TUM-DI-LAB Report

K. Harsha, A. Grundner, K. Wang

Jul 19, 2018

Abstract

Recent research has suggested great benefits from applying machine learning tools for the verification of parameters in PDEs. Building upon this research, we implement and analyze the estimation of parameters in PDEs using Gaussian Processes. Knowing only the parameter-dependend (linear) relationship between noisy data, we can infer this parameter by placing a Gaussian Prior on the data and by optimizing a certain log-marginal likelihood function. Here we rely heavily on the fact, that a linear transformation of a Gaussian Process is again a Gaussian Process.

After introducing the concept of Gaussian Processes, we apply this methodology to the Heat Equation, a modified version of the Burgers' Equation and to the Wave Equation. By doing this, we show how the framework can be successfully used in one or more dimensions and to some extent for the estimation of multiple parameters and for those in non-linear transformations.

Contents

CHAPTER 0

Preface

A great challenge for researchers with many applications in the applied sciences is to use vast available data sets and blend them with differential equations. For instance these methods can be used for verification and validation processes, which need to be undertaken in order to design well-functioning simulation software. This is also of special interest to the Max Planck Institute. Concretely, we are interested in finding the relationship between two black-box functions u and f based on possibly noisy data $\{X_u, u(X_u)\}$ and $\{X_f, f(X_f)\}$. That is, in specifying the form of the transformation $\mathcal{L}_x^\phi u(x) = f(x)$. We presume to know the form of the transformation $\mathcal{L}_x^\phi$ up to a set of unknown parameters ϕ. This setup falls under the broad range of so-called *inverse problems*, which occur often in diverse scientific disciplines. An example application would be to take the classical problem of heat conduction in a medium with unknown conductivity properties. The distribution of temperature is then governed by the Heat Equation, which is a type of PDE. The thermal diffusivity coefficient would, however, be unknown and needs to be estimated. Using Gaussian Processes to create a framework shedding light on the optimal parameters of the transformation $\mathcal{L}_x^\phi$ has many advantages. They can be used as flexible priors, describing a distribution over functions and provide a powerful training procedure, coming from a probabilistic viewpoint. They can themselves be seen as a one-layer neural network with infinitely many hidden units. In contrast to meshless methods, the optimal (hyper-)parameters can be *learned* by minimizing the negative log-likelihood function and don't have to be guessed or tuned manually. Whereas the main difference to latent force models, which are one of the few existing frameworks for combining machine learning tools with differential equations, is, that there is no need to solve the differential equation either analytically or numerically.

This paper is based mostly on the research conducted by Raissi et al. *[13]*. In their paper, applications consist of a fractional equation, an integro-differential equation, a reaction-diffusion PDE and the Heat Equation. Whilst their code was written in Matlab, we implement their methodology in Python and by doing this, also making it more accessible.

Notes regarding the contents

In order to retain comparability, in each chapter we will work with a set of points X with elements in $[0,1]^n$ where n is the number of dimensions. We always use either 10 or 20 points with the corresponding function values (X, Y_u, Y_f) as data samples. In this range, the estimates are good and the computation cost low.

We will mostly stick with a noise parameter of $s = 10^{-7}$, due to a trade-off between having a well-conditioned matrix and accuracy: Our data samples are generated without any noise, so the lower the noise parameter, the more accurate our estimation should be. Setting $s = 0$ would increase the likelihood of having to work with an ill-conditioned or singular matrix, since two columns in the covariance matrix corresponding to two points being close to each other would have almost equal values. We want to avoid this, since, when calculating the negative log-likelihood, we have to calculate the inverse of the covariance matrix. When working with noisy data on the other hand, one could optimize over the parameter s as well.

Most code cells from our notebooks are missing in the report. This has been done to improve readability. For the complete notebooks, we refer to our GitHub repository *[5]*.

CHAPTER 1

About Gaussian Processes

1.1 Introduction to Gaussian processes

This chapter develops the theory behind Gaussian processes and Gaussian fields. In simple terms, stochastic processes are families of random variables parameterized by a scalar variable $t \in \mathbb{R}$ denoting time. A random field, on the other hand, is a stochastic process parameterized by $x \in \mathbb{R}^d$ for $d > 1$ often denoting space. We begin with a review of univariate and multivariate normal distributions and use them to understand the properties of Gaussian processes.

1.1.1 Gaussian distributions

The univariate Gaussian (normal) distribution has a probability density given by

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

where $x, \mu, \sigma \in \mathbb{R}$. If a random variable X is Gaussian distributed with mean μ and variance σ, we commonly write $X \sim \mathcal{N}(\mu, \sigma)$.

If $Y \sim \mathcal{N}(\mu, \sigma)$ and $\alpha \in \mathbb{R}$, then

$$\alpha Y \sim \mathcal{N}(\alpha\mu, \alpha^2\sigma).$$

A p-dimensional multivariate Gaussian (or normal) distribution has a joint probability density given by

$$p(\mathbf{X}|\mu, \Sigma) = (2\pi)^{-p/2}|\Sigma|^{-1/2} \exp\left(-\frac{1}{2}(\mathbf{X}-\mu)^T\Sigma^{-1}(\mathbf{X}-\mu)\right),$$

where $\mu \in \mathbb{R}^p$ is the mean vector and $\Sigma \in GL(p, \mathbb{R})$ is the (symmetric and positive semi-definite) covariance matrix. This is commonly denoted as $\mathbf{X} \sim \mathcal{N}(\mu, \Sigma)$ or $\mathbf{X} \sim \mathcal{N}_p(\mu, \Sigma)$ to specify the dimension.

If $Y = BX + b$ where $B \in \mathbb{R}^{q\times p}, rank(B) = q$ and $b \in \mathbb{R}^q$, then

$$Y \sim \mathcal{N}(B\mu + b, B\Sigma B^T).$$

Let $\mathbf{X} \sim \mathcal{N}_p(\mu, \Sigma)$ and decompose $\mathbf{X}, \mu, \Sigma$ as

$$\mathbf{X} = \begin{pmatrix} X_1 \\ X_2 \end{pmatrix}, \mu = \begin{pmatrix} \mu_1 \\ \mu_2 \end{pmatrix} \text{ and } \Sigma = \begin{pmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{21} & \Sigma_{22} \end{pmatrix}$$

with $\Sigma_{11} \in \mathbb{R}^{n \times n}$ and $\Sigma_{22} \in \mathbb{R}^{m \times m}$ and the remaining variables respectively.

Then the conditional probability of X_2 is given by

$$X_2 | X_1 \sim N_{p_2}(\mu_2 + \Sigma_{21}\Sigma_{11}^{-1}\left(X_1 - \mu_1\right), \Sigma_{22} - \Sigma_{21}\Sigma_{11}^{-1}\Sigma_{12}).$$

1.1.2 Gaussian processes

The following definitions and theorems are to introduce the concept of Gaussian processes and fields. For detailed discussions and proofs, please refer to *[9]*.

Definition (Stochastic process):

Given a set $\mathcal{T} \subset \mathbb{R}$, a measurable space $(H, \mathcal{H})$, and a probability space $(\Omega, \mathcal{F}, \mathbb{P})$, an H-valued *stochastic process* is a set of H -valued random variables $\{X(t) : t \in \mathcal{T}\}$. We simply write $X(t)$ to denote the process. To emphasize the dependence on ω and that $X : \mathcal{T} \times \Omega \to \mathbb{R}$, we may write $X(t, \omega)$.

Definition (Second-order process):

A real-valued stochastic process is *second-order* if $X(t) \in L^2(\Omega)$ for each $t \in \mathcal{T}$. The mean function is defined by $\mu(t) := \mathbb{E}[X(t)]$ and the covariance function (also referred to as the *kernel*) is defined by $k(s, t) := Cov(X(s), X(t)))$ for all $s, t \in \mathcal{T}$.

Definition (Real-valued Gaussian process):

A real-valued second-order stochastic process $\{X(t) : t \in \mathcal{T}\}$ is *Gaussian* if $\mathbf{X} = [X(t_1), \ldots, X(t_M)]^T$ follows a multivariate Gaussian distribution for any $t_1, \ldots, t_M \in \mathcal{T}$ and any $M \in \mathbb{N}$.

Theorem:

Let $\mathcal{T} \subset \mathbb{R}$. The following statements are equivalent.

1. There exists a real-valued second-order stochastic process $X(t)$ with mean function $\mu(t)$ and kernel $k(s, t)$.
2. The function μ maps $\mathcal{T} \to \mathbb{R}$ and the function k maps $\mathcal{T} \times \mathcal{T} \to \mathbb{R}$. Furthermore k is symmetric and positive semi-definite.

Corollary:

The probability distribution $\mathbb{P}_X$ on $\left(\mathbb{R}^{\mathcal{T}}, \mathcal{B}\left(\mathbb{R}^{\mathcal{T}}\right)\right)$ of a real-valued Gaussian process $X(t)$ is uniquely determined by its mean $\mu : \mathcal{T} \to \mathbb{R}$ and kernel $k : \mathcal{T} \times \mathcal{T} \to \mathbb{R}$.

Definition (Random field):

For a set $D \subset \mathbb{R}^d$, a *(real-valued) random field* $\{u(x) : x \in D\}$ is a set of real-valued random variables on a probability space $(\Omega, \mathcal{F}, \mathbb{P})$. In the subsequent text, we drop $\omega \in \Omega$ and simply write $u(x)$, although it should be noted that $u : D \times \Omega \to \mathbb{R}$.

Definition (Second-order field):

For a set $D \subset \mathbb{R}^d$, a *random field* $\{u(x) : x \in D\}$ is *second-order* if $u(x) \in L^2(\Omega)\ \forall x \in D$. We say a second-order random field has mean function $\mu(x) \in L^2(\Omega)$ and kernel

$$k(\mathbf{x}, \mathbf{y}) = \mathrm{Cov}(u(\mathbf{x}), u(\mathbf{y})) := \mathbb{E}[(u(\mathbf{x}) - \mu(\mathbf{x}))(u(\mathbf{y}) - \mu(\mathbf{y}))], \quad \mathbf{x}, \mathbf{y} \in D$$

Definition (Gaussian random field):

A *Gaussian random field* $\{u(x) : x \in D\}$ is a second-order random field such that $u = [u(x_1), u(x_2), \ldots, u(x_M)]^T$ follows the multivariate Gaussian distribution for any $x_1, \ldots, x_M \in D$ and any $M \in \mathbb{N}$. We denote it here as $\mathbf{u} \sim \mathbf{GP}(\mu, k)$ where $\mu_i = \mu(x_i)$ and $k_{ij} = k(x_i, x_j)$.

An important thing to note is, that by sampling an element u from a Gaussian Process, we are thereby sampling a set of function values for the points in the domain D and can thus view u as a function itself.

Since we will deal with different dimensions throughout the text, we will use the term '(Gaussian) process' for both of these cases to improve readability.

1.1.3 Kernels

This subchapter is to give an overview over the most popular kernels for a Gaussian Process.

Squared Exponential Kernel

It is also called Radial Basis Function kernel (RBF kernel), or Gaussian kernel, which is as follows:

$$k_{\mathrm{SE}}(x, x') = \sigma^2 \exp\left(-\frac{\|x - x'\|_2^2}{2l^2}\right)$$

The *length-scale* l determines the width of the kernel; in other words, the larger l is, the smoother the function is. The *signal variance* σ^2 controls the variance of the sampled functions. All the standard kernels have this parameter in front as a scale factor.

It has become the default kernel for GPs and pyGPs, and we have also chosen this kernel for our project, which will be explained in the later section.

Rational Quadratic Kernel

$$k_{\mathrm{RQ}}(x, x') = \sigma^2 \left(1 + \frac{\|x - x'\|_2^2}{2\alpha\ell^2}\right)^{-\alpha}$$

This kernel is equivalent to adding together many RBF kernels with different length-scales, or can be seen as an infinite sum of RBF kernels. If $\alpha \to \infty$, then the RQ is identical to the RBF.

Periodic Kernel

$$k_{\mathrm{Per}}(x, x') = \sigma^2 \exp\left(-\frac{2\sin^2(\pi\|x - x'\|_2/p)}{\ell^2}\right)$$

It is obvious that the periodic kernel (derived by David Mackay) is designed for functions with repeating structures. Its parameters are easily interpretable:

The period p is the distance between repetitions of the function.

The length-scale l has the same interpretation as the length-scale in the RBF kernel.

Linear Kernel

$$k_{\mathrm{Lin}}(x, x') = \sigma^2 (x - c)^T (x' - c)$$

The linear kernel, unlike other kernels, is a non-stationary covariance function, which means that it does not solely depend on $x - x'$. Thus by fixing the hyperparameters and moving the data, the model will yield different predictions.

Our Choice

Since our project is mainly based on the Raissi's paper, so we also follow his choice of the kernel. The reason has been stated in his 2017 paper:

> In particular, the squared exponential covariance function chosen above implies smooth approximations. More complex function classes can be accommodated by appropriately choosing kernels. For example, non-stationary kernels employing nonlinear warpings of the input space can be constructed to capture discontinuous response.

We have used the pyGPs package to test the kernels written above and customized kernels (see our project on GitHub). It seems that the RBF kernels work for most functions at hand.

1.2 Simple example of a Gaussian process

The following example illustrates how we move from process to distribution and also shows that the Gaussian process defines a distribution over functions.

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

$m(x) = \frac{x^2}{4}$

$k(x, x') = exp(-\frac{1}{2}(x - x')^2)$

$y = f + \epsilon$

$\epsilon \sim \mathcal{N}(0, \sigma^2)$

```
In [1]: ## Importing necessary packages
        import numpy as np
        import matplotlib.pyplot as plt
```

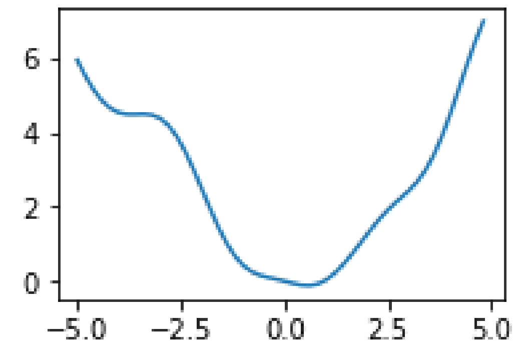
```
In [2]: ## Generating x-axis of 50 linearly spaced data points
        ## in the range between -5 and 5.
        x = np.arange(-5,5,0.2)
        n = x.size
        s = 1e-7        # Noise parameter for y: s = sigma^2
```

```
In [3]: ## Defining the mean function
        m = np.square(x) * 0.25
```

```
In [4]: ## Defining the covariance matrix k_y with respect to x
        a = np.repeat(x, n).reshape(n, n)
        k_y = np.exp(-0.5*np.square(a - a.transpose())) \
            + s*np.identity(n)
```

```
In [5]: ## We sample an n-dimensional vector of function values
        ## for y
        r = np.random.multivariate_normal(m, k_y, 1)
        y = np.reshape(r, n)
```

```
In [6]: ## The missing function values are filled in smoothly
        ## by the matplotlib-package
        plt.figure(figsize = (3,2))
        plt.plot(x,y)
        plt.show()
```



1.3 Linear operators on GPs

1.3.1 Regularity of Stochastic Processes

Definition (mean-square continuity):

Let $\{X(t) : t \in \mathcal{T}\}$ be a mean-zero process. The kernel k is continuous in (t,t) if and only if $\mathbb{E}\left[(X(t+h) - X(t))^2\right] \to 0$ as $h \to 0$. In particular, if $k \in \mathrm{C}(\mathcal{T} \times \mathcal{T})$, then $\{X(t) : t \in \mathcal{T}\}$ is *mean-square continuous*.

Definition (mean-square derivative):

A process $\{X(t) : t \in \mathcal{T}\}$ is *mean-square differentiable* with *mean-square derivative* $\frac{dX(t)}{dt}$ if, for all $t \in \mathcal{T}$, we have as $h \to 0$

$\|\frac{X(t+h)-X(t)}{h} - \frac{dX(t)}{dt}\|_{L^2(\Omega)} = \mathbb{E}\left[|\frac{X(t+h)-X(t)}{h} - \frac{dX(t)}{dt}|^2\right]^{1/2} \to 0.$

Theorem:

Let $\{X(t) : t \in \mathcal{T}\}$ be a stochastic process with mean zero. Suppose that the kernel $k \in \mathrm{C}^2(\mathcal{T} \times \mathcal{T})$. Then $X(t)$ is mean-square differentiable and the derivative $\frac{dX(t)}{dt}$ has kernel $\frac{\partial^2 k(s,t)}{\partial s \partial t}$.

Proof:

For any $s, t \in \mathcal{T}$ and real constants $h_s, h_t > 0$,

$$\begin{aligned}\mathrm{Cov}\left(\frac{X(s+h_s) - X(s)}{h_s}, \frac{X(t+h_t) - X(t)}{h_t}\right) &= \frac{1}{h_s h_t}\mathbb{E}[(X(s+h_s) - X(s))(X(t+h_t) - X(t))] \\ &= \frac{1}{h_s h_t}(k(s+h_s, t+h_t) - k(s+h_s, t) - k(s, t+h_t) + k(s,t))\end{aligned}$$

A simple calculation with the Taylor series shows that the right-hand side converges to $\frac{\partial^2 k(s,t)}{\partial s \partial t}$ as $h_s, h_t \to 0$.

With a similar approach and setting as in the previous theorem, we can calculate the covariance between a Gaussian process and its mean-square derivative.

$$\mathrm{Cov}\left(X(s), \frac{X(t+h) - X(t)}{h}\right) = \frac{1}{h}\mathbb{E}[(X(s))(X(t+h) - X(t))] = \frac{1}{h}(k(s, t+h) - k(s,t))$$

The right hand side converges to $\frac{\partial}{\partial t} k(s,t)$ as $h \to 0$.

Theorem (mean-square regularity):

Let $u(x)$ be a mean-zero second-order random field. If the kernel $k \in C(D \times D)$, then $u(x)$ is mean-square continuous so that $\|u(\mathbf{x}+\mathbf{h}) - u(\mathbf{x})\|_{L^2(\Omega)} \to 0$ as $h \to 0$ $\forall x \in D$. If $k \in C^2(D \times D)$, then $u(x)$ is mean-square differentiable. That is, a random field $\frac{\partial u(x)}{\partial x_i}$ exists such that

$$\|\frac{u\left(\mathbf{x}+he_i\right) - u(\mathbf{x})}{h} - \frac{\partial u(\mathbf{x})}{\partial x_i}\|_{L^2(\Omega)} \to 0 \quad \text{as } h \to 0$$

and $\frac{\partial u(x)}{\partial x_i}$ has the kernel $k_i(x,y) = \frac{\partial^2 C(x,y)}{\partial x_i \partial y_i}$.

Especially this theorem tells us, how zero-mean Gaussian Processes transform, when taking derivatives. Raissi describes in his paper, that the following even holds for general linear transformations *[13]*:

Let $u \sim GP(0, k_{uu})$ and $\mathcal{L}_x$ be a linear transformation. Then for $f = \mathcal{L}_x u$ it holds:

1. $f \sim GP(0, k_{ff})$
2. The covariance function of f is given by $k_{ff} = \mathcal{L}_x \mathcal{L}_{x'} k_{uu}$.
3. The covariance between $u(x)$ and $f(x')$ is given by $k_{uf} = \mathcal{L}_{x'} k_{uu}$, whereas the covariance between $f(x)$ and $u(x')$ is given by $k_{fu} = \mathcal{L}_x k_{uu}$.

CHAPTER 2

Parameter estimation with Gaussian Processes

2.1 1D Linear operator with one parameter

This chapter introduces a basic example of the framework developed in Chapter 3. We take a one-dimensional system with a single parameter and extract an operator out of it.

:raw-latex:‘\begin{align*} \mathcal{L}_x^\phi u(x) &= f(x) \\ \mathcal{L}_x^\phi &:= \phi \cdot + \frac{d}{dx}\cdot \end{align*}‘

It is trivial to verify linearity of the operator:

:raw-latex:‘\begin{align*} u, f : [0, 1] &\rightarrow \mathbb{K}, \alpha, \beta \in \mathbb{R} \\ \mathcal{L}_x^\phi (\alpha u + \beta f) &= \phi (\alpha u + \beta f) + \frac{d}{dx}(\alpha u + \beta f) \\ &= \alpha \phi u + \beta \phi f + \alpha \frac{d}{dx}u + \beta \frac{d}{dx}f \\ &= \alpha \mathcal{L}_x^\phi u + \beta \mathcal{L}_x^\phi f \end{align*}‘

One of the solutions to this system might be:

:raw-latex:‘\begin{align*} u(x) &= x^3 \\ f(x) &= \phi x^3 + 3x^2 \\ x &\in [0, 1] \end{align*}‘

We define Gaussian priors on the input and output:

:raw-latex:‘\begin{align*} u(x) &\sim \mathcal{GP}(0, k_{uu}(x,x’,\theta)) \\ f(x) &\sim \mathcal{GP}(0, k_{ff}(x,x’,\theta,\phi)) \end{align*}‘

A noisy data model for the above system can be defined as:

:raw-latex:‘\begin{align*} y_u &= u(X_u) + \epsilon_u; \epsilon_u \sim \mathcal{N}(0, \sigma_u^2I)\\ y_f &= f(X_f) + \epsilon_f; \epsilon_f \sim \mathcal{N}(0, \sigma_f^2I) \end{align*}‘

For the sake of simplicity, we ignore the noise terms ϵ_u and ϵ_f while simulating the data. They’re nevertheless beneficial, when computing the negative log marginal likelihood (NLML) so that the resulting covariance matrix is mostly more well-behaved for reasons as they were outlined after the preface.

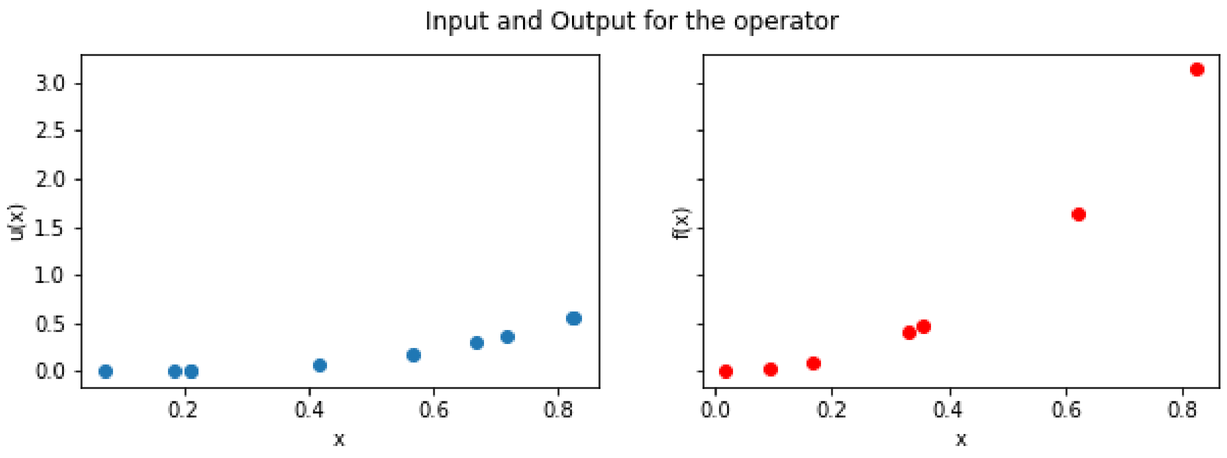
For the parameter estimation problem for the linear operator described above, we are given $\{X_u, y_u\}$, $\{X_f, y_f\}$ and we need to estimate ϕ.

2.1.1 Step 1: Simulate data

We use $\phi = 2$.

```
In [2]: def get_simulated_data(n1, n2, phi):
            x_u = np.random.rand(n1)
            y_u = np.power(x_u, 3)
            x_f = np.random.rand(n2)
            y_f = phi*np.power(x_f, 3) + 3*np.power(x_f,2)
            return(x_u, y_u, x_f, y_f)
```

```
In [4]: plt.show()
```



2.1.2 Step 2: Evaluate kernels

We use the RBF kernel defined as:

:raw-latex:‘\begin{align*} k_{uu}(x_i, x_j; \theta) = \theta exp(-\frac{1}{2l}(x_i-x_j)^2) \end{align*}‘

throughout the report. It is worth noting that this step uses information about $\mathcal{L}_x^\phi$ but not about $u(x)$ or $f(x)$. The derivatives are computed using *sympy*.

```
In [5]: x_i, x_j, theta, l, phi = sp.symbols('x_i x_j theta l phi')
        kuu_sym = theta*sp.exp(-l*((x_i - x_j)**2))
        kuu_fn = sp.lambdify((x_i, x_j, theta, l), kuu_sym, "numpy")
        def kuu(x, theta, l):
            k = np.zeros((x.size, x.size))
            for i in range(x.size):
                for j in range(x.size):
                    k[i,j] = kuu_fn(x[i], x[j], theta, l)
            return k
```

:raw-latex:‘\begin{align*} k_{ff}(x_i,x_j;\theta,\phi) &= \mathcal{L}_{x_i}^\phi \mathcal{L}_{x_j}^\phi k_{uu}(x_i, x_j; \theta) \\ &= \mathcal{L}_{x_i}^\phi \left(\phi k_{uu} + \frac{\partial}{\partial x_j}k_{uu} \right) \\ &= \phi^2 k_{uu} + \phi \frac{\partial}{\partial x_j}k_{uu} + \phi \frac{\partial}{\partial x_i}k_{uu} + \frac{\partial}{\partial x_i}\frac{\partial}{\partial x_j}k_{uu} \\ &= \theta exp(-\frac{1}{2l}(x_i-x_j)^2)\left[\phi^2 + 2\phi |x_i-x_j| + (x_i-x_j)^2 + 1 \right] \end{align*}‘

```
In [6]: kff_sym = phi**2*kuu_sym \
                    + phi*sp.diff(kuu_sym, x_j) \
                    + phi*sp.diff(kuu_sym, x_i) \
                    + sp.diff(kuu_sym, x_j, x_i)
        kff_fn = sp.lambdify((x_i, x_j, theta, l, phi), kff_sym, "numpy")
```

```
def kff(x, theta, l, phi):
    k = np.zeros((x.size, x.size))
    for i in range(x.size):
        for j in range(x.size):
            k[i,j] = kff_fn(x[i], x[j], theta, l, phi)
    return k
```

:raw-latex:‘\begin{align*} k_{fu}(x_i,x_j;\theta,\phi) &= \mathcal{L}_{x_i}^\phi k_{uu}(x_i, x_j; \theta) \\ &= \phi k_{uu} + \frac{\partial}{\partial x_i}k_{uu} \\ &= \theta exp(-\frac{1}{2l}(x_i-x_j)^2) \left[(\frac{1}{2})2|x_i-x_j| + \phi \right] \\ &= \theta exp(-\frac{1}{2l}(x_i-x_j)^2)(\phi + |x_i-x_j|) \end{align*}‘

```
In [7]: kfu_sym = phi*kuu_sym + sp.diff(kuu_sym, x_i)
        kfu_fn = sp.lambdify((x_i, x_j, theta, l, phi), kfu_sym, "numpy")
        def kfu(x1, x2, theta, l, phi):
            k = np.zeros((x2.size, x1.size))
            for i in range(x2.size):
                for j in range(x1.size):
                    k[i,j] = kfu_fn(x2[i], x1[j], theta, l, phi)
            return k
```

:raw-latex:‘\begin{align*} k_{uf}(x_i,x_j;\theta,\phi) &= \mathcal{L}_{x_j}^\phi k_{uu}(x_i, x_j; \theta) \\ &= \phi k_{uu} + \frac{\partial}{\partial x_j}k_{uu} \\ &= \theta exp(-\frac{1}{2l}(x_i-x_j)^2) \left[(\frac{1}{2})2|x_i-x_j| + \phi \right]\\ &= \theta exp(-\frac{1}{2l}(x_i-x_j)^2)(\phi+|x_i-x_j|) \end{align*}‘

```
In [8]: def kuf(x1, x2, theta, l, phi):
            return kfu(x1, x2, theta, l, phi).T
```

2.1.3 Step 3: Compute the negative log marginal likelihood(NLML)

The following covariance matrix is the result of our discussion at the end of Chapter 1.3.1, with an added noise parameter:

:raw-latex:‘\begin{align*} K = \begin{bmatrix} k_{uu}(X_u, X_u; \theta) + \sigma_u^2I & k_{uf}(X_u, X_f; \theta, \phi) \\ k_{fu}(X_f, X_u; \theta, \phi) & k_{ff}(X_f, X_f; \theta, \phi) + \sigma_f^2I \end{bmatrix} \end{align*}‘

For simplicity, assume $\sigma_u = \sigma_f$.

:raw-latex:‘\begin{align*} \mathcal{NLML} = \frac{1}{2} \left[log|K| + y^TK^{-1}y + Nlog(2\pi) \right] \end{align*}‘

where $y = \begin{bmatrix} y_u \\ y_f \end{bmatrix}$.

```
In [9]: def nlml(params, x1, x2, y1, y2, s):
            params = np.exp(params)
            K = np.block([
                [
                    kuu(x1, params[0], params[1]) + s*np.identity(x1.size),
                    kuf(x1, x2, params[0], params[1], params[2])
                ],
                [
                    kfu(x1, x2, params[0], params[1], params[2]),
                    kff(x2, params[0], params[1], params[2]) + s*np.identity(x2.size)
                ]
            ])
            y = np.concatenate((y1, y2))
            val = 0.5*(np.log(abs(np.linalg.det(K))) \
                       + np.mat(y) * np.linalg.inv(K) * np.mat(y).T)
            return val.item(0)
```

2.1.4 Step 4: Optimize hyperparameters

```
In [10]: nlml_wp = lambda params: nlml(params, x_u, x_f, y_u, y_f, 1e-6)
         m = minimize(nlml_wp, np.random.rand(3), method="Nelder-Mead")
```

```
In [12]: np.exp(m.x)
```

```
Out[12]: array([11.90390211,  0.47469623,  2.00120508])
```

The estimated value comes very close to the actual value.

For the current model, we get the following optimal values of the hyperparameters:

Parameter	Value
θ	11.90390211
l	0.47469623
ϕ	2.00120508

2.2 2D Linear operator with one parameter

Along the same lines as the previous example, we take a two-dimensional linear operator corresponding to a two-dimensional system with a single parameter. As previously explained, we set our problem up as follows:

$$\begin{gathered}
\mathcal{L}_{\bar{x}^\phi u(\bar{x})} = f(\bar{x}), \quad \bar{x} := (x_1, x_2)^T \\
\mathbb{X} := [0,1]^2; \quad \mathbb{K} \subset \mathbb{R}; \quad u, f \in C(\mathbb{X}, \mathbb{K}) \\
\mathcal{L}_{\bar{x}^\phi} : C(\mathbb{X}, \mathbb{K}) \to C(\mathbb{X}, \mathbb{K}) \\
\mathcal{L}_{x}^{\phi} := \phi \cdot + \frac{d}{dx_1} \cdot + \frac{d^2}{dx_2^2} \cdot
\end{gathered}$$

It is easy to verify that $\mathcal{L}_{\bar{x}^\phi}$ is linear and continuous. A suitable solution for the operator is

:raw-latex:‘\begin{align*} u(\bar{x}) &= x_1 x_2 - x_2^2 \\ f(\bar{x}) &= \phi x_1 x_2 - \phi x_2^2 + x_2 - 2 \end{align*}‘

Now, we use the same Gaussian priors as before, except that they are now defined on a two-dimensional space.

:raw-latex:‘\begin{align*} u(\bar{x}) &\sim \mathcal{GP}(0, k_{uu}(\bar{x},\bar{x}’,\theta)) \\ f(\bar{x}) &\sim \mathcal{GP}(0, k_{ff}(\bar{x},\bar{x}’,\theta,\phi)) \\ y_u &= u(X) + \epsilon_u; \; \epsilon_u \sim \mathcal{N}(0, \sigma_u^2I) \\ y_f &= f(X) + \epsilon_f; \; \epsilon_f \sim \mathcal{N}(0, \sigma_f^2I) \end{align*}‘

The parameter estimation problem for the linear operator described above is to estimate ϕ, given $\{x, y_u, y_f\}$. Notice that we evaluate u, f on the same set of points because it makes sense from a physics point of view.

2.2.1 Simulate data

We use $\phi = 2$ and try to estimate it.

```
In [2]: def get_simulated_data(n, phi):
            x = np.random.rand(n,2)
            y_u = np.multiply(x[:,0], x[:,1]) - x[:,1]**2
            y_f = phi*y_u + x[:,1] - 2
            return (x, y_u, y_f)
```

2.2.2 Evaluate kernels

The implementation of the kernels is straightforward as done earlier. Consequently, the corresponding code has been omitted from the report.

1. $k_{uu}(\bar{x}_i, \bar{x}_j; \theta) = \theta e^{(-l_1(x_{i,1}-x_{j,1})^2 - l_2(x_{i,2}-x_{j,2})^2)}$
2. $k_{ff}(\bar{x}_i, \bar{x}_j; \theta, \phi) = \mathcal{L}^{\phi}_{\bar{x}_i}\mathcal{L}^{\phi}_{\bar{x}_j} k_{uu}(\bar{x}_i, \bar{x}_j; \theta)$
   $= \mathcal{L}^{\phi}_{\bar{x}_i}\left(\phi k_{uu} + \frac{\partial}{\partial x_{j,1}} k_{uu} + \frac{\partial^2}{\partial^2 x_{j,2}} k_{uu}\right)$
   $= \phi^2 k_{uu} + \phi \frac{\partial}{\partial x_{j,1}} k_{uu} + \phi \frac{\partial}{\partial x_{i,1}} k_{uu}$
   $+ \phi \frac{\partial^2}{\partial^2 x_{j,2}} k_{uu} + \frac{\partial}{\partial x_{i,1}} \frac{\partial}{\partial x_{j,1}} k_{uu} + \phi \frac{\partial^2}{\partial^2 x_{i,2}} k_{uu}$
   $+ \frac{\partial}{\partial x_{i,1}} \frac{\partial^2}{\partial^2 x_{j,2}} k_{uu} + \frac{\partial^2}{\partial^2 x_{i,2}} \frac{\partial}{\partial x_{j,1}} k_{uu} + \frac{\partial^2}{\partial^2 x_{i,2}} \frac{\partial^2}{\partial^2 x_{j,2}} k_{uu}$
3. $k_{fu}(\bar{x}_i, \bar{x}_j; \theta, \phi) = \mathcal{L}^{\phi}_{\bar{x}_i} k_{uu}(\bar{x}_i, \bar{x}_j; \theta)$
   $= \phi k_{uu} + \frac{\partial}{\partial x_{i,1}} k_{uu} + \frac{\partial^2}{\partial x_{i,2}^2} k_{uu}$

```
In [8]: (x, yu, yf) = get_simulated_data(10, 2)
        nlml((1, 1, 1, 0.69), x, yu, yf, 1e-6)

Out[8]: 10.124377463652147
```

2.2.3 Optimize hyperparameters

```
In [9]: nlml_wp = lambda params: nlml(params, x, yu, yf, 1e-7)
        m = minimize(nlml_wp, np.random.rand(4), method="Nelder-Mead")

In [11]: np.exp(m.x)

Out[11]: array([1.29150445e+06, 4.54133519e-05, 5.27567407e-04, 2.00497047e+00])
```

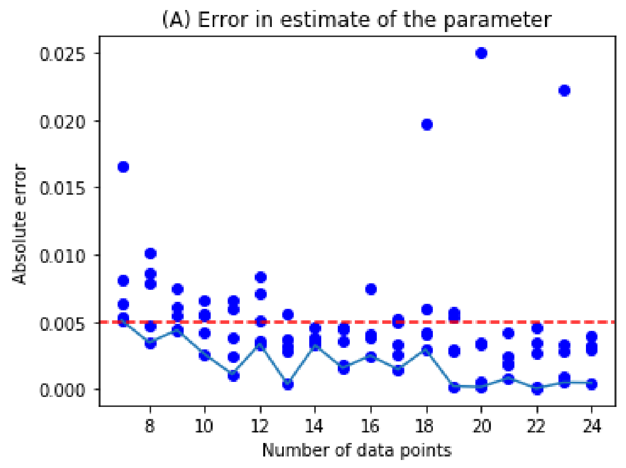
The optimized hyperparameters are:

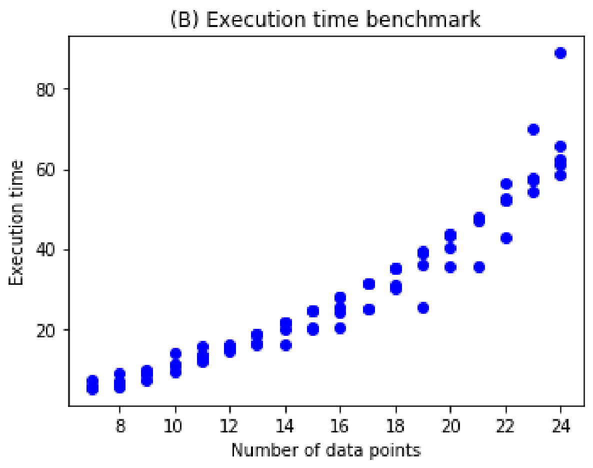
Parameter	Value
θ	1.29150445e+06
l_1	4.54133519e-05
l_2	5.27567407e-04
ϕ	2.00497047

2.2.4 Analysis

To investigate error properties for this model, we run simulations from 5 to 25 datapoints with 5 samples of random data in each case which is 100 samples. The absolute error in the parameter estimate, $|\phi_{est} - \phi_{true}|$ is plotted in the left figure below. The execution time is plotted to the right; it includes the time taken to simulate the data and the time taken by the optimization routine.

```
In [76]: plt.show()
```





It is common practice to select the best of a set of minimisation results with different starting points. In this case, assuming we take the minimum out of 5 iterations for a given sample(n data points), the absolute error is bounded by 0.005 for n > 10. The blue line connects the minimum absolute error for every n. We employ this trick in further notebooks.

The execution time shows a monotonically increasing trend when increasing the number of data points.

2.3 1D Linear operator with two parameters

In the previous examples we focus on systems with only one parameter. To see if the framework can be applied to a more general case, we use a 1D linear operator on two parameters, and verify the prediction.

Here we set up the linear operator as follows:

:raw-latex:‘\begin{align*} \mathcal{L}_x^\phi u(x) &= f(x) \\ \mathcal{L}_x^\phi &:= \phi_1 \cdot + \phi_2\frac{d}{dx}\cdot \end{align*}‘

A suitable solution can be:

:raw-latex:‘\begin{align*} u (x) & = \sin (x) \\ f (x) & = \phi _ { 1 } \sin (x) + \phi _ { 2 } \cos (x) \\ x & \in [0,1] \end{align*}‘

In this example, we assume $\phi_1 = 2$, $\phi_2 = 5$, and estimate ϕ_1 and ϕ_2.

2.3.1 Simulate data

```
In [2]: ##Initiating f(x) and u(x) with 20 data points
        x = np.random.rand(20)
        phi1 = 2.0
        phi2 = 5.0
        y_u = np.sin(x)
        y_f = phi1*np.sin(x) + phi2*np.cos(x)
```

2.3.2 Evaluate kernels

Corresponding kernels are defined as following:

$k_{uu}(x_i, x_j; \theta) = \theta \exp\left(-\frac{1}{2l}(x_i - x_j)^2\right)$

$$\begin{aligned}&k_{ff}\left(x_i, x_j; \theta, \phi_1, \phi_2\right)\\ &= \mathcal{L}_{x_i}^\phi \mathcal{L}_{x_j}^\phi k_{uu}\left(x_i, x_j; \theta\right)\\ &= \mathcal{L}_{x_i}^\phi \left(\phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_j} k_{uu}\right)\\ &= \phi_1^2 k_{uu} + \phi_1\phi_2 \frac{\partial}{\partial x_j} k_{uu} + \phi_1\phi_2 \frac{\partial}{\partial x_i} k_{uu} + \phi_2^2 \frac{\partial}{\partial x_i}\frac{\partial}{\partial x_i} k_{uu}\end{aligned}$$

$$\begin{aligned}&k_{fu}(x_i, x_j; \theta, \phi_1, \phi_2)\\ &= \mathcal{L}_{x_i}^\phi k_{uu}(x_i, x_j; \theta)\\ &= \phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_i} k_{uu}\end{aligned}$$

$$\begin{aligned}&k_{uf}\left(x_i, x_j; \theta, \phi_1, \phi_2\right)\\ &= \mathcal{L}_{x_j}^\phi k_{uu}\left(x_i, x_j; \theta\right)\end{aligned}$$

2.3.3 Optimize hyperparameters

```
In [14]: phi ## Estimated phi1 and phi2 using noiseless data points

Out[14]: [1.9999955379492513, 5.000001322351089]
```

Parameter	Value
ϕ_1	1.9999
ϕ_2	5.0000

We see that the error rate is less than 0.01% for the hyperparameter estimation. This example shows, that one can use our framework for multiple parameter estimation.

2.4 1D linear operator with a zero parameter

We have shown that parameter estimation works well with positive parameters. In some complex problems, one might include extra terms in the predicted function, whose parameter equals to 0 in the end. To check if the framework works in such case, we construct the following example:

:raw-latex:‘\begin{align*} \mathcal{L}_x^\phi &:= \phi_1 \cdot + \phi_2\frac{d}{dx}\cdot + \phi_3\frac{d^2}{dx^2}\cdot \\ u(x) &= sin(x) \\ f(x) &= \mathcal{L}_x^\phi u(x)\\ &=\phi_1 sin(x) + \phi_2 cos(x) - \phi_3 sin(x) \\ &=(\phi_1- \phi_3)sin(x) + \phi_2 cos(x) \\ x &\in [0, 1] \\ \end{align*}‘

2.4.1 Simulate data

We assume $\phi_1 = 2$ and $\phi_3 = 5$, and $cos(x)$ is the extra term with $\phi_2 = 0$, then the function f is given by:

:raw-latex:‘\begin{align*} f(x) &= -3sin(x) \end{align*}‘

2.4.2 Evaluate kernels

Corresponding kernels are defined as follows:

$$k_{uu}\left(x_i, x_j; \theta\right) = \theta \exp\left(-\frac{1}{2l}\left(x_i - x_j\right)^2\right)$$

$$\begin{aligned}&k_{ff}\left(x_i, x_j; \theta, \phi_1, \phi_2, \phi_3\right)\\ &= \mathcal{L}_{x_i}^\phi \mathcal{L}_{x_j}^\phi k_{uu}\left(x_i, x_j; \theta\right)\\ &= \mathcal{L}_{x_i}^\phi \left(\phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_j} k_{uu} + \phi_3 \frac{\partial^2}{\partial x_j^2} k_{uu}\right)\\ &= \left(\phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_i} k_{uu} + \phi_3 \frac{\partial^2}{\partial x_i^2} k_{uu}\right)\left(\phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_j} k_{uu} + \phi_3 \frac{\partial^2}{\partial x_j^2} k_{uu}\right)\end{aligned}$$

$$k_{fu}\left(x_i, x_j; \theta, \phi_1, \phi_2, \phi_3\right)$$
$$= \mathcal{L}_{x_i}^{\phi} k_{uu}\left(x_i, x_j; \theta\right)$$
$$= \phi_1 k_{uu} + \phi_2 \frac{\partial}{\partial x_i} k_{uu} + \phi_3 \frac{\partial^2}{\partial x_i^2} k_{uu}$$

$$k_{uf}\left(x_i, x_j; \theta, \phi_1, \phi_2, \phi_3\right)$$
$$= \mathcal{L}_{x_i}^{\phi} k_{uu}\left(x_i, x_j; \theta\right)$$

The kernels apply to the general 1D system with three parameters.

2.4.3 Optimize hyperparameters

```
In [16]: phi # [phi1 - phi3, phi2]
```

```
Out[16]: [-3.0110910877279604, 0.006341664007503534]
```

Parameter	Value
$\phi_1 - \phi_3$	-3.0001
ϕ_2	0.18e-05

The parameter estimation is very close to our prediction. The linear operator can be applied to most 1D linear PDE problems, which is quite powerful. While dealing with some specific problems, we can add more terms in the form of the transformation, then the parameter estimation with Gaussian Processes determines which of these terms are redundant.

CHAPTER 3

Linear PDEs

3.1 Heat equation

The heat equation is a parabolic partial differential equation that describes the distribution of heat (or variation in temperature) in a given region over time. The general form of the equation in any coordinate system is given by:

:raw-latex:‘\begin{align*} \frac{\partial u}{\partial t} - \phi \nabla^2 u = f \end{align*}‘

We will work here with the heat equation in one spatial dimension. This can be formulated as:

:raw-latex:‘\begin{align*} \mathcal{L}_{\bar{x}}^{\phi}u(\bar{x}) = \frac{\partial}{\partial t}u(\bar{x}) - \phi \frac{\partial^2}{\partial x^2}u(\bar{x}) = f(\bar{x}), \end{align*}‘ where $\bar{x} = (t, x) \in \mathbb{R}^2$.

The fundamental solution to the heat equation gives us:

:raw-latex:‘\begin{align*} u(x,t) &= e^{-t}sin(2\pi x) \\ f(x,t) &= e^{-t}(4\pi^2 - 1)sin(2\pi x) \end{align*}‘

3.1.1 Simulate data

```
In [2]: np.random.seed(int(time.time()))
        def get_simulated_data(n):
            t = np.random.rand(n)
            x = np.random.rand(n)
            y_u = np.multiply(np.exp(-t), np.sin(2*np.pi*x))
            y_f = (4*np.pi**2 - 1) * np.multiply(np.exp(-t), np.sin(2*np.pi*x))
            return (t, x, y_u, y_f)

        (t, x, y_u, y_f) = get_simulated_data(10)
```

3.1.2 Evaluate kernels

We use a reduced version of the kernel here.

1. $k_{uu}(x_i, x_j, t_i, t_j; \theta) = e^{\left[-\theta_1 (x_i - x_j)^2 - \theta_2 (t_i - t_j)^2\right]}$

2. $k_{ff}(\bar{x}_i, \bar{x}_j; \theta, \phi) = \mathcal{L}^{\phi}_{\bar{x}_i}\mathcal{L}^{\phi}_{\bar{x}_j} k_{uu}(\bar{x}_i, \bar{x}_j; \theta) = \mathcal{L}^{\phi}_{\bar{x}_i}\left[\frac{\partial}{\partial t_j}k_{uu} - \phi\frac{\partial^2}{\partial x_j^2}k_{uu}\right]$
   $= \frac{\partial}{\partial t_i}\frac{\partial}{\partial t_j}k_{uu} - \phi\left[\frac{\partial}{\partial t_i}\frac{\partial^2}{\partial x_j^2}k_{uu} + \frac{\partial^2}{\partial x_i^2}\frac{\partial}{\partial t_j}k_{uu}\right] + \phi^2\frac{\partial^2}{\partial x_i^2}\frac{\partial^2}{\partial x_j^2}k_{uu}$
3. $k_{fu}(\bar{x}_i, \bar{x}_j; \theta, \phi) = \mathcal{L}^{\phi}_{\bar{x}_i} k_{uu}(\bar{x}_i, \bar{x}_j; \theta) = \frac{\partial}{\partial t_i}k_{uu} - \phi\frac{\partial^2}{\partial x_i^2}k_{uu}$

3.1.3 Optimize hyperparameters

```
In [10]: %%timeit
         nlml_wp = lambda params: nlml(params, t, x, y_u, y_f, 1e-7)
         minimize(
             nlml_wp,
             np.random.rand(3),
             method="Nelder-Mead",
             options={'maxiter' : 5000, 'fatol' : 0.001})
```

```
2.13 s ± 267 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
In [11]: def minimize_restarts(t, x, y_u, y_f, n = 10):
             nlml_wp = lambda params: nlml(params, t, x, y_u, y_f, 1e-7)
             all_results = []
             for it in range(0,n):
                 all_results.append(
                     minimize(
                         nlml_wp,
                         np.random.rand(3),
                         method="Nelder-Mead",
                         options={'maxiter' : 5000, 'fatol' : 0.001}))
             filtered_results = [m for m in all_results if 0 == m.status]
             return min(filtered_results, key = lambda x: x.fun)
```

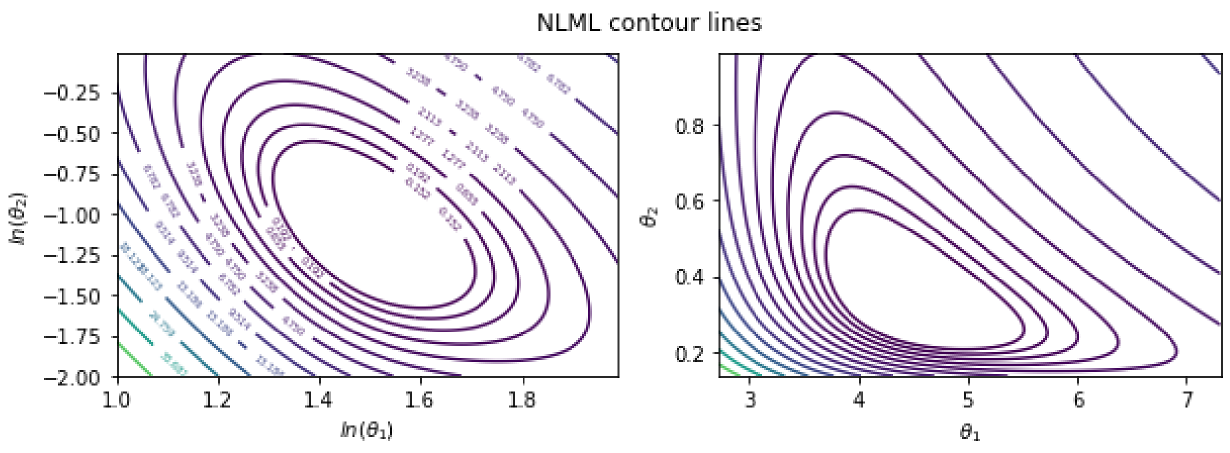
Estimated value of α

```
In [13]: np.exp(m.x[2])

Out[13]: 0.9701718328188159
```

3.1.4 Analysis

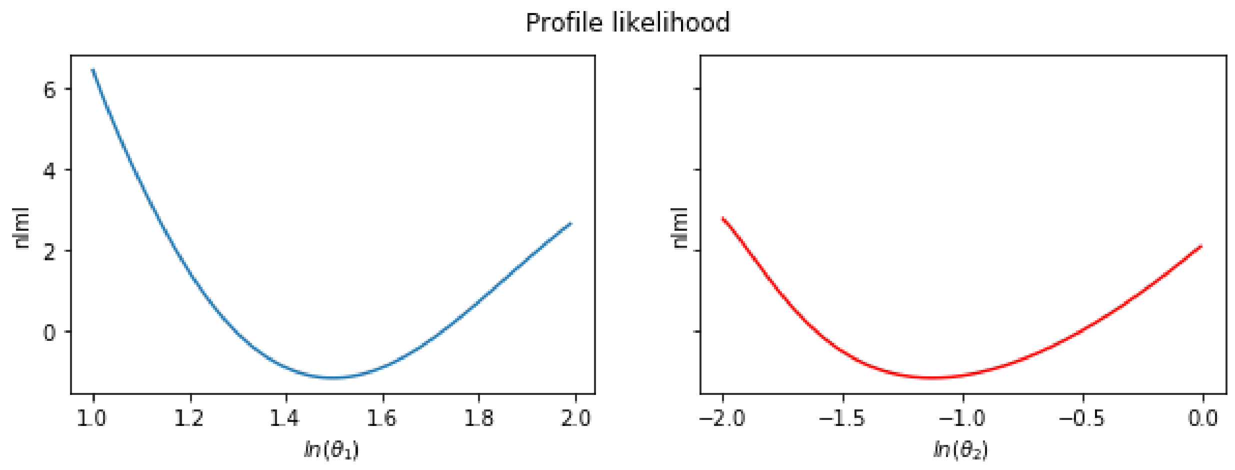
Contour lines for the likelihood

```
In [16]: plt.show()
```



Profile likelihood

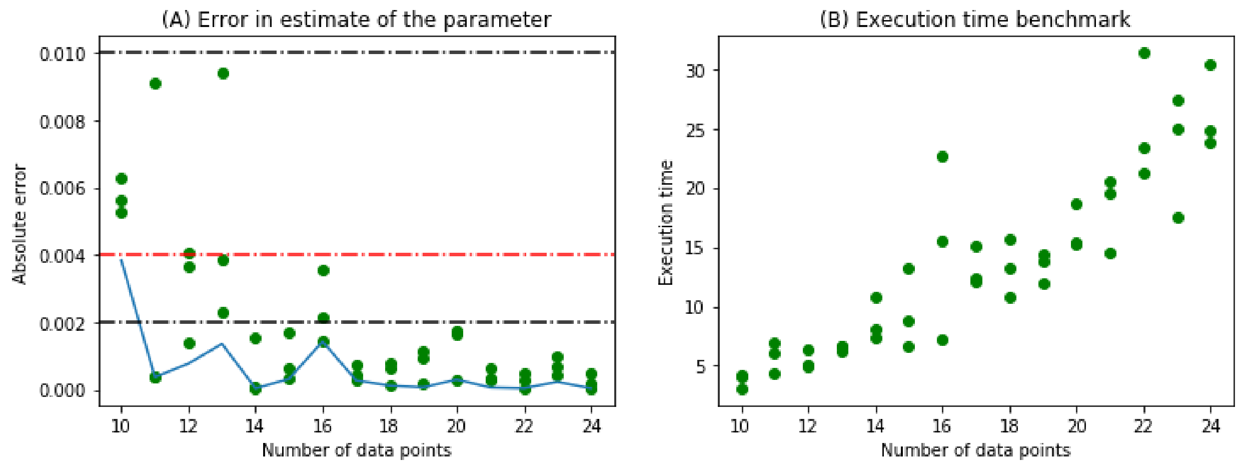
```
In [20]: plt.show()
```



Errors

We generate 5 set of samples for each number of data points(n) in the range (5,25) as earlier. The absolute error in the parameter estimate and the computation times are plotted in the following figure.

```
In [135]: plt.show()
```



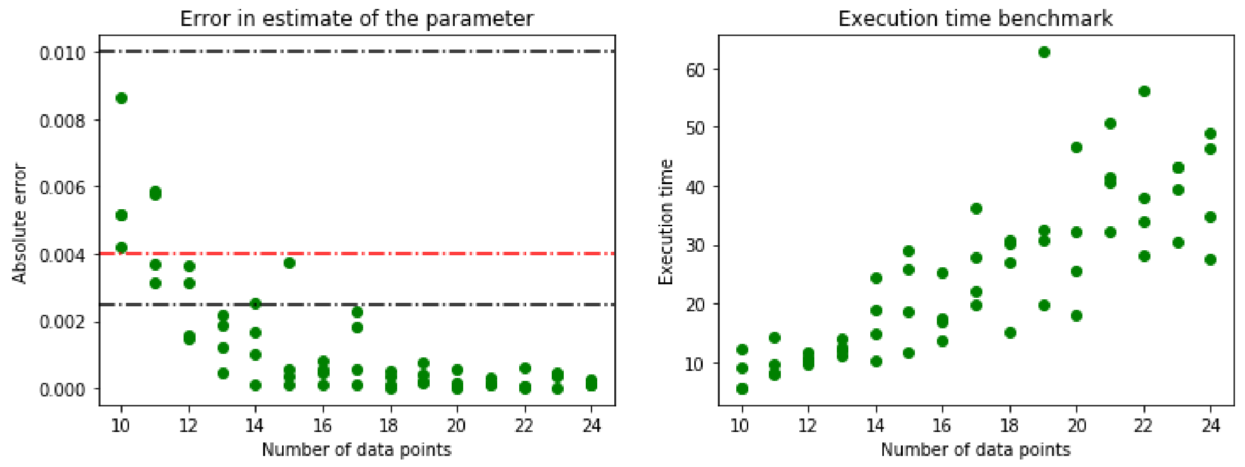
The minimum error for each value of n is bounded by 0.002 for n > 10. The computation time also increases monotonically with n.

With the full kernel

```
In [30]: nlml((1,1,1,0), t, x, y_u, y_f, 1e-6)

Out[30]: 1100767.8910308597

In [31]: %%timeit
         nlml_wp = lambda params: nlml(params, t, x, y_u, y_f, 1e-7)
         minimize(
             nlml_wp,
             np.random.rand(4),
             method="Nelder-Mead",
             options={'maxiter' : 5000, 'fatol' : 0.001})

6.93 s ± 1.94 s per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

The reduced kernel takes less time for optimization as compared to the full one.

```
In [137]: plt.show()
```



The same analysis as earlier in the chapter was done for the full kernel given by $\theta exp((\mathbf{x}-\mathbf{y})^T\Sigma(\mathbf{x}-\mathbf{y}))$ where $\Sigma = diag([l_1, l_2])$.

It can be noticed that the error in parameter estimate is slightly lower for the full kernel but the difference is not very significant. Meanwhile, the execution times are significantly different for both the cases. For 10 data points, the minimizer takes an average of 2.13 seconds for the reduced kernel while it is 6.93 seconds for the full kernel. This shows that not all hyperparameters might be necessary to get acceptable results. For some specific problems, having an intuition on the choice of kernels might be fruitful in the end.

3.2 Wave equation

Using our framework, we want to infer the constant c from the Wave Equation. The Wave Equation is given by

$$\frac{\partial^2 u}{\partial t^2} = c\nabla^2 u,$$

where $u = u(x_1, x_2, \dots, x_n, t)$ and $c > 0$ is some constant *[17]*. In one spatial dimension it boils down to:

$$\frac{\partial^2 u}{\partial t^2} - c\frac{\partial^2 u}{\partial x^2} = 0. \tag{3.1}$$

We generate the data from a solution of the equation (3.1) corresponding to $c = 1$ and get an estimation of $c = 1.0003$.

3.2.1 Problem Setup

$$u_{tt} - cu_{xx} = 0$$

The general solution is given by: $u(x,t) = F(x - ct) + G(x + ct)$ with F, G some functions.

Take $F(x) = x^2$ and $G(x) = \sin(x)$ and $c = 1$.

Thus: $u(x,t) = (x-t)^2 + \sin(x+t)$.

Set $f = 0$.

Consider u to be a Gaussian process:

$u \sim \mathcal{GP}(0, k_{uu}(x_i, x_j; \tilde{\theta}))$ with the hyperparameters $\tilde{\theta} = \{\theta, l_x, l_t\}$.

And the linear operator:

$\mathcal{L}_x^c = \frac{d^2}{dt^2} \cdot -c\frac{d^2}{dx^2} \cdot$

so that

$\mathcal{L}_x^c u = f$

Problem at hand: Estimate c (should be $c = 1$ in the end).

3.2.2 Step 1: Simulate data

$x \in [0,1]^n, \; t \in [0,1]^n$

```
In [3]: def get_simulated_data(n = 20):
            t = np.random.rand(n)
            x = np.random.rand(n)
            y_u = np.multiply(x-t, x-t) + np.sin(x+t)
            y_f = 0*x
            return(x, t, y_u, y_f)

        (x, t, y_u, y_f) = get_simulated_data()
```

3.2.3 Step 2: Evaluate kernels

1. $k_{uu}(y_i, y_j; \tilde{\theta}) = \theta exp(-\frac{1}{2l_x}(x_i - x_j)^2 - \frac{1}{2l_t}(t_i - t_j)^2)$, where $y_i = (x_i, t_i)$, $y_j = (x_j, t_j)$.
2. $k_{ff}(y_i, y_j; \tilde{\theta}, c) = \mathcal{L}_{y_i}^c \mathcal{L}_{y_j}^c k_{uu}(y_i, y_j; \tilde{\theta})$
   $= \frac{d^4}{dt_i^2 dt_j^2} k_{uu} - c\frac{d^4}{dt_i^2 dx_j^2} k_{uu} - c\frac{d^4}{dx_i^2 dt_j^2} k_{uu} + c^2 \frac{d^4}{dx_i^2 dx_j^2} k_{uu}$
3. $k_{fu}(y_i, y_j; \tilde{\theta}, c) = \mathcal{L}_{\tilde{x}_i}^c k_{uu}(y_i, y_j; \tilde{\theta}) = \frac{d^2}{dt_i^2} k_{uu} - c\frac{d^2}{dx_i^2} k_{uu}$
4. $k_{uf}(y_i, y_j; \tilde{\theta}, c)$ is given by the transpose of $k_{fu}(y_i, y_j; \tilde{\theta}, c)$.

3.2.4 Steps 3 and 4: Compute NLML and optimize the hyperparameters

```
In [10]: m = minimize_restarts(x, t, y_u, y_f, 5)
         np.exp(m.x[3]) # This is the optimized value for our parameter c

Out[10]: 0.9978007101765757
```

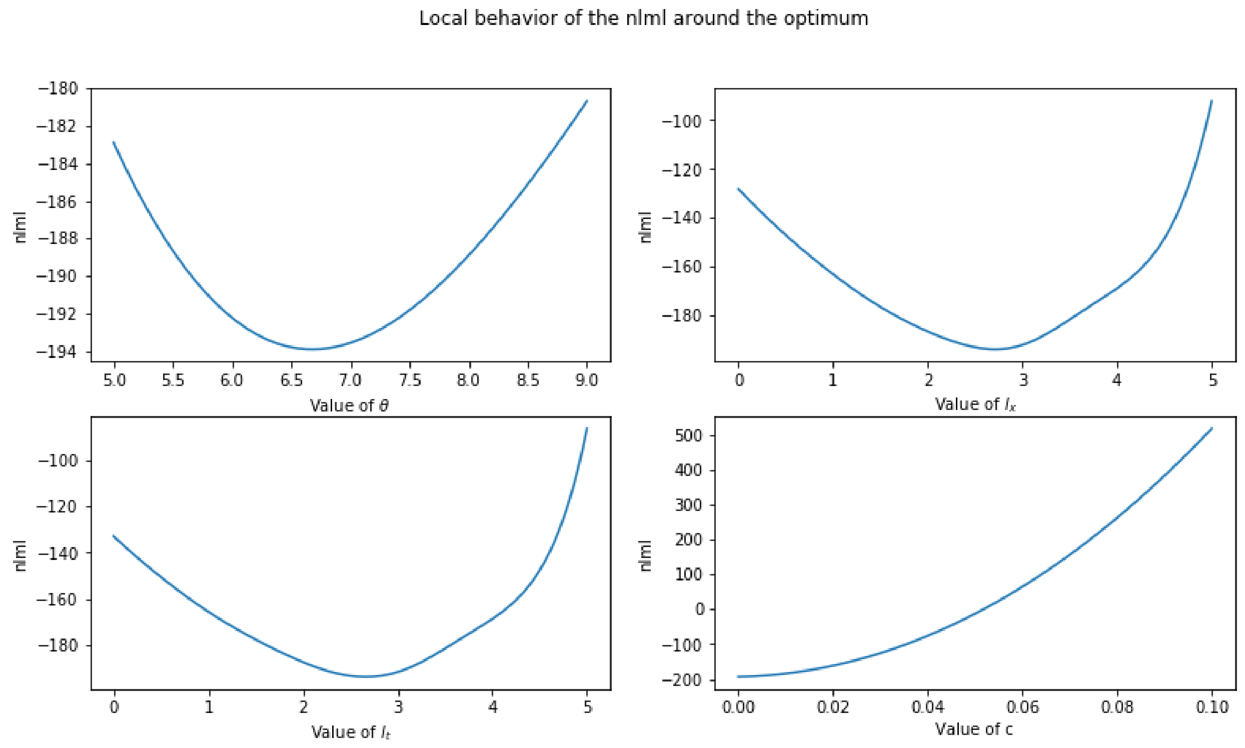
3.2.5 Step 5: Plotting the behavior for varied parameters

The logarithms of the optimal hyperparameters are given by (arranged in $[\theta, l_x, l_t, c]$):

```
In [11]: m.x

Out[11]: array([ 6.67851868e+00,  2.71167786e+00,  2.66415024e+00, -2.20171181e-03])
```

We want to plot the behavior of the nlml-function around the minimizer:

```
In [13]: show_1(lin0, lin1, lin3, res0, res1, res2, res3);
```



3.2.6 Step 6: Analysis of the error

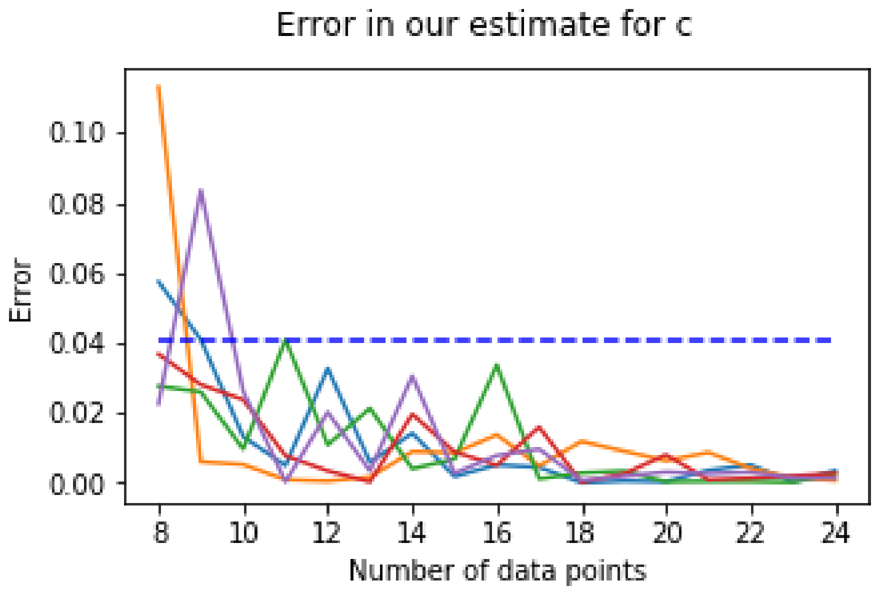
In this section we want to analyze the error of our algorithm using two different ways and plot its time complexity.

1. Plotting the error in our estimate for c:

The error is given by $|c_{estimate} - c_{true}|$.

We ran the algorithm five times and plotted the respective outcomes in different colors:

```
In [33]: show_3(lin, ones, res)
```



We see that for n sufficiently large (in this case $n \geq 10$), we can assume the error to be bounded by 0.041.

2. Plotting the error between the solution and the approximative solution:

Another approach of plotting the error is by calculating the difference between the approximative solution and the true solution.

That is: Let $\tilde{c}$ be the parameter, resulting from our algorithm. Set $\Omega := \{(x_i, t_i) \mid x_i \in x, t_i \in t\} \subseteq [0,1] \times [0,1]$.

Then we can calculate the solution of the PDE

$$\frac{d^2}{dt^2}\tilde{u}(x,t) - \tilde{c}\frac{d^2}{dx^2}\tilde{u}(x,t) = 0. \tag{3.2}$$

and set the error to $\|\tilde{u}(x,t) - u(x,t)\|_\Omega$. The norm can be chosen freely.

In our case, finding the solution to a given $\tilde{c}$ is not difficult. It is given by

$$\tilde{u}(x,t) = u(x, \sqrt{\tilde{c}}t) = (x - \sqrt{\tilde{c}}t)^2 + \sin(x + \sqrt{\tilde{c}}t) \tag{3.3}$$

We thus get:

$$\|\tilde{u}(x,t) - u(x,t)\|_\Omega = \|(x - \sqrt{\tilde{c}}t)^2 + \sin(x + \sqrt{\tilde{c}}t) - (x-t)^2 - \sin(x+t)\|_\Omega$$

With the L^2-norm, this is

$$\Big(\sum_{(x_i,t_i)\in\Omega} |(x_i - \sqrt{\tilde{c}}t_i)^2 + \sin(x_i + \sqrt{\tilde{c}}t_i) - (x_i - t_i)^2 - \sin(x_i + t_i)|^2\Big)^{1/2}$$

Short proof of (3.3):

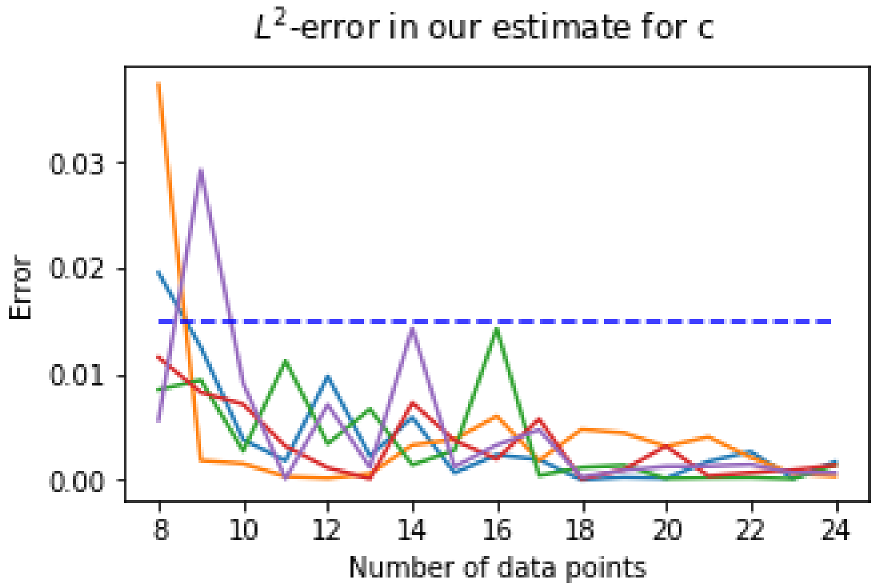
We assume $\tilde{c} \geq 0$ and want to find some $\alpha \in \mathbb{R}$ such that

$$\frac{d^2}{dt^2}\tilde{u}(x,\alpha t) - \tilde{c}\frac{d^2}{dx^2}\tilde{u}(x,\alpha t) = 0.$$

By setting $\alpha = \sqrt{\tilde{c}}$ we have:

$$\begin{aligned}\frac{d^2}{dt^2}\tilde{u}(x,\alpha t) - \tilde{c}\frac{d^2}{dx^2}\tilde{u}(x,\alpha t) &= \alpha^2[\frac{d^2}{dt^2}u(x,t)](x,\alpha t) - \tilde{c}[\frac{d^2}{dx^2}u(x,t)](x,\alpha t) \\ &= \tilde{c}\left(\frac{d^2}{dt^2}u(x,t) - \frac{d^2}{dx^2}u(x,t)\right)(x,\alpha t) \overset{(3.2)}{=} 0.\end{aligned}$$

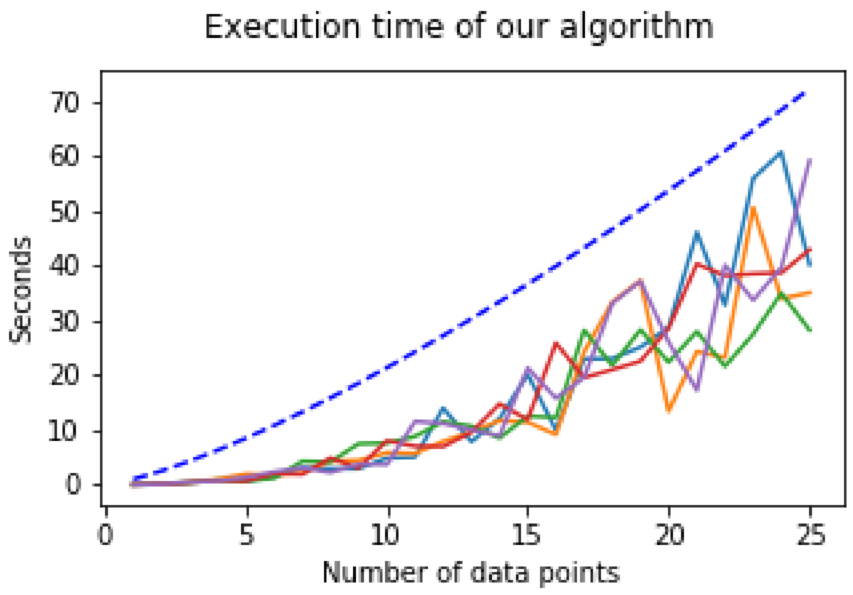
```
In [39]: show_4(lin, ones, res, diff)
```



The L^2-error is in our case bounded by 0.015 for $n \geq 10$.

3. Plotting the execution time:

```
In [42]: show_5(lin, timing)
```



Curiously, the time complexity seems to be around $\mathcal{O}(n^{4/3})$ (blue-dashed line).

Assuming an equal amount of function evaluations in the Nelder-Mead algorithm for different values of n, we would have been expecting a time complexity of $\mathcal{O}(n^3)$, due to the computation of the inverse of an $n \times n$-matrix in every evaluation of *nlml*. This could probably be seen with larger values of n.

CHAPTER 4

Non-linear PDEs

4.1 The inviscid Burgers' Equation

Burgers' Equation is an important non-linear PDE to applied mathematics. In its general form it is defined as follows:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial^2 u}{\partial x^2},$$

where $x \in \mathbb{R}$ and $t > 0$. Given an initial condition, i.e. when specifying $u(x, 0)$, there exists a closed form expression for the solution $u(x, t)$ *[16]*. Due to the complicated nature of the solution, we want to focus on the *inviscid Burgers' Equation*, where the diffusion term is equal to zero, i.e. $\nu = 0$, and the non-linearity is still present:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = 0$$

In order to deal with the non-linear term, we will have to replace u by some constant. For this, the mean of the function values of u is the most natural choice. Without this replacement, we would have a non-linear transformation of a Gaussian Process, which is not guaranteed to be a Gaussian Process.

An interesting and easy solution to the inviscid Burgers' Equation is given by

$$u(x, t) = \frac{x}{1+t}.$$

Introducing a parameter c, which we want to infer, we have the following:

4.1.1 Problem Setup

$$u_t + cuu_x = 0 \tag{4.1}$$

The equation (4.1) has $u(x, t) = \frac{x}{1+t}$ as a solution, if we assume $c = 1$.

Then $u_0(x) := u(x, 0) = x$

Using the backward Euler scheme, the equation can be re-written as:

$$\frac{u_n - u_{n-1}}{\tau} + cu_n \frac{d}{dx} u_n = 0$$

We set $u_n = \mu_{n-1}$, where μ_{n-1} is the mean of u_{n-1}, to deal with the non-linearity:

$$u_n + \tau c \mu_{n-1} \frac{d}{dx} u_n = u_{n-1}$$

Consider u_n to be a Gaussian process.

$u_n \sim \mathcal{GP}(0, k_{uu}(x_i, x_j; \theta, l))$

And the linear operator:

$\mathcal{L}_x^c = \cdot + \tau c \mu_{n-1} \frac{d}{dx} \cdot$

so that

$\mathcal{L}_x^c u_n = u_{n-1}$

Problem at hand: Estimate c. Using 20 data points, we will be able to estimate c to be 0.9994.

For the sake of simplicity, take $u := u_n$ and $f := u_{n-1}$.

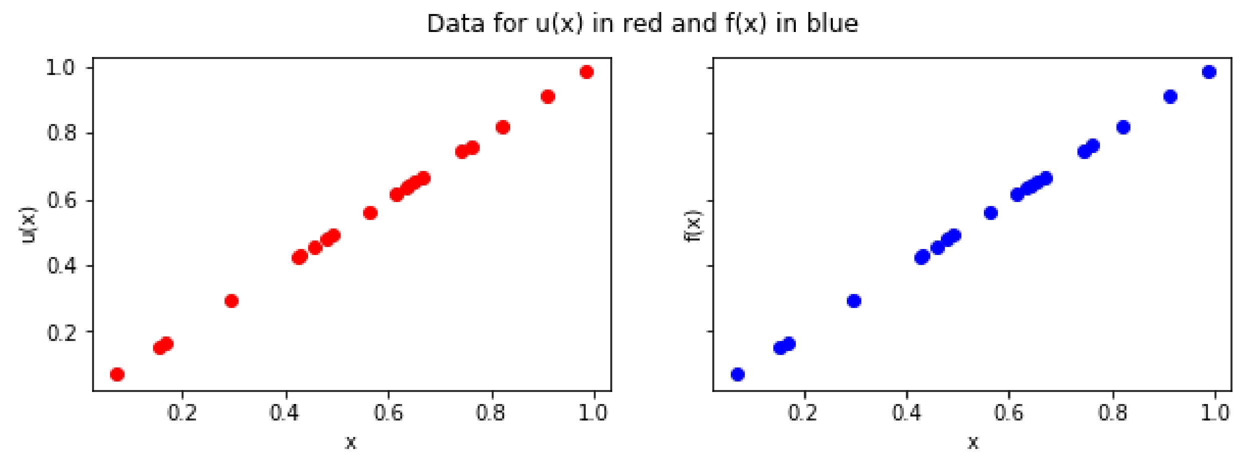
4.1.2 Step 1: Simulate data

Take data points at $t = 0$ for (u_{n-1}) and $t = \tau$ for (u_n), where τ is the time step.

$x \in [0, 1], \ t \in \{0, \tau\}$

```
In [3]: tau = 0.001
        def get_simulated_data(tau, n=20):
            x = np.random.rand(n)
            y_u = x/(1+tau)
            y_f = x
            return (x, y_u, y_f)

        (x, y_u, y_f) = get_simulated_data(tau)
```

```
In [6]: show_1(x,y_u,y_f)
```



4.1.3 Step 2: Evaluate kernels

1. $k_{uu}(x_i, x_j; \theta, l) = \theta exp(-\frac{1}{2l}(x_i - x_j)^2)$, where $\theta, l > 0$
2. $k_{ff}(x_i, x_j; \theta, l, c) = \mathcal{L}_{x_i}^c \mathcal{L}_{x_j}^c k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau c \mu_{n-1} \frac{d}{dx_i} k_{uu} + \tau c \mu_{n-1} \frac{d}{dx_j} k_{uu} + \tau^2 c^2 \mu_{n-1}^2 \frac{d^2}{dx_i x_j} k_{uu}$
3. $k_{fu}(x_i, x_j; \theta, l, c) = \mathcal{L}_{x_i}^c k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau \mu_{n-1} c \frac{d}{dx_i} k_{uu}$
4. $k_{uf}(x_i, x_j; \theta, l, c)$ is given by the transpose of $k_{fu}(x_i, x_j; \theta, l, c)$

4.1.4 Steps 3 and 4: Compute NLML and optimize the hyperparameters

```
In [ ]: m = minimize(nlml, np.random.rand(3), args=(x, y_u, y_f, 1e-7), method=\
                     "Nelder-Mead", options = {'maxiter' : 1000})
```

```
In [11]: m.x[2]  # This is our inferred value for c
```

```
Out[11]: 0.9994300669651587
```

4.1.5 Step 5: Analysis w.r.t. the number of data points (up to 25):

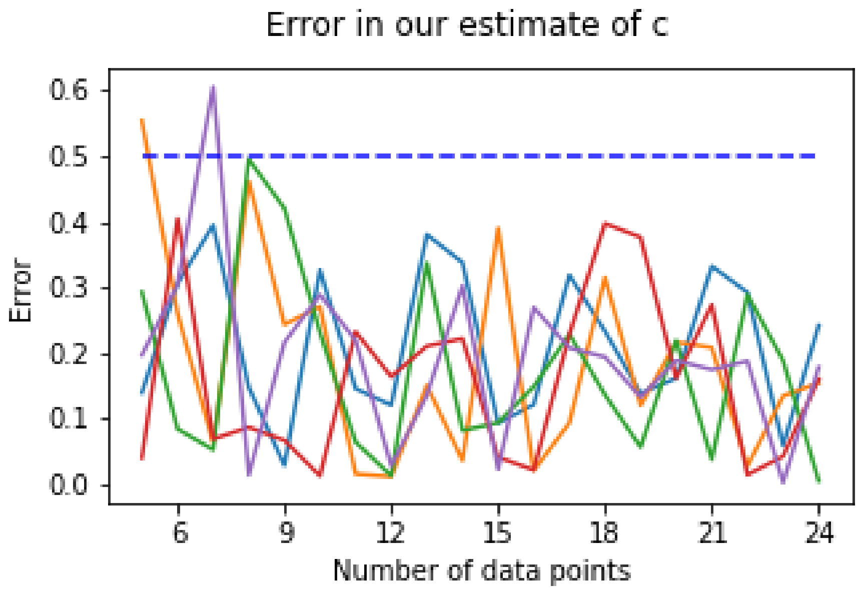
In this section we want to analyze the error of our algorithm using two different ways and plot its time complexity.

1. Plotting the error in our estimate for c:

The error is given by $|c_{estimate} - c_{true}|$.

We have altogether ran the algorithm five times, each time increasing the number of data points:

```
In [150]: show_2(lin, ones, res)
```



We see that for n sufficiently large (in this case $n \geq 5$), we can assume the error to be bounded by 0.5. It seems to be difficult to (even roughly) describe the limiting behavior of the error w.r.t. the number of data points.

2. Plotting the error between the solution and the approximative solution:

Another approach of plotting the error is by calculating the difference between the approximative solution and the true solution. That is: Let $\tilde{c}$ be the parameter, resulting from our algorithm. Set $\Omega := ([0,1] \times 0) \cup ([0,1] \times \tau)$. Then we can calculate the solution of the PDE

$$\frac{d}{dt}\tilde{u}(x,t) + \tilde{c}\tilde{u}(x,t)\frac{d}{dx}\tilde{u}(x,t) = 0. \tag{4.2}$$

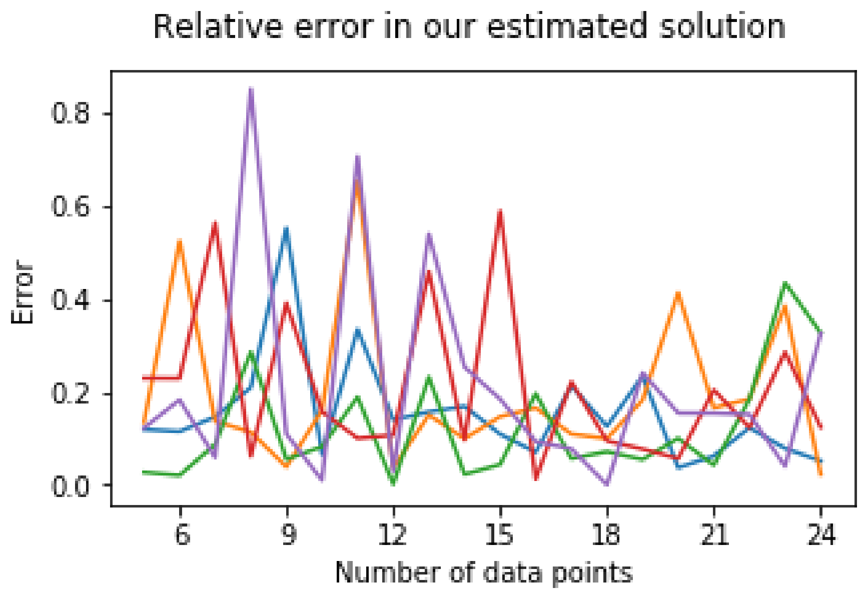
and set the error to $\|\tilde{u}(x,t) - u(x,t)\|_\Omega$. The norm can be chosen freely. In our case, finding the solution to a given $\tilde{c}$ is very simple. It is given by $\tilde{u}(x,t) = \frac{1}{\tilde{c}}u(x,t) = \frac{1}{\tilde{c}}\frac{x}{1+t}$. We thus get:

$$\|\tilde{u}(x,t) - u(x,t)\|_\Omega = \|\frac{1}{\tilde{c}}u(x,t) - u(x,t)\|_\Omega = |\frac{1}{\tilde{c}} - 1|\|u(x,t)\|_\Omega \propto |\frac{1}{\tilde{c}} - 1| \tag{4.3}$$

The plots of the errors look as follows:

```
In [155]: show_2(lin, ones, res)
```

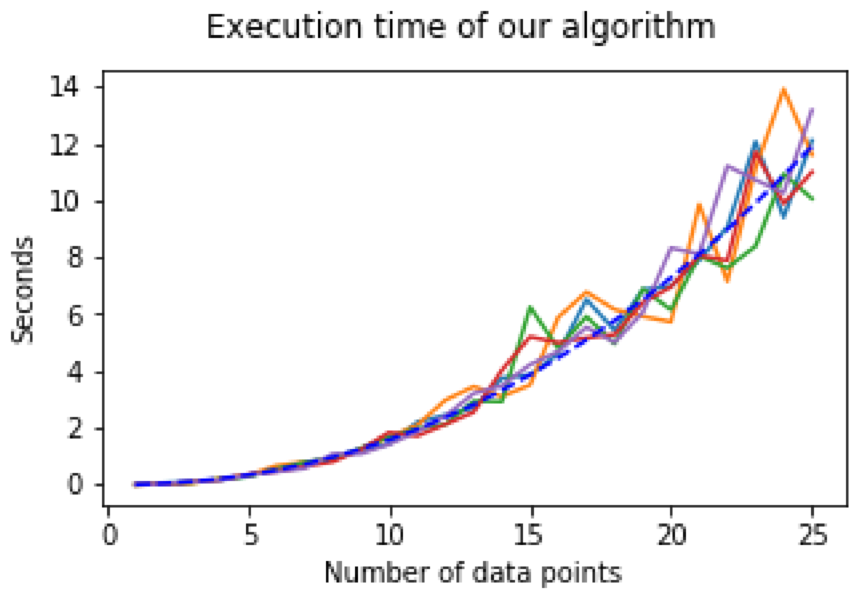
```
<Figure size 360x216 with 0 Axes>
```



3. Plotting the execution time:

All in one plot (the blue dashed line follows $f(x) = 0.01x^{2.2}$):

```
In [92]: show_3(lin, timing)
```



The time complexity seems to be around $\mathcal{O}(n^{2.2})$ (blue-dashed line).

4.2 The inviscid Burgers' Equation - A different approach

In this chapter it is our goal to infer, that a parameter is best set to zero. For this we take a slightly modified version of the Burgers' Equation and can still work with the solution $u(x,t) = \frac{x}{1+t}$ from the previous chapter:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = \nu\frac{\partial u}{\partial x}$$

For the solution to be valid, $\nu = 0$ must hold. That again would yield the inviscid Burgers' Equation. Note that on the right hand side of the equation, we are only taking the first derivative w.r.t. x -in contrast to Burgers' Equation-, since taking the second derivative $\frac{\partial^2 u}{\partial x^2} = 0$ would imply the validity of our solution independent of the value of ν. To show that taking different time schemes is possible, we use the forward Euler scheme in this example.

4.2.1 Problem Setup

$$u_t + uu_x = \nu u_x \tag{4.4}$$

Setting $u(x,t) = \frac{x}{1+t}$, we expect $\nu = 0$ as a parameter.

Then $u_0(x) := u(x,0) = x$.

Using the forward Euler scheme, the equation can be re-written as:

$$\frac{u_n - u_{n-1}}{\tau} + u_{n-1}\frac{d}{dx}u_{n-1} = \nu\frac{d^2}{dx^2}u_{n-1} \tag{4.5}$$

and setting the factor $u_{n-1} = \mu_{n-1}$ (analogously to the previous subchapter this is the mean of u_{n-1}) to deal with the non-linearity:

$$\tau\nu\frac{d^2}{dx^2}u_{n-1} - \tau\mu_{n-1}\frac{d}{dx}u_{n-1} + u_{n-1} = u_n$$

Consider u_{n-1} to be a Gaussian process.

$u_{n-1} \sim \mathcal{GP}(0, k_{uu}(x_i, x_j; \theta, l))$

And the linear operator:

$\mathcal{L}_x^\nu = \cdot + \tau\nu\frac{d}{dx}\cdot - \tau\mu_{n-1}\frac{d}{dx}\cdot$

so that

$\mathcal{L}_x^\nu u_{n-1} = u_n$

Problem at hand: Estimate ν (should be $\nu = 0$ in the end).

For the sake of simplicity, take $u := u_{n-1}$ and $f := u_n$.

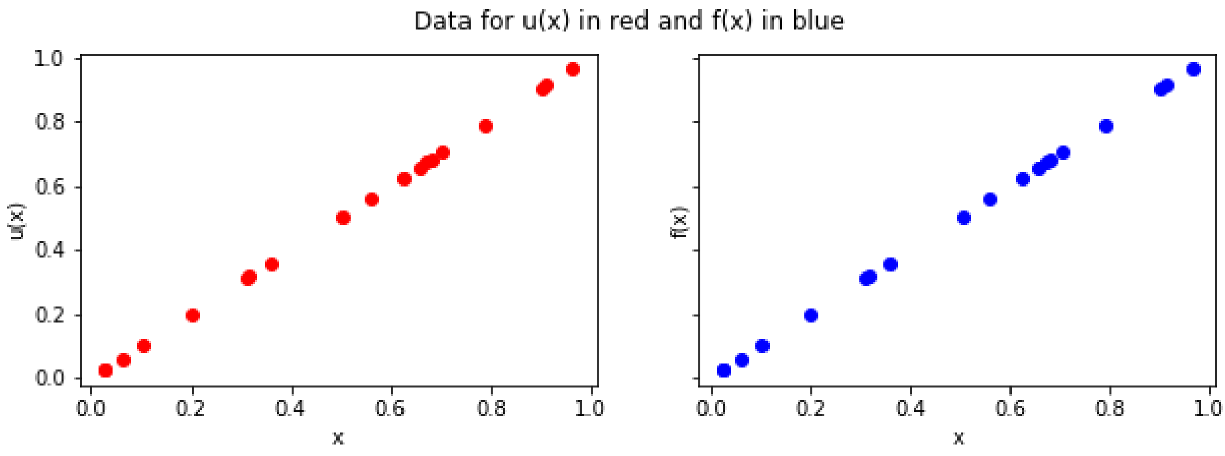
4.2.2 Step 1: Simulate data

Take data points at $t = 0$ for (u_{n-1}) and $t = \tau$ for (u_n), where τ is the time step.

$x \in [0, 1], \; t \in \{0, \tau\}$

```
In [10]: tau = 0.001
         def get_simulated_data(tau, n=20):
             x = np.random.rand(n)
             y_u = x
             y_f = x/(1+tau)
             return (x, y_u, y_f)

         (x, y_u, y_f) = get_simulated_data(tau)
```

```
In [81]: show_1(x, y_u, y_f)
```



4.2.3 Step 2: Evaluate kernels

1. $k_{uu}(x_i, x_j; \theta, l) = \theta exp(-\frac{1}{2l}(x_i - x_j)^2)$
2. $k_{ff}(x_i, x_j; \theta, l, \nu) = \mathcal{L}_{x_i}^{\nu} \mathcal{L}_{x_j}^{\nu} k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau\nu \frac{d}{dx_i} k_{uu} - \tau\mu_{n-1} \frac{d}{dx_i} k_{uu} + \tau\nu \frac{d}{dx_j} k_{uu} + \tau^2\nu^2 \frac{d}{dx_i} \frac{d}{dx_j} k_{uu} - \tau^2\nu\mu_{n-1} \frac{d^2}{dx_i dx_j} k_{uu} - \tau\mu_{n-1} \frac{d}{dx_j} k_{uu} - \tau^2\nu\mu_{n-1} \frac{d^2}{dx_i dx_j} k_{uu} + \tau^2\mu_{n-1}^2 \frac{d^2}{dx_i dx_j} k_{uu}$
3. $k_{fu}(x_i, x_j; \theta, l, \nu) = \mathcal{L}_{x_i}^{\nu} k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau\nu \frac{d}{dx_i} k_{uu} - \tau\mu_{n-1} \frac{d}{dx_i} k_{uu}$
4. $k_{uf}(x_i, x_j; \theta, l, \nu)$ is given by the transpose of $k_{fu}(x_i, x_j; \theta, l, \nu)$.

4.2.4 Steps 3 and 4: Compute NLML and optimize the hyperparameters

```
In [76]: m = minimize(nlml, np.random.rand(3), args=(x, y_u, y_f, 1e-7), method=\
                       "Nelder-Mead", options = {'maxiter' : 1000})
```

```
In [77]: m.x[2]    # This is our inferred value for \nu
```

```
Out[77]: 0.00021347647791778476
```

4.2.5 Step 5: Analysis w.r.t. the number of data points (up to 25):

In this section we want to analyze the error of our algorithm and plot its time complexity.

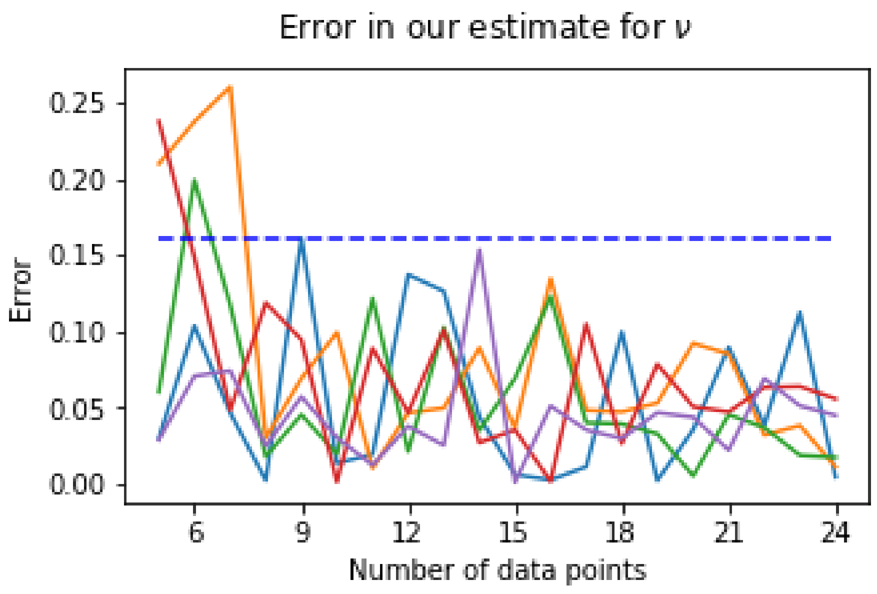
1. Plotting the error in our estimate:

The error is given by $|\nu_{estimate} - \nu_{true}|$.

We plot the error with respect to the number of data samples for five runs of the program:

```
In [100]: show_2(lin, res)
```

```
<Figure size 360x216 with 0 Axes>
```



We see that for n sufficiently large (in this case $n \geq 8$), we can assume the error to be bounded by 0.16.

2. Plotting the error between the solution and the approximative solution:

Another approach of plotting the error is by calculating the difference between the approximative solution and the true solution. That is: Let $\tilde{\nu}$ be the parameter, resulting from our algorithm. Set $\Omega := ([0,1] \times 0) \cup ([0,1] \times \tau)$. Then we can calculate the solution of the PDE

$$\frac{d}{dt}\tilde{u}(x,t) + \tilde{u}(x,t)\frac{d}{dx}\tilde{u}(x,t) = \tilde{\nu}\frac{d}{dx}\tilde{u}(x,t). \tag{4.6}$$

and set the error to $\|\tilde{u}(x,t) - u(x,t)\|_\Omega$. The solution is given by $\tilde{u}(x,t) = u(x,t) + \tilde{\nu} = \frac{x}{1+t} + \tilde{\nu}$. We thus get:

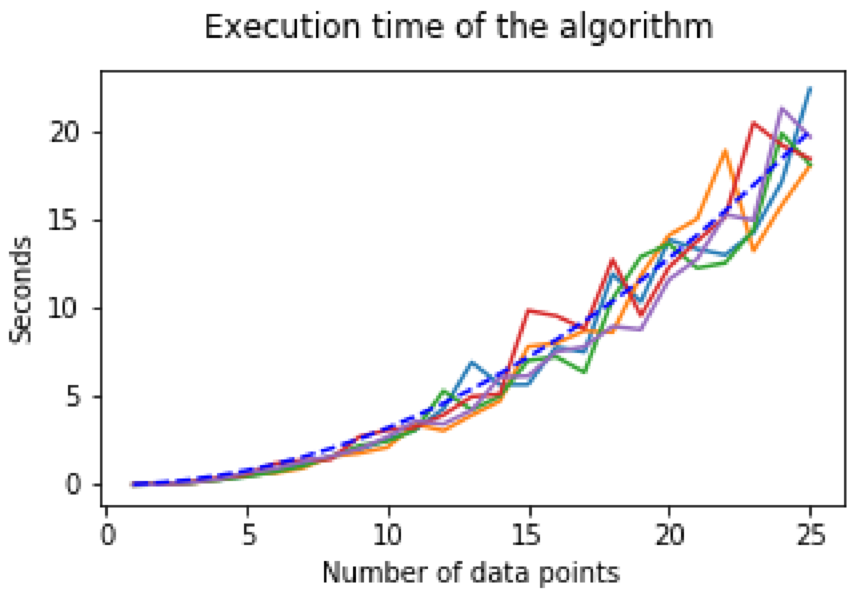
$$\|\tilde{u}(x,t) - u(x,t)\|_\Omega = \|u(x,t) + \tilde{\nu} - u(x,t)\|_\Omega \propto |\tilde{\nu}| \tag{4.7}$$

Here, the two error terms coincide.

3. Plotting the execution time:

The blue-dashed line follows $f(x) = 0.032x^2$.

```
In [59]: show_3(lin, timing)
```



We observe a time complexity of roughly $\mathcal{O}(n^2)$ (blue-dashed line).

4.2.6 Comparison with a no-mean version

Looking at the setup following (4.4), we might ask, how much better our result would be, when replacing the factor u_{n-1} by the function itself, instead of the mean. Clearly, this is only for comparative purposes and would be impossible in an application, since the function u_{n-1} is assumed to be unknown.
We will see, that the error will be smaller, in our case by a factor of 178 for $n \geq 8$.

Setting the factor $u_{n-1} = u_0(x) = x$ in (4.5), we get:

$\tau\nu\frac{d^2}{dx^2}u_{n-1} - \tau x\frac{d}{dx}u_{n-1} + u_{n-1} = u_n$

The linear operator looks just slightly different:

$\mathcal{L}_x^{\nu} = \cdot + \tau\nu\frac{d}{dx}\cdot - \tau x\frac{d}{dx}\cdot$

so that

$\mathcal{L}_x^{\nu}u_{n-1} = u_n.$

Our kernels evaluate to:

1. $k_{uu}(x_i, x_j; \theta, l) = \theta exp(-\frac{1}{2l}(x_i - x_j)^2)$
2. $k_{ff}(x_i, x_j; \theta, l, \nu) = \mathcal{L}_{x_i}^{\nu}\mathcal{L}_{x_j}^{\nu}k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau\nu\frac{d}{dx_i}k_{uu} - \tau x\frac{d}{dx_i}k_{uu} + \tau\nu\frac{d}{dx_j}k_{uu} + \tau^2\nu^2\frac{d}{dx_i}\frac{d}{dx_j}k_{uu} - \tau^2\nu x\frac{d^2}{dx_i dx_j}k_{uu} - \tau x\frac{d}{dx_j}k_{uu} - \tau^2\nu x\frac{d^2}{dx_i dx_j}k_{uu} + \tau^2 x^2\frac{d^2}{dx_i dx_j}k_{uu}$
3. $k_{fu}(x_i, x_j; \theta, l, \nu) = \mathcal{L}_{x_i}^{\nu}k_{uu}(x_i, x_j; \theta, l)$
   $= k_{uu} + \tau\nu\frac{d}{dx_i}k_{uu} - \tau x\frac{d}{dx_i}k_{uu}$
4. $k_{uf}(x_i, x_j; \theta, l, \nu)$ is given by the transpose of $k_{fu}(x_i, x_j; \theta, l, \nu)$.

After constructing the *nlml* we get:

```
In [17]: m = minimize(nlml, np.random.rand(3), args=(x, y_u, y_f, 1e-7), method=\
                         "Nelder-Mead", options = {'maxiter' : 1000})
         m.x[2] # This is our prediction for \nu
```

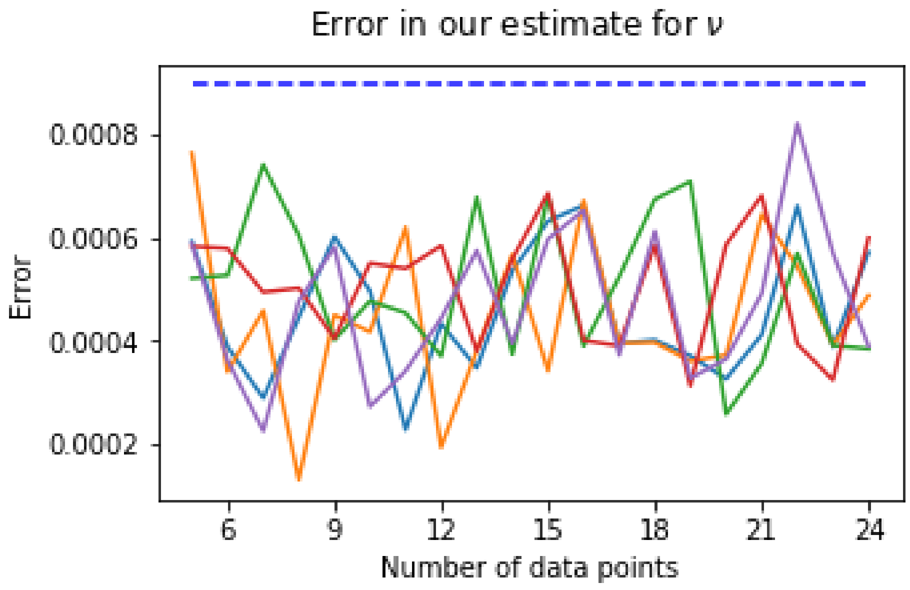
```
Out[17]: 0.00042724953581625225
```

We can analyze the error in multiple runs and look at the execution time:

1. Plotting the error in our estimate for ν

The error is given by $|\nu_{estimate} - \nu_{true}|$.

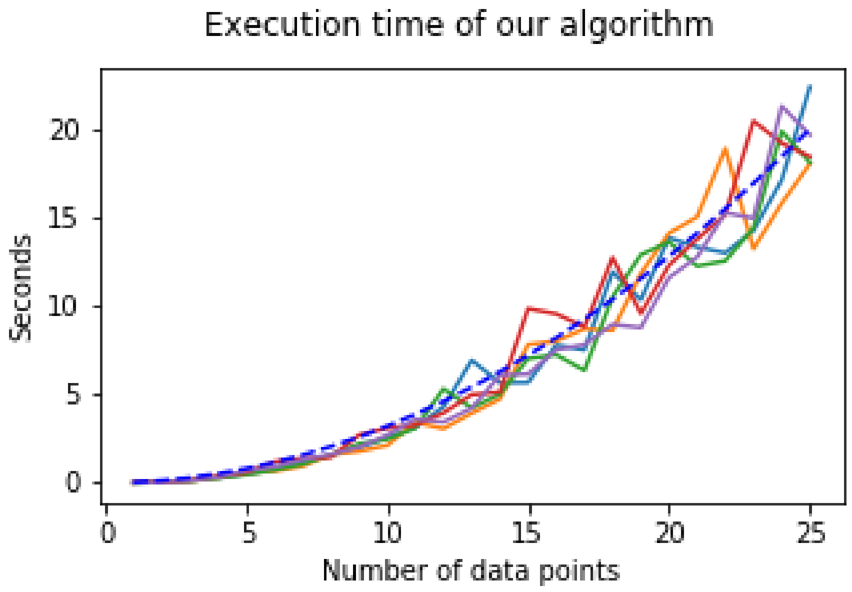
```
In [30]: show_4(lin, est, res)
```



We see that for n sufficiently large (in this case $n \geq 5$), we can assume the error to be bounded by 0.0009.

2. Plotting the execution time:

```
In [57]: show_5(lin, timing)
```



The time complexity seems to be as before roughly $\mathcal{O}(n^2)$.

CHAPTER 5

Approach with pyGPs

5.1 Parameter Estimation for a linear operator using pyGPs

When starting to work on this project, we were looking for a suitable Python package giving us the capabilities we needed concerning Gaussian Processes. We therefore implemented the early code using the pyGPs-package *[17]*. It is a package best suited for classical Gaussian Process Regression or Classification. After installing pyGPs and testing it, using the provided test data, there appeared an 'Intel MKL FATAL ERROR: Cannot load mkl_intel_thread.dll' - error, which we were not able to resolve. By switching from Windows to a Linux Cluster, this issue could be circumvented.

Our approach with pyGPs went as follows:

1. We assume Gaussian Priors with zero mean and an RBF Kernel for u and f, i.e.:

$$u(x) \sim \mathcal{GP}(0, k_{uu}(x, x'; \sigma_u, l_u))$$
$$f(x) \sim \mathcal{GP}(0, k_{ff}(x, x'; \sigma_f, l_f))$$

2. Given the data $\{X_u, Y_u\}$ and $\{X_f, Y_f\}$, we can now optimize the hyperparameters of the two Gaussian Processes separately using pyGPs.
3. Since $f = \mathcal{L}_x^\phi u(x)$, we know that the covariance matrix for f is given by $k_f = \mathcal{L}_{x'}^\phi \mathcal{L}_x^\phi k_{uu}$ (cf. Chapter 1.3.1). As an approximation, we set $k_f = k_{ff}$. As a consequence, also

   $$k_f(x_i, x_i) = k_{ff}(x_i, x_i)$$

   must hold for all data points x_i. Rearranging leads to some function

   $$\phi = g(\sigma_u, \sigma_f, l_u),$$

   which we can evaluate.

Now this approach worked for simple examples (e.g. in the Example implementation), though it failed for more complicated ones. As an attempt to resolve this problem, we wanted to avoid the approximation in step 3 and work with the correct covariance matrix instead. The pyGPs-package allows the user to define custom covariance functions. This was of no avail mainly due to incongruities in the pyGPs source code regarding the derivatives of covariance functions in combination with insufficient documentation in that regard and the resulting complexity of the task itself.

As an example, we have included a custom covariance function in the appendix. We finally ceased expanding upon this approach, since it is not clear, whether the independent optimization of the hyperparameters for the functions u and f would yield the result we are striving for.

5.1.1 Example implementation

Assumptions:

$\mathcal{L}_x^\phi u(x) = f(x)$

$u(x) \sim \mathcal{GP}(0, k_{uu}(x, x'; \theta))$

$f(x) \sim \mathcal{GP}(0, k_{ff}(x, x'; \theta, \phi))$

$\theta = \{\sigma, l\}$

Chosen operator: $\mathcal{L}_x^\phi u(x) = \phi * u(x)$

We choose the following two functions to sample the data (The factor 12 can be varied):

$$u(x) = \sqrt{x}$$

$$f(x) = 12 * u(x) = 12\sqrt{x}$$

Problem at hand: Given $\{X_u, y_u\}$ and $\{X_f, y_f\}$, estimate ϕ.

We clearly expect ϕ to be estimated as being as close to 12 as possible.

We will get an estimate of 12.05 using 15 evenly spaced data samples in the interval $[0, 2\pi]$.

We employ a GP with a RBF kernel for u and f:

$k_{uu}(x_i, x_j; \theta_u) = \sigma_u^2 \exp(-\frac{1}{2l_u^2}(x_i - x_j)^2)$

$k_{ff}(x_i, x_j; \theta_f) = \sigma_f^2 \exp(-\frac{1}{2l_f^2}(x_i - x_j)^2)$

We use the known transformation behavior of Gaussian Processes:

$k_{ff}(x_i, x_j; \theta, \phi)$
$= \mathcal{L}_{x_i}^\phi \mathcal{L}_{x_j}^\phi k_{uu}(x_i, x_j; \theta)$
$= \phi^2 \sigma_u^2 \exp(-\frac{1}{2l_u^2}(x_i - x_j)^2)$

Equating the two expressions we have for k_{ff} and comparing a diagonal entry (where $x_i = x_j$), it follows that $\sigma_f^2 = \phi^2 \sigma_u^2$, i.e.:

$$\phi = \frac{\sigma_f}{\sigma_u}$$

```
In [9]: def main():

            import numpy as np
            import pyGPs

            # Generating data:
            # Keeping it as simple as possible, using sine instead of sqrt the pyGPs-
            # optimizer won't be able to calculate the optimal hyperparameters,
            # independent of the method
            x_u = np.linspace(0, 2*np.pi, 15)
            y_u = np.sqrt(x_u)

            x_f = x_u
            y_f = 12.0*np.sqrt(x_f)

            # The function u is assumed to be a Gaussian Process.
            # After a linear transformation, f has to be a Gaussian Process as well.

            model_u = pyGPs.GPR()
            model_u.setData(x_u, y_u)
            model_u.optimize(x_u, y_u)

            model_f = pyGPs.GPR()
            model_f.setData(x_f, y_f)
            model_f.optimize(x_f, y_f)

            # Note that in hyp only the logarithm of the hyperparameter is stored!
            # Characteristic length-scale l is equal to np.exp(hyp[0]) (Default: 1)
            # Signal variance sigma is equal to np.exp(hyp[1]) (Default: 1)

            # This should give 12 as output:
            print(np.exp(model_f.covfunc.hyp[1])/np.exp(model_u.covfunc.hyp[1]))

        # Prevents execution by Sphinx:
        if __name__ == '__main__':

            main()
```

```
Number of line searches 40
Number of line searches 40

12.049201003414321
```

CHAPTER 6

Conclusion

It has been demonstrated that estimating parameters using a Gaussian process approach works well for a variety of PDEs arising from linear operators. Even though we have worked with time schemes in the example of Burgers' Equation, this methodology in general circumvents the need for any discretization methods. When working with truly noisy data, the parameter s (which we have set to 1e-7 in most cases) can be varied or even optimized as well. This approach unfortunately can't really be applied to non-linear equations, only by using workarounds like replacing the non-linear term with the mean as we did in Burgers' Equation, essentially transforming the non-linear equation to a linear one. Using this remedy, we still get good results as we saw in Chapters 4.2.4 and 4.2.5. In our case, five to eight data samples were sufficient to learn the parameters within a margin of error of 0.058 and 0.0009 in the linear, and 0.5 and 0.16 in the non-linear cases. Using 20 data samples, we were able to estimate the parameters to an accuracy of 0.6 percent.

6.1 Problems with the RBF kernel

One of the problems we noticed with the RBF kernel is that for more than 30 data points, the final covariance matrix was frequently ill-conditioned. This happens more often, when the length-scale is large. With the increased number of points, the probability of points being close to each other increases and then the respective columns of the covariance matrix will be almost equal, especially when working with a large length-scale. Hence, this kernel is not good enough for practical purposes on large datasets. The immediate solution that comes to mind to tackle this problem is to do a singular value decomposition. *[1]* outlines the RBF-QR algorithm to avoid these issues. Stable computations can also be acheived using Hermite polynomials *[2]*, *[3]*.

6.2 Non-linearity

In Burgers' Equation we approximated the non-linear term by a constant term. This yields good results for some special cases but is not a generic approach. The problem arises because the product of Gaussian processes does not result in a Gaussian process. Nevertheless, we could utilize the fact that the product of Gaussian distributions is also Gaussian in order to come up with a proper solution. Another approach is to assume priors over the kernel hyperparameters and infer parameters with MCMC sampling schemes *[4]*.

6.3 Kernel computations

The current framework involves two computations for every operator over the kernel. It is easy to do this by hand for simple operators but even then there is scope for manual error. It would be nice to have a tool to compute the transformed kernels automatically. Some progress has been made at this front resulting in a package for symbolic kernel computations available on this project's GitHub repository *[5]*.

APPENDIX A

Appendix

A.1 Demo of pyGPs

Here, we demonstrate a short program to show how to use the pyGPs-package for regression tasks.

```
In [4]: import numpy as np
        import pyGPs
```

We are generating data by taking 50 evenly spaced values between -5 and 5 as our set of points X. For our respective set of points Y we are generating Gaussian-distributed values with a parabola shaped mean and an RBF Kernel with an added error term as our covariance:

```
In [5]: X = np.arange(-5, 5, 0.2)
        s = 1e-8                                                  # error term
        n = X.size
        m = 1/4*np.square(X)                                      # mean
        a = np.repeat(X, n).reshape(n, n)
        k = np.exp(-0.5*(a - a.transpose())**2) + s*np.identity(n)    # covariance
        Y = np.random.multivariate_normal(m, k, 1)
        Y = Y.reshape(n)                                          # Converting y from a
                                                                  # matrix to an array
```

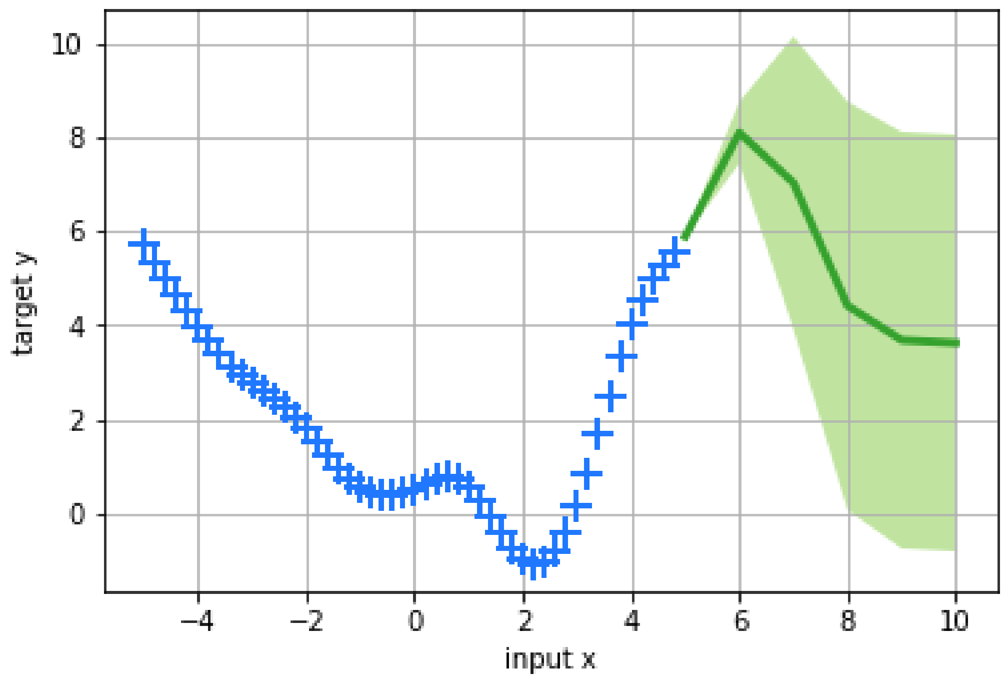
We treat the values $x \in X$ (whilst $X \subseteq \mathbb{R}$) as our input points and the values $y \in Y$ ($Y \subseteq \mathbb{R}$) as corresponding output points. Handing these over to our model variable as data, we can use pyGPs' Gaussian Process Regression (With a zero mean, an RBF Kernel and Gaussian likelihood as default).

Note, that we're using a zero mean Gaussian Process prior for our data, which has been generated by a parabola-shaped mean Gaussian. This shows the flexibility of the model.

Using this, we can predict output values for values of x outside of our initial range between -5 and 5.

```
In [17]: model = pyGPs.GPR()
         model.setData(X,Y)
         # model.setPrior(mean=pyGPs.mean.Zero(), kernel=pyGPs.cov.RBF()) is redundant
         model.optimize(X,Y)
         model.predict(np.array([5,6,7,8,9,10]))
         model.plot()
```

```
Number of line searches 40
```



A.2 Own Covariance function for the Heat Equation with pyGPs

The definition of this kernel was an attempt to train a pyGPs-GPR-model with the kernel given by k_{ff} in Chapter 3.1.2, in accordance with the approach as it was described in Chapter 5.

The coefficients were calculated using the heat_equation_with_pygps-v2.ipynb notebook on our GitHub-Page.

The optimal parameters for k_{uu} were estimated as $[\sigma_u, l_u] = [2e - 05, 1291919.81]$.

```
In [1]: # The kernel has to be added to cov.py, which is located (with Windows and
        # Anaconda) in C:\Users\SurfaceAdmin\Anaconda3\Lib\site-packages\pyGPs\Core

        # Unfortunately the kernel is basically equal to zero and the optimizer
        # will stick with the default value for the hyperparameter, which in turn
        # returns 1.0 for phi.

        # Own Kernel function (the try-except block was only added to prevent
        # error-messages concerning the unknown Kernel-class after execution of
        # the code):

        try:
            class MyKernel2(Kernel):

                def __init__(self, log_phi=0.):
                    self.hyp = [log_phi]

                def getCovMatrix(self,x=None,z=None,mode=None):
                    self.checkInputGetCovMatrix(x,z,mode)
                    p = np.exp(self.hyp[0])             # phi
                    A = 0
                    if not x is None:
```

```
            n, D = x.shape
            A = np.zeros((n,n))
        if not z is None:
            nn, D = z.shape
            A = np.zeros((nn,nn))
        if mode == 'self_test':
            A = np.zeros((nn,1))
        elif mode == 'train':                     # compute covariance matrix for
                                                  # the training set
            A = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    A[i][j] = (p**2*(1.43e-34*(x[i][1] - x[j][1])**4 - \
                            1.11e-27*(x[i][1] - x[j][1])**2 + 7.17e-22) - \
                            2.39e-22*(x[i][0] - x[j][0])**2 + 3.09e-16)* \
                            np.exp(-3.87e-7*(x[i][0] - x[j][0])**2 - \
                            3.87e-7*(x[i][1] - x[j][1])**2)

        elif mode == 'cross':   # compute covariance between data sets x and z
            m = z.shape[0]
            A = np.zeros((n,m))
            for i in range(n):
                for j in range(m):
                    A[i][j] = (p**2*(1.43e-34*(x[i][1] - z[j][1])**4 - \
                            1.11e-27*(x[i][1] - z[j][1])**2 + 7.17e-22) - \
                            2.39e-22*(x[i][0] - z[j][0])**2 + 3.09e-16)* \
                            np.exp(-3.87e-7*(x[i][0] - z[j][0])**2 - \
                            3.87e-7*(x[i][1] - z[j][1])**2)
        return A

# We are taking the derivative w.r.t. p, but are multiplying it with 2*p or p,
# since that seems to be the pattern in the source code of pyGPs as well:

    def getDerMatrix(self,x=None,z=None,mode=None,der=None):
        self.checkInputGetCovMatrix(x,z,mode)
        p = np.exp(self.hyp[0])           # phi
        n = 0
        if not x is None:
            n, D = x.shape
        if not z is None:
            nn, D = z.shape
        if mode == 'self_test':           # self covariances for the test cases
            A = np.zeros((nn,1))
        elif mode == 'train':             # compute covariance matrix for
                                          # the dataset x
            A = np.zeros((n,n))
            for i in range(n):
                for j in range(n):
                    A[i][j] = (p**2*(1.43e-34*(x[i][1] - x[j][1])**4 - \
                            1.11e-27*(x[i][1] - x[j][1])**2 + 7.17e-22) - \
                            2.39e-22*(x[i][0] - x[j][0])**2 + 3.09e-16)* \
                            np.exp(-3.87e-7*(x[i][0] - x[j][0])**2 - \
                            3.87e-7*(x[i][1] - x[j][1])**2)

        elif mode == 'cross':             # compute covariance between data
                                          # sets x and z
            A = np.zeros((n,nn))
            for i in range(n):
                for j in range(nn):
                    A[i][j] = (p**2*(1.43e-34*(x[i][1] - z[j][1])**4 - \
                            1.11e-27*(x[i][1] - z[j][1])**2 + 7.17e-22) - \
                            2.39e-22*(x[i][0] - z[j][0])**2 + 3.09e-16)* \
                            np.exp(-3.87e-7*(x[i][0] - z[j][0])**2 - \
                            3.87e-7*(x[i][1] - z[j][1])**2)
        if der == 0:    # compute derivative matrix wrt 1st parameter
            if mode == 'train':
                A = np.zeros((n,n))
                for i in range(n):
                    for j in range(n):
                        A[i][j] = 4*p**2*(1.43e-34*(x[i][1] - x[j][1])**4 - \
                                1.11e-27*(x[i][1] - x[j][1])**2 + 7.17e-22)*\
                                np.exp(-3.87e-7*(x[i][0] - x[j][0])**2 - \
                                3.87e-7*(x[i][1] - x[j][1])**2)
            elif mode == 'cross':
                A = np.zeros((n,nn))
                for i in range(n):
                    for j in range(nn):
                        A[i][j] = 4*p**2*(1.43e-34*(x[i][1] - z[j][1])**4 - \
                                1.11e-27*(x[i][1] - z[j][1])**2 + 7.17e-22)*\
                                np.exp(-3.87e-7*(x[i][0] - z[j][0])**2 - \
                                3.87e-7*(x[i][1] - z[j][1])**2)
        else:
            raise Exception("Calling a derivative in RBF that doesn't exist")
        return A
except:
pass
```

```
In [2]: # Has to be added to pyGPs/Testing/unit_test_cov.py and then unit_test_cov.py
        # has to be executed.
        # For testing purposes only:

        def test_cov_new(self):
            k = pyGPs.cov.MyKernel()     # specify your covariance function
            self.checkCovariance(k)
```

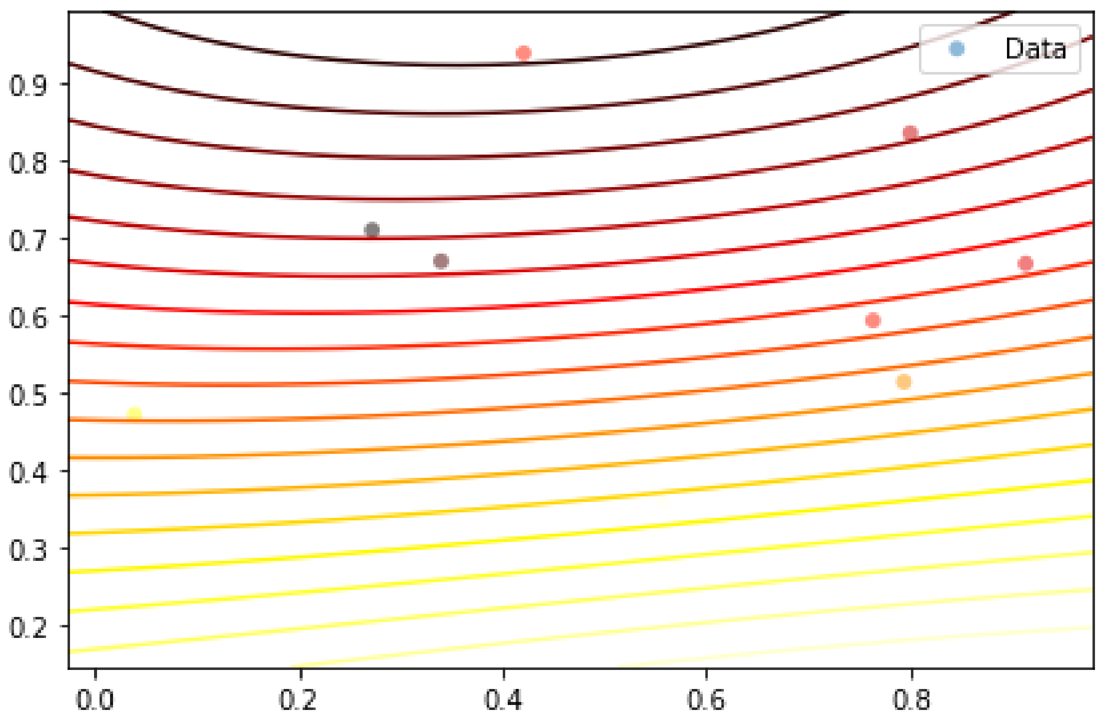
A.3 Exploring the GPy package

```
In [1]: import time
        import numpy as np
        import scipy as sp
        import matplotlib.pyplot as plt

In [2]: n = 10
        np.random.seed(int(time.time()))
        t = np.random.rand(n)
        x = np.random.rand(n)
        y_u = np.multiply(np.exp(-t), np.sin(2*np.pi*x))
        y_f = (4*np.pi**2 - 1) * np.multiply(np.exp(-t), np.sin(2*np.pi*x))

In [3]: import GPy

In [4]: kernel = GPy.kern.RBF(input_dim=2)
        kernel

Out[4]: <GPy.kern.src.rbf.RBF at 0x10660f9b0>

In [5]: m = GPy.models.GPRegression(np.matrix([t, x]).T, y_u.reshape(y_u.size,1),kernel)
In [6]: from IPython.display import display
        display(m)
```

```
<GPy.models.gp_regression.GPRegression at 0x10660ff28>
```

```
In [7]: fig = m.plot()
        display(GPy.plotting.show(fig, filename='basic_gp_regression_notebook_2d'))
```

```
 /usr/local/lib/python3.6/site-packages/matplotlib/figure.py:1743: UserWarning:This figure includes A
```



```
In [8]: m.optimize(messages=True)
```

```
Running L-BFGS-B (Scipy implementation) Code:
  runtime   i      f              |g|
    00s00  0003   1.797920e+01   3.159042e+02
    00s03  0012   3.656938e+00   9.295373e+00
    00s06  0023   6.417812e-01   7.386639e-03
    00s09  0035   6.255299e-01   4.347322e-06
    00s10  0040   6.255208e-01   1.657704e-08
    00s12  0045   6.255205e-01   6.876193e-11
    00s12  0046   6.255205e-01   6.876193e-11
Runtime:     00s12
Optimization status: Converged
```

```
Out[8]: <paramz.optimization.optimization.opt_lbfgsb at 0x1067d2ba8>
```

```
In [9]: m.optimize_restarts(num_restarts = 10)
```

```
Optimization restart 1/10, f = 0.6255204911801915
Optimization restart 2/10, f = 0.6255204901419562
Optimization restart 3/10, f = 0.6255214894302625
Optimization restart 4/10, f = 0.6255204755438637
Optimization restart 5/10, f = 0.625520537160762
Optimization restart 6/10, f = 0.6255204592848189
Optimization restart 7/10, f = 0.6255204754204922
Optimization restart 8/10, f = 0.6255213104151442
Optimization restart 9/10, f = 0.6255213338652572
Optimization restart 10/10, f = 0.6255205894920346
```

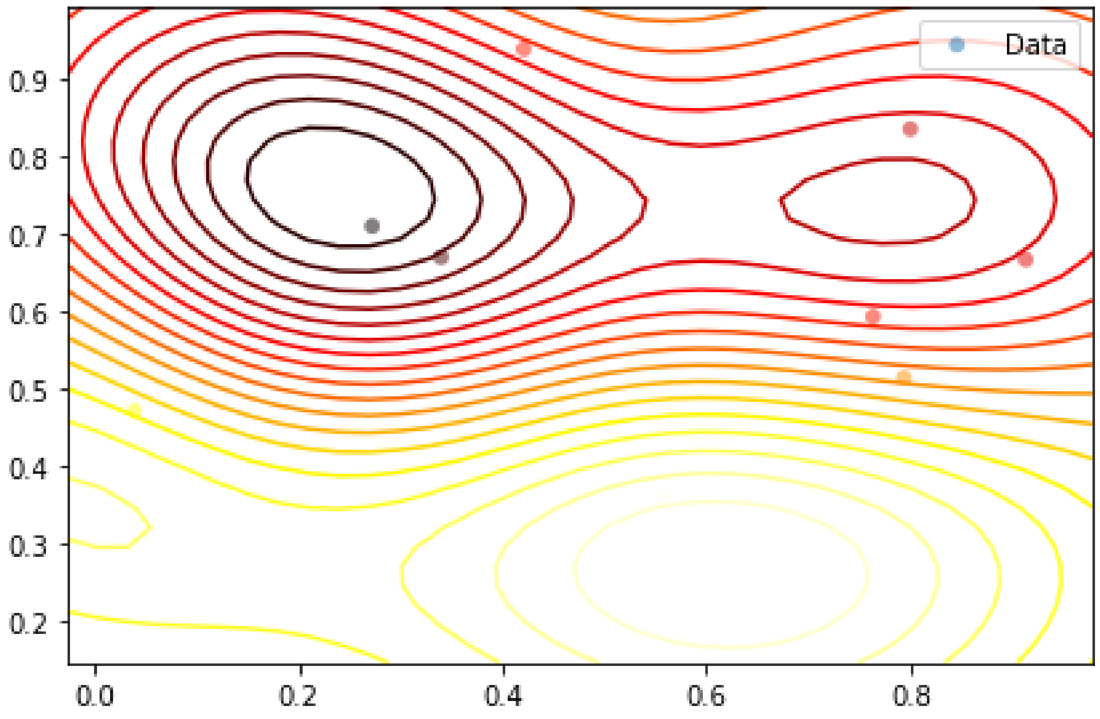
```
Out[9]: [<paramz.optimization.optimization.opt_lbfgsb at 0x1067d2ba8>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10317ffd0>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x1067d2908>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x103fce0f0>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10667c668>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10780b080>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10781fdd8>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10781f3c8>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10667c400>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x10781fda0>,
         <paramz.optimization.optimization.opt_lbfgsb at 0x1067d2a58>]
```

```
In [10]: display(m)
```

```
<GPy.models.gp_regression.GPRegression at 0x10660ff28>
```

```
In [11]: fig = m.plot()
         GPy.plotting.show(fig, filename='basic_gp_regression_notebook_optimized')
```

```
 /usr/local/lib/python3.6/site-packages/matplotlib/figure.py:1743: UserWarning:This figure includes A
```



Bibliography

[1] Gregory E. Fasshauer and Michael J. McCourt. Stable Evaluation of Gaussian Radial Basis Function Interpolants. *SIAM Journal on Scientific Computing*, 34(2):A737–A762, jan 2012. URL: http://epubs.siam.org/doi/10.1137/110824784, arXiv:arXiv:1302.5877, doi:10.1137/110824784.

[2] Bengt Fornberg, Elisabeth Larsson, and Natasha Flyer. Stable Computations with Gaussian Radial Basis Functions. *SIAM Journal on Scientific Computing*, 33(2):869–892, jan 2011. URL: http://epubs.siam.org/doi/10.1137/090750688http://epubs.siam.org/doi/10.1137/09076756X, arXiv:arXiv:1302.5877, doi:10.1137/09076756X.

[3] Anna Yurova and Katharina Kormann. Stable evaluation of Gaussian radial basis functions using Hermite polynomials. *ArXiv e-prints*, sep 2017. URL: https://arxiv.org/pdf/1709.02164.pdfhttp://arxiv.org/abs/1709.02164, arXiv:1709.02164.

[4] Ben Calderhead, Mark Girolami, and Neil D Lawrence. Accelerating Bayesian Inference over Nonlinear Differential Equations with Gaussian Processes. *Advances in Neural Information Processing Systems 21*, pages 217–224, 2009.

[5] Ahmed Ratnani, Kumar Harsha, Arthur Grundner, and Kangkang Wang. Machine Learning for Hidden Physics and Partial Differential Equations. 2018. URL: https://github.com/ratnania/mlhiphy.

[6] E Solak, R Murray-Smith, W.E. Leithead, D.J. Leith, and C.E. Rasmussen. Derivative observations in Gaussian process models of dynamic systems. *Nips 15*, pages 8, 2002. URL: https://papers.nips.cc/paper/2287-derivative-observations-in-gaussian-process-models-of-dynamic-systems.pdf.

[7] Carl E. Rasmussen and Christopher K. I. Williams. *Gaussian processes for machine learning.* Volume 14. The MIT Press, 2004. ISBN 026218253X. URL: http://www.gaussianprocess.org/gpml/chapters/RW.pdf.

[8] R B Platte and T A Driscoll. Polynomials and potential theory for Gaussian radial basis function interpolation. *SIAM Journal on Numerical Analysis*, 2005. doi:10.1137/040610143.

[9] Gabriel J. Lord, Catherine E. Powell, and Tony Shardlow. *An Introduction to Computational Stochastic PDEs*. Cambridge University Press, Cambridge, 2014. ISBN 9781139017329. URL: http://ebooks.cambridge.org/ref/id/CBO9781139017329, doi:10.1017/CBO9781139017329.

[10] Marion Neumann, Shan Huang, Daniel E Marthaler, and Kristian Kersting. pyGPs – A Python Library for Gaussian Process Regression and Classification. *Journal of Machine Learning Research*, 16:26112616, 2015.

[11] Carl Jidling, Niklas Wahlström, Adrian Wills, and Thomas B. Schön. Linearly constrained Gaussian processes. *ArXiv e-prints*, mar 2017. URL: http://arxiv.org/abs/1703.00787, arXiv:1703.00787.

[12] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Inferring solutions of differential equations using noisy multi-fidelity data. *Journal of Computational Physics*, 335:736–746, 2017. URL: http://dx.doi.org/10.1016/j.jcp.2017.01.060, arXiv:1607.04805, doi:10.1016/j.jcp.2017.01.060.

[13] Maziar Raissi, Paris Perdikaris, and George Em Karniadakis. Machine learning of linear differential equations using Gaussian processes. *Journal of Computational Physics*, 348:683–693, 2017. URL: http://dx.doi.org/10.1016/j.jcp.2017.07.050, arXiv:1701.02440, doi:10.1016/j.jcp.2017.07.050.

[14] J. Rashidinia, M. Khasi, and G. E. Fasshauer. A stable Gaussian radial basis function method for solving nonlinear unsteady convection–diffusion–reaction equations. *Computers and Mathematics with Applications*, 2018. doi:10.1016/j.camwa.2017.12.007.

[15] Pedro Henrique de Almeida Konzen, Esequia Sauter, Fabio Souto de Azevedo, and Paulo Ricardo de Ávila Zingano. Numerical simulations with the finite element method for the Burgers' equation on the real line. *ArXiv e-prints*, may 2016. URL: http://arxiv.org/abs/1605.01109, arXiv:1605.01109.

[16] Michael P. Lamoureux. The mathematics of PDEs and the wave equation. In *Seismic Imaging Summer School*. Pacific Institute for the Mathematical Sciences, 2006. URL: http://www.mathtube.org/sites/default/files/lecture-notes/Lamoureux\protect\T1\textbraceleft\T1\textbackslash{}_\protect\T1\textbracerightMichael.pdf.

[17] Marion Neumann, Shan Huang, Daniel E. Marthaler, and Kristian Kersting. Pygps – a python library for gaussian process regression and classification. *Journal of Machine Learning Research*, 16:2611–2616, 2015. URL: http://jmlr.org/papers/v16/neumann15a.html.