimate is iterated over time, yielding the estimated state of the system at each step. In addition, the Kalman filter is computationally feasible since the number of computations needed are linear with the size of the input. Section 3 explains the theoretical foundation of the Kalman filter. First, some motivation is given for its usefulness, and how it is related to Gaussian processes. Then, the iterative process of predicting and updating state estimates is described in further detail.

In Section 4 we describe *ordinary differential equations* (ODE) and *initial value problems* (IVP). Probabilistic numerical methods can be used to numerically estimate solutions to

these problems, while incorporating uncertainty. We then present a certain probabilistic numerical algorithm for solving initial value problems, developed by Michael Schober et al. [21]. This algorithm is known as a *probabilistic filtering ODE solver* (PFOS).

Section 5 returns to the problem of tractography. As it turns out, tractography can be viewed as an initial value problem, with the diffusion MRI data taking the role of the ordinary differential equation. Thus, we apply the PFOS to the problem of tractography and present the results.

Finally, we discuss our results in Section 6 and assess the effectiveness of the PFOS applied on tractography.

1.1 The Bayesian Approach

In order to understand the theory and tools used in this project, it is important to know that the statistical approach is Bayesian. In statistics, a distinction is made between two groups of practitioners: The frequentists and the Bayesians [10, p. 339]. In the frequentist approach, parameters for a given model are fixed but usually unknown [10]. These parameters have a sampling distribution that captures the uncertainty we have, given that sampling the parameters will give different estimates. The parameters themselves are still considered fixed, and an infinite number of samples will converge in expectancy to these fixed values.

In Bayesian thinking, parameters are not considered fixed, but treated as random variables [10]. These parameters can then be described by a probability distribution. The probability distribution does not need data initially, since this distribution reflects our beliefs about the probability of the parameters, before any observations. This is called the *prior distribution*.

As an example take a series of coin flips and denote tails as 0 and heads as 1. We believe the expected value of a coin flip is 0.5, and so we could use this as an outset for the prior probability distribution of the expected value. If we instead were suspicious of the coin, we might have a prior uniform distribution denoting equal probability for any expected value between zero and one. If the coin turns out to be fair, we would then discover, by flipping the coin and observing the outcomes, that the probability distribution of the expected value is centered around 0.5 and not uniform as we assumed before observing data.

Bayesian analysis allows us to update our belief about the parameters using the observations we have made. In our case, we can compute the posterior distribution by the product of the prior distribution and the likelihood function, divided by the marginal likelihood. This is the case because the prior and the posterior belong to the same class of distributions (Gaussians), and so are conjugate priors with respect to the likelihood function [20]. If the prior distribution is not very informative, then observing data will have a strong impact on the shape of the posterior. This will cause the posterior to look markedly different from our prior, which is good because we can then update our beliefs about the distribution of the parameters. Otherwise, if the prior is informative, i.e. it is a better description of the parameter distribution, then observing data will only make the posterior look a little different from the prior. In the long run, observing more and more data will dominate how the posterior distribution looks, and whatever belief we initially had, becomes less relevant.

In practice, when observations are done sequentially, the posterior distribution calculated from one sample observation is usually used as the prior in the next observation. While observing more and more samples, the current belief about the parameter distribution is updated at every sample. This is directly beneficial when modelling dynamical system, as will be described further in Section 3 about the Kalman filter. In this context, the Bayesian approach provides the mathematical foundations to model these systems over time. The notable advantage is that uncertainty and noise can be incorporated into the modelling this way, namely because Bayesian statistics is concerned with uncertainty with respect to each single observation [13].

1.2 Mathematical Foundations

A vector is notated in **bold**, e.g. $\mathbf{y} = (y_1, y_2, \ldots, y_n)$, and bold capital letters (such as $\mathbf{A}$) are used for matrices.

The **joint probability** $\Pr(\mathbf{y})$ of n random variables $(y_1, y_2, \ldots, y_n)$ is the probability values associated with the random variables occurring at the same time. As all probabilities, it integrates to 1 in the continuous case:

$$\Pr(\mathbf{y}) = \int_{y_1} \int_{y_2} \cdots \int_{y_n} \Pr[Y_1 = y_1, Y_2 = y_2, \ldots, Y_n = y_n] \, \mathrm{d}y_n \ldots \mathrm{d}y_2 \, \mathrm{d}y_1 = 1$$

A **marginal probability** only concerns a subset of the random variables and so 'reduces' the dimensionality. Let $\mathbf{y_a}, \mathbf{y_b} \subseteq \mathbf{y}$ be disjoint sets of the random variables in $\mathbf{y}$, and let $\Pr_{\mathbf{a},\mathbf{b}}$ be their joint distribution. The marginal probability $\Pr_{\mathbf{a}}$ of $\mathbf{y_a}$ is then

$$\Pr_{\mathbf{a}}(a_1, a_2, a_3, \ldots, a_m) = \int \Pr_{\mathbf{a},\mathbf{b}}(a_1, a_2, a_3, \ldots, a_m, \mathbf{y_b}) \, \mathrm{d}\mathbf{y_b},$$

where $|\mathbf{y_a}| = m$. The marginal probability is itself a joint probability if $|\mathbf{y_a}| > 1$, i.e. if there are more than one random variable in $\mathbf{y_a}$. Variables are independent if their joint distributions are equal to the product of their marginal distributions:

$$\Pr(\mathbf{y}) = \prod_{i=1}^{n} \Pr(y_i) \iff y_1, y_2, \ldots, y_n \text{ are independent.}$$

A **conditional probability** is the probability of observing one variable given another:

$$\Pr(\mathbf{y_a} \mid \mathbf{y_b}) = \frac{\Pr(\mathbf{y_a}, \mathbf{y_b})}{\Pr(\mathbf{y_b})}, \qquad \Pr(\mathbf{y_b} \mid \mathbf{y_a}) = \frac{\Pr(\mathbf{y_a}, \mathbf{y_b})}{\Pr(\mathbf{y_a})}$$

rearranging this gives:

$$\Pr(\mathbf{y_a}, \mathbf{y_b}) = \Pr(\mathbf{y_b}) \Pr(\mathbf{y_a} \mid \mathbf{y_b}), \qquad \Pr(\mathbf{y_a}, \mathbf{y_b}) = \Pr(\mathbf{y_a}) \Pr(\mathbf{y_b} \mid \mathbf{y_a})$$

combining these gives *Bayes' Theorem*:

$$\Pr(\mathbf{y_a} \mid \mathbf{y_b}) = \frac{\Pr(\mathbf{y_a}) \cdot \Pr(\mathbf{y_b} \mid \mathbf{y_a})}{\Pr(\mathbf{y_b})}.$$

Bayes' theorem is the foundation of many of the calculations done in this project. We assume from now on that the reader is familiar with these concepts.

2
Gaussian Processes

This section deals with the theory of Gaussian Processes and tries to provide the explanations needed in order to understand its relevance to the project. Initially the Gaussian distribution is introduced. Building on this, the idea of a Gaussian Process is then explained, at first informally to foster an intuitive understanding, and then in a formalized way to make the theory mathematically solid. The applications of Gaussian Processes in regression tools are then emphasized. Finally, we give examples of applications on concrete data.

2.1 The Gaussian Distribution

In order to understand the Gaussian Process theory, one must first understand the Gaussian distribution. The Gaussian distribution is probably one of the most well-known probability density functions and also one of the most used. We have the univariate Gaussian distribution:

$$f(x) = \frac{1}{\sigma\sqrt{2\pi}} \exp\left(-\frac{(x-\mu)^2}{2\sigma^2}\right)$$

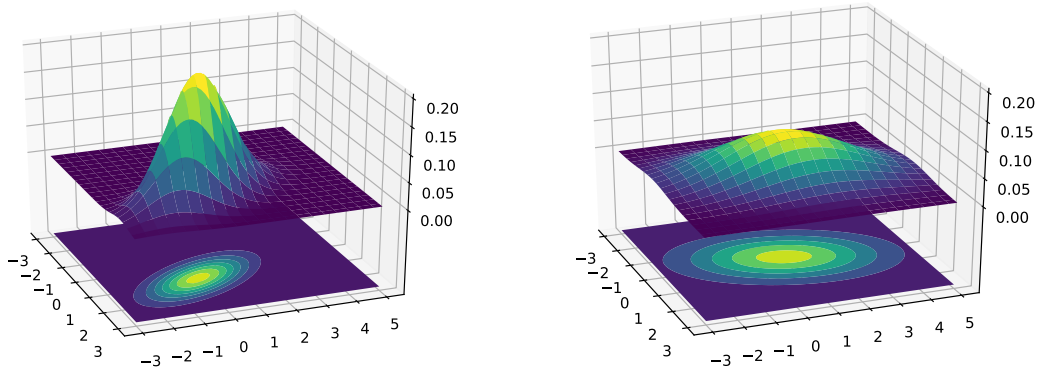
Here, $f(x)$ describes a Gaussian probability density function. We have a scalar parameter μ for the mean of the distribution and a scalar parameter σ for the standard deviation. In statistics, the Gaussian distribution is used frequently because of the *Central Limit Theorem*, which states that the (normalized) sums of random variables taken from unknown distributions, under some assumptions, will converge in distribution to the Gaussian distribution as the number of samples increases. That is, even though the distribution of the observed values is unknown, with enough samples, the methods and tools that depend on means, that works with Gaussian distributions can still be applied. This makes the theory and methods associated with the Gaussian distribution attractive, as it lends itself to a wide range of problems.

There is another reason for the prevalence of Gaussian distributions, which is the ease of computation that Gaussian distributions afford. This is perhaps even more important with respect to Gaussian Processes. It is computationally easy to manipulate Gaussian variables, such as adding or multiplying them. Not only so, but the Gaussian distribution is closed under these operations, meaning that the results will also follow a Gaussian distribution. That is, the product of two Gaussian probability density functions is a Gaussian probability density function. This is especially important later, when we have to calculate posterior distributions.

Similar to the univariate Gaussian, the multivariate Gaussian distribution is defined as

$$\mathcal{N}(\mathbf{x}; \mu, \mathbf{\Sigma}) = \left((2\pi)^{N/2}|\mathbf{\Sigma}|^{1/2}\right)^{-1} \exp\left(-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^\top \mathbf{\Sigma}^{-1}(\mathbf{x}-\boldsymbol{\mu})\right),$$

where $\mathbf{x}, \mu \in \mathbb{R}^N$ and $\mathbf{\Sigma} \in \mathbb{R}^{N\times N}$, N being the number of dimensions and μ being the *mean vector* and Σ being the *covariance matrix*. The covariance matrix has to be symmetric and positive semi-definite, i.e. have non-negative eigenvalues. Since the mean μ is now a



(a) $\mu = [1, 0]^\top$, $\mathbf{\Sigma} = \begin{bmatrix} 1/2 & -1/2 \\ -1/2 & 3/2 \end{bmatrix}$ (b) $\mu = [0, 1]^\top$, $\mathbf{\Sigma} = \begin{bmatrix} 1 & 1/2 \\ 1/2 & 4 \end{bmatrix}$

Figure 2.1: Contour plots of two bivariate Gaussian distributions. The mean μ defines the center, and the covariance matrix $\mathbf{\Sigma}$ defines the shape.

vector of N dimensions, and $\mathbf{\Sigma}$ is a matrix, the calculations with multivariate Gaussian distributions are now done with linear algebraic methods. Informally, the mean defines the center of the distribution, and the covariance matrix defines the shape. Figure 2.1 visualizes this in 2 dimensions.

The univariate Gaussian distribution is a special case of the multivariate Gaussian. Both enjoy the properties as described above, with the addition that we have closure under projections and marginalization. This is very convenient, since, if we are only interested in a few variables of the total, this allows us to essentially only look at the subvector of the mean, and the submatrix of the covariance matrix that concerns the variables we are interested in. This is computationally tractable, since the computations will not concern all the variables, but only the subset that are of interest to us. Another desirable property is that conditional Gaussian distributions, i.e. a Gaussian distribution conditioned on some previous knowledge, is also a Gaussian distribution, assuming that the knowledge also follows a Gaussian distribution. Again, this will later be useful in Bayesian inference modelling with calculations of prior and posterior distributions.

2.2 Parametric and Non-Parametric Models

There are two main approaches to inference when doing supervised learning – parametric and non-parametric [5]. Both approaches attempt to figure out which model function has the best predictive fit with some given training data. In the parametric approach, you restrict the classes of functions to consider. An example would be to only consider linear or quadratic functions. The goal is then to find *function parameters* that make your regression fit the training data well. For example you may wish to find the slope and intersect of a linear function. In this case, you can use relatively simple regression methods using loss functions to find the optimal parameters. The problem with this approach is that it is often difficult to choose which classes of functions to consider and which to discard. If you are careless, you'll use a low-order function, resulting in a regression that fits the data badly, or a high-order function that fits the data too well, making predictions useless.

A non-parametric model attempts to describe the fitting function without considering the function's specific parameters. The benefit to this approach is that you do not have to specify what parameters the function may have, meaning you do not have to only consider functions of a certain class (say, quadratic functions), which may lead to less under- or over-fitting. The fact that you do not possess the actual parameters of the function makes interpretation and analysis of the regression quite complicated though.

2.3 Gaussian Processes

A Gaussian process is an example of a non-parametric model. A Gaussian process attempts to attribute each potential function with a prior probability and then sample from the distribution of all potential functions. Functions that are considered more likely are attributed with a higher probability (e.g. functions that fit the data well). This is a non-parametric model because we never actually compute any parameters of the function itself (although there are other forms of (hyper-)parameters involved that we will get to in Section 2.3.1 and 2.4).

Gaussian processes allows for this approach to be computationally feasible in practice, even when you do not restrict the classes of functions you consider. In essence, a Gaussian process does not choose the classes of functions to consider based on the specific properties of the functions, i.e. whether it is linear or quadratic. Instead, it considers or excludes functions based on how likely they are deemed to be, allowing functions with much larger degrees of freedom. The difficult part of a Gaussian process is then to figure out how to attribute each possible function with a certain probability density. If you only need to know the function values at a finite number of points, then the Gaussian process will give the same answer as if all infinite points were taken into account. This is known as the *marginalization property* [5].

The definition of a Gaussian Process is as follows:

Definition 1 ([5])**.** *"A* Gaussian process *is a collection of random variables, any finite number of which have a joint Gaussian distribution."*

A Gaussian process can be thought of as a distribution over functions, i.e a random function. As stated above, Gaussian distributions are relatively easy to work with and are well-behaved, in that operations on them often result in other Gaussian distributions. Hence, it seems convenient to use a multivariate Gaussian distribution to define our distribution over functions. Let's say we want to model a function $f : \mathbb{R}^D \to \mathbb{R}$.

A sample of a multivariate Gaussian distribution is a vector of some specified length. We will model the function f using this vector $\mathbf{f}$. At first, this seems odd, since a function is usually defined over an infinite domain, but let's say that we only sample a finite subset of f's domain. Then we will say that $\mathbf{f}$ is distributed by a multivariate Gaussian distribution:

$$\mathbf{f} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{2.1}$$

Now, imagine that we extend the range of f that we wish to sample. Due to the marginalization property, no matter how many extra points of f that we sample, the distribution of the original points remain the same [5]. That is, if we sampled a larger range (possibly infinite) $[\mathbf{f}, \mathbf{f}_+]^\top \sim \mathcal{N}(\boldsymbol{\mu}_+, \boldsymbol{\Sigma}_+)$ then $\mathbf{f}$ alone is *still* distributed as in Equation 2.1. This essentially means that whether we look at a finite or an infinite range of f, we still get the same result on the individual points. This is really important, as it allows us to work on

a finite range of the function, while still getting the same results as if we were looking at the whole infinite range of the function.

To make the definition of this multivariate distribution more useful, we will define a *mean function* $m : \mathbb{R}^D \to \mathbb{R}$ and a *covariance function* $k : \mathbb{R}^D \times \mathbb{R}^D \to \mathbb{R}$, that we will use to define $\boldsymbol{\mu}$ and $\boldsymbol{\Sigma}$.

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

These definitions follow exactly from the usual definition of mean and covariance [5]. We will then write that the function f follows a distribution of functions, that is, a Gaussian process like this:

$$f \sim \mathcal{GP}\left(m, k\right)$$

Now, if we wish to sample a domain $\mathbf{X} = [\mathbf{x}_1, \ldots, \mathbf{x}_N]$ of a function $f(\mathbf{x})$, gathering the values of that domain in the vector $\mathbf{f}$, we construct a mean vector $\boldsymbol{\mu}$ and a covariance matrix $\boldsymbol{\Sigma}$ using these functions. We will extend the functions m and k to $M : \mathbb{R}^{D \times N} \to \mathbb{R}^N$ and $K : \mathbb{R}^{D \times N} \times \mathbb{R}^{D \times N} \to \mathbb{R}^{N \times N}$ so they will be able to take the entire range $\mathbf{X}$:

$$\boldsymbol{\mu} = M(\mathbf{X}) = \begin{bmatrix} m(\mathbf{x}_1) \\ m(\mathbf{x}_2) \\ \vdots \\ m(\mathbf{x}_N) \end{bmatrix} \qquad \boldsymbol{\Sigma} = K(\mathbf{X}, \mathbf{X}') = \begin{bmatrix} k(\mathbf{x}_1, \mathbf{x}'_1) & \cdots & k(\mathbf{x}_1, \mathbf{x}'_N) \\ \vdots & \ddots & \vdots \\ k(\mathbf{x}_N, \mathbf{x}'_1) & \cdots & k(\mathbf{x}_N, \mathbf{x}'_N) \end{bmatrix}.$$

Then, $\mathbf{f}$ has the following distribution:

$$\mathbf{f} \sim \mathcal{N}(M(\mathbf{X}), K(\mathbf{X}, \mathbf{X}))$$

The mean and covariance functions give us flexibility, because they allow us to define these separately. Especially the choice of covariance function is significant, as it defines the shape of the distribution. Figure 2.2 shows an example of three functions sampled with $M(\mathbf{X}) = \mathbf{0}$ and using the squared exponential covariance function.

The central task in Gaussian process inference that we now face, is choosing how to define $M(\mathbf{X})$ and $K(\mathbf{X}, \mathbf{X})$ concretely. The mean vector can in some cases, for simplicity's sake, be set to 0, $M(\mathbf{X}) = \mathbf{0}$. The slightly more interesting choice of covariance function is much less obvious. There are many types of covariance functions, also known as *kernels* and each of them requires us to choose additional parameters, known as *hyperparameters* (so as to distinguish them from weight parameters in linear regression). Properly choosing a kernel and adjusting its hyperparameters is a complicated task. The kernel is what contains many of the assumptions of the problem that is modelled by the process; figuring out what can be assumed is therefore usually connected to the task of choosing a kernel.

2.3.1 Squared Exponential Kernel

A kernel that is often used is the *Squared Exponential kernel* (SE), and will be used here as well as in the examples to follow. The Squared Exponential kernel is a stationary kernel, i.e. the covariance between any two locations $\mathbf{x}$ and $\mathbf{x}'$ is a function of the distance between the two. In this case, distance is measured as the Euclidean norm $||\mathbf{x} - \mathbf{x}'||$ between the two locations. Simplified, the closer in distance two random variables are, the higher the correlation is between them. The Squared Exponential kernel is defined as [5]

$$k_{SE}\left(\mathbf{x}, \mathbf{x}'\right) = \sigma \cdot \exp\left(-\frac{||\mathbf{x} - \mathbf{x}'||^2}{2\ell^2}\right).$$

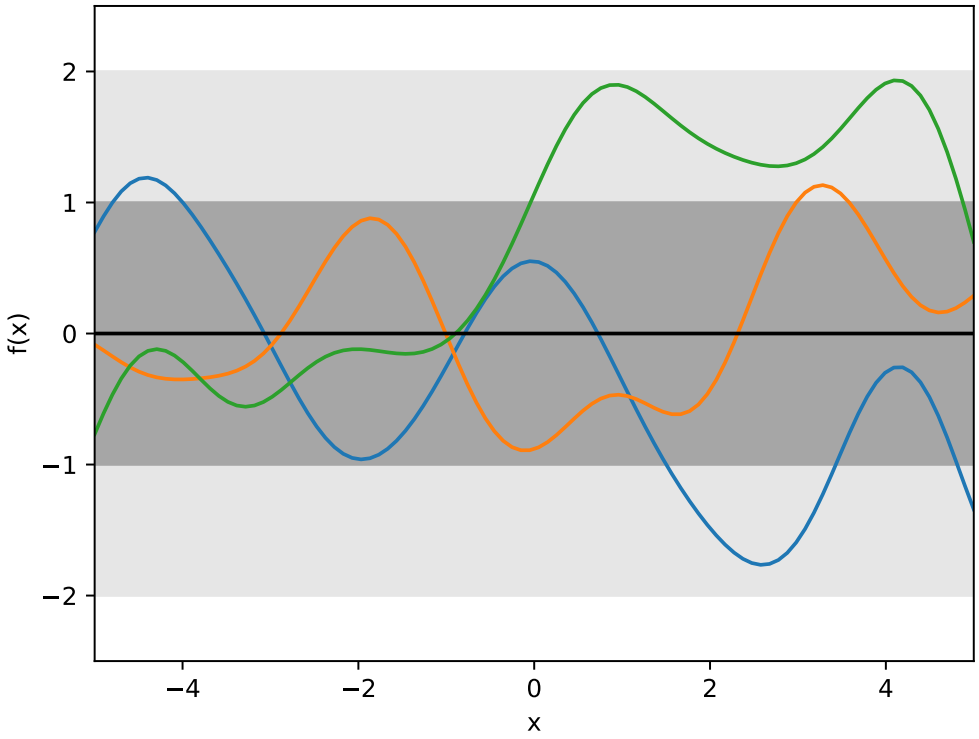


Figure 2.2: Three functions sampled with a **0** mean function and squared exponential covariance function. The black line shows the mean. The grey areas show one and two standard deviations from the mean. The functions look smooth, but in reality they are plotted using 100 points.

If the domain is one-dimensional, the distance is simply the absolute difference $|x - x'|$. The variables σ and ℓ are hyperparameters, ℓ denoting the characteristic length-scale of the kernel, and σ simply a variance/scaling factor. The length-scale roughly indicates how far away two points need to be before large changes in function values can be expected. A large length-scale parameter would make the function look more smooth, since it would require a long distance to significantly change the function's value. For the same reason, one should take the length-scale into account when extrapolating from the data, since a small length-scale would mean rapid change in the function, and perhaps less reliability in the extrapolation [22]. The variance factor σ indicates how close to the function mean the observed function values are. The Squared Exponential kernel is smooth [5]. This might be the reason why it is used so much, but it is argued as well that many problems are not realistically modelled by the Squared Exponential for the same reason, i.e. it is not feasible to assume that the problem can be modelled so smoothly.

2.4 Optimization of Hyperparameters

By varying the hyperparameters of a kernel, significantly different results can be obtained. Some values of hyperparameters produce regressions that fit the data well, while others are nonsense. We would like a mechanism for determining which values of hyperparameters are *good* and which are bad. For this purpose, we will use the marginal likelihood of the

data to determine how likely the data observed is, *given* the values of the hyperparameters. In this way, we can compare the likelihood of different sets of hyperparameters and choose the one with the highest likelihood.

In other words, we want the greatest likelihood of our predicted values $\mathbf{y}$ given our data $\mathbf{X}$ and our variable hyperparameters θ. This likelihood, $p(\mathbf{y} \mid X, \theta)$ is given by the formula for the multivariate Gaussian distribution, with $M = M(X)$ as the mean vector and $K_y = K(X, X) + \sigma_n^2 I$ as the covariance matrix (where σ_n^2 is a noise factor on the covariance matrix) [5]:

$$\Pr(\mathbf{y} \mid X, \theta) = \left((2\pi)^{N/2}|K_y|^{1/2}\right)^{-1} \exp\left(-\frac{1}{2}(\mathbf{y} - M)^\top K_y^{-1}(\mathbf{y} - M)\right)$$

To make the above equation less unwieldy, we can take its logarithm. This is fine, since we simply wish to find a maximum and taking the logarithm does not change where the maximum is, since the logarithm is a monotonically increasing function.

$$\ln \Pr(\mathbf{y} \mid X, \theta) = -\frac{N}{2}\ln(2\pi) - \frac{1}{2}\ln|K_y| - \frac{1}{2}(\mathbf{y} - M)^\top K_y^{-1}(\mathbf{y} - M) \tag{2.2}$$

In order to maximize the likelihood, we could take the gradient, set it to 0 and isolate the hyperparameters. However, the gradient is not trivial. Therefore, we will use gradient ascent in order to maximize the likelihood. Unfortunately, there are many local maximums, which means that we are not guaranteed to find a global maximum.

Gradient ascent is an algorithm for approximating minimums/maximums of a function that would otherwise be hard to optimize. The algorithm very roughly works like this:

1. Choose initial hyperparameter values as a starting point (the algorithm can be favorably iterated with different initial values).
2. Calculate the gradient at the current point.
3. Follow the direction of the gradient with some step length to get to a new point.
4. Repeat step 2-3 until the gradient is sufficiently close to 0 (i.e. when the algorithm is sufficiently close to a minimum/maximum).

For step 2, we need the partial derivatives of the likelihood with respect to each hyperparameter, θ_i. The partial derivative is given by

$$\frac{\partial}{\partial\theta_i}\ln p(\mathbf{y} \mid X, \theta) = -\frac{1}{2}\mathrm{tr}\left(K^{-1}\frac{\partial K}{\partial\theta_i}\right) + \frac{1}{2}(\mathbf{y} - M)^\top K^{-1}\frac{\partial K}{\partial\theta_i}K^{-1}(\mathbf{y} - M)$$

The first term of Equation 2.2 is a constant and disappears. The second and the third term are derived from these two formulas respectively:

$$\frac{\partial}{\partial\theta}\ln|K| = \mathrm{tr}\left(K^{-1}\frac{\partial K}{\partial\theta}\right), \qquad \frac{\partial K^{-1}}{\partial\theta} = -K^{-1}\frac{\partial K}{\partial\theta}K^{-1}$$

So in order to calculate this derivative, we must have the derivative of our kernel. This is generally not a problem to obtain.

Once we have the partial derivatives w.r.t each hyperparameter, we combine them in a vector in order to get the gradient. This is the direction that we follow in the gradient ascent algorithm. Once the algorithm terminates, we will have found a set of hyperparameters in a local maximum, potentially a global maximum. Running the algorithm multiple times on different starting points, we pick the hyperparameters with the highest likelihood, considering them optimal. An illustration of the algorithm is found in our example in Section 2.5.2, figure 2.6.

2.5 Gaussian Process Regression

We will now consider the situation where you are given (or have observed) a set of data points, that is, a set of n input vectors $\mathbf{x}_i$, $i \in [n]$, and their corresponding output values $f(\mathbf{x}_i)$. These data points form our observation set $D = \{(\mathbf{x}_i, f(\mathbf{x}_i)) \mid i = 1, \ldots, n\}$, or simply $D = (X, \mathbf{f})$. We want to use the observation set as training data for a Gaussian process, in order to make predictions about a certain other set of input vectors that we will call the test set and denote as X_*. We will denote the predicted values of $f(\mathbf{x}_*)$ as $\mathbf{f}_*$. The joint distribution of $\mathbf{f}$ and $\mathbf{f}_*$ can then be obtained by combining them in one vector:

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(M\left(\begin{bmatrix} X \\ X_* \end{bmatrix} \right), \begin{bmatrix} K(X,X) & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix} \right)$$

Here $K(X, X)$ denotes the part of the covariance matrix that describes the covariance between all pairs of the vectors in X. The submatrix $K(X, X_*)$ denotes the part of the covariance matrix that describes the covariance between all pairs of vectors between X and X_*. Finally $K(X_*, X_*)$ denotes the part that describes the covariance between all pairs of vectors in X_*. Because a covariance matrix must be symmetric, $K(X, X_*)^\top = K(X_*, X)$.

In order to make predictions about $\mathbf{f}_*$, we need to *condition* the joint distribution on $\mathbf{f}$. That is, we want the distribution of the test set output values, $\mathbf{f}_*$, given the training data $\mathbf{f}$. We denote this as $\mathbf{f}_* \mid \mathbf{f}$. The conditional distribution is as follows [1, 5]:

$$\begin{aligned} \mathbf{f}_* \mid \mathbf{f} \sim \mathcal{N}(&M(X_*) + K(X_*, X)K(X, X)^{-1}(\mathbf{f} - M(X)), \\ &K(X_*, X_*) - K(X_*, X)K(X, X)^{-1}K(X, X_*)) \end{aligned} \tag{2.3}$$

The conditional probability (2.3) is important because it constitutes the posterior model of the problem, i.e. given the observations $\mathbf{f}$, we think $\mathbf{f}_*$ is likely to be distributed as the conditional probability distribution. It can be said to be what we now think about the model, taking into account the prior beliefs (the prior distribution) and the new observed data.

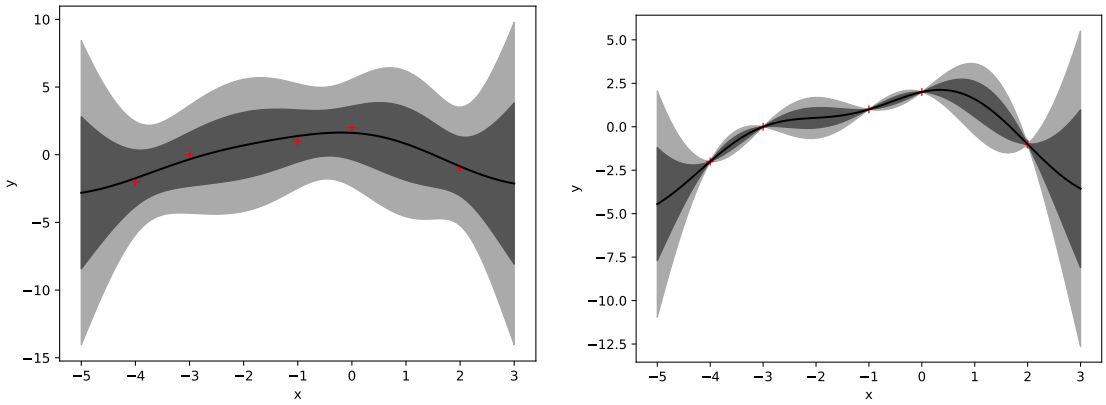
2.5.1 Incorporating Noise

Equation 2.3 works fine, but it assumes that the observed data is true with absolute certainty. Usually our training data set is not a hard truth but is merely noisy observations. We should strive to incorporate this noise into our model.

The problem with the noise-free Equation 2.3 is that the model becomes too certain about the values at the points of the training set. In order to make the model uncertain about these function values, we must consider the training set as noisy measurements.

Thus let us consider the same basic situation as before, but now we have that the n observed data points are not exactly equal to the underlying function $f(\mathbf{x}_i)$. Instead, we only have the values $y = f(\mathbf{x}_i) + \varepsilon$, gathered in the vector $\mathbf{y}$, where $\varepsilon \sim \mathcal{N}(0, \sigma_n^2)$ is noise that is independent and identically distributed by a Gaussian distribution. We denote the noise's variance as σ_n^2 and we add this variance to the diagonal of the covariance matrix, to signify the extra uncertainty about the values themselves. Then we have:

$$\begin{bmatrix} \mathbf{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left(M\left(\begin{bmatrix} X \\ X_* \end{bmatrix} \right), \begin{bmatrix} K(X,X) + \sigma_n^2 I & K(X,X_*) \\ K(X_*,X) & K(X_*,X_*) \end{bmatrix} \right)$$



(a) (b)

Figure 2.3: A simple data set of 5 observations, modelled with (a) and without (b) assumed noise in the observations. The red crosses are data points, the black line is the mean and the grey areas represent one and two standard deviations.

The posterior will then be:

$$\begin{aligned}\mathbf{f}_* \mid \mathbf{y} \sim \mathcal{N}(&M(X_*) + K(X_*, X)(K(X, X) + \sigma_n^2 I)^{-1}(\mathbf{f} - M(X)),\\ &K(X_*, X_*) - K(X_*, X)(K(X, X) + \sigma_n^2 I)^{-1} K(X, X_*))\end{aligned} \tag{2.4}$$

So we see that incorporating noise into our model is quite simple, it just changes Equation 2.3 slightly. Figure 2.3 illustrates the difference on a dummy data set.

2.5.2 Gaussian Process Regression Example

We will now present an example of GP regression, using the concepts outlined in this section. The main example is with a simple data set consisting of daily temperature measurements from a Russian town during a whole year [2, 7]. To begin with, we introduce the temperature data set. Then a prior model is constructed, using the squared exponential kernel and hyperparameters that are not optimized. Finally, we optimize the model by using gradient ascent to find hyperparameters that yield a higher likelihood. This final model is then displayed and described.

Data

The data set in this example consists of 365 temperature measurements. The measurements starts on January 1$^{\text{st}}$, measuring temperature once a day, going through until December 31$^{\text{st}}$. See figure 2.4 for the plotted data.

By looking at figure 2.4, in the first hundred days or so, the temperatures seem to vary a lot between minus ten and minus forty degrees. During the summer period, temperatures are measured more concentrated around a few negative degrees. The last part of the year is yet again more varied in temperature from day to day, compared to the summer. Unsurprisingly, daily temperatures are consistently lower in the winter and fall, than during the summer.

Our goal is to do regression on this data set so that we may make predictions about the temperature in following years.

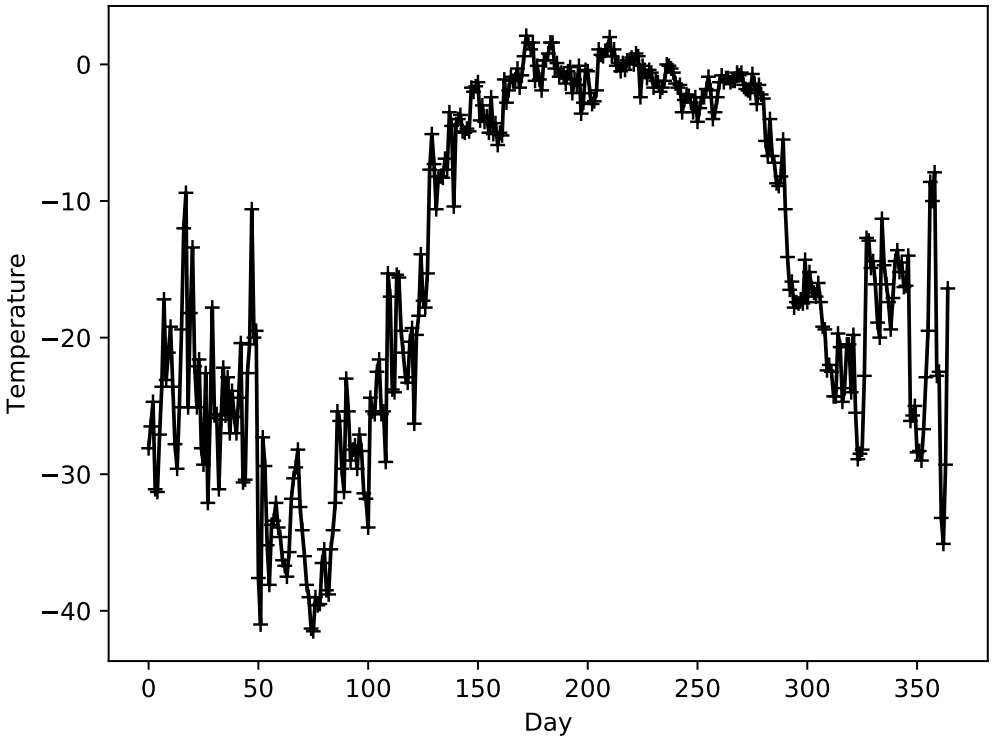


Figure 2.4: The plotted data set. Each '+' is a measured Celsius temperature; there is one for each day of the year. The location is Siberia, which might explain why the vast majority of the temperature measurements are negative.

Regression

To perform the regression, we use Equation 2.4. The testing data points, X_*, are given as $4 \cdot 365 = 1460$ points equally distributed between 0 and 364. Some of these points are the same points as in the training data, X.

The mean and covariance matrix are calculated as in Equation 2.4. We have chosen the noise added to the covariance matrix as 15. The data is plotted along with the mean and some number of standard deviations. The standard deviations are the square roots of the diagonal of the covariance matrix (these are after all the variances σ^2, hence the standard deviation σ is their square roots).

The choice of kernel greatly affects the end result. Figure 2.5 shows the mean and two barely visible standard deviations for four different types of kernels. They are only for illustrative purposes, as we will only use the Squared Exponential kernel from here on. We have set $M(X) = \mathbf{0}$ and all hyperparameters are simply set to one. Even so, we see that the data is fitted at least somewhat nicely with most of the kernels.

Optimizing hyperparameters of squared exponential kernel

To get a better fit, let us take the squared exponential kernel and optimize its hyperparameters. We use gradient ascent in order to find the hyperparameters that lead to the

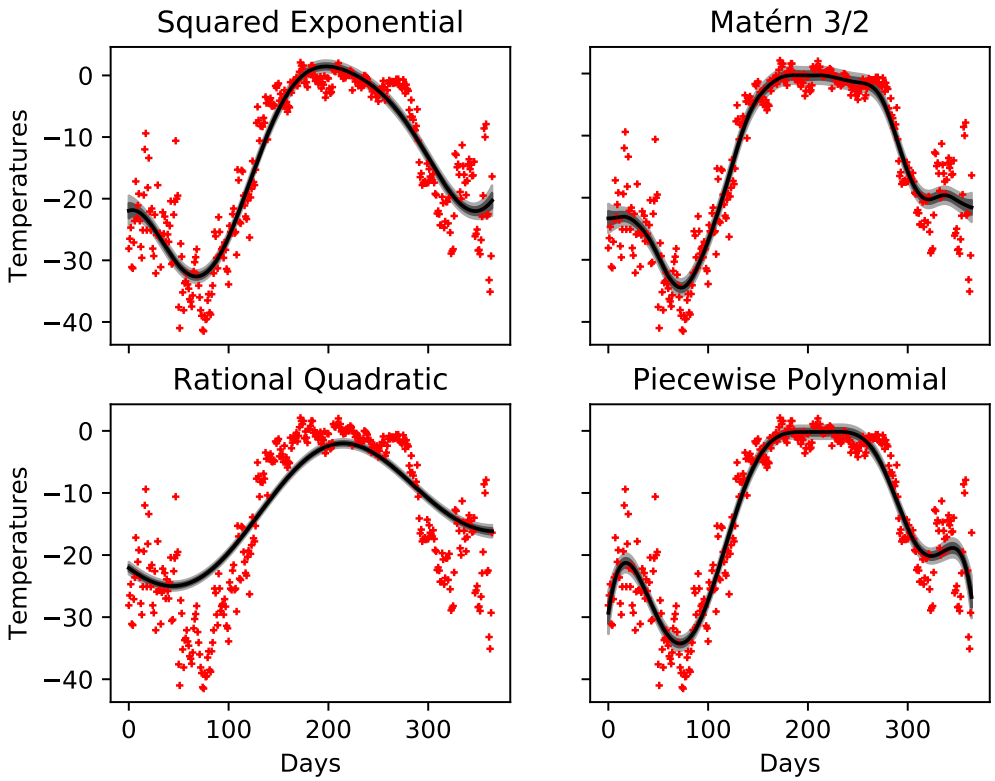


Figure 2.5: The data modelled using four different covariance (kernel) functions. The red marks are data points; the black line is the mean line, and the barely visible grey shading denotes two standard deviations. Each kernel yields a different model.

maximum likelihood of the data. Recall the two hyperparameters of the squared exponential kernel – the variance factor and the length-scale. Taking the derivative of the squared exponential kernel with respect to these parameters yields:

$$\frac{\partial k_{SE}(x,x')}{\partial \sigma} = \exp\left(-\frac{||x-x'||^2}{2\ell^2}\right),$$
$$\frac{\partial k_{SE}(x,x')}{\partial \ell} = \sigma\frac{||x-x'||^2}{\ell^3}\exp\left(-\frac{||x-x'||^2}{2\ell^2}\right).$$

We use these equations to calculate the gradient at the current point (current values of hyperparameters). We make use of an adaptive learning rate to make our gradient ascent better: First, we perform backtracking, meaning that we never take a step in the direction of the gradient if the likelihood at the new point would be lower. If we get into a situation where that would happen, we reduce our step length by half and try again (potentially reducing it by half again, until the step produces a higher likelihood). Second, we perform acceleration, meaning that if a successful step was taken, we increase the step length by 10% – essentially, if it seems that we are going in the right direction, we speed up to reach the maximum as fast as possible. Third, we scale the step length by the magnitude of the gradient, which helps to speed up and slow down appropriately as we get close to a maximum. We stop the ascent when the gradient becomes sufficiently small.

These techniques mean that we avoid some pitfalls that a naive gradient ascent algorithm

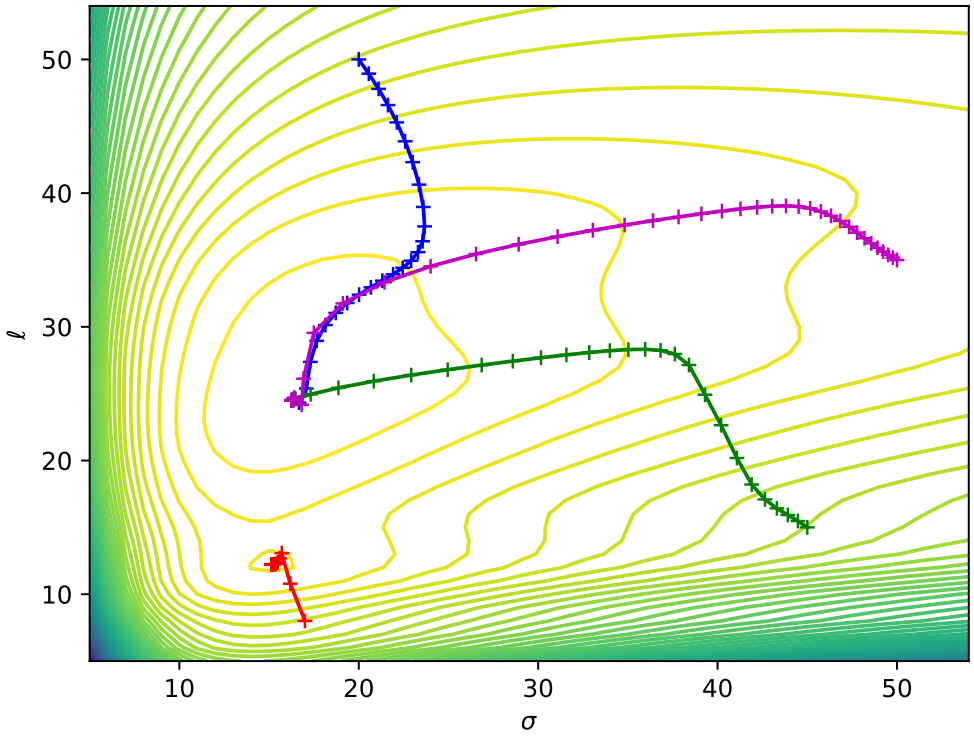


Figure 2.6: Contour plot of the marginal likelihood of the hyperparameters, σ (variance factor) and ℓ (length-scale), and the paths of four gradient ascent algorithm runs. Notice the local maximum found by the red path.

might face. For example, we avoid stepping back and forth between two points on either side of a maximum. We also avoid long running times, since we increase the step length when appropriate, instead of having a static step length. Also, the precision of the final result can be changed pretty much arbitrarily within floating points precision, although running time becomes longer and longer the more precise you need the result to be.

Figure 2.6 shows a contour plot of the likelihood as a function of the variance factor σ and the length-scale ℓ hyperparameters. Four coloured paths are also shown, which display the path of our gradient ascent algorithm from different starting points. Each cross on the paths shows a step taken by the algorithm. Notice how the step length changes on the longer stretches of the purple path, showing the acceleration technique. Also notice that the green path curves around (37,28) without overshooting past the "hill". Especially interesting is the red path, which shows another local maximum, although the maximum found by the other paths is superior. Using gradient ascent, we reach the optimal hyperparameters of $\sigma \approx 16.256$ and $\ell \approx 24.502$. Plugging these into our regression yields figure 2.7. This seems to fit somewhat better than the squared exponential kernel in figure 2.5. The fit is still not terribly great though, as there are too many points outside two standard deviations from the mean. This may be because the kernel we use is stationary, so it only takes the distance between days into account. This can be somewhat alleviated by changing the noise factor that we simply set to 15 earlier. This noise factor can also be considered a hyperparameter on its own, in order to optimize it. We felt this was

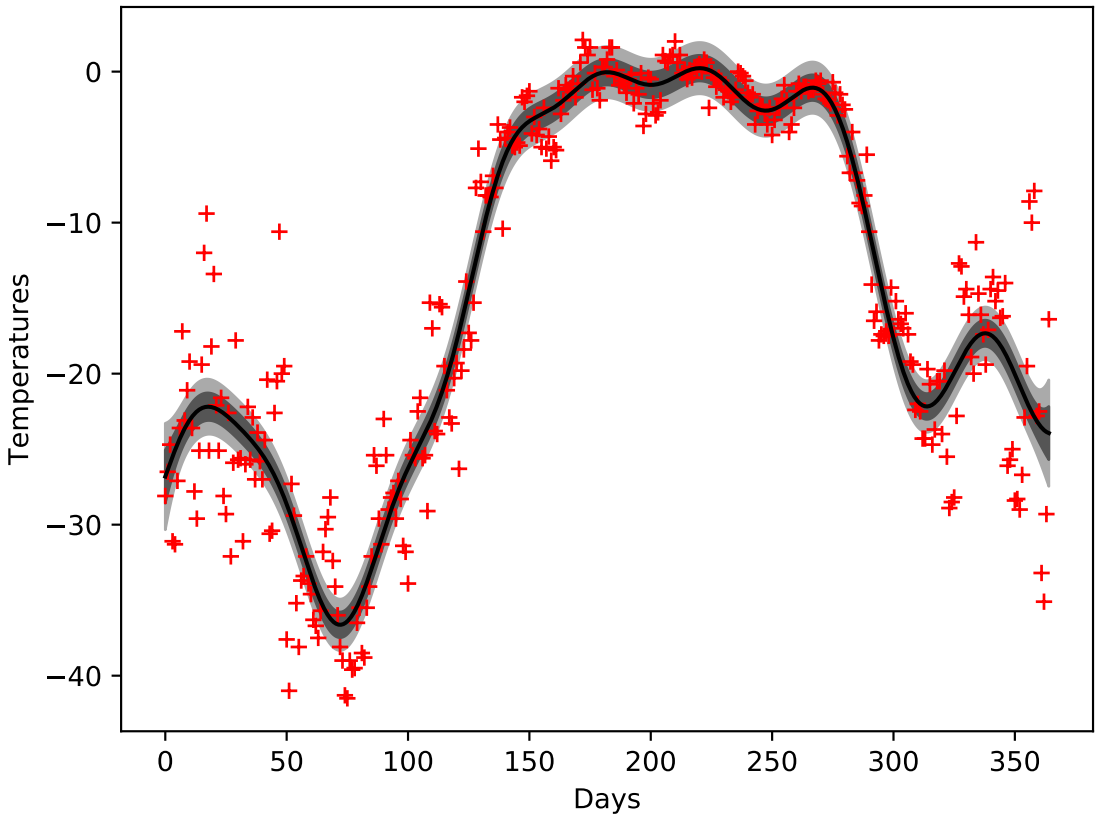


Figure 2.7: Regression of the temperature data using the squared exponential kernel with optimized hyperparameters of $\sigma \approx 16.256$ and $\ell \approx 24.502$. The black line is the mean while the grey areas show one and two standard deviations.

unnecessary for the purposes of this example though.

3
The Kalman Filter

The *Kalman filter* is an algorithm used for estimating states of dynamical systems. A dynamical system is a system that contains some *state* which may vary over time. The state may be a variable or a function, depending on the system. The state may be observed and measured, but this measurement is noisy and inaccurate. We would like to have a way to estimate the state from such noisy measurements in an optimal manner.

At some *time* value t, the true value of the state is $y(t)$, which is estimated based on a prediction of the value, and a noisy observation of the value at time t. The next state time is then $t + \Delta t$, and so on. The Kalman filter works on some prior knowledge of the current state t, expressed as a prior distribution; a prediction step is then made using this prior knowledge. Usually, a noisy measurement of the value $y(t)$ is then observed, and the final estimate is updated based on the prediction and the observation. This estimate is also expressed as a (posterior) distribution. This three-fold process of predicting, observing and updating the estimate is iterated over time, yielding the estimated state of the system at each step. The filter is used in a variety of practical settings, including GPS, target tracking, robotics and, importantly for this thesis, in probabilistic numerics, as we will discuss in Section 4. In addition, the Kalman filter is computationally feasible since the number of computations needed is linear in the size of the input. That is, the algorithm works iteratively, requiring only the last state of the system to calculate the next, in contrast to requiring all the previous states. This online property makes it favorable in real-time problems, and in particular when data is obtained sequentially.

This section explains the theoretical foundations of the Kalman filter. First, some motivation is given for its usefulness. Then, the iterative process of predicting and updating state estimates is described in mathematical terms. Finally, because the filtering model alone does not update previous estimates according to new observations, this smoothing procedure of updating previous estimates is also explained.

3.1 Motivation and Example

Imagine that you are faced with a set of measurements $\mathbf{y}_{[n]} = \mathbf{y}_1, \mathbf{y}_2, \ldots, \mathbf{y}_n$, describing the states $\mathbf{x}_{[n]} = \mathbf{x}_1, \ldots, \mathbf{x}_n$ of a system at time steps $t \in [n]$, that you are interested in. For example, the data set may consist of measurements of the distance between a target and a tracking station and the target's speed at the measured time. Since the measurements are noisy, the measurements are not the true values for the distance and speed. Based on this information, you want to estimate the distance and speed of the target in some time interval $[t_1, t_n]$, in our example at each integer value from 1 to n.

In order to estimate the states of the system in this interval, you want to condition on the measurements $\mathbf{y}_{[n]}$ available. To do this, you can calculate the joint posterior distribution $\Pr(\mathbf{x}_{[n]} \mid \mathbf{y}_{[n]})$ for all the states, given all the observations, using the well-known Bayes' rule:

$$\Pr\left(\mathbf{x}_{[n]} \mid \mathbf{y}_{[n]}\right) = \frac{\Pr\left(\mathbf{y}_{[n]} \mid \mathbf{x}_{[n]}\right) \Pr\left(\mathbf{x}_{[n]}\right)}{\Pr\left(\mathbf{y}_{[n]}\right)},$$

where each vector $\mathbf{x}_t \in \mathbf{x}_{[n]}$ is a vector known as the *state vector* containing the variables of interest at time t, in the example the target's distance and speed. The corresponding measurement is given by $\mathbf{y}_t \in \mathbf{y}_{[n]}$. We thus have that the joint posterior distribution $\Pr(\mathbf{x}_{[n]} \mid \mathbf{y}_{[n]})$ is proportional to the product of the likelihood of the measurements $\Pr(\mathbf{y}_{[n]} \mid \mathbf{x}_{[n]})$ and whatever prior knowledge we have about the states $\Pr(\mathbf{x}_{[n]})$ (the prior distribution). The term $\Pr(\mathbf{y}_{[n]})$ is a normalizing term that assures the posterior is a properly defined probability distribution [13].

Now imagine that you want to make further estimates, and receive a new measurement of the target's distance and speed at time $n + 1$. The problem with the above approach is that you would have to calculate the joint posterior distribution *all over again*, taking all measurements and previous states into account. Because of this, each new step taken into account will require more computations, as the size of the data increases. If we continually receive new measurements, the associated calculations will eventually slow down to a crawl, making this approach impractical for large data sets.

3.2 The Filter

The Kalman filter provides a solution to the above problem by making the estimation procedure computationally feasible. The filter only relies on the best estimate of the state, up to the point in time where we expect a new observation, instead of computing the full joint posterior. That is, it instead computes marginal distributions on the current state, which is computationally favorable [13].

First, a *predictive distribution* $\Pr(\mathbf{x}_n \mid \mathbf{y}_{[n-1]})$ is calculated, called the prediction step, which is the estimate of the next state given the current observations. Usually, an observation $\mathbf{y}_n$ is then made, which is used in the *filtering distribution* $\Pr(\mathbf{x}_n \mid \mathbf{y}_{[n]})$, to estimate the next state $\mathbf{x}_n$, given both the current observations $\mathbf{y}_{[n-1]}$, and the next observation $\mathbf{y}_n$. This is the update step of the Kalman filter.

Predicting and updating the state for an iteration can be done in a constant amount of operations, if we make some assumptions of the underlying model of the dynamic system we are describing. In short, we assume that a state $\mathbf{x}_n$ only depends on the previous state $\mathbf{x}_{n-1}$ and not the entire history of states $\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_{n-1}$ (*Markov property*). It is also assumed that the system is linear, i.e. state transitions and similar aspects of the system can be described by matrix multiplications, and that noise follows a Gaussian distribution. The underlying model is described in detail in Section 3.2.1.

In the example where the system is distance and speed of a target in respect to a tracking station, the Kalman filter would estimate the target's distance and speed based only on its previous distance and speed. To do this, the filter makes use of a state transition matrix $\mathbf{A}_t$, which governs the *linear* transformation from one state to another at time t. Assuming knowledge of the physical laws regarding distance and speed, these can be used to inform this transition. Furthermore, the model might govern the influence of control inputs such as turning the steering wheel in case of a vehicle, or accelerating the speed by some control over the target. These are the known influences on the state, and the filter makes sure to take these into account when estimating the next state. Control input

will however not be applicable to this thesis, and so they are not dealt with any further. Still important, there are usually influencing factors that are not sufficiently modelled by either control or state transition, for example draft and resistance on the target, or elevation. Therefore the model also includes noise $\mathbf{q}$, which reflects these unknown factors that are not otherwise incorporated into the model. This noise is in our case assumed to be normally distributed, i.e. $\mathbf{q} \sim \mathcal{N}(\mathbf{0}, \mathbf{Q})$, where $\mathbf{Q}$ is a covariance matrix for the noise. In much the same manner, the observations are assumed to adhere to a linear measurement model $\mathbf{H}_n$, and observation noise $\mathbf{r} \sim \mathcal{N}(\mathbf{0}, \mathbf{R})$ [13, p. 56]:

$$\mathbf{y}_n = \mathbf{H}_n \mathbf{x}_n + \mathbf{r}.$$

Formally, the state prediction can thus be modelled, without control inputs, as

$$\begin{aligned} \Pr\left(\mathbf{x}_n \mid \mathbf{y}_{[n-1]}\right) &\sim \mathcal{N}\left(\mathbf{m}_n^-, \mathbf{P}_n^-\right), \\ \mathbf{m}_n^- &= \mathbf{A}_{n-1}\mathbf{m}_{n-1}, \\ \mathbf{P}_n^- &= \mathbf{A}_{n-1}\mathbf{P}_{n-1}\mathbf{A}_{n-1}^\top + \mathbf{Q}_{n-1}, \end{aligned}$$

where $\mathbf{m}_{n+1}^-$ is the mean of the predicted state, and $\mathbf{P}_{n+1}^-$ is the covariance [13, p. 37]. Again, the underlying model assumptions allow us to denote the transition of the state linearly, i.e. by matrix multiplication, and to denote the predicted state as a Gaussian distribution, since the noise is assumed to be Gaussian as well. In a similar fashion, the updated *filter distribution* can be expressed:

$$\begin{aligned} \Pr\left(\mathbf{x}_n \mid \mathbf{y}_{[n]}\right) &= \mathcal{N}\left(\mathbf{m}_n, \mathbf{P}_n\right), \\ \mathbf{m}_n &= \mathbf{m}_n^- + \mathbf{K}_n\left(\mathbf{y}_n - \mathbf{H}_n\mathbf{m}_n^-\right), \\ \mathbf{P}_n &= \left(\mathbf{I} - \mathbf{K}_n\mathbf{H}_n\right)\mathbf{P}_n^-, \\ \mathbf{K}_n &= \mathbf{P}_n^-\mathbf{H}_n^\top\mathbf{S}_n^{-1}, \\ \mathbf{S}_n &= \mathbf{H}_n\mathbf{P}_n^-\mathbf{H}_n^\top + \mathbf{R}, \end{aligned}$$

where $\mathbf{m}_n$ is the updated mean, and $\mathbf{P}_n$ is the updated covariance of the state. Here $\mathbf{S}_n$ corresponds to the covariance of the difference observed between the predicted state, and the observed state. The matrix $\mathbf{K}_n$ denotes the *Kalman gain* of the step, which in short does a weighted average of the predicted estimate and the observed value, giving more weight to the more certain value of the two. A high Kalman gain will follow new observations more closely, which makes the filter more responsive to changes in observations, but yields a more jagged, jumpy function. A lower Kalman gain follows the predicted estimate more closely, which yields a smoother function. The has the disadvantage of making the model less responsive to changes on observations.

Combining the prediction and update step equations above, we reach Algorithm 1 which illustrates how you would implement the Kalman filter in practice.

In summation, the Kalman filter can, by continuously predicting, observing and updating, efficiently estimate the state of such a dynamic system over time. A simple example of the applied filter can be seen in figure 3.1.

3.2.1 The model

The Kalman filter is assumed to make estimates of true, but hidden states of a dynamic system. The hidden states are not themselves known, but noisy observations can be made

Algorithm 1: The Kalman filter

Initialize $\mathbf{x}_0 \sim \mathcal{N}(\mathbf{m}_0, \mathbf{P}_0)$.
while *more estimations are desired* **do**
    /* Predict next state */
    $\mathbf{m}_n^- = \mathbf{A}_{n-1}\mathbf{m}_{n-1}$
    $\mathbf{P}_n^- = \mathbf{A}_{n-1}\mathbf{P}_{n-1}\mathbf{A}_{n-1}^\top + \mathbf{Q}_{n-1}$
    /* Update the estimate based on prediction and observation */
    $\mathbf{y_n} = \mathbf{H}_n\mathbf{x}_n + \mathbf{r}$
    $\mathbf{m}_n = \mathbf{m}_n^- + \mathbf{K}_n(\mathbf{y}_n - \mathbf{H}_n\mathbf{m}_n^-)$
    $\mathbf{P}_n = \mathbf{P}_n^- - \mathbf{K}_n\mathbf{S}_n\mathbf{K}_\mathbf{n}^\top$
    $\mathbf{K}_n = \mathbf{P}_n^-\mathbf{H}_n^\top\mathbf{S}_n^{-1}$
    $\mathbf{S}_n = \mathbf{H}_n\mathbf{P}_n^-\mathbf{H}_n^\top + \mathbf{R}$
return $\mathbf{x}_{[n]}$, $\mathbf{y}_{[n]}$, $\mathbf{m}_{[n]}$, $\mathbf{P}_{[n]}$,

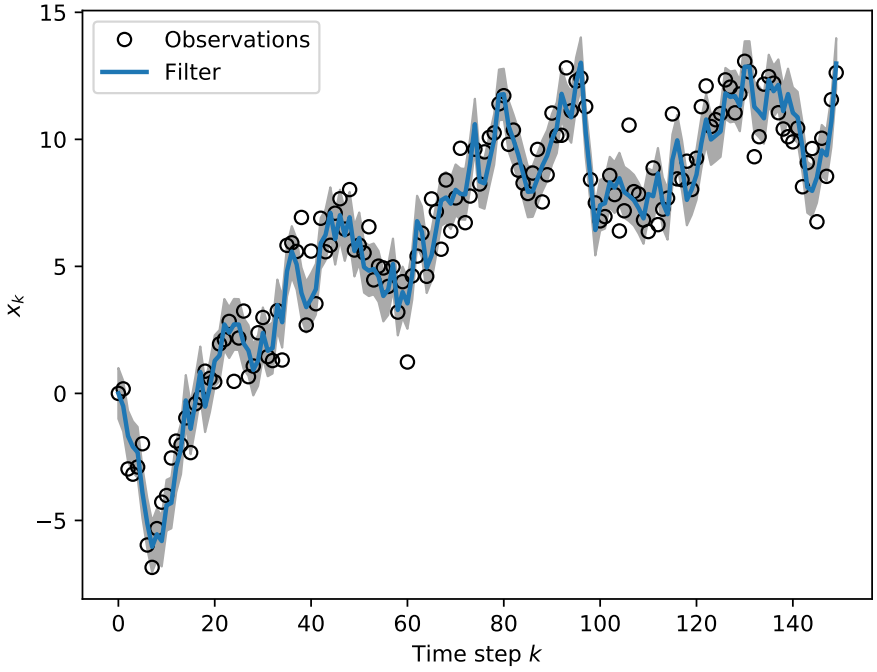


Figure 3.1: The Kalman filter applied to a random walk. The black circles denote the observed states, and the blue line denotes the Kalman filter state estimates at each time step k. The grey area represents one standard deviation.

of them, and these are the available information. Furthermore, the underlying assumption for the state estimations is that it is a *Markov sequence* [13, p. 10]. Markov sequences have some desired properties: The ensuing state $\mathbf{x}_n$ is assumed to only depend on the current state $\mathbf{x}_{n-1}$. The process 'has no memory' of past states, and so is only conditioned on the current state:

$$\Pr(\mathbf{x}_n \mid \mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-1}) = \Pr(\mathbf{x}_n \mid \mathbf{x}_{n-1})$$

In words, there is no information about the next state $\mathbf{x}_n$, in the time steps previous to the current time step $\mathbf{x}_{n-1}$, which in probabilistic terms means that the value of $\mathbf{x}_n$ is independent from the states $\mathbf{x}_1, \mathbf{x}_2, \ldots, \mathbf{x}_{n-2}$. The property is useful, since the conditional computations are significantly simpler due to this assumption. It is an arguably fair assumption to make of certain physical systems, such as the motion of an object governed by physical laws: Knowing the state of an object at one point in time, such as location, direction and velocity, one can calculate an estimate of its trajectory using the physical laws of motion. Such an assumption might not be realistic, as the age of a car definitely influences its movement; it might break down or run slower, etc.

In addition, we assume that the state of the system can initially be described by a prior probability distribution $\Pr(\mathbf{x}_0)$, which reflects any initial information we might have about the states. It is also assumed that the measurements $\mathbf{y}_{[n]}$ on the true states $\mathbf{x}_{[n]}$ are modelled by a conditional probability distribution on the measurement, given the state. This distribution is denoted $\Pr(\mathbf{y}_{[n]} \mid \mathbf{x}_{[n]})$.

3.3 Smoothing the Model

In filtering estimates, each system state at time t is estimated using the observed measurements seen before and at time t. It might however be of interest to update these estimates later, when more measurements have been made. This updating of previous estimates given new data is called *smoothing* the model. For linear Gaussian models, the smoothing function is known as the Rauch-Tung-Striebel smoother (RTS) [13, p. 134]. Other smoothing functions exist, but this is the one of interest for this thesis, since the modelling that is being done is on linear Gaussian models.

Given n measurements, the smoother updates each estimate $\mathbf{x}'_t$, $t \in [n]$, such that the posterior distribution is $\Pr(\mathbf{x}_t \mid \mathbf{y}_{[n]})$, instead of the previous $\Pr(\mathbf{x}_t \mid \mathbf{y}_{[t]})$. In short, it takes any future observations into account when doing the estimate. In practice this is done by recursing backwards through the sequence of estimates, updating each estimate at every step. This operation is linear in computation, and has conceptual similarities with the filter, drawing on the advantage that each estimate at time step t can be updated looking only at the current estimate and the updated estimate at step $t + 1$ (since the algorithm moves backward from n down to 1).

4
Probabilistic Numerics

This section will introduce the concept of *ordinary differential equations* (ODEs) and their structure. Then, we will introduce a type of problem known as *initial value problems* (IVPs), which consists of an ODE and a starting point.

It is sometimes possible to analytically solve a given IVP. However, this is only the case when the ODE of the IVP is well-behaved. Unfortunately, for many applications this is not the case. In such situations, the only feasible way to solve these IVPs are by numerical estimation.

We will then turn our attention to a probabilistic approach for numerically solving IVPs. This probabilistic numerical method is especially useful when the ODE has some elements of uncertainty. With the probabilistic method, a distribution of solutions can be constructed, from which samples can be drawn.

We are interested in probabilistic numerical IVP solving because of the applications it has in tractography. In particular, the measurements obtained from the brain by a diffusion MRI scan can be considered to induce an ODE. However, this ODE cannot be analytically integrated and there is a lot of noise in the data. Thus probabilistic numerical IVP solving is very relevant.

4.1 Ordinary Differential Equations

Differential equations are equations that relate functions with their derivatives. For example, $y'(x) = \frac{1}{2}y(x)$ is a differential equation that relates the function $y(x)$ to its derivative $y'(x)$. An *ordinary* differential equation is a differential equation that only involves functions of one unknown variable. The above equation is also an example of this as y only depends on one variable, namely x.

When solving ODEs analytically, the result is a class of functions (a function with an arbitrary constant) satisfying the equation. The above equation has the solution $y(x) = c \cdot \exp\left(\frac{1}{2}x\right)$, where c can be any arbitrary constant. Thus there are infinitely many solutions.

4.1.1 Initial Value Problems

An initial value problem consists of an ODE and an initial value $y_0 = y(x_0)$ at some point x_0. The ODE only describes how the function value changes, but not where it started or what its value is at any given point. The initial value provides a starting point from which the ODE can be used to infer the rest of the function values – we know a starting point and we know how the function changes, thus we can figure out the function's values.

Solving an IVP results in a restriction of the class of functions that solve the ODE. The class of functions that solve the ODE will be restricted to only the function that passes through the value y_0, thus fixing the arbitrary constant c.

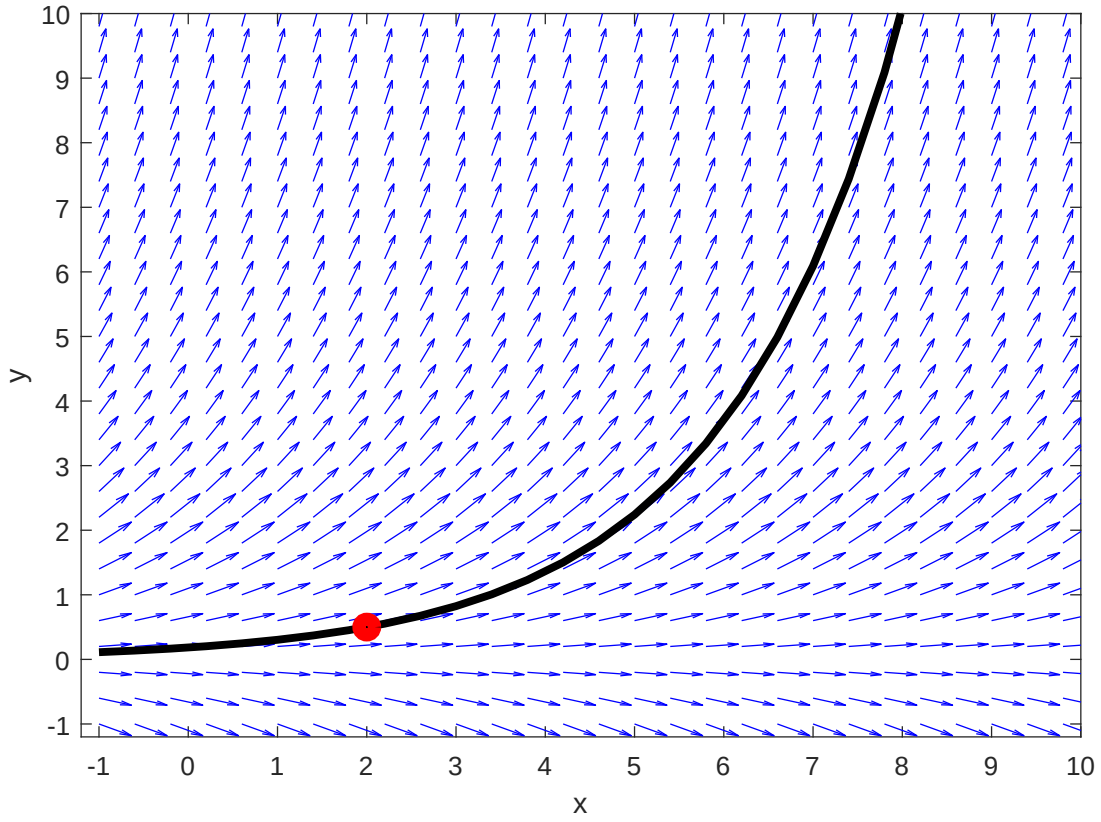


Figure 4.1: Illustration of an IVP. The blue vector field represents the ODE with infinitely many solutions. The point y_0, here shown in red, restricts the solution to a single function, the black line.

For the example above, if we were given the initial value of $y(2) = \frac{1}{2}$ it would restrict the solution of the ODE to be exactly $y(x) = 0.18394 \exp\left(\frac{1}{2}x\right)$, since 0.18394 is the only value for c that solves the ODE and passes through the initial value.

Figure 4.1 illustrates the idea of an initial value problem, using the example IVP presented above.

For our purposes, we are only interested in IVPs with first-order ODEs, meaning they only involve the first derivative of the function y. They have the general form

$$y'(x) = \frac{d}{dx}y(x) = f(x, y(x)).$$

We will use this form below when discussing the Runge-Kutta method.

4.2 Numerical solutions to IVPs

The above example was easy to analyze and therefore it was easy for us to obtain a single clear result. The example was also simple, however, and this is very rarely the case in reality. When your ODE is complicated, it may be extremely difficult, counterproductive or even impossible to analytically solve. In these cases, the only option left is to approximate a solution numerically. That is, use the ODE directly to infer the function values around the starting point. This has the disadvantage that we do not obtain a universal

and truly correct solution — we only obtain the estimates of function values at certain points originating from the initial value. Given enough computational resources, we can estimate function values at any finite range.

4.2.1 Runge-Kutta Method

Methods for performing this numerical estimation have already been developed. A well-known method is the Runge-Kutta method. It starts at (x_0, y_0) and estimates the next point (x_1, y_1) by using the value of the derivative at intermediate points between the current point and the next. The next point is determined by a fixed step-size Δx, which is chosen before starting the algorithm. Each iteration, the current point (x_n, y_n) is used as a new starting point from which the next point (x_{n+1}, y_{n+1}) is computed and this process can be iterated for as many points as desired.

Specifically, the Runge-Kutta method estimates the next function value $y(x_{n+1}) = y_{n+1}$ from the current one $y(x_n) = y_n$ by using these equations,

$$\begin{aligned} x_{n+1} &= x_n + \Delta x, \\ y_{n+1} &= y_n + \frac{1}{6}(k_1 + 2k_2 + 2k_3 + k_4), \end{aligned}$$

where the k's are estimations of the slope at different increments between y_n and y_{n+1}. They are defined as follows:

$$\begin{aligned} k_1 &= \Delta x \cdot f\left(x_n, y_n\right), \\ k_2 &= \Delta x \cdot f\left(x_n + \frac{\Delta x}{2}, y_n + \frac{k_1}{2}\right), \\ k_3 &= \Delta x \cdot f\left(x_n + \frac{\Delta x}{2}, y_n + \frac{k_2}{2}\right), \\ k_4 &= \Delta x \cdot f\left(x_n + \Delta x, y_n + k_3\right). \end{aligned}$$

To be precise, this method is the *fourth order* (referring to the number of increments k) Runge-Kutta method, but this is by far the most common version of the method and we will simply refer to it as "the Runge-Kutta method".

4.2.2 Probabilistic solutions to IVPs

The disadvantage with the purely numerical solutions is that they are not entirely accurate; they are estimates. This means that the result we obtain contains some error or uncertainty. However, algorithms like Runge-Kutta do not directly quantify this uncertainty. We would like to have an algorithm that takes this uncertainty into account, so that we may gauge how certain we are about the results. Ideally, we would like the algorithm to calculate a probability distribution over the results so that we may analyze them using parametric statistics, e.g. confidence intervals. It would be especially useful to be able to take samples from this distribution as well as examine the distribution parameters.

Fortunately, this is an active research area and such algorithms have already been developed. One such algorithm for solving initial value problems probabilistically has been developed by Michael Schober, Simo Särkkä and Philipp Hennig [21]. The algorithm uses the topics that we have talked about so far to produce a probabilistic ODE solver. It combines the Runge-Kutta method with a Kalman filter to produce a Gaussian process which can then be analyzed and sampled from.

Schober, Särkkä and Hennig defines their ODE solver themselves as:

Definition 2 ([21])**.** *"A* probabilistic filtering ODE solver (PFOS) *is the Kalman filter applied to an initial value problem with an underlying Gauss–Markov linear, time-invariant SDE and Gaussian observation likelihood model" .*

Let us unravel this rather cumbersome definition. "Gauss-Markov" refers to the usage of a Gaussian process (it happens to also be a Markov process, which means that it only uses the previous point for estimation of the next). SDE is an acronym for "stochastic differential equation", meaning it is a differential equation with some notion of uncertainty or noise. Linear and time-invariant refer to the structure of this SDE, linear meaning it only uses linear operations and time-invariant meaning that the parameters of the function do not depend on the argument x (which is often time). By Gaussian observation likelihood model, it is meant that the observed data also includes uncertainty or noise distributed by a Gaussian distribution.

The algorithm functions by looping through the Kalman filter algorithm after an initialization period. The initialization consists of running the Runge-Kutta method for the first couple of iterations to reach a *steady state* in the Kalman filter [21]. After this initialization, the Kalman filter algorithm is used. In each loop iteration, a new prediction is made of the next point. This prediction is used to evaluate the differential equation at the next point and the result is used to update the predicted point for the next iteration.

The iterations run until a certain x-value, that the user sets. When it is done, the result is an approximation of the function values at each of the desired points, in the form of a mean vector and a covariance matrix. Theoretically, sampling from these should be possible, but we would like to use smoothing on the result before we sample.

Therefore, to finalize the process, the points are run through the Rauch–Tung–Striebel smoothing algorithm. This produces a Gaussian process with a mean vector and covariance matrix which can be used for sampling.

Example usage of the PFOS

Using the PFOS requires the definition of an ODE and an initial value. If we simply plug in the example that we used above, we have the following definition for the ODE

$$y'(x) = \frac{1}{2}y(x),$$

and the initial value

$$y(2) = \frac{1}{2}.$$

Figure 4.2 shows the result of the PFOS applied on the above IVP. As can be seen, the PFOS' solution is nearly identical to the analytical solution (not exactly identical however). Also notice that the PFOS only finds the points ahead of the initial value, since it follows the derivative.

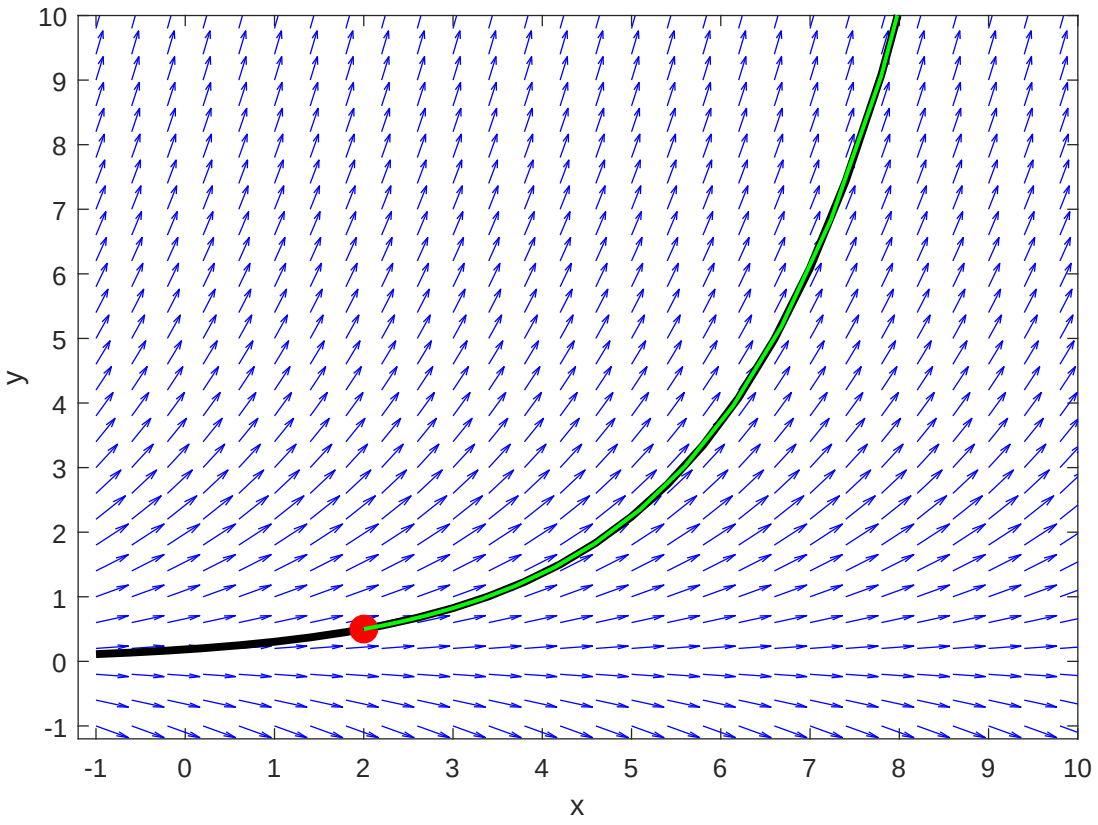


Figure 4.2: The probabilistic filtering ODE solver applied to a simple initial value problem. The black line is the analytical solution and the green line is the PFOS' solution, in the form of the mean vector.

5
Application in Tractography

In this section we introduce the subject of tractography and our application of the previously mentioned methods to this problem. We will begin by defining the problem and describing how the data we use was generated. We will then explain how our brain data can be considered an ordinary differential equation on which the probabilistic filtering ODE solver (PFOS) of Schober et al. can be applied. We will then describe how we applied this solver on the problem and present our results.

The model algorithm constructs an n-dimensional Gaussian process as output, which represents the n estimates of the solution to the IVP given to it. In this case, the task is to infer the location of brain tracts. The Gaussian process is constructed by use of the Kalman filter: at each step, the ensuing location is estimated by first predicting its location, then observing a measure by plugging the prediction into the differential equation for the IVP (which is done by interrogating the preprocessed brain data, as described in Section 5.3). The prediction is then updated based on this 'observation' of the derivative. In the end, the filtered estimates are smoothed as described in Section 3.3. In short, the approach combines all the theory of Sections 2, 3 and 4 into the modelling.

We will apply the PFOS to different tracts in the brain. The results of the application will be a 3-dimensional probabilistic estimate of the neural tracts in the brain, in the form of a Gaussian process. It is probabilistic in the sense that we will be able to gauge the amount of uncertainty in the results by examining the distribution parameters (the covariance matrix) and/or a large amount of samples.

It should be mentioned that we do not really have any medical expertise or knowledge about the brain that we could use to more rigorously interpret the results. Our goal is merely to evaluate the effectiveness of Schober et al.'s solver on this problem. As with many things in computer science and mathematics, the problem and its solution are interesting solely for the applications in other fields. As computer scientists, we serve to use our field to enhance others'.

5.1 Problem definition

Tractography is the problem of finding *neural tracts* in the brain. Neural tracts are bundles of nerves connecting different parts of the brain. Knowledge of these tracts is interesting as it helps understand the structure of the brain, which may lead to a better understanding of the brain's function.

The problem of tractography is to use the diffusion MRI data (see below) to estimate the nerve tracts in the brain. The probabilistic approach has previously been used in tractography, where shortest paths between points have been estimated [17, 15].

5.2 Description of Data

In order to learn anything about the human brain without cutting in people, *diffusion-weighted magnetic resonance imaging* (Diffusion MRI) is used. Diffusion MRI measures

the diffusion of water in biological tissues. These measurements can be used to generate data representing the flow of water in the brain, since this flow and the neural tracts correlate.

The diffusion MRI data has been preprocessed to be easier to work with. The data essentially consists of a giant 3-dimensional matrix, where each entry corresponds to a point in the 3-dimensional image of the brain. Each entry contains a vector that points in the direction of maximal water flow at that point. To be precise, the vector is more like a representation of a line — the vector may as well point in the opposite direction. Essentially, all we know is that the water flows along this vector, either in the same or opposite direction. In this way, the brain data can be viewed as a big, 3-dimensional vector field.

5.2.1 Data Acknowledgement

We did not procure this data ourselves — we were given a preprocessed data set from a single subject (100307) of the preprocessed Human Connectome Project (HCP) diffusion data [11, 14, 12, 4, 8, 16, 18]. We use the principal eigenvector from the DTI tensor, obtained by using FSL's dtifit. A brain mask was computed using FSL's BET and a tissue classification was computed using FSL's FAST [9, 3].

5.2.2 Relation to Ordinary Differential Equations

As described above, the brain data that we work with is essentially just a big 3-dimensional vector field. However, an ODE is actually equivalent to a vector field, as was illustrated earlier in figure 4.1. The central idea of the method now becomes apparent; consider the brain data (3D vector field) as an ODE. Choose some interesting starting point within the brain and use the PFOS with the brain data and this starting point. Because we trick the PFOS to merely see the brain data as an ODE, it will generate a function (i.e. a path) which starts at the starting point and continues in the direction of the water flow. In this way, we will be able to generate and visualize the nerve tracts inside the brain. This is in part why it also makes use of the Kalman filter: Observations are generated as we move on the estimated function trajectory. For this reasons, we cannot a priori observe in batches, even if we wanted to; the observations are generated online, and continuously depend on previous estimates.

5.3 Applying the PFOS on the Brain Data

There are some challenges with applying the PFOS on the brain data:

Global definition: The brain data is not an *actual* ODE, we are just tricking the PFOS into considering it one. Regular ODEs are globally defined, i.e. they have a derivative value at every possible point. However, the brain data only has a discrete set of voxels but the PFOS should still be able to get a derivative value from voxels in-between the voxel centers.

Vector directions: The vectors in the brain data are really just a representation of a line, and the opposite vector represents the same line, so the ODE needs to be able to consider it as a vector in the opposite direction as well.

Limited range: The PFOS's result must be contained inside the brain and cannot be allowed to run outside the skull. This requires some handling if the PFOS reaches

the edge of the brain. There are also some parts of the brain and skull that we would like to avoid running into (i.e. non-brain, cortico-spinal fluids etc.).

5.3.1 Construction of Ordinary Differential Equation

As described above, the brain data is equivalent to an ODE. Thus, the ODE we give to the PFOS will simply be a function that makes a look-up into the brain data and returns the derivative at the requested point. That way, when the PFOS requests a derivative from the ODE, it will receive the derivative that corresponds to the brain data's water flow. This is the core concept of the ODE but we still need to address the above challenges.

Global definition

In order for the ODE to be globally defined, we need to somehow extend the definition of the ODE outside the voxel centers, since we only have data points at discrete values. We do this by interpolation. We considered two different ways to do this: Nearest neighbour interpolation and linear interpolation.

Let us imagine a simple 2D example. We have the brain data as a matrix $\mathbf{B}$ and we have a point $p = (p_1, p_2)$ that lies withing the bounds of the brain data. We wish to calculate a derivative δ for this point. Assume that neither p_1 or p_2 are integers, meaning we cannot query the brain data at p directly. Now imagine that we have the vectors surrounding the point p

$$\begin{aligned}
\mathbf{B}_{\lfloor p_1 \rfloor, \lfloor p_2 \rfloor} &= \mathbf{y_1} = (y_{11}, y_{12})^\top, \\
\mathbf{B}_{\lfloor p_1 \rfloor, \lceil p_2 \rceil} &= \mathbf{y_2} = (y_{21}, y_{22})^\top, \\
\mathbf{B}_{\lceil p_1 \rceil, \lfloor p_2 \rfloor} &= \mathbf{y_3} = (y_{31}, y_{32})^\top, \\
\mathbf{B}_{\lceil p_1 \rceil, \lceil p_2 \rceil} &= \mathbf{y_4} = (y_{41}, y_{42})^\top,
\end{aligned}$$

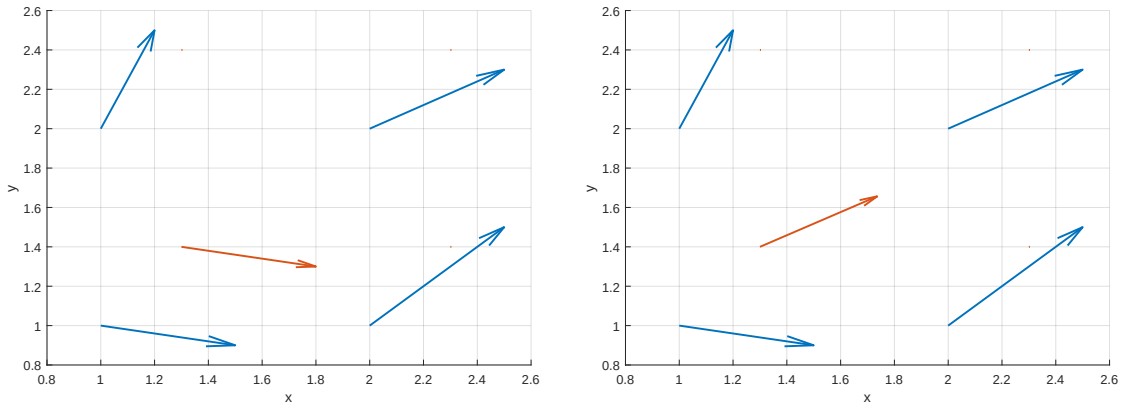
where y_n are shorthands for the vectors around p, saved in b. The "$\lfloor \rfloor$" and "$\lceil \rceil$" denotes floor and ceiling functions.

With nearest neighbour interpolation we would simply return the vector closest to the requested point. This has the disadvantage that the derivative may change very suddenly as the nearest vector changes. We would simply calculate the derivative by rounding the point p and query the brain data at the resulting point

$$\delta = \mathbf{B}_{\text{round}(p_1), \text{round}(p_2)}$$

With linear interpolation we would calculate the average vector of the vectors immediately around the requested points (4 vectors in 2D, 8 in 3D). The vectors would be linearly weighted by their distance to p. This method results in a somewhat more smooth transition as we move from one area of vectors to the next. We calculate it as follows:

$$\begin{aligned}
D &= \sum_n ||\mathbf{y}_n - p||, \\
d_n &= \frac{D}{||\mathbf{y}_n - p||}, \\
w_n &= \frac{d_n}{\sum_m d_m}, \\
\delta &= \sum_n w_n \mathbf{y}_n
\end{aligned}$$



(a) (b)

Figure 5.1: Example and comparison of nearest neighbour and linear interpolation. Blue arrows represent data while red arrows are interpolated values where there isn't any data. (a) shows the effect of using nearest neighbour, which may result in a bad representation of the data. (b) shows the effect of linear interpolation, which fits the overall data better.

where W is the sum of the distances from p to the vectors and w_n is the weight factor for vector $\mathbf{y_n}$.

Figure 5.1 illustrates the difference between the two methods in a simplified 2-dimensional example. When developing the ODE we tried both methods and decided that linear interpolation worked best.

Vector directions

In order for the ODE to be able to consider the vectors as flow in both directions, we need to flip the vectors under certain conditions. Specifically, we will flip a vector if the flipped vector aligns better with the previous derivative than the non-flipped vector does. This corresponds to the inner product of the previous derivative and the vector being negative. In this way, we always consider the vector in the most favourable direction with regards to our current derivative. It also means that our derivative will not change too much at once, since it cannot turn by more than 90 degrees in a single iteration of the PFOS.

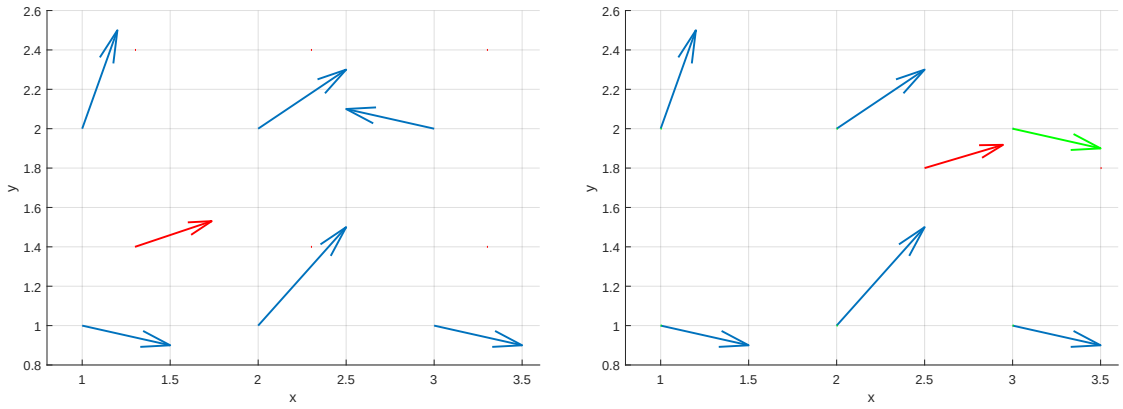
Figure 5.2 illustrates the idea of flipping the vectors in a simplified 2-dimensional example.

Limited range

Because we make a look up at every step, the model can naturally be constrained to remain in the 3D matrix of the brain. This is handled by taking advantage of the data structure, which denotes areas outside of the brain with zero values. The bounds on the 3D matrix is also known, which allows us to check if the model has moved past the limits of the matrix itself. Both properties are checked during each step of the filtering. If any of the two are true, the differential equation simply returns a zero vector as derivative, halting any further movement of the model.

In addition, there are certain physiological and data related constraints that can be taken into account. One is the fractional anisotrophy, FA, of the data:

$$\text{FA} = \sqrt{\frac{1}{2}} \frac{\sqrt{(\lambda_1 - \lambda_2)^2 + (\lambda_2 - \lambda_3)^2 + (\lambda_3 - \lambda_1)^2}}{\sqrt{\lambda_1^2 + \lambda_2^2 + \lambda_3^2}},$$



(a) (b)

Figure 5.2: Illustration of vector flipping during derivative interpolation. Blue arrows represent data while red arrows are interpolated values where there isn't any data. (a) shows the previous step, where the red arrow is interpolated from its four surrounding blue arrows. (b) shows the next step. When the vector at (3, 2) is encountered, it is flipped because its flipped direction fits the previous derivative better (inner product is negative). The green arrow represents the flipped vector that is used for interpolation of the red arrow in (b).

where λ_1, λ_2 and λ_3 are the eigenvalues of the data (in reality, the FA is based on the underlying raw non-processed data, not the data we use) [6]. Simply explained, the FA is an indication of how certain the measured directions are at the current location. If the FA is low, there is no certain direction to follow, and thus one could arguably stop the modelling process from going any further. We decide on a certain threshold for the FA, and then check if the PFOS reaches a location where the FA is below this threshold. If the FA is too low, the differential equation returns the zero vector as derivative. For our results, we have used a standard practice FA threshold of 0.2.

We have also made use of a white/grey matter mask to make sure that we do not exit the parts of the brain that contain nerve fibers. If a tract moves outside this mask, we stop it again by setting the derivative to zero. This filters out a few errant tracts found by the PFOS, though the results largely stay similar.

5.3.2 Construction of Initial Value

With the ODE defined, we just need to define an initial value for it to be a proper IVP. This essentially amounts to choosing some starting point inside the brain and applying the PFOS from that point. Doing this, we can obtain a single tract, or curve estimate, through the brain, which varies depending on the starting point. Figure 5.3 illustrates this.

From this single tract, which is a Gaussian process, samples can be drawn. Figure 5.4 shows a zoomed in view of samples taken from the same tract as in figure 5.3. It also shows contours of the tracts, denoting two standard deviations, i.e. a 95% confidence interval. As can be seen, the samples are very close together and the confidence interval is quite narrow, suggesting that there is some uncertainty that our model does not capture. We discuss this further in Section 6.3.1.

However, we do not want to find just a single tract. We want to find many tracts, to get

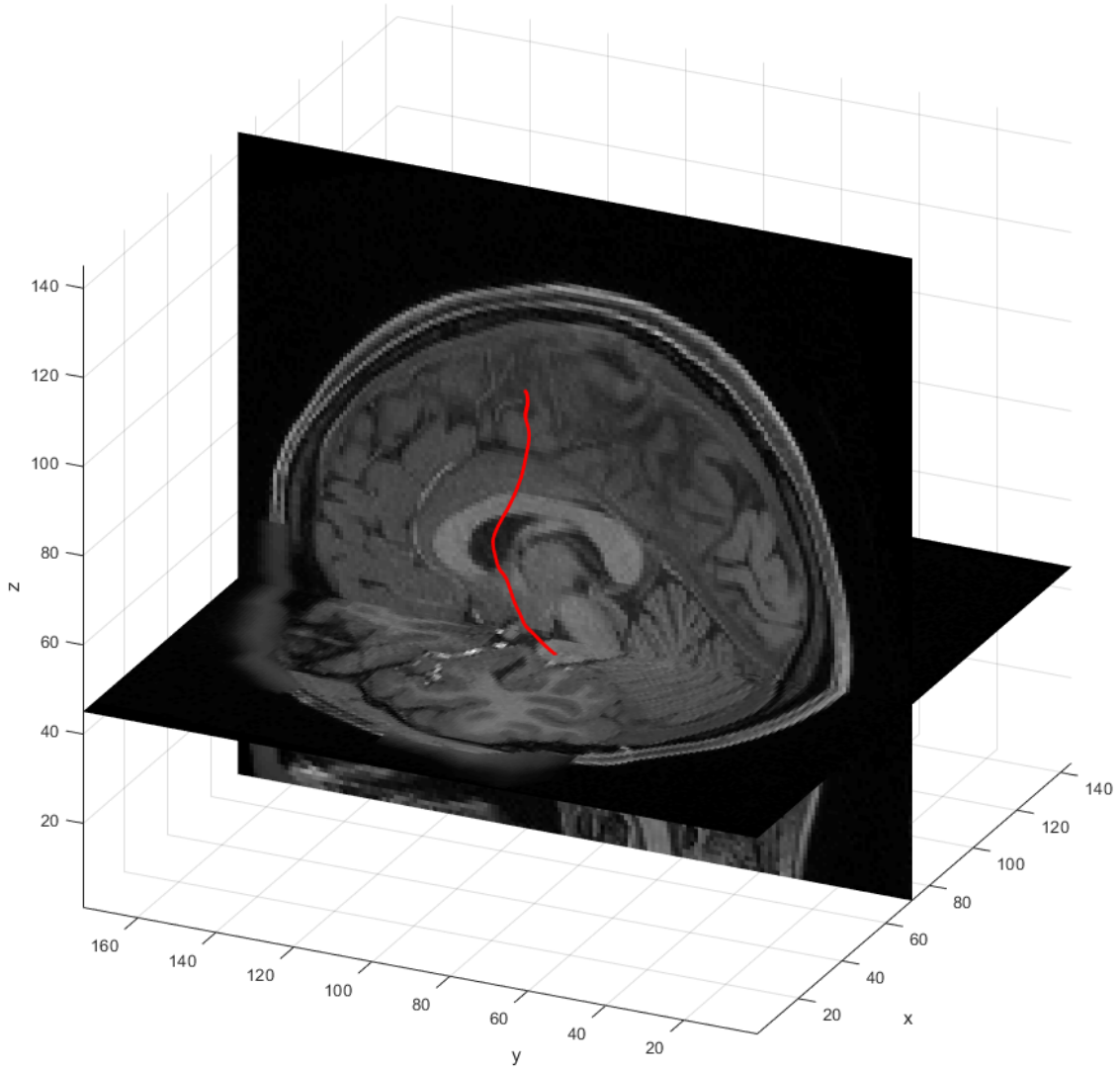


Figure 5.3: Single tract found on the left side of the brain, represented by its mean function.

a better idea of the network of the tracts in the brain. To do this, we run the PFOS with many different starting points. We choose which starting points to use by looking at a data set containing certain "regions of interest". These regions in the brain lie at the ends of prominent tracts in the brain [19].

The regions are stored as a mask over the brain. When applying the PFOS, we iterate through the entire brain and apply the PFOS on the current point if the point is included in the region of interest. We look at three different prominent tracts: Fornix, the inferior fronto-occipital fasciculus (IFOF) and the corticospinal tract (CST). All three tracts have both a left and a right version, corresponding to the halves of the brain, i.e. 6 tracts in total. Each tract has two regions of interest at each end, resulting in 12 regions of interest in total.

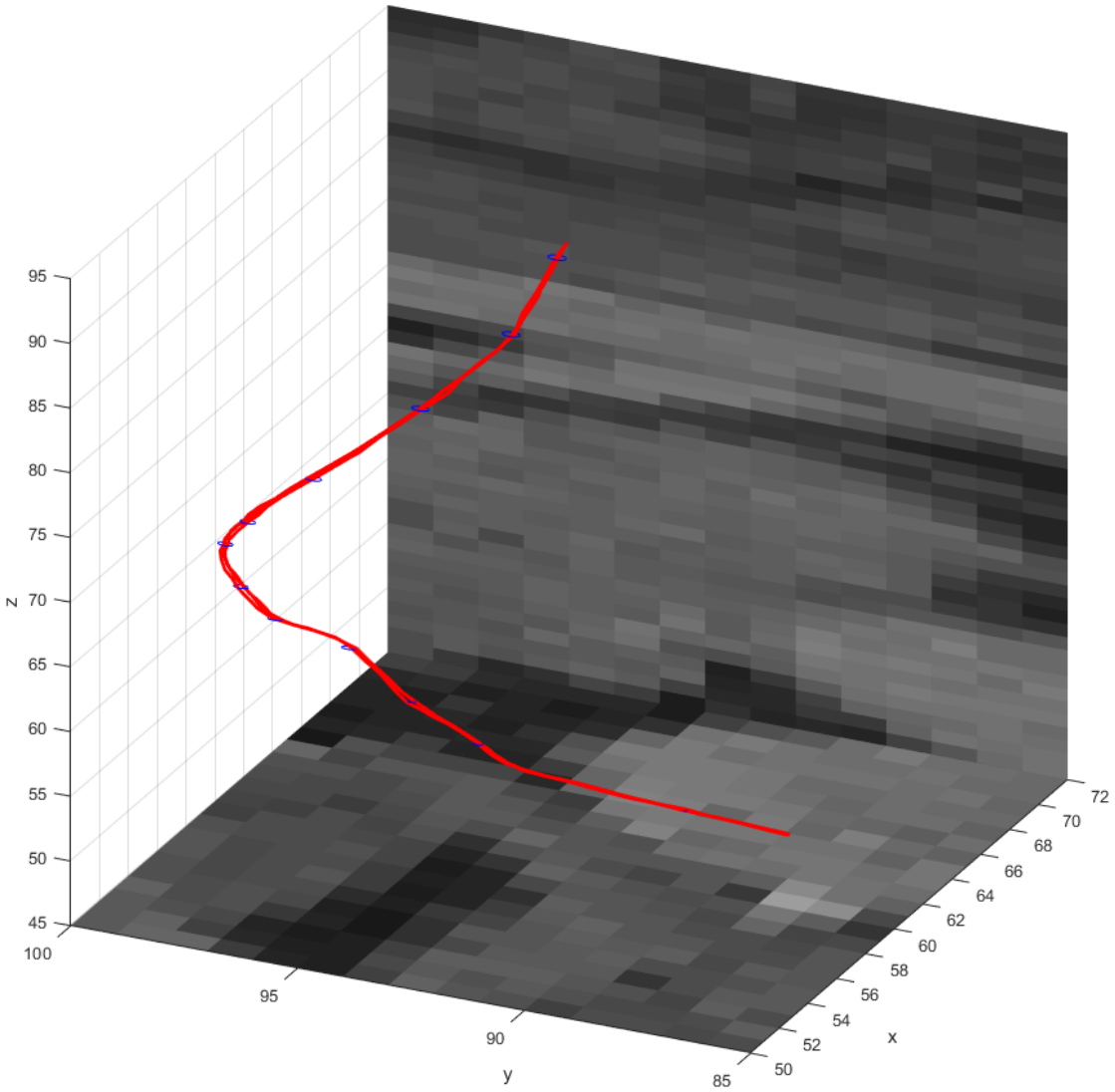


Figure 5.4: Three samples drawn from the same tract as in figure 5.3. The red lines are samples while the blue circles denote the 95% confidence interval (two standard deviations).

5.4 Results

Here we present the results obtained by applying the PFOS on the brain data. The modelling algorithm is applied three times on the data; each iteration models one of the three major brain tracts, the Fornix, the Inferior fronto-occipital fasciculus (IFOF) and the Corticospinal tract (CST). We will show the result for each region of interest separately. To evaluate the results, we visually compare them with the tract atlas obtained as in Kasenburg et. al. [19] i.e. areas where the tracts are expected to be located. We consider this tract atlas a kind of ground truth for comparison.

The results are visualized together with brain slices from a corresponding MRI image, showing the brain from the side, back and above, and as 3D figures with an orthographic projection.

5.4.1 IFOF

Figures 5.5 and 5.6 show the tracts computed by running the IFOF regions of interest. Figure 5.5 shows the tracts compared to the ground truth. Figure 5.6 shows the tracts in 3-dimensional space, seen from both brain halves. We managed to find the IFOF tract reasonably well — the overlap between our found tracts and the ground truth is significant. We do find other tracts as well though. The slice values for figure 5.5 are $x = 48$, $y = 87$ and $z = 56$. There are 2187 points in the regions of interest, from which the algorithm has been initialized.

5.4.2 CST

Figures 5.7 and 5.8 show the tracts computed by running the CST regions of interest. Figure 5.7 shows the tracts compared to ground truth. Figure 5.8 shows the tracts in 3-dimensional space, seen from both brain halves. The CST region of interest is much, much larger than the other regions. It takes about 20-30 minutes to finish the computation of the CST tracts, compared to the few minutes of the other regions. However, while the PFOS does find the CST tract, it seems like it finds a lot of other unrelated tracts as well. This is especially evident in figure 5.8. We suspect this is due to the size of the CST region of interest. The slice values for figure 5.7 are $x = 90$, $y = 90$ and $z = 63$. There are 8807 points in the regions of interest, from which the algorithm has been initialized.

5.4.3 Fornix

Figures 5.9 and 5.10 show the tracts computed by modelling the tracts originating in the Fornix region of interest. Figure 5.9 shows the tracts compared to HCP's ground truth. Figure 5.10 shows the tracts in 3-dimensional space, seen from both brain halves. As can be seen, we did not manage to find the Fornix tract very well — Fornix has by far been the most difficult tract to adapt this method to. We are not alone in this problem. The Fornix tract is known to be difficult to find. The slice values for figure 5.9 are $x = 69$, $y = 98$ and $z = 50$. There are 853 points in the regions of interest, from which the algorithm has been initialized.

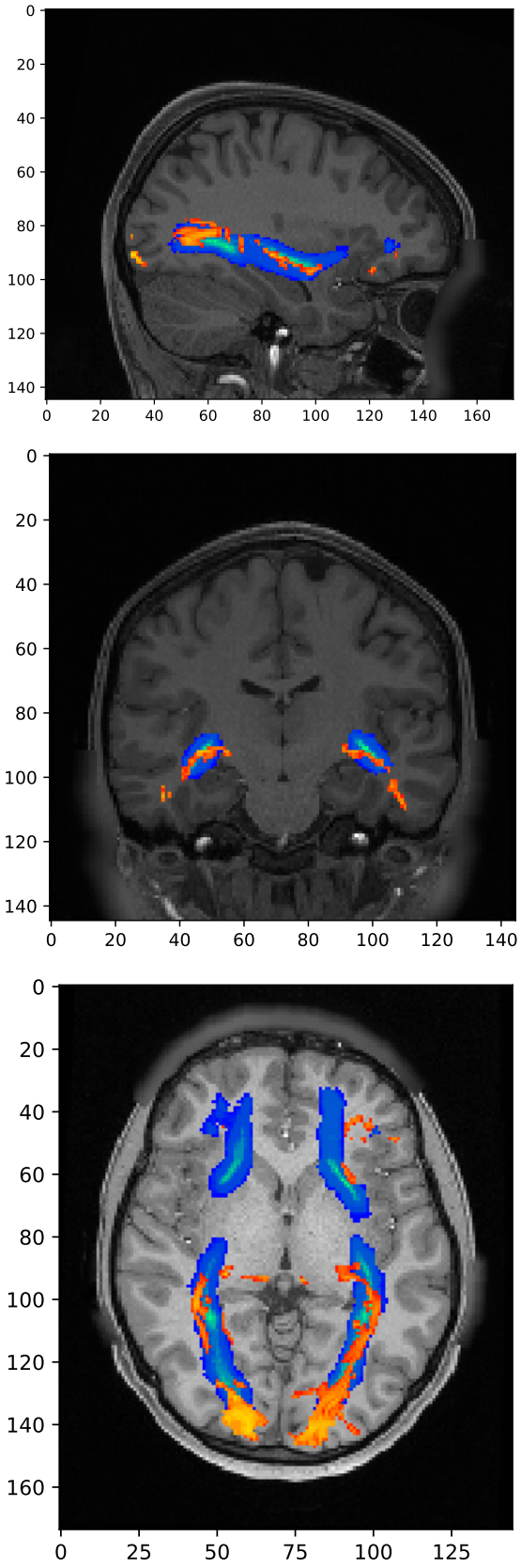


Figure 5.5: The IFOF tracts results, shown from three different angles. The blue area is the tract as provided by HCP. The orange area is the tract as calculated by the PFOS.

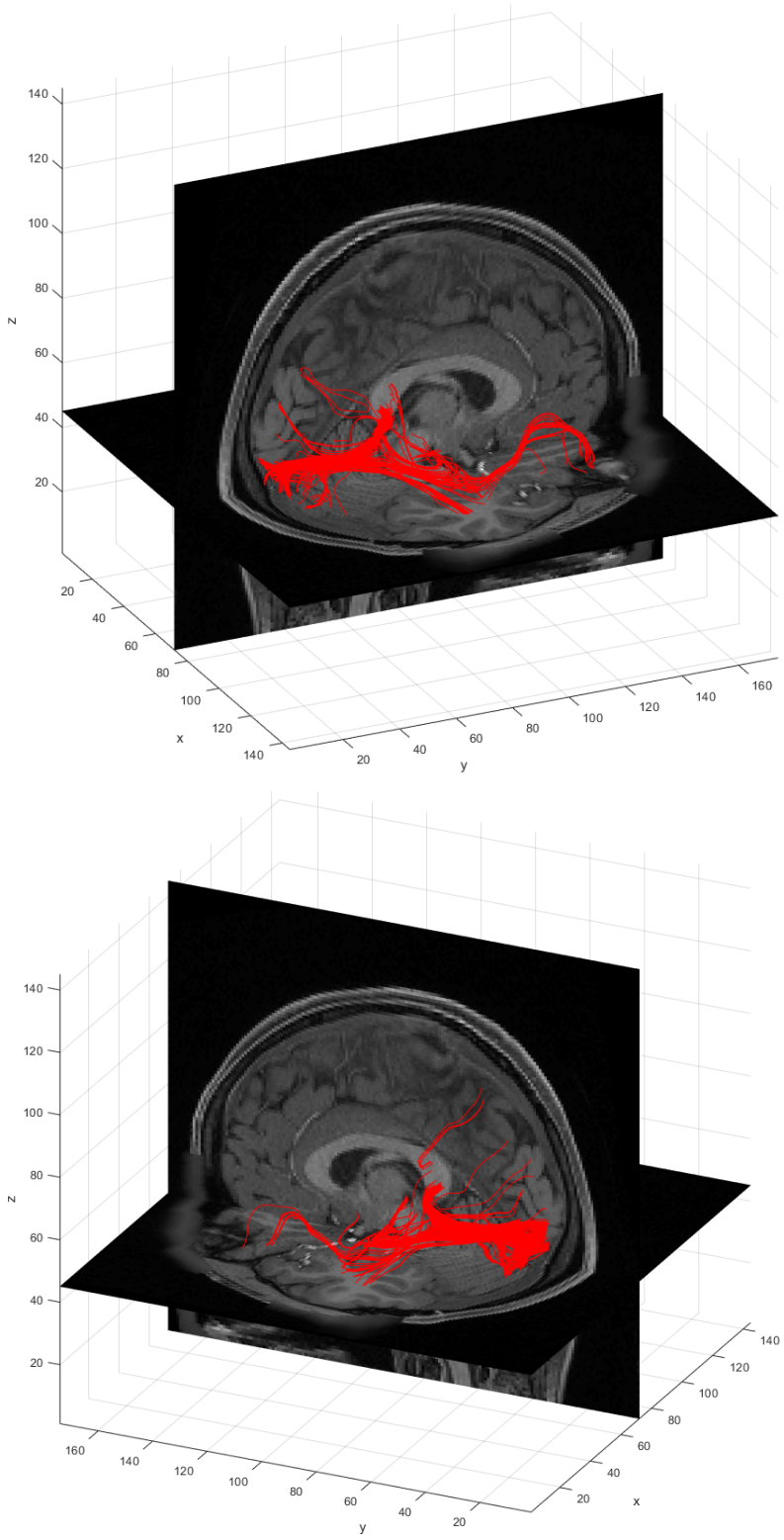


Figure 5.6: The IFOF tracts results, seen from two angles. The red lines show the tracts found by the PFOS.

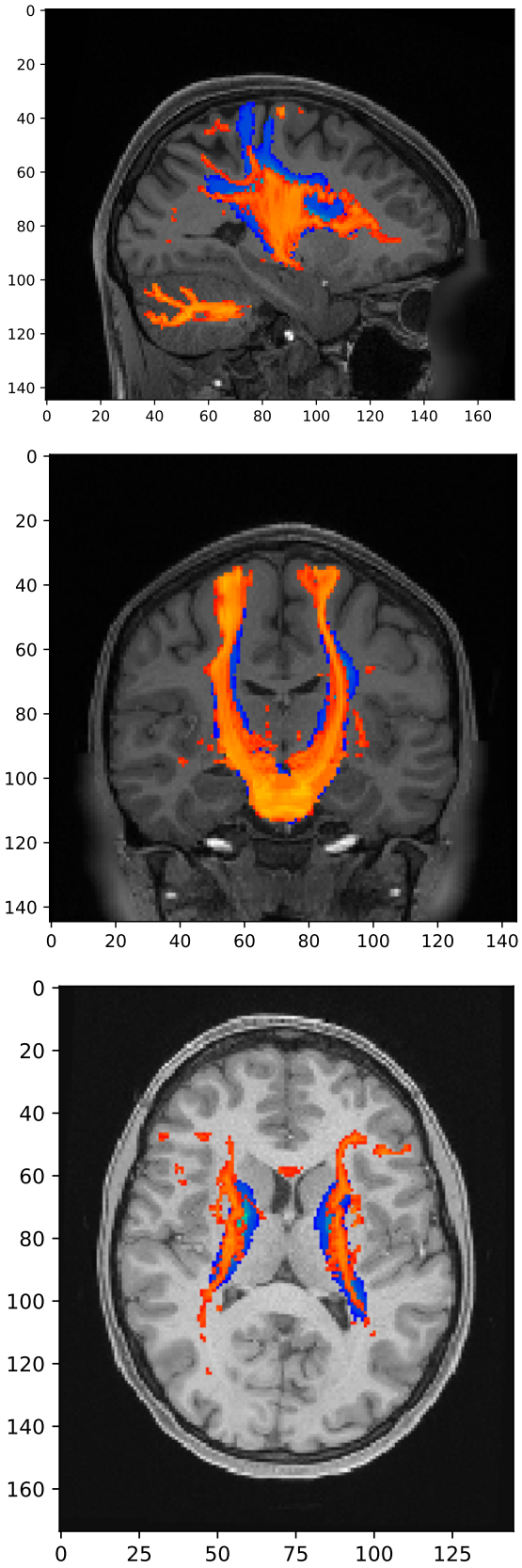


Figure 5.7: The CST tracts results, shown from three different angles. The blue area is the tract as provided by HCP. The orange area is the tract as calculated by the PFOS.

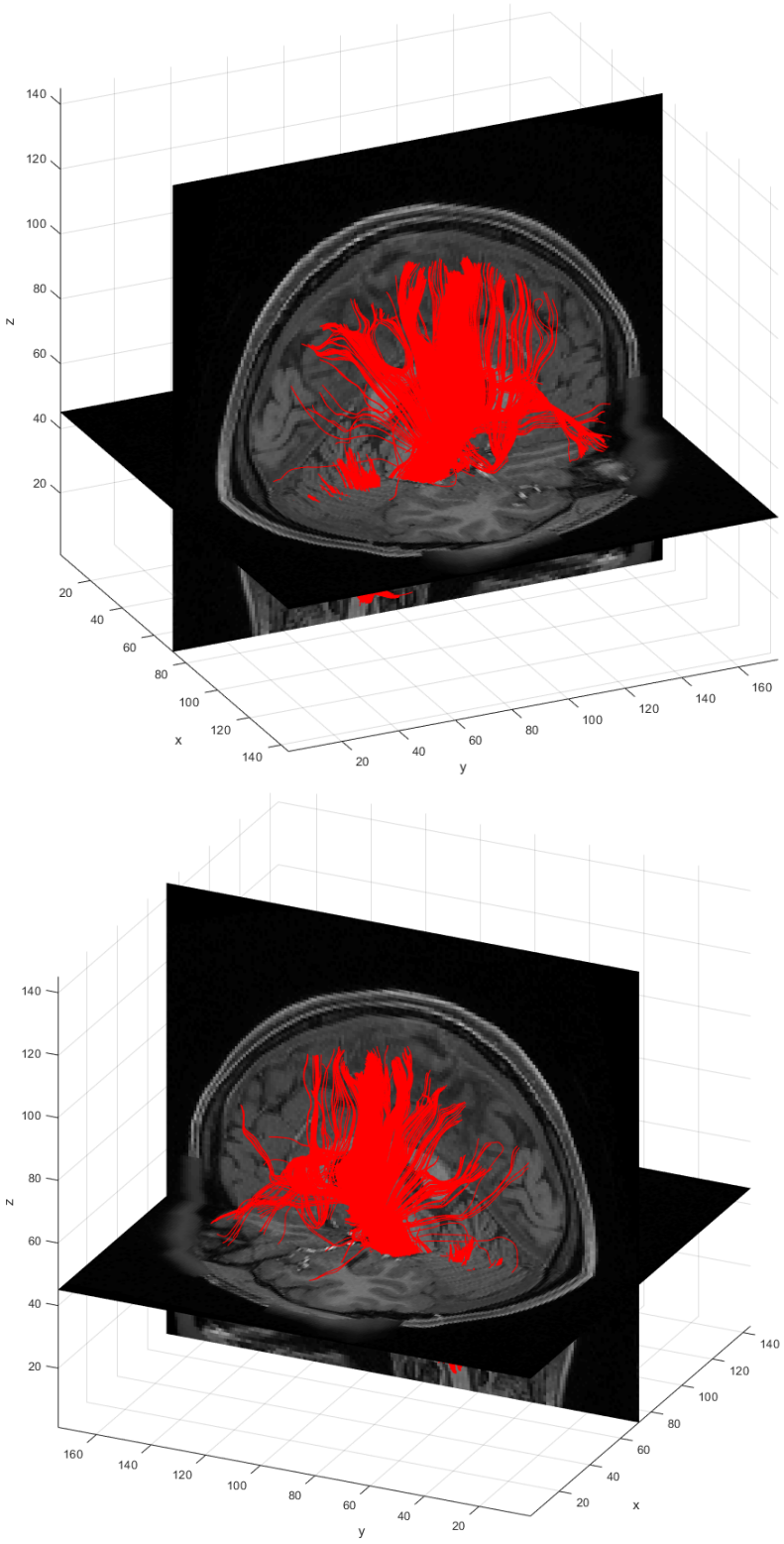


Figure 5.8: The CST tracts results, seen from two angles. The red lines show the tracts found by the PFOS.

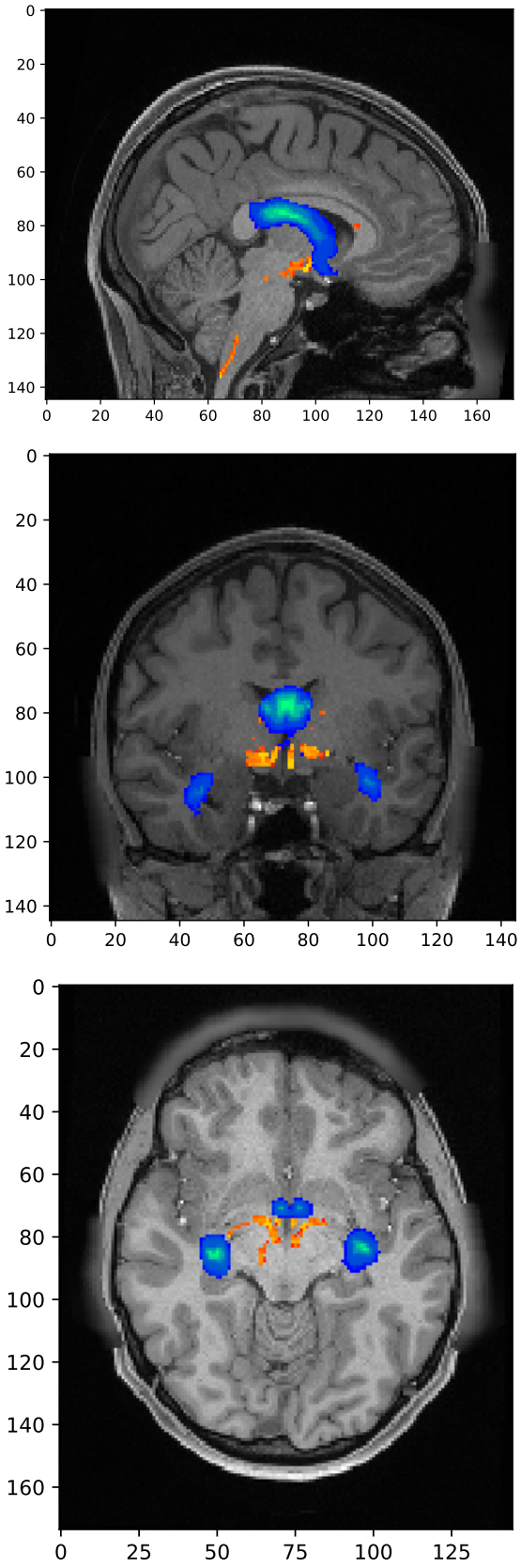


Figure 5.9: The Fornix tracts results, shown from three different angles. The blue area is the tract as provided by HCP. The orange area is the tract as calculated by the PFOS.

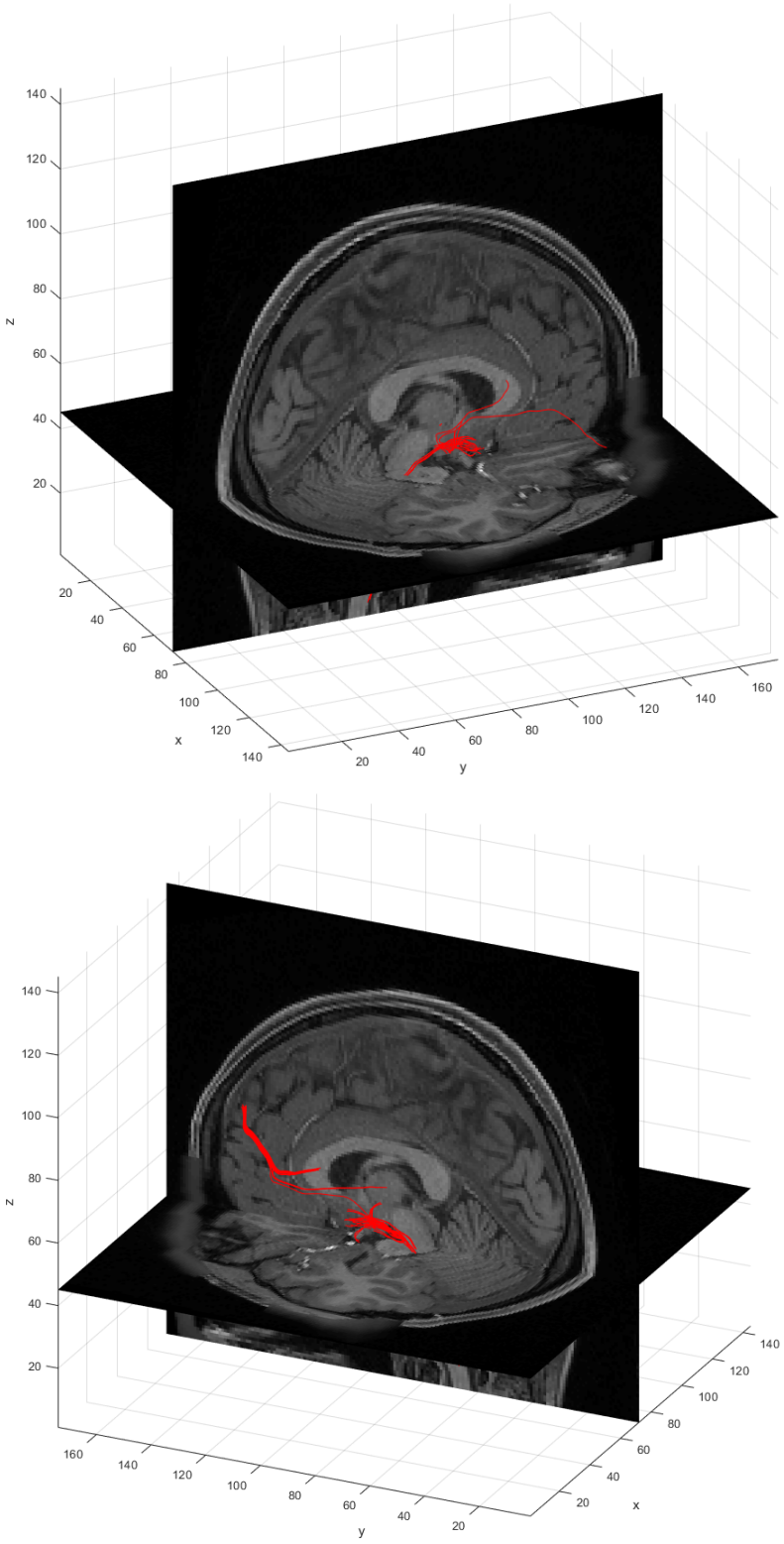


Figure 5.10: The Fornix tracts results, seen from two angles. The red lines show the tracts found by the PFOS.

6
Discussion

In this section we will discuss our results and make some conclusions on the effectiveness of the probabilistic filtering ODE solver on the problem of tractography. We will also discuss various improvements that could be applied to the model as well as the model's pitfalls and limitations.

6.1 Quality of Results

Here we will discuss the quality of our results, i.e. the tracts that we have found. It is important to note that we do not have a quantitative measure for how good or bad our results are. We are left with comparing our results to the ground truth from the HCP.

As can be seen by figure 5.5, we have managed to estimate the IFOF tract with good accuracy. While there are some errant tracts, the majority of the found tracts appear to lie on or around the ground truth. In general we are quite satisfied with the model's performance on the IFOF tract.

The CST tract is the most "extreme" tract, in a way. The region of interest is far larger than for the other tracts and the corresponding tracts seem more extensive as a result. As figure 5.7 shows, we manage to almost entirely cover the ground truth areas, to the point that they are barely visible. While this is good, since we would like to find these tracts, the CST tract also finds *a lot* of false positives in the form of unrelated tracts. Figure 5.8 also shows this, as it is clearly the most extensive tract of the three tracts we examined. With the CST region of interest, the model has a high tendency to go into the cerebellum in particular, as can be seen in the first image of figure 5.7.

We can see by figure 5.9 that some tracts inside the Fornix ground truth area have been found, but not very many. The tracts that have been found are very thin in comparison to the ground truth and there are many errant tracts outside the ground truth area. Figure 5.10 shows the very sparse tracts found from the Fornix region of interest. It seems that the filtering done by the fractional anisotrophy and the white and grey matter mask has been slightly too zealous. Applying the PFOS on the Fornix tract with lower thresholds does result in a more thorough tract but also finds a lot more errant tracts as well. The fit of the estimate is poor compared to the CST and the IFOF estimates.

The amount of starting points in the region of interest for each brain tract seem to be of significant influence; the more initial values, the more promising the result looks. This may not be the cause, as some tracts such as the Fornix can simply just be harder to estimate. Fornix is hard to find mostly due to its curvature.

The results seem to fit the ground truth to a certain degree, but it also has a tendency to traverse tracts that are unrelated to the tract we're trying to find. This is especially true for the Fornix and the CST tracts. We believe this mostly has to do with tracts that cross each other and the general random movement of the PFOS's path through the tracts. If tract A connects perpendicularly to tract B and the PFOS happens to be following along on the edge of tract A, it may very easily catch some of the vectors of tract B and start

following that tract instead. We try to limit this behaviour to some degree by using the fractional anisotrophy as a threshold, but it is difficult to use that to control which paths the PFOS should traverse and which it should not. The masking by white and grey matter helps to remove some of these errant tracts but not all. We suspect that there is no easy way to get around this erratic tract-switching.

Even with the small amount of erratic tracts, we find the results satisfactory and we believe the method has potential for more use.

6.2 Conclusion

We have estimated brain tracts using probabilistic numerics and diffusion MRI data. We argue that the combination of Gaussian processes, the Kalman filter and the field of probabilistic numerics is a useful tool in the topic of tractography, since this combination makes it possible to capture the uncertainty, while maintaining computational feasibility. A central decision was how to interpolate the derivatives from the brain data. We have examined different approaches and have produced an algorithm for calculating these derivatives, while taking different kinds of thresholds into account. The algorithm uses linear interpolation on adjacent derivatives and vector flipping, as described in 5.3.

Furthermore, the uncertainty of the estimations are explicitly incorporated into the output estimates, which can readily be visualized. Based on the results, we have generated 3D figures of the tract estimates and heat maps on slices of the brain. These are shown in figures 5.5, 5.6, 5.7, 5.8 and 5.9, 5.10. In addition, we give examples of a single tract including noise, and a single tract mean estimate. The results seem to correspond well with expert opinion in the form of the ground truth values from the HCP, but we give no quantitative measure of the estimates' correctness.

Especially the IFOF and CST tracts have been estimated with satisfactory accuracy. Even if we did not manage to estimate the Fornix tract very well, we do believe that our model has potential and that with further work (as described below) it could be improved.

6.3 Further work

Here we discuss various things that could be done to improve the model.

6.3.1 Including uncertainty in the derivatives

As it currently stands, our method has some uncertainty in the resulting tracts, due to the fact that the PFOS is a numerical approximation. This is what causes samples from the resulting Gaussian process to be slightly different.

However the samples are not *very* different and the method probably does not account for all the uncertainty that is present. There are likely many reasons for this but we believe one big cause comes from the certainty of the derivatives.

When the PFOS tries to approximate the true function from the derivatives, it considers the derivatives as absolutely true — this is problematic because our derivatives (the brain water flow) does have a lot of uncertainty. This uncertainty may not be accounted for because of the way the PFOS works.

It may be possible to remedy this by modifying the PFOS to consider the derivative as a distribution of derivatives. This would require us to find some way to quantify the

uncertainty in the derivatives, however. Even without that issue, modifying the PFOS in this way may change the algorithm so severely that many of the properties that Schober et al. argued for are no longer present. In short, we are unsure of the implications of including uncertainty in the derivatives, if this is even possible. We also believe it would require quite an extensive amount of work.

6.3.2 Quantitative validation of results

As mentioned in Section 6.1, we do not have a concrete mathematical way to measure the quality of our results. Combined with including noise in the derivatives, it may be possible to measure a likelihood for the tracts. This may even allow us to calculate a likelihood for each individual tract and then filter those with low likelihood.

Such a measure would also allow us to say with more confidence how good or bad our results are. Unfortunately we have not had the time to figure out a way to calculate such a measure.

6.3.3 Improving the efficiency of the code

The computational efficiency of the PFOS solver on a single brain tract is very good — it takes less than a second on one of the authors' machine to run the filter and smoother.

However, we run the filter and smoother on huge amounts of points when running through the regions of interest discussed in Section 5.3.2. On the CST region, it takes upwards of 30 minutes to run through all the tracts. We believe this could be severely improved by using concurrency. Each tract is computed independent of the other tracts, so in theory it should be possible to run the algorithm concurrently, thereby greatly increasing the efficiency on multi-core processor systems. This could potentially increase the running speed proportional to the number of cores.

In the extreme case, one could imagine running the tracts on a GPU. A GPU would be able to run many thousands of tracts concurrently, potentially making the computation almost as fast as a single tract on a single-core CPU.

We attempted adapting our code to multi-core CPU concurrency, however this proved to be very difficult due to the fact that the code is written in Matlab. Matlab provides some basic concurrency functionality in the form of `parfor`, a parallelized for-loop, but this imposes constraints on the code inside the loop, due to memory safety concerns. Matlab does not provide any means of handling concurrency and memory safety manually, which makes it very hard to use concurrency in our code. We actually did try to modify our code to be able to use `parfor`, however it only made it slower.

We suspect that the code would be significantly faster if it was rewritten in another programming language like C/C++, Java or similar fast lower-level languages. However, we do realize that such a rewriting would redouble the complexity of the code and would require a lot of work. Plotting results etc., would probably require some Matlab or Python code to be used anyway. Then again, this computation would probably not have to be run that often, so it being slow is not a huge detriment.

6.3.4 Expanding the derivatives to full tensors

The data used in this project is preprocessed to exclude all but the principal eigenvector of the diffusion tensor imaging. There is no reason other than simplicity for this decision,

and further work on the modelling could favorably include the additional information that is available. This would also make it easier to incorporate the uncertainty. This would require significant changes and considerations to how the derivatives are calculated.

6.3.5 Applying the method to a larger cohort

Another potential valuable effort could be to broaden the modelled data to several brains instead of one. This could yield much needed information about the robustness of the model, as it might not work equally well on all subjects. It might also give information about brain tracts in general, since modelling observations on more brains might give insights to this aspect.

7
Appendix

7.1 Project and Work-flow Evaluation

Throughout the semester, we have collaborated continuously on the thesis project. Weekly meetings have been held between us and our supervisors. Here we have had detailed discussions on how to proceed, and received feedback on drafted sections of the report. Internally between us, we have met at DIKU and worked together 2-3 times a week on average, with increasing frequency as the project developed. Throughout the semester, we have kept logs on tasks and questions we have had, and short recaps of our meetings. The majority of the project has been done together, but some delegations have been made on drafting the report. That is, we have each had delegated writing to do, which we have subsequently reviewed together and edited accordingly. We are in general very happy with our team work.

The project has in general been more theoretical and oriented toward the mathematical aspects of computer science than we are used to. This is reflected in the sections of this thesis, where roughly the first half is wholly theoretical foundations. The work process has accordingly been heavily concerned with understanding these foundations. The focus on the theory has made the project quite challenging, but also interesting and enlightening.

7.2 Code Structure

The code we have developed can be found in the accompanying zip file. Alternatively, you can send a mail to nordam@di.ku.dk and we can grant you access to our private GitHub repository, containing the code as well.

The code is divided in code for the temperature Gaussian process example, figures for the report and the tractography code. The tractography code is probably the most interesting to look at and we will explain the contents of that here. The tractography code is written in Matlab.

The tractography folder contains four main components:

`pfos`: A folder containing the GitHub repository for Schober et al.'s PFOS solver. The functions we use (`odeFilter.m` and `odeSmoother.m`) are located in `pfos/matlab/solver/filter`.

`mriDerivLinearIntFlip.m`: Our ordinary differential equation. This is the function that is fed to the PFOS functions. It does linear interpolation of the nearby data in the brain data and flips vectors accordingly, as described in 5.3. There are a couple of other outdated differential equations which we wrote as we were testing the nearest neighbour method.

`singleTract.m`: Runs the PFOS on a single specified starting point, producing a single tract through the brain.

`ROITracts.m`: Runs the PFOS on the three regions of interest (IFOF, CST and Fornix), producing many tracts.

Both `ROITracts.m` and `singleTract.m` use `mriDerivLinearIntFlip.m` as their derivative functions. There are many configuration options at the start of both of the files. The configuration allows the user to decide what they want to plot and how to plot it, i.e. by specifying threshold values or another derivative function.

$$
\begin{aligned}
\mathbf{m}_n^s &= \mathbf{m}_n + \mathbf{G}_n \left(\mathbf{m}_{n+1}^s - \mathbf{m}_{n+1}^- \right) \\
\mathbf{P}_n^s &= \mathbf{P}_n + \mathbf{G}_n \left(\mathbf{P}_{n+1}^s - \mathbf{P}_{n+1}^- \right) \mathbf{G}_n^\top \\
\mathbf{G}_n &= \mathbf{P}_n \mathbf{A}_n^\top \left(\mathbf{P}_{n+1}^- \right)^{-1} \\
\Pr\left(\mathbf{x}_n \mid \mathbf{y}_{[N]} \right) &= \mathcal{N}\left(\mathbf{m}_n^s, \mathbf{P}_n^s \right)
\end{aligned}
$$

References

[1] Richard von Mises and Hilda Geiringer (Auth.) *Mathematical Theory of Probability and Statistics*. Elsevier Inc, 1964. ISBN: 978-1-4832-3213-3.

[2] R. Tatusko and J. A. Mirabito. *Cooperation in climate research: An evalutaion of the activities conducted under the US-USSR agreement for environmental protection since 1974*. 1990.

[3] Mark Jenkinson et al. "Improved optimization for the robust and accurate linear registration and motion correction of brain images." eng. In: *NeuroImage* 17.2 (2002), pp. 825–841.

[4] Jesper L R Andersson, Stefan Skare, and John Ashburner. "How to correct susceptibility distortions in spin-echo echo-planar images: application to diffusion tensor imaging." eng. In: *NeuroImage* 20.2 (2003), pp. 870–888. DOI: 10.1016/S1053-8119(03)00336-7.

[5] C. Edward Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. MIT Press, 2006. ISBN: 978-0-262-18253-9.

[6] Peter J Basser and Carlo Pierpaoli. "Microstructural and physiological features of tissues elucidated by quantitative-diffusion-tensor MRI". In: *Journal of magnetic resonance* 213.2 (2011), pp. 560–570.

[7] O. Bulgyina and V. Razuvaev. *Daily temperature and precipitation data for 518 russian meteorological stations*. 2012.

[8] Bruce Fischl. "FreeSurfer". In: *Neuroimage* 62.2 (2012), pp. 774–781.

[9] Mark Jenkinson et al. "FSL". eng. In: *NeuroImage* 62.2 (2012), pp. 782–790. DOI: 10.1016/j.neuroimage.2011.09.015.

[10] Michael Baron. *Probability and statistics for computer scientists*. CRC Press, 2013.

[11] David C Van Essen et al. "The WU-Minn Human Connectome Project: An overview." eng. In: *NeuroImage* 80 (2013), pp. 62–79. DOI: 10.1016/j.neuroimage.2013.05.041.

[12] Matthew F Glasser et al. "The minimal preprocessing pipelines for the Human Connectome Project." eng. In: *NeuroImage* 80 (2013), pp. 105–124.

[13] Simo Särkka. *Bayesian Filtering and Smoothing*. Vol. 3. IMS Textbooks series. Cambridge University Press, 2013.

[14] S. N. Sotiropoulos et al. "Effects of image reconstruction on fiber orientation mapping from multichannel diffusion MRI: reducing the noise floor using SENSE." eng. In: *Magn Reson Med* 70.6 (2013), pp. 1682–1689. DOI: 10.1002/mrm.24623.

[15] Michael Schober et al. "Probabilistic shortest path tractography in DTI using Gaussian Process ODE solvers". In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2014, pp. 265–272.

[16] Jesper LR Andersson and Stamatios N Sotiropoulos. "Non-parametric representation and prediction of single-and multi-shell diffusion-weighted MRI data using Gaussian processes". In: *Neuroimage* 122 (2015), pp. 166–176.

[17] Søren Hauberg et al. “A random riemannian metric for probabilistic shortest-path tractography”. In: *International Conference on Medical Image Computing and Computer-Assisted Intervention*. Springer. 2015, pp. 597–604.

[18] Jesper LR Andersson and Stamatios N Sotiropoulos. “An integrated approach to correction for off-resonance effects and subject movement in diffusion MR imaging”. In: *Neuroimage* 125 (2016), pp. 1063–1078.

[19] Niklas Kasenburg et al. “Training shortest-path tractography: Automatic learning of spatial priors”. In: *NeuroImage* 130 (2016), pp. 63–76.

[20] Simon Rogers and Mark Girolami. *A first course in machine learning*. CRC Press, 2016.

[21] M. Schober, S. Särkkä, and P. Hennig. “A probabilistic model for the numerical solution of initial value problems”. In: *ArXiv e-prints* (Oct. 2016). arXiv: `1610.05261 [math.NA]`.

[22] Ariadne. *Covariance functions*. 2018. URL: `https://bit.ly/2HknE8M` (visited on 02/25/2018).