Master's thesis

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

Vilius Ciuzelis

Nonlinear optimization for Hyperparameter computation in Gaussian Processes machine learning

Master's thesis in Engineering Cybernetics
Supervisor: Lars Imsland
June 2019

NTNU
Norwegian University of Science and Technology

Vilius Ciuzelis

Nonlinear optimization for Hyperparameter computation in Gaussian Processes machine learning

Master's thesis in Engineering Cybernetics
Supervisor: Lars Imsland
June 2019

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Engineering Cybernetics

NTNU
Norwegian University of
Science and Technology

Summary

The goals of this thesis are:

1. Literature study on Gaussian Processes (GP), Optimization theory, and Nonlinear Conjugate Gradient (NCG) method.
2. Novel implementation of a GP algorithm in computer software.
3. Analysis of hyperparameter optimization using the NCG methods.
4. Case examples of the GP implementation and performance comparison to existing software.

Gaussian Process has showed great potential in ability to interpret highly non-linear models extremely well with minimal tuning. Two test cases have been designed to test the functionality of the GP implementation. The first one is a 1-dimensional sinusoidal wave regression with some noise. The second example is a 4-dimensional 2-link planar robot manipulator arm model with friction in the joints.

The results of first test case showed great potential and were comparable to the performance of the existing software. The runtime of the implementation was low and well scalable (26 seconds for 1000 runs of the optimization algorithm). The results of the second test case produced faulty results and point to problems in the implementation. Regardless of the final results, this thesis shows that Gaussian Processes is a great tool to have in the machine learning toolbox.

THIS PAGE LEFT INTENTIONALLY BLANK

Sammendrag

Målene til denne masteroppgaven er:

1. Litteraturstudie på Gaussiske Prosesser (GP), Optimeringsteori og metoden av ulineære konjugerte gradienter (NCG).
2. Ny implementasjon av en GP-algoritme i programvare.
3. Analyse av optimering av hyperparameter ved bruk av en NCG metode.
4. Testcaser av GP implementasjonen og sammenligne ytelse med eksisterende dataprogramvare.

Gaussiske Prosesser har vist stor potensiale i å kunne tolke høyst ulineære modeller med ekstrem presisjon med lite tuning. To testcaser har blitt designet for å teste funksjonalitet i GP implementasjonen. Den første er regresjon av en 1-dimensjonal sinus kurve med støy. Den andre er en 4-dimensjonal-robot manipulator modell med friksjon i leddene.

Resultatene til første case har vist stor potensiale og er sammenlignbare med ytelse til andre eksisterende implementasjoner. Kjøretiden var lav og skalerbar (26 sekunder for 1000 kjøringer av optimaliseringsalgoritmen). Resultatene til den andre cases har gitt defekte resultater og viser at implementasjonen ikke er feilfri. Uansett sluttresultatet, så viser dette arbeidet at GP er et fantastisk verktøy i maskinlæringskassen.

THIS PAGE LEFT INTENTIONALLY BLANK

Preface

This thesis is the finishing work that concludes my Master of Science degree in engineering cybernetics studies at Norwegian University of Science and Technology. Research performed here builds on a previous research on Gaussian Processes by me. It was my project thesis from January 2019 and is called “Gaussian Processes in non-linear regression”. It was also supervised by Lars Imsland at the department of Engineering Cybernetics at NTNU.

The basis for this thesis is heavily based on the book on Gaussian Processes by Rasmussen and Williams – “Gaussian Processes in Machine Learning” from 2006. The second edition of “Numerical Optimization” from 2006 by Jorge Nocedal and Stephen J. Wright was basis for the optimization theory and algorithms. The rest of the research literature can be found in bibliography in the end of the document.

I have received external help and guidance provided by Lars Imsland in the form of meetings throughout the research period (January 18th to June 13th). I want to thank him for patience and advice, and thank my family and my girlfriend for the support.

Contents

List of Tables

List of Figures

List of Algorithms

Nomenclature

$\text{cholesky}(A)$	-	cholesky decomposition. Returns L where $LL^{\mathsf{T}} = A$
D	-	number of dimensions in training data
δ_{pq}	-	Kronecker delta function which equals to 1 iff $p = q$ and 0 otherwise
$\mathbf{f}_*$ or f_*	-	Gaussian process posterior prediction
$\bar{\mathbf{f}}_*$ or $\bar{\mathrm{f}}_*$	-	Gaussian process predictive mean
g	-	gravitational constant $\approx 9.81\ m/s^2$
$\mathcal{GP}$	-	Gaussian process with a mean and a covariance function
I	-	moment of inertia
μ	-	mean value
$\boldsymbol{M}$	-	matrix of characteristic length-scales
m_i	-	mass of link i
$\mathcal{N}(\mu, \sigma^2)$	-	Normal (or Gaussian) distribution
n	-	number of training inputs
n_*	-	number of test inputs
$k(\mathbf{x}, \mathbf{x}')$	-	kernel (or similarity) function evaluated at $\mathbf{x}$ and $\mathbf{x}'$

$K(X,X)$	-	$n \times n$ covariance matrix
$\mathcal{L}$	-	Lagrangian
l	-	characteristic length-scale parameter
$m(\mathbf{x})$	-	mean function
σ_f^2	-	variance of the noise-free signal
σ_n^2	-	variance of the noise
q_i	-	joint i angle
$\dot{q}_\imath$	-	joint i angular velocity
$\ddot{q}_\imath$	-	joint i angular acceleration
T_*	-	kinetic co-energy
τ_i	-	torque of link i
$\boldsymbol{\theta}$	-	hyperparameters for GP
tr(A)	-	trace is the sum of the elements on the main diagonal of a square matrix. $\sum_{i=1}^{n} a_{ii} = a_{11} + a_{22} + \cdots + a_{nn}$
v_i	-	velocity of link i
V	-	potential energy
X	-	a $D \times n$ matrix of training inputs
X_*	-	a $D \times n_*$ matrix of test inputs
$\mathbf{x}_\mathrm{i}$	-	a $D \times 1$ vector of training input i
$\mathbf{y}$	-	a $1 \times n$ vector of training targets

1. Introduction

Gaussian Processes (GP) method is a powerful machine learning algorithm first introduced back in 1949 by Norbert Wiener. The method has many application areas, where regression and classification are the two most prominent ones. The main goal in a regression problem is to find a function that *fits* the data at hand as accurately as possible. There is virtually an infinite amount of functions that fit any finite set of data, so how does GP find "the solution" in finite time? Instead of finding all the individual functions, that are an infinity of, GP works with the probability distribution of these functions. The best guess is then the mean of this distribution. GP also provides the confidence of the prediction at every prediction point, a feature that can be used to design fault-tolerant and robust control models.

This thesis is intended to further develop the discussion surrounding the control theory using the non-parametric models, especially Gaussian Processes. It is written in an easy-to-understand language and all parts of the process are extensively discussed and described. To the contrary of some (faulty) implementations out there, the code in the Appendix B is backed up with concrete algorithms in the literature. It can be used as groundwork for further development of machine learning in process control.

Chapter II starts by presenting a planar robot manipulator model and deriving the equations of motion by using the Lagrangian method. Forward and backward kinematics are then used to find the relationship between the joint angles and the end effector position, which are later used in controller. Following a discussion about the model dynamics identification challenges and the choices for a good excitation trajectory. Then, the basics of the machine learning algorithm GP are introduced. Optimization theory, which is crucial for choice of hyperparameters, that inherently define the GP itself, is introduced and the necessary formulas are derived. Appendix A provides extensive explanation to the underlying mathematical principles necessary to understanding of this chapter.

Chapter III discusses the implementation of the ideas presented in Chapter II in computer software. MATLAB programming language has been chosen for this purpose. Code optimization ideas are also discussed throughout the presentation of the code. Two test cases have been made and are extensively described in the chapter. The first one is a benchmark case made for comparison with other existing software. The chapter starts by describing the general notion of the implementation common for both cases, before diving deeper into each case.

The results of the simulations are discussed in Chapter IV. Simulation plots and graphs

are presented and evaluated. A short discussion on the validity and usefulness of the results in the process control concludes the chapter.

Implementation improvements and critique is given in Chapter V. Following a few words on implementation pitfalls and challenges. Chapter VI suggests ideas for further research in the field of machine learning in control systems.

2. Background theory

2.1. Robot model

This thesis focuses on the model of the 2-link planar robot manipulator arm depicted in the Figure 1. The model is the simplified version of an industrial robot arm. The mathematical derivations throughout this chapter is trivial to extended to any number of joints. Since the focus of this thesis is the optimization and machine learning part, the simplified model has been chosen to merely save time when implementing the GP framework. This model, simple as it is, showcases the workings of GP just as well.

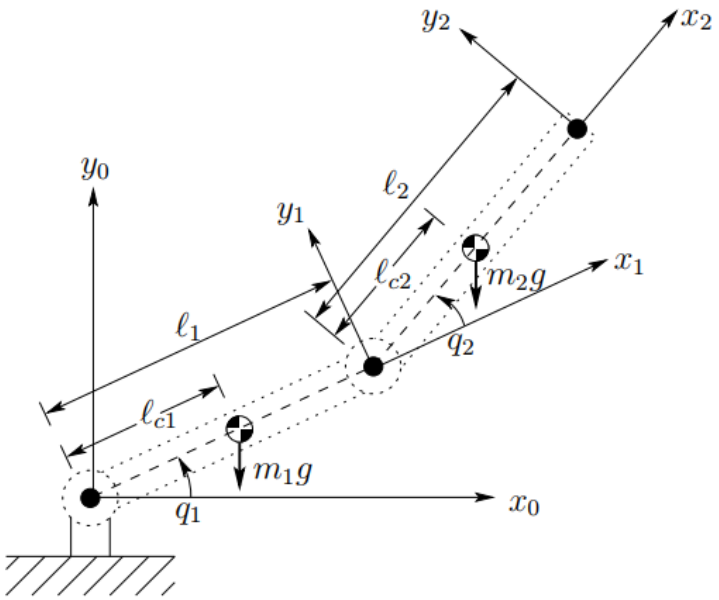


Figure 1. 2-link planar robot manipulator arm model. Illustration borrowed from [42].

The robot consists of 2 links and 2 joints. The lengths of the first and the second link are defined as l_1 and l_2, respectively. The base of the first link is attached to the ground forming joint 1. The second link is attached onto the top of the first link and will therefore be called joint 2. The angle the first link forms with respect to the horizontal ground is q_1, while the angle between the first and the second link is q_2. The links are formed as thin, uniform beams with masses m_1 and m_2, with mass centers in the middle of the links. The inertia about the center of mass for each link l_i is then given by

$$I_{i,center} = \frac{m_i l_i^2}{12}$$

The end of the second link, and therefore the robot, is known as end effector, or EE for short. This is the part of the robot which is of most interest as it is the position and reach of the EE that decides what the robot's capabilities are. The most common way of controlling the EE is by changing the joint angles that are driven by (usually) stepper motors inside the links. Therefore, the relationship between joint torques and the model parameters will be derived next.

2.1.1. Equations of Motion

As mentioned in the last section, the equations of motion (EOM) describe the relationship between the forces and the model motion, (kinematics). There are several well-known methods of obtaining EOMs, some of them being the Newton-Euler method, Lagrange's method and the Euler-Poincaré equation. All methods produce the same equations, but some are better suited for given applications than others. Lagrange's method is considered less complex to derive and very well suited for robotic manipulators. It produces simple expression for systems with a clear set of generalized coordinates, with the joint angles being excellent candidates. Whereas the systems described in terms or rotation matrices and angular velocities might find Newton-Euler method a better choice. The following derivation is extensively described in Appendix A, where the full derivation can be found in [1].

Let the total energy in the system be defined as the difference between the kinetic (co-energy) and the potential energies:

$$\mathcal{L} = \sum_i \frac{1}{2} m_i v_i^2 - \sum_i m_i g l_i$$

Equations of motion (EOM) are then found using the Lagrangian:

$$\frac{d}{dt}\left(\frac{\partial \mathcal{L}}{\partial \dot{q}_i}\right) - \frac{\partial \mathcal{L}}{\partial q_i} = \tau_i$$

Express for $\boldsymbol{\tau} = [\tau_1\ \tau_2]$ to arrive at EOMs:

$$\begin{aligned}\tau_1 = {} & \left(I_1 + I_2 + m_1 L_{c1}^2 + m_2 (L_1^2 + L_{c2}^2 + 2 L_1 L_{c2} \cos q_2)\right)\ddot{q}_1 \\ & + (I_2 + m_2 L_{c2}^2 + m_2 L_1 L_{c2} \cos q_2)\ddot{q}_2 - m_2 L_1 L_{c2} \sin q_2\ (2\dot{q}_1\dot{q}_2 + \dot{q}_2^2) \\ & + (m_1 L_{c1} + m_2 L_1) g \cos q_1 + m_2 L_{c2} g \cos(q_1 + q_2) \\ \tau_2 = {} & (I_2 + m_2 L_{c2}^2 + m_2 L_1 L_{c2} \cos q_2)\ddot{q}_1 + (I_2 + m_2 L_{c2}^2)\ddot{q}_2 + m_2 L_1 L_{c2} \dot{q}_1^2 sin\ q_2 \\ & + m_2 L_{c2} g \cos(q_1 + q_2)\end{aligned}$$

These equations can be written in a more compact form

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{G}(\boldsymbol{q}) = \boldsymbol{\tau}$$

where $\boldsymbol{M}(\boldsymbol{q})$ is a positive-definite mass matrix, $\boldsymbol{G}(\boldsymbol{q})$ is the gradient of the gravity potential, and $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ is the matrix contains so-called *Christoffel* symbols of the first kind. Using this compact form, the EOMs are fully ready to be implemented in computer software.

2.1.2. Kinematics

As mentioned earlier, kinematics describe the motion of the system without considering the forces causing the motion. *Forward kinematics* expresses the position of the end effector as the coordinates in the xy-plane as a function of the joint angles $\boldsymbol{q}$ and the links $\boldsymbol{l}$. These formulas are trivial to derive and will not be presented here. They take the following form

$$\begin{pmatrix} x_e \\ y_e \end{pmatrix} = \begin{pmatrix} l_1 \cos q_1 + l_2 \cos(q_1 + q_2) \\ l_1 \sin q_1 + l_2 \sin(q_1 + q_2) \end{pmatrix}$$

This result can be used for various purposes, one of them being tracking of the end effector measuring the joint angles.

Inverse kinematics problem is the reverse process of the forward kinematics problem, that is: given a position of the end effector in the xy-plane, compute possible joint angles and link geometries which correspond to that particular end effector position. Several forms of the solution exist, each one with different characteristics when it comes to computation complexity, orientation of the links, and avoidance of singularities. The formula below, for instance, does not have any singularities. For complete derivation of this formula, see Appendix A.

$$q(x_e, y_e) = \begin{bmatrix} \operatorname{atan2}(y_e, x_e) \pm \operatorname{atan2}\left(l_2 \sin q_2,\ (l_1 + l_2 \cos q_2)\right) \\ \operatorname{atan2}\left(\pm \sqrt{1 - \left(\frac{x_e^2 + y_e^2 - l_1^2 - l_2^2}{2 l_1 l_2}\right)^2},\ \frac{x_e^2 + y_e^2 - l_1^2 - l_2^2}{2 l_1 l_2} \right) \end{bmatrix}$$

2.1.3. Identification trajectory

Black-box model estimates use an input set to excite the model in such a way, that the excitation is able to identify most of the underlying dynamics of the system. In the presence of noise, such excitation trajectories play a crucial role when identifying model parameters. Therefore, the trajectory must be chosen carefully. Sometimes, the desired excitation might be difficult to achieve with physical systems. Even though there has been developed several optimal excitation methodologies, no one method suits every application.

In order to discover most of the coupled and highly non-linear dynamics of robotic manipulators, a sufficiently exciting motion trajectory is needed. One might use an intuitive approach and try to excite the robot in every direction with various velocities and hope that most of the dynamics will be discovered. This method would most likely yield a sub-optimal result compared to optimal techniques, see Figure 2. There exists however a method of maximizing the excitation motion throughout the identification motion, for instance as in [2]. Further investigations on the topic are left up to the reader.

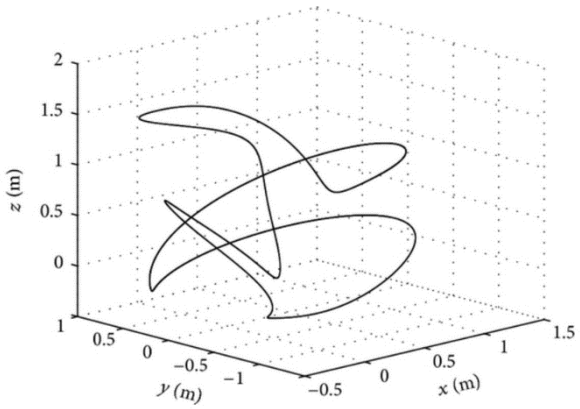


Figure 2. Possible end-effector path for parameter identification. Illustration borrowed from [41]

This thesis has used simple geometry figures, such as circles, to identify the dynamics. That is most likely a sub-optimal trajectory for this system, so it should be updated in the future research. Figure 3 shows the chosen identification trajectory used in the simulations.

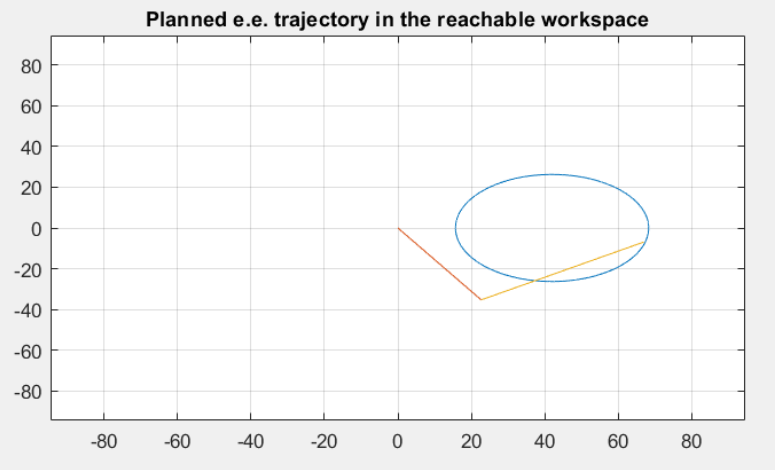


Figure 3. End effector trajectory for parameter identification used in the simulations.

2.2. Gaussian Processes

GP is used for making understandings about the relationships between the training and target data. Said in other words, input-output relationship or the conditional distribution of the targets, given the inputs [3]. By a more formal definition in the same book, a Gaussian Process "is a collection of random variables, any finite number of which have a joint Gaussian distribution". Given a dataset of $D \times n$ observations, which we call $\mathcal{D} = (\mathbf{X}, \mathbf{y})$, a GP is fully expressed by its mean and covariance functions $m(\mathbf{x})$ and $k(\mathbf{x}, \mathbf{x}')$. These functions are defined as

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})]$$
$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}\big[\big(f(\mathbf{x}) - m(\mathbf{x})\big)\big(f(\mathbf{x}') - m(\mathbf{x}')\big)\big]$$

where $f(\mathbf{x})$ is the process. A Gaussian Process using formulas above can be denoted as

$$f(\mathbf{x}) \sim \mathcal{GP}\big(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')\big)$$

2.2.1. Multivariate normal distribution

GP building blocks are multivariate normal distributions (MVN) or simply multivariate Gaussian distributions. It is a crucial part of the GP methodology. Given a multivariate normal distribution of a k-dimensional vector $\mathbf{X} = (\mathrm{X}_1, \ldots, X_k)^\top$

$$\mathbf{X} \sim \boldsymbol{\mathcal{N}}_k(\boldsymbol{\mu}, \boldsymbol{\Sigma})$$

where $\boldsymbol{\mu}$ is a k-dimensional mean vector

$$\boldsymbol{\mu} = \mathrm{E}[\mathbf{X}] = \big[\mathrm{E}[X_1], \mathrm{E}[X_2], \ldots, \mathrm{E}[X_k]\big]^\top$$

and $\boldsymbol{\Sigma}$ is a square k-dimensional covariance matrix, where each entry is

$$\Sigma_{\mathrm{i,j}} = \mathrm{E}\big[(X_i - \mu_i)(X_j - \mu_j)\big] = Cov\big[X_i, X_j\big]$$

The most important part to take from this is the requirement of the normal distribution when using GP. Extensive description and analysis of the multivariate normal distributions can be found in [3].

2.2.2. Mean

Mean function is useful when the data is expected to be at a given shape. In most cases, the mean is left 0 throughout the inference. The reason behind that is that the mean is usually well enough explained through covariance. There have therefore not been used any mean functions in this thesis. The mean is simply 0 for the rest of the thesis.

2.2.3. Covariance

Covariance functions, also called kernel functions or just kernels, describe the relationship between input-output pairs. The learning part of a GP inference is finding the "correct" properties for the covariance function. Any function that produces a positive definite covariance matrix is, in theory, a valid covariance function. The requirement of positive-definiteness stems from the definition of similarity between two points. Most kernels define this similarity simply as the distance between the points. There are several techniques of measuring this distance: Euclidean, Manhattan, Minkowski, Cosine, the squared Mahalanobis distance, and Jaccard [4]. Other generalized measures exist but will not be explained here and are left up to the reader himself to research. This thesis focuses on the distance measure by using the squared Euclidean distance.

In general, squared Euclidean distance between two $D \times 1$ points $(\mathbf{x}_\mathrm{i}, \mathbf{x}_\mathrm{j})$ is defined as:

$$distance_{SE}(\mathbf{x}_\mathrm{i}, \mathbf{x}_\mathrm{i}) = \left\|\mathbf{x}_\mathrm{i} - \mathbf{x}_\mathrm{j}\right\|^2 = \sum_{m=1}^{D} \left(x_{im} - x_{jm}\right)^2$$

To accommodate the vast range of possible nonlinear models, a number of kernel functions have been developed. These kernels have different properties concerning effects like, among others, stationarity, isotropy, and smoothness.

Stationarity refers to a stochastic process whose unconditional joint probability distribution, mean, and variance do not change in time. It means that $\mathbf{x_i} - \mathbf{x_j}$ only depends on the values of $\mathbf{x_i}$ and $\mathbf{x_j}$, but not their position in time. Isotropy deals with the measurement of distance. If a function is only dependent on values and not the measurement direction, then the function is called isotropic. Smoothness is defined by the expected closeness (or similarity) between input-output pairs. If the expectancy is high, the resulting function will tend to favor a more rapidly changing model rather than a slower producing a *smoother* model.

The choice of the kernel function is not arbitrary. The most common and widely used

kernel function is the Squared Exponential (SE). Despite its widespread use, this kernel has been critiqued for being "too smooth" and therefore unrealistic in most cases [3]. There is however nothing that indicates that this kernel is not suitable for the application in this thesis. It is defined as

$$k(\mathbf{x}_\mathrm{i}, \mathbf{x}_\mathrm{j}) = \sigma_\mathrm{f}^2 \exp\left(-\frac{1}{2}(\mathbf{x}_\mathrm{i} - \mathbf{x}_\mathrm{j})^\top \boldsymbol{M}(\mathbf{x}_\mathrm{i} - \mathbf{x}_\mathrm{j})\right) + \sigma_n^2 \delta_{\mathbf{x}_\mathrm{i}\mathbf{x}_\mathrm{j}}$$

where σ_f^2 is the variance of the noise-free signal, σ_n^2 is the variance of the noise, and $\delta_{\mathbf{x}_\mathrm{i}\mathbf{x}_\mathrm{j}}$ is a Kronecker delta which is 1 if $\mathbf{x}_\mathrm{i} = \mathbf{x}_\mathrm{j}$ and 0 otherwise. $\boldsymbol{M}$ is simply a symmetric matrix containing *the characteristic length-scales* $\boldsymbol{l}$. Characteristic length-scale decides how fast pace of change the sample functions are to have. Low l values will yield a more rapidly changing functions, while greater values will tend to smooth out the functions. It might take one of the following forms, depending on the values of hyperparameters:

$$M_1 = l^{-2}\boldsymbol{I}, \qquad M_2 = diag(\boldsymbol{l})^{-2}, \qquad M_3 = \Lambda\Lambda^\top + diag(\boldsymbol{l})^{-2}$$

For input with multiple dimensions, one can define a suitable covariance function with a property called in the literature Automatic Relevance Determination (ARD). ARD kernels weight each input dimension differently. Features that do not contribute enough for the explanation of the model are weighted lower for being not relevant, hence the R in ARD. This effect is automatic. There exist numerous ARD kernels suited for different applications, but the one used in this thesis is the Squared Exponential (SE) ARD kernel. Other kernels with ARD property are listed below. The SE-ARD kernel takes the following form:

$$k(\mathbf{x}_\mathrm{i}, \mathbf{x}_\mathrm{j}|\theta) = \sigma_f^2 \exp\left[-\frac{1}{2}\sum_{m=1}^{d}\frac{(x_{im} - x_{jm})^2}{l_m^2}\right]$$

Table 1 contain the most widely used kernels with the ARD property. The first step of a successful GP process is usually the selection of functions to be used in the inference – the prior distribution.

ARD Squared Exponential	$\sigma_f^2 \exp\left[-\frac{1}{2}\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}\right]$
ARD Exponential	$\sigma_f^2 \exp\left[-\sqrt{\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}}\right]$
ARD Matern 3/2	$\sigma_f^2\left(1+\sqrt{3}\sqrt{\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}}\right)\exp\left[-\sqrt{3}\sqrt{\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}}\right]$
ARD Matern 5/2	$\sigma_f^2\left(1+\sqrt{5}\sqrt{\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}}+\frac{5}{3}\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}\right)\exp\left[-\sqrt{5}\sqrt{\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}}\right]$
ARD Rational Quadratic	$\sigma_f^2\left(1+\frac{1}{2\alpha}\sum_{m=1}^{d}\frac{(x_{im}-x_{jm})^2}{l_m^2}\right)^{-\alpha}$ where when $\alpha \to \infty$, the kernel is identical to the SE-ARD kernel

Table 1. Example of kernel functions with the ARD property

2.2.4. Prior

Prior prediction, also called prior distribution, or just prior, is used to encode any prior knowledge about model to help infer the correct results. That is done by choosing the initial hyperparameter values, the mean, and the covariance function. A simple inference on the test inputs can then be performed, which reveals the function pool that will be drawn from in the inference. For example, if the data at hand is very rough, say, similar to the Brownian motion, probably the best suited kernel would be a Matérn 5/2 or a SE with low characteristic length values.

Note that the prior distribution is solely dependent on the test data. One can sample the

desired number of sample functions from GP prior using the multivariate normal distribution

$$\mathbf{f}_* \sim \mathcal{N}\big(\mathbf{0}, K(X_*, X_*)\big)$$

Prior functions can be sampled using the following operation:

$$\mathbf{f}_{\text{prior}} = \text{cholesky}(K(X_*, X_*) + \epsilon I)\boldsymbol{u} \tag{1}$$

where $\boldsymbol{u} \sim \boldsymbol{\mathcal{N}}(\mathbf{0}, I)$, that is a $n_* \times N$ matrix of random functions drawn from a normal (Gaussian) distribution with 0 mean and unit standard deviation, and ϵ is a small constant, usually in order of 10^{-6}. Most software tools have a tool for generating these random functions, often called a Gaussian generator. An example of such a sample of prior functions is shown in Figure 4.

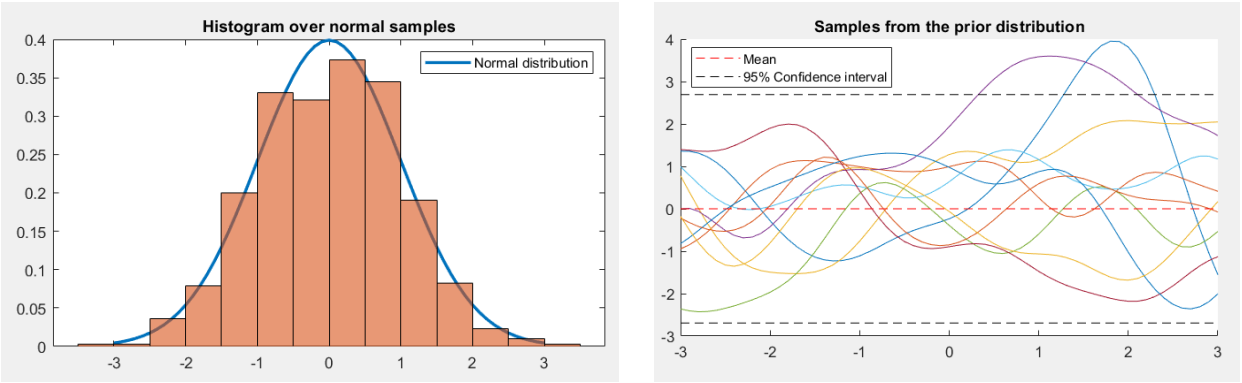


Figure 4. Example of multiple sample functions from the prior distribution.

2.2.5. Inference

Inference is the main process of the GP. The input data, together with the optimized hyperparameters, is used to approximate the mean of the given data and produce the confidence interval as to how accurate the predictions might be. The joint distribution of the observations at the test points are [3]:

$$\begin{bmatrix}\mathbf{y}\\ \mathbf{f}_*\end{bmatrix} \sim \mathcal{N}\left(\mathbf{0}, \begin{bmatrix}K(X,X) + \sigma_n^2 & K(X,X_*)\\ K(X_*,X) & K(X_*,X_*)\end{bmatrix}\right)$$

The fully derived conditional on Gaussian prior distribution is then

$$\bar{\mathbf{f}}_* = K(X_*,X)[K(X,X) + \sigma_n^2 I]^{-1}\mathbf{y}$$
$$\text{cov}(\mathbf{f}_*) = K(X_*,X_*) - K(X_*,X)[K(X,X) - \sigma_n^2 I]^{-1}K(X,X_*)$$

where $[K(X,X) - \sigma_n^2 I]^{-1}$ should be computed by using the Cholesky factorization. This yields a more efficient and numerically stable inversion process. When predicting noisy test data, one must add the noise variance σ_n^2 to the diagonal of the covariance matrix $\text{cov}(\mathbf{f}_*)$.

2.2.6. Posterior

The posterior distribution can be constructed to show which functions conform to the inference model produced by the GP. The operation is identical to when generating the prior, except for that covariance matrix now uses the input data, rather than only the test inputs:

$$\mathbf{f}_{\text{posterior}} = \text{cov}(\mathbf{f}_*)\boldsymbol{u} = \big(K(X_*, X_*) - K(X_*, X)[K(X,X) - \sigma_n^2 I]^{-1} K(X, X_*)\big)\boldsymbol{u}$$

The posterior combines the mean and the variance to show the final prediction, which an example of is shown in Figure 5.

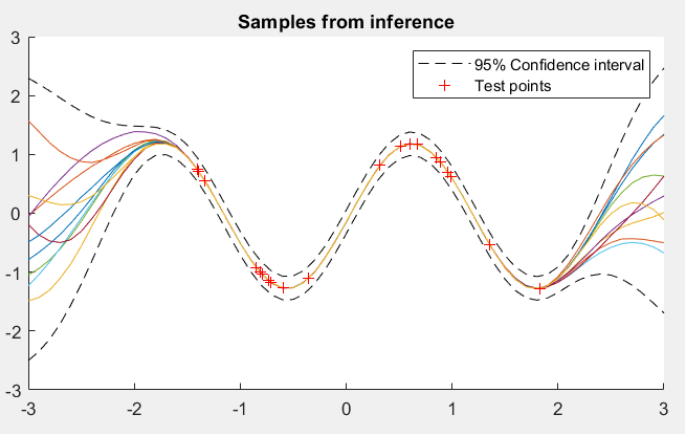


Figure 5. Posterior distribution.

2.2.7. Model validation

Model validation concerns measuring the accuracy of a given GP inference or prediction. The simplest method of measuring validity of predictions is the *mean squared error* (MSE), which is defined as:

$$MSE = mean\left(y_* - \overline{f}_*(X_*)\right)^2$$

where $\overline{f}_*$ is the inferred mean, y_* are the target input, and X_* is the test point set. As this measure

is sensitive to the overall scale of the target values [5], the *standardized mean squared error* is usually used:

$$SMSE = \frac{MSE}{\sigma_y}$$

SMSE of 0 indicate perfect correlation, while 1 indicate pure guessing.

2.2.8. Hyperparameters

Hyperparameters play the key role in the Gaussian Processes. They are the defining parameters that form the shape of the covariance matrix. Recall the squared exponential kernel $k_{SE}(x_i, x_j) = \sigma_f^2 \exp \frac{(x_i - x_j)^2}{l^2}$, parameters σ_f and l are the hyperparameters. They can be set using either intuition or use an automated technique. Intuition tends to produce sub-optimal results as the process of finding the optimal parameters is not trivial, especially when the dimension of the data grows. Intuition, or some prior knowledge of the system at hand, is however useful when initializing the hyperparameters and setting the GP prior.

Automated techniques for finding suitable hyperparameters almost always include gradient-based approaches. There exist gradient-free approaches, but they are not used as much in practice. Grid search and derivative-free-optimization (DFO) are examples of these. The grid search method, that searches for optima values in a grid of values, tends to be too computationally expensive, whereas the DFO methods usually yield worse performance than the gradient-based counterparts and should therefore only be used when the information of the gradients of the optimizing function is not available or too costly to compute.

The hyperparameters are found by minimizing a cost, or a loss, function $\mathcal{L}(y_{true}, y_{guess})$ with respect to hyperparameters. There exist several powerful cost functions, each with different properties, such that Bayesian – log marginal likelihood (LML), or a Cross Validation-based method – Leave One Out Cross Validation (LOO-CV). The LML is a little more computationally efficient and will therefore be focused on in this thesis. There is some research however, that points to the CV-based methods being more robust against model misspecification [6].

Maximization of LML is equivalent to minimization of the negative – negative log marginal likelihood (NLML). NLML produces a self-regulating cost function that produces a scalar value. It contains an automatic *Occam's razor* between the data explanation and the complexity of that explanation, see Figure 7. The better the explanation of the data, the more

complex model will get. That makes sense, because a perfectly exact model will fit every input exactly producing an effect known as *overfitting*. On the other hand, low complexity models tend to generalize too much and might not explain the data at all. This will cause *underfitting*. Both effects are clearly seen in Figure 6.

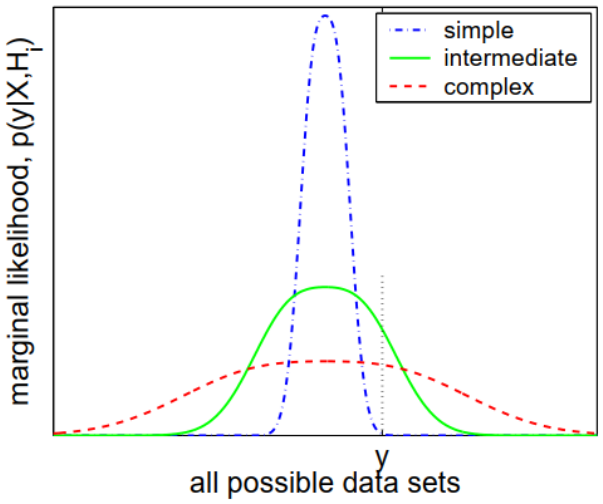


Figure 7. Self-regulating nature of the marginal likelihood. Illustration borrowed from [3].

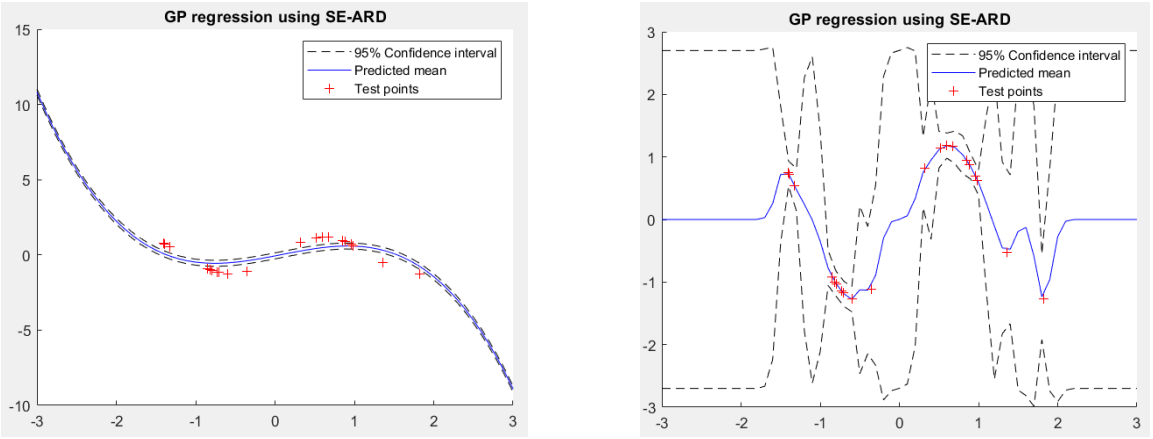


Figure 6. Examples of underfitting and overfitting.

The formal definition of the negative log marginal likelihood is

$$-\log p(\mathbf{y}|X,\boldsymbol{\theta}) = \underbrace{\frac{1}{2}\mathbf{y}^{\top}K_y^{-1}\mathbf{y}}_{data-fit} + \underbrace{\frac{1}{2}\log|K_y|}_{complexity\ penalty} + \underbrace{\frac{n}{2}\log 2\pi}_{normalization\ constant}$$

where $K_y = K(X,X) + \sigma_n^2 I$. For a gradient based NLML optimization approach, the gradient of the NLML is given below:

$$-\frac{\partial}{\partial\theta_j}\log p(\mathbf{y}|X,\boldsymbol{\theta}) = -\frac{1}{2}\mathbf{y}^{\top}K_y^{-1}\mathbf{y} + \frac{1}{2}\mathrm{tr}\left(K_y^{-1}\frac{\partial}{\partial\theta_j}\right)$$
$$= -\frac{1}{2}\mathrm{tr}\left(\left(\boldsymbol{\alpha}\boldsymbol{\alpha}^{\top} - K_y^{-1}\right)\frac{\partial K_y}{\partial\theta_j}\right)$$

where $\boldsymbol{\alpha} = K_y^{-1}\mathbf{y}$ and $\boldsymbol{\theta} = (\sigma_f^2, \sigma_n^2, l)$. Partial derivatives of K_y with respect to the hyperparameters are trivial to compute for most kernels. For other kernels, the derivatives might not readily be available and approximation methods might be used. The finite difference or automatic differentiation can be used. See [Appendix A] for the complete list of the standard kernel derivatives with respect to hyperparameters. Optimization algorithms often used in GP involving nonlinear problems are either quasi-Newton or nonlinear conjugate gradient methods. See page 20 for the introduction of the method of conjugate gradients.

2.3. Optimization theory

As mentioned earlier, optimization of the hyperparameters is the key element to a successful GP prediction. This chapter will try to introduce basics of the optimization theory necessary for the understanding of the GP. An excellent source for the extensive derivations and proofs is found in [5].

Every optimization problem starts with a function to be minimized or maximized:

$$\min_{x\in\mathbb{R}^n} f(x) \qquad \text{or} \qquad \max_{x\in\mathbb{R}^n} -f(x)$$

Given an initial point x_0, search for the next point x_1 that produces a better result – lower value of $f(x)$. Possibly the best direction to look for next x is the direction in which the value of the function $f(x)$ *descends* the most – a *descent* direction. Define this direction as $p_k = -\frac{d}{dx}f(x) = -\nabla f(x)$, or in general terms as $p_k = -B_k^{-1}\nabla f_k$, where B_k varies for different methods. For example, B_k is an identity matrix I in the steepest descent method, exact Hessian $\nabla^2 f(x)$ in Newton's method and approximation to the Hessian in quasi-Newton methods. Conjugate gradient methods use the combination of the previous and the current search directions: $-\nabla f_k + \beta_k p_{k-1}$. More on the choice of β_k in the introduction of NCG on page 20.

When the desired direction is found, one has to determine how far along this direction lies the improvement point x_1, in other words, find the step size α_k:

$$x_{k+1} = x_k + \alpha_k p_k$$

It is often found using interpolation or using some multiple of a constant. Next section discusses the methods available to take the right size steps as to ensure the convergence of the algorithm.

2.3.1. Interpolation

Given a set of value pairs (x, y), and possibly the derivatives at those points, interpolation is a method of *approximating new points* that belong to that same set. See Figure 8 for the illustration of the concept.

In the line search algorithms, the interpolation technique is used to find the optimal step size, given a step size interval. There exist several interpolation algorithms, based on the current information of the optimization function. Some examples are the quadratic, cubic, and the three-point interpolation. [5] suggests using a quadratic interpolation followed by the cubic one. The quadratic interpolation is expressed as

$$\phi_q(\alpha) = \left(\frac{\phi(\alpha_0) - \phi(0) - \alpha_0\phi'(0)}{\alpha_0^2}\right)\alpha^2 + \phi'(0)\alpha + \phi(0)$$

The step size α_{min} that minimizes ϕ_q is at the zero point of the derivative

$$\phi_q' = 0 = 2\left(\phi(\alpha_0) - \phi(0) - \frac{\alpha_0\phi'(0)}{\alpha_0^2}\right)\alpha_{min} + \phi'(0)$$

$$\alpha_{min} = -\frac{\phi'(0)\alpha_0^2}{2[\phi(\alpha_0) - \phi(0) - \alpha_0\phi'(0)]}$$

Not all steps will yield fast solution or even convergence. Therefore, some conditions need to be imposed, to be able to find the correct step size fast.

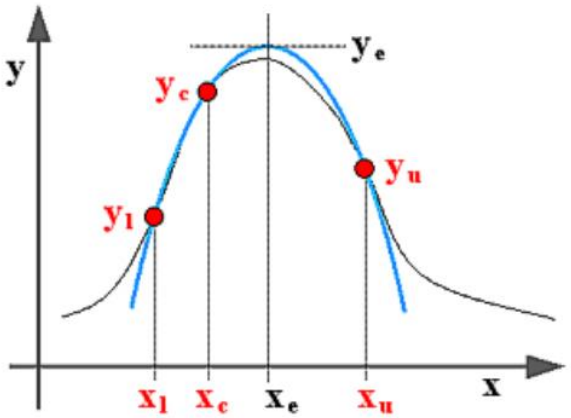


Figure 8. By interpolating the given value pairs, it is possible to approximate the interior point of the function.

Wolfe conditions

Wolfe conditions is a popular set of inexact line search abortion conditions, that ensure fast convergence. The set consists of two equations, where the first one is called the *sufficient decrease condition*, or Armijo condition, and the second one is referred to as the *curvature condition*. The sufficient decrease condition require a sizeable decrease in the function value for any step size. The curvature condition, on the other hand, forbid too small steps to ensure convergence. Together, the conditions ensure a reliable method of finding the near-optimal step size. See Figure 9 for illustration of the conditions. The official definition of wolfe conditions is:

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^\top p_k,$$
$$\nabla f(x_k + \alpha_k p_k)^\top p_k \geq -c_2 \nabla f_k^\top p_k$$

where $0 < c_1 < c_2 < \frac{1}{2}$. In practice, the parameters c_1 and c_2 are usually equal 10^{-4} and 0.1 respectively, for conjugate gradient methods [5]. A small modification of the conditions offer an even greater convergence and are often used in NCG. The modified condition exclude points that are far from stationary points of $f(x_k + \alpha_k p_k)$. These conditions are called the *strong Wolfe conditions:*

$$f(x_k + \alpha_k p_k) \leq f(x_k) + c_1 \alpha_k \nabla f_k^\top p_k,$$
$$|\nabla f(x_k + \alpha_k p_k)^\top p_k| \geq -c_2 \nabla f_k^\top p_k$$

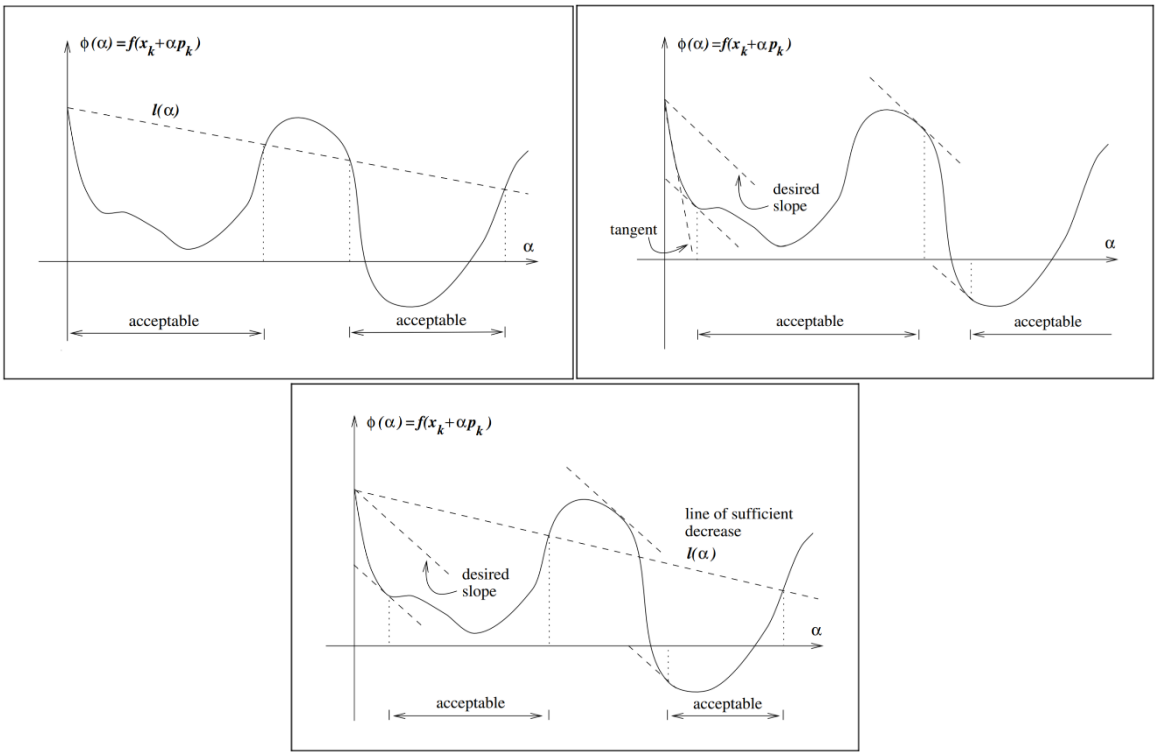


Figure 9. Illustration of the Wolfe conditions. Upper left: the sufficient decrease condition, upper right: the curvature condition, lower: the steps satisfying the Wolfe conditions. The illustration is from [3]

2.3.2. Nonlinear Conjugate Gradients

Conjugate gradient methods have firstly been proposed for solving large linear systems of equations, as an alternative to Gaussian elimination. With little adaptation, the method can also be used to solve nonlinear optimization problems. The attractive properties of both methods are the fast convergence (faster than the steepest descent) and no need of matrix storage, as opposed to the methods involving Newton directions.

Linear conjugate gradient (CG) method was first proposed by Hestenes and Stiefel in the 1950s. The first *nonlinear* conjugate gradient (NCG) algorithm was proposed by Fletcher and Reeves in the 1960s. The use of NCG algorithms has been widespread and involves, among others, neural net training and nonlinear regression. The convergence of linear methods is closely tied to the eigenvalues of the coefficient matrix. The more spread are the eigenvalues, the slower is the algorithm. See [5] for more info on the topic.

The key elements of the CG algorithms are the use of gradients and the property called

conjugates. Define a set of nonzero vectors $\{p_0, p_1, \dots, p_l\}$ to be *conjugate* with respect to the symmetric positive definite matrix A if

$$p_i^\top A p_j = 0, \qquad \text{for all } i \neq j.$$

It can also be shown that any set of vectors satisfying the above property is also linearly independent. All conjugate vectors are conjugate to each other, therefore there is no need of storing the previous values of conjugate vectors.

As mentioned, the second important part of an NCG algorithm is the use of gradients. Most NCG variations use available gradient information of the current and the previous point, while some others also require hessian or approximate hessian, such as Newton-Rhapson or Secant variations [7]. It is not always trivial to find the hessian analytically; approximation methods can be used – finite differences or automatic differentiation [5].

Since this thesis only uses NCG, it is left up to the reader to investigate the CG algorithms by himself. The general outline of all NCG methods is depicted in the Algorithm 1 below.

input: x_0 (starting point), f (minimizer)

1: **set** $x = x_0$, $p = -\nabla f(x_0)$,
2: **repeat**
3: **find** step length $\alpha \geq 0$
4: **update** $x = x_{previous} + \alpha \cdot p$
5: **set** β using gradient information
6: **update** $p = -\nabla f(x) + \beta \cdot p_{previous}$
end (**repeat**)

Algorithm 1. A general NCG algorithm.

The Algorithm 1 is usually repeated until some convergence conditions are achieved. For instance, when α or p approaches zero, when the progress of decrease of the minimizer is sufficiently small or simply stop after a given amount of iterations.

All NCG algorithms are similar, except for one property: descent direction update. That is the only feature that distinguish the different NCG methods. Below, some of the most prominent NCG methods are defined. The Fletcher-Reeves (FR) method is one of the first ones proposed and has good convergence properties:

$$\beta_{k+1}^{FR} = \frac{\nabla f_{k+1}^{\top} \nabla f_{k+1}}{\nabla f_k^{\top} \nabla f_k}$$

The Polak-Ribière (PR) method does not guarantee that p_k will remain a descent direction even when the line search is performed using the strong Wolfe conditions.

$$\beta_{k+1}^{PR} = \frac{\nabla f_{k+1}^{\top} (\nabla f_{k+1} - \nabla f_k)}{\| \nabla f \|^2}$$

A small improvement to the method though, provides that property. The algorithm is called Polar-Ribière plus (PR+), which usually outperforms FR. It is formulated as:

$$\beta_{k+1}^{PR+} = \max\{\beta_{k+1}^{PR}, 0\}$$

Another method, that promises global convergence is, again, a modification of the PR algorithm, called FR-PR:

$$\beta_k = \begin{cases} -\beta_k^{FR} & if \quad \beta_k^{PR} < -\beta_k^{FR} \\ \beta_k^{PR} & if \quad |\beta_k^{PR}| \leq \beta_k^{FR} \\ \beta_k^{FR} & if \quad \beta_k^{PR} > \beta_k^{FR} \end{cases}$$

2.3.3. Preconditioning

For the methods that use the hessian information, the convergence can be greatly improved by manipulating the hessian. The method is called *preconditioning* and it is a method of matrix manipulation to improve the *condition number* of a matrix, or a Hessian in this case. The condition number for a given symmetric, positive-definite matrix M in $M^{-1}Ax = M^{-1}b$ is defined as the ratio between the largest and smallest eigenvalue of that matrix.

$$\kappa = \lambda_{max}/\lambda_{min}$$

Analysis of condition number is beyond the scope of this thesis. To sum it up, the convergence of an optimization algorithm is worst for great values of κ. Matrices with great condition numbers are called *ill-conditioned*.

There exist several precondition techniques of preconditioning with different properties, such that i) the total performance increase, ii) inexpensiveness of computation and storage of M, or iii) inexpensiveness of solution $My = r$, which is a cheap approximation to

the solution $Ax = b$ [5]. Some promising general-purpose preconditioners are *symmetric successive overrelaxation* (SSOR), incomplete Cholesky factorization, and banded preconditioners. Further investigation on preconditioning methods is left up to the reader. No preconditioners were found needed in the simulations of the problems described in this thesis.

3. Simulation

3.1. General

Simulations described in this chapter were carried out using a Windows 10 operating system laptop with Intel Core i7-7700 CPU @ 3.60 GHz and 8 GB RAM. All simulations were entirely written in MATLAB programming language. The version of MATLAB at the time is pre-release version 2018a. As the code was written avoiding the short-term implementations in MATLAB (i.e. the inline command), the code should be flexible to tackle some older and newer versions of MATLAB.

The code used in the simulations is written by the author unless specifically stated otherwise. Both examples start with generating the input data. This data is then standardized according to the formula below:

$$standardized\ data = \frac{data - data\ mean}{data\ standard\ deviation}$$

This is done to help algorithms recognize patterns in the data. Then the optimization can be started by initializing the hyperparameters: initialize length-scale and function variance to 1. The experience also shows that higher initial values for the target function variance yield better performance when optimizing the other hyperparameters, even when expecting low noise. A small number of restarts of the optimization algorithm with different initial starting points is a possible remedy for when local optima is reached [8].

Algorithm 3.5 (Line Search Algorithm).
Set $\alpha_0 \leftarrow 0$, choose $\alpha_{\max} > 0$ and $\alpha_1 \in (0, \alpha_{\max})$;
$i \leftarrow 1$;
repeat
  Evaluate $\phi(\alpha_i)$;
  if $\phi(\alpha_i) > \phi(0) + c_1\alpha_i\phi'(0)$ or $[\phi(\alpha_i) \geq \phi(\alpha_{i-1})$ and $i > 1]$
    $\alpha_* \leftarrow$**zoom**(α_{i-1}, α_i) and **stop**;
  Evaluate $\phi'(\alpha_i)$;
  if $|\phi'(\alpha_i)| \leq -c_2\phi'(0)$
    set $\alpha_* \leftarrow \alpha_i$ and **stop**;
  if $\phi'(\alpha_i) \geq 0$
    set $\alpha_* \leftarrow$**zoom**(α_i, α_{i-1}) and **stop**;
  Choose $\alpha_{i+1} \in (\alpha_i, \alpha_{\max})$;
  $i \leftarrow i + 1$;
end (repeat)

Algorithm 2. Line search algorithm, ref [3].

Line search methods consist of two phases: the bracketing phase, where a smaller interval of test values, known to contain a minimum, is determined, and an interpolation phase, which finds the local minimum on the given interval. See Algorithm 2 and Algorithm 3. The criteria for the abortion of the search is the strong Wolfe conditions. The workings of both algorithms are rather easy to understand and are extensively described in [5]. NCG abort conditions are 100 iterations or step length being too small ($< 1.4^{-10}$) 5 times in a row.

Algorithm 3.6 (zoom).
repeat
Interpolate (using quadratic, cubic, or bisection) to find a trial step length α_j between α_{lo} and α_{hi};
Evaluate $\phi(\alpha_j)$;
if $\phi(\alpha_j) > \phi(0) + c_1\alpha_j\phi'(0)$ or $\phi(\alpha_j) \geq \phi(\alpha_{\text{lo}})$
$\alpha_{\text{hi}} \leftarrow \alpha_j$;
else
Evaluate $\phi'(\alpha_j)$;
if $|\phi'(\alpha_j)| \leq -c_2\phi'(0)$
Set $\alpha_* \leftarrow \alpha_j$ and **stop**;
if $\phi'(\alpha_j)(\alpha_{\text{hi}} - \alpha_{\text{lo}}) \geq 0$
$\alpha_{\text{hi}} \leftarrow \alpha_{\text{lo}}$;
$\alpha_{\text{lo}} \leftarrow \alpha_j$;
end (repeat)

Algorithm 3. Part 2 of line search algorithm, ref [3].

3.2. Example test case

The following test case has been used as an example case by Rasmussen on the GPML documentation web page [9].The training inputs X are 20 random points with zero mean and unit variance. The training targets y are then generated by using the training inputs and adding some noise:

$$y = \sin(3X) + \epsilon_{noise}$$

The test inputs are simply 61 equally spaced points between -3 and 3. See Figure 10 for the illustration. The data is then standardized as described earlier. Two versions of optimization are then run. The first one is all the NCG versions using different β parameters, and the second one is the steepest descent method, meaning $\beta = 0 \rightarrow p_k = -\nabla f(x_k)$.

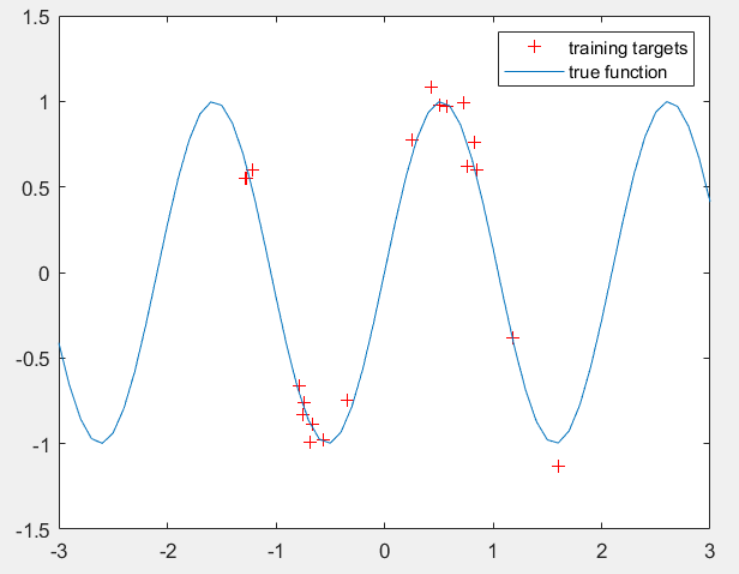


Figure 10. Example case input data.

Since the problem is only two-dimensional, it is easy to display the functionality of the algorithm graphically. The NLML forms, in this case, a drop shaped form with the optimum at around $(0.7, 1.45)$ with the lengthscale parameter on the x-axis and the noise variance on the y-axis. Note the many and small steps the SD algorithm while the NCG needs only a few. See Figure 11 for the illustration. The runtime for different choices of β parameter in the NCG algorithm are shown in Figure 12, with the clear winner being the FR-PR method.

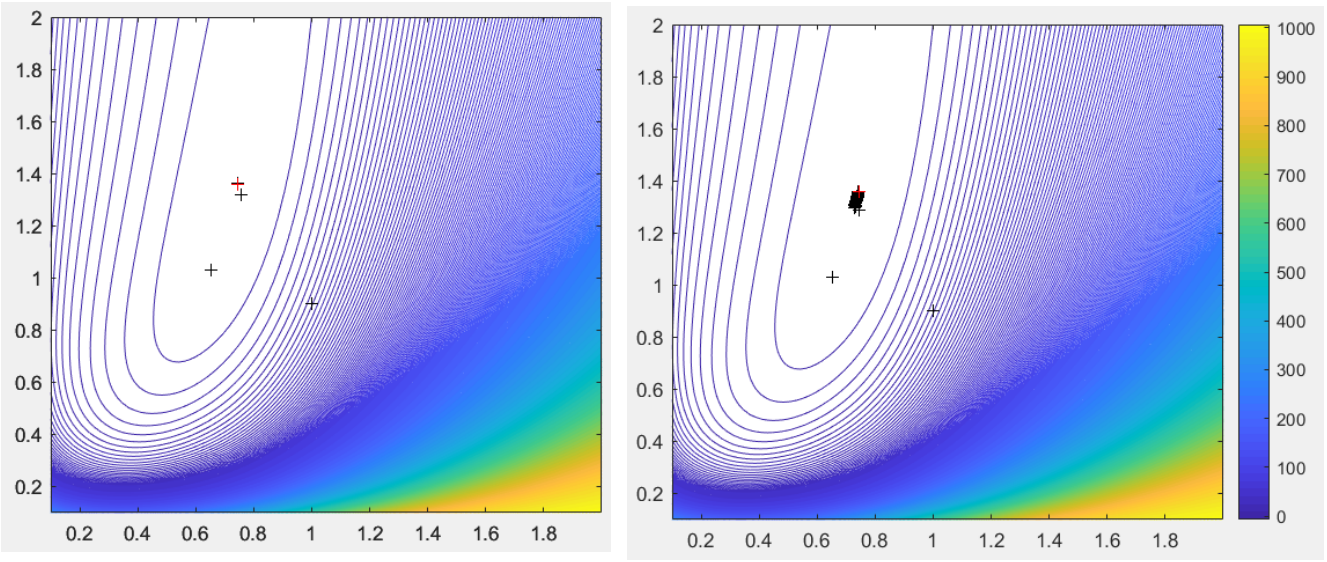


Figure 11. The contours of the NLML distribution for the test case: left – FR-PR NCG, right: Steepest Descent. The red cross is the found minimum, black crosses are intermediate steps.

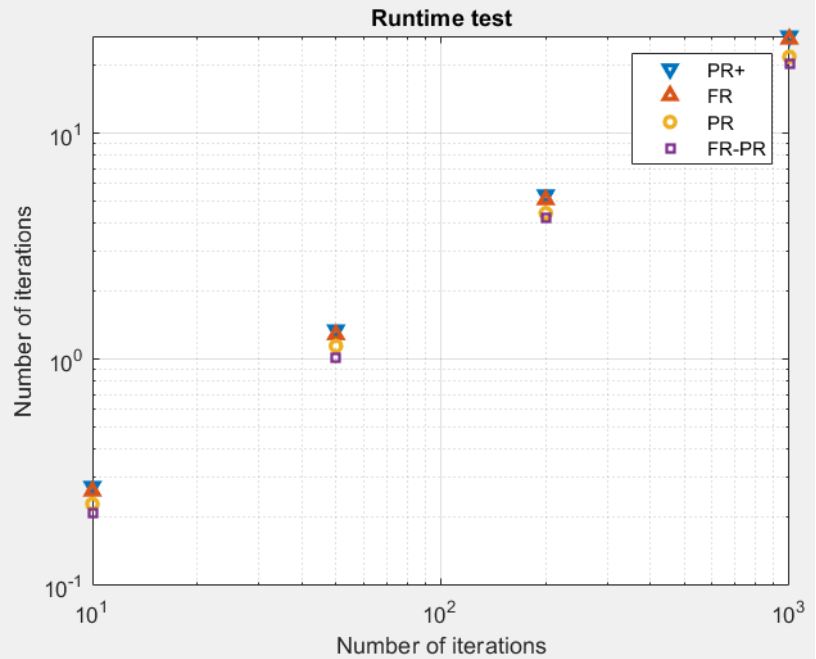


Figure 12. Runtime test on the simple example case.

3.3. Robot model test case

Physical robot manipulator arm model parameters are based on a Kuka LBR iiwa 14 R280 (GmbH) robot, Figure 13. The robot has 7 degrees of freedom (DoF) and therefore has 7 joints. The mass of the robot is assumed to be uniform, such that the masses of the respective links are obtained by simply linear combination of the different links of the robot.

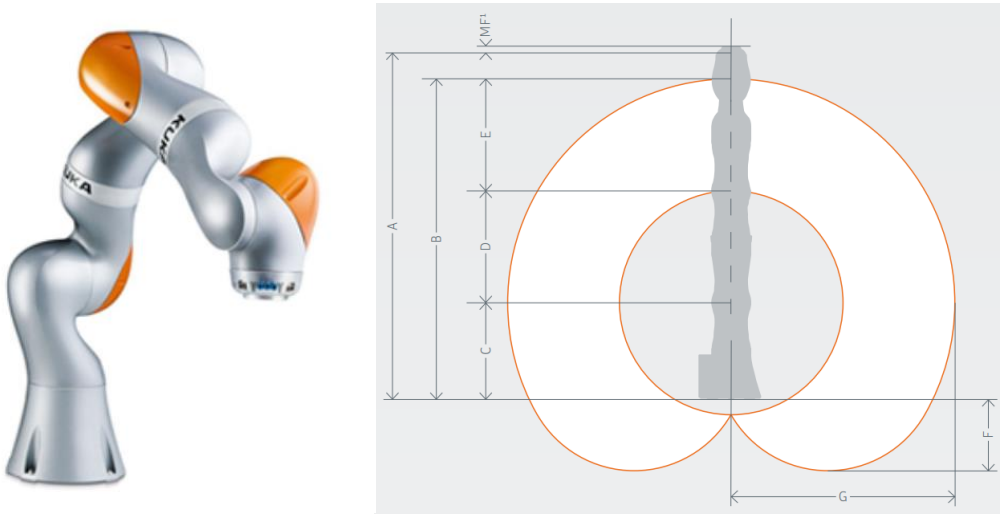


Figure 13. KUKA intelligent industrial work assistant 14 R280, illustrations from [25]

Friction coefficients in the robot joints is chosen to be simply $\begin{bmatrix} 10000 & 0 \\ 0 & 5000 \end{bmatrix}$. The behavior of the unforced system represents life-like behavior.

Model parameter	LBR iwwa 14's parameter	Value	Explanation
$\boldsymbol{l_1}$	D	420 mm	Length of the 1st link
$\boldsymbol{l_2}$	$A - C - D$	526 mm	Length of the 2nd link
$\boldsymbol{m_T}$	m_T	29,9 kg	Total mass of the robot
$\boldsymbol{m_1}$	–	9,6 kg	Mass of the 1st link
$\boldsymbol{m_2}$	–	12,0 kg	Mass of the 2nd link
$\boldsymbol{I_1}$	–	5640 $kgmm^2$	Inertia of the 1st link
$\boldsymbol{I_2}$	–		Inertia of the 2nd link
$\boldsymbol{g}$	–	$-9{,}81\ kgm/s^2$	Gravity constant

A proportional-derivative controller works fine in simulations. The explicit integration action in the controller is not necessary as the system has an internal integrator ($\ddot{\boldsymbol{q}} \rightarrow \dot{\boldsymbol{q}}$). The

controller is of the form:

$$\boldsymbol{\tau} = \boldsymbol{k}_d(-\dot{\boldsymbol{q}}) + \boldsymbol{k}_p(\boldsymbol{q}_r - \boldsymbol{q})$$

where $\boldsymbol{q}_r$ is the reference trajectory and $\boldsymbol{k}_d$ and $\boldsymbol{k}_d$ are the control constants. After a quick tuning the system performs satisfactory with the controller gains found in Table 2 below.

$\boldsymbol{k_p}$	$\begin{bmatrix} 100000 \cdot m_1 l_1 \\ 12500 \cdot m_2 l_2 \end{bmatrix}$
$\boldsymbol{k_d}$	$\begin{bmatrix} 15000 \cdot m_1 l_1 \\ 1000 \cdot m_2 l_2 \end{bmatrix}$

Table 2. PD controller gains

The actual values of the gains might not be entirely feasible in a real-world application as they might be too great. These gains yield an extremely fast convergence rate and fast error dynamics. Therefore, they perform sufficiently well based on the application of the current model. The error dynamics are shown in Figure 14.

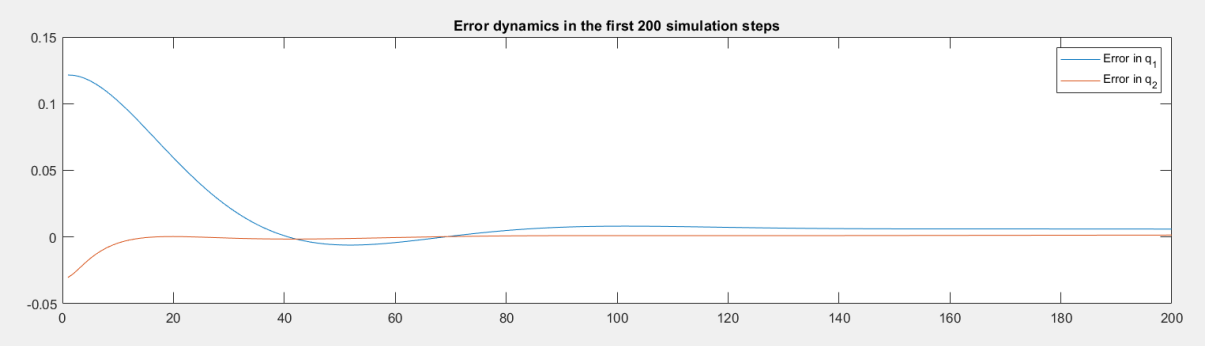


Figure 14. Error dynamics for the robot model test case.

The data generated by the robot model is six vectors of size 1×5100: the joint angles q_1, q_2, the joint angle velocities $\dot{q_1}$, $\dot{q_2}$, and the joint torques τ_1 and τ_2. The GP is then used to model the torque as a function of the joint angles and their velocities together with friction dependent on the angle velocities:

$$\tau_1 = f(q_1, q_2, \dot{q_1}, \dot{q_2}) + g(\dot{q_1})$$
$$\tau_2 = f(q_1, q_2, \dot{q_1}, \dot{q_2}) + g(\dot{q_2})$$

4. Results

4.1. Example test case

The results of the simulations are shown in the figures below. Figure 15 shows the runtime and the total steps taken before the convergence was reached. When comparing the steepest descent with the NCG, the latter wins clearly by the measure of convergence. While the resulting hyperparameters are equal, the required number of steps were unsurprisingly higher in the SD method. Figure 16 shows comparable results between the novel implementation and the results using the in-built MATLAB `fitrgp` function. The validation SMSE scores were 0.0428 and 0.0123, respectively. It is worth to note that the runtime for the in-built function was 2 times faster than the novel implementation. Figure 17 shows the samples of the posterior prediction.

```
--> Hyperparameter optimization: optimizer = NCG, kernel = SE-ARD

|=============================================================================================================|
| ITER |    FUN VAL    |                        X VAL                        |  DATA FIT  | CMPX PEN | NORM CNST |
|=============================================================================================================|
| 1    | 7.06146       | 1 0.9                                               | 24.3       | -35.6    | 18.4      |
| 2    | -8.59994      | 0.654 1.03                                          | 4.42       | -31.4    | 18.4      |
| 3    | -8.81196      | 0.756 1.32                                          | 3.89       | -31.1    | 18.4      |
| 4    | -8.84157      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 5    | -8.84157      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 6    | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 7    | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 8    | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 9    | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 10   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 11   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 12   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 13   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 14   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
        Aborted due to lack of progress
Elapsed time is 0.377664 seconds.

|=============================================================================================================|
| ITER |    FUN VAL    |                        X VAL                        |  DATA FIT  | CMPX PEN | NORM CNST |
|=============================================================================================================|
| 81   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 82   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 83   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 84   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 85   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 86   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 87   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 88   | -8.84158      | 0.743 1.36                                          | 3.48       | -30.7    | 18.4      |
| 89   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 90   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 91   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 92   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 93   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
| 94   | -8.84158      | 0.743 1.36                                          | 3.47       | -30.7    | 18.4      |
        Aborted due to lack of progress
Elapsed time is 0.631219 seconds.
```

Figure 15. Example test case MATLAB command window output. Upper: FR-PR NCG algorithm. Lower: Steepest descent

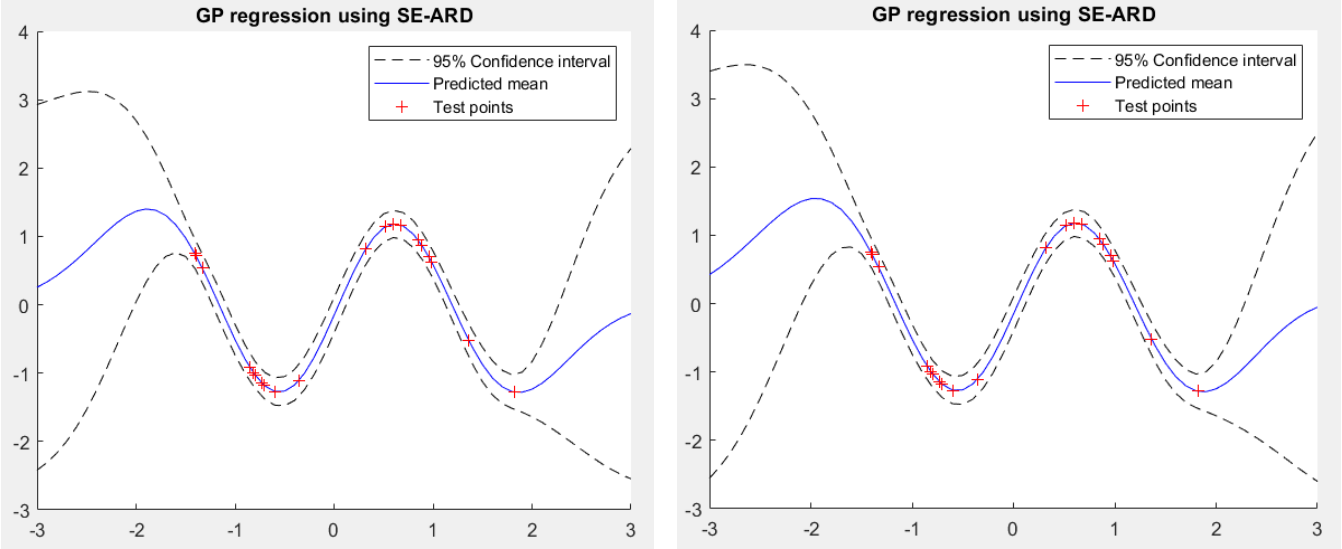


Figure 16. The resulting prediction from the GP inference on the example test case. Left – Novel implementation, right – fitrgp result.

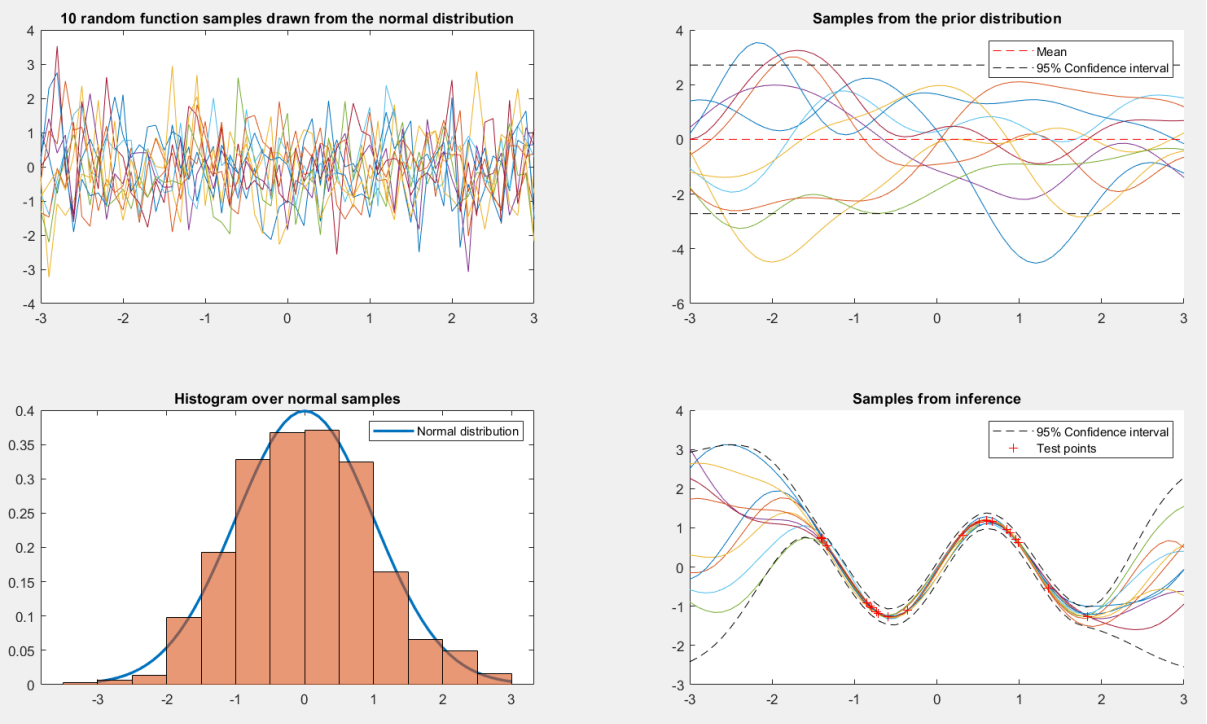


Figure 17. Example test case results. Top left – functions used in the predictions. Top right – the prior prediction. Lower left – the distribution of the functions used in the predictions. Lower right – the posterior prediction.

4.2. Robot model test case

The following figures describe the results of the simulation of the robot model test case. Surprisingly, the steepest descent method yielded a better NLML score of -94.7814 compared to the NCG method with FR-PR beta update with the s core of only -65.5756. The runtime of the latter algorithm was also shorter. This defies every intuition prior to the simulations and therefore can be an indicator that the results are faulty. Note this fact for the further analysis of the results. The resulting output windows of both methods are depicted in Figure 18.

```
--> Hyperparameter optimization: optimizer = NCG, kernel = SE-ARD
```

ITER	FUN VAL	X VAL	DATA FIT	CMPX PEN	NORM CNST
1	-54.9896	1 1 1 1 0.9	49.1	-196	91.9
2	-56.8045	1.03 1.14 1.09 0.961 0.88	52.3	-201	91.9
3	-63.7008	1.07 1.43 1.29 0.549 0.999	40.2	-196	91.9
4	-94.5159	0.842 0.989 0.864 3.25 2.71	10.1	-197	91.9
5	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
6	-94.7814	0.843 1.03 0.896 3.25 2.69	11.7	-198	91.9
7	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
8	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
9	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
10	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
11	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
12	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9
13	-94.7814	0.843 1.03 0.896 3.25 2.68	11.7	-198	91.9

Aborted due to lack of progress
Elapsed time is 4.257108 seconds.

ITER	FUN VAL	X VAL	DATA FIT	CMPX PEN	NORM CNST
1	-54.9896	1 1 1 1 0.9	49.1	-196	91.9
2	-56.8045	1.03 1.14 1.09 0.961 0.88	52.3	-201	91.9
3	-64.3609	1.13 1.75 1.5 0.416 0.976	42.8	-199	91.9
4	-64.5188	1.12 1.71 1.47 0.462 0.97	43.9	-200	91.9
5	-65.5567	1.12 1.71 1.47 0.543 0.978	46.1	-204	91.9
6	-65.5737	1.12 1.71 1.47 0.533 0.977	45.7	-203	91.9
7	-65.5755	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
8	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
9	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
10	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
11	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
12	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9
13	-65.5756	1.12 1.71 1.47 0.53 0.977	45.6	-203	91.9

Aborted due to lack of progress
Elapsed time is 5.123435 seconds.

Figure 18. MATLAB command window ouput for the robot model test case.

Looking at the NLML plots, both methods show the failure to converge. The algorithm stops due to the step size being zero, while the descent direction p_k is anything but the zero vector. Figure 19 shows the final results of the NLML distribution where x and y axes are the first and second lengthscale parameters– angles q_1 and q_2, respectively. Figure 20 shows some of the step search curves and the failure to find the correct minima.

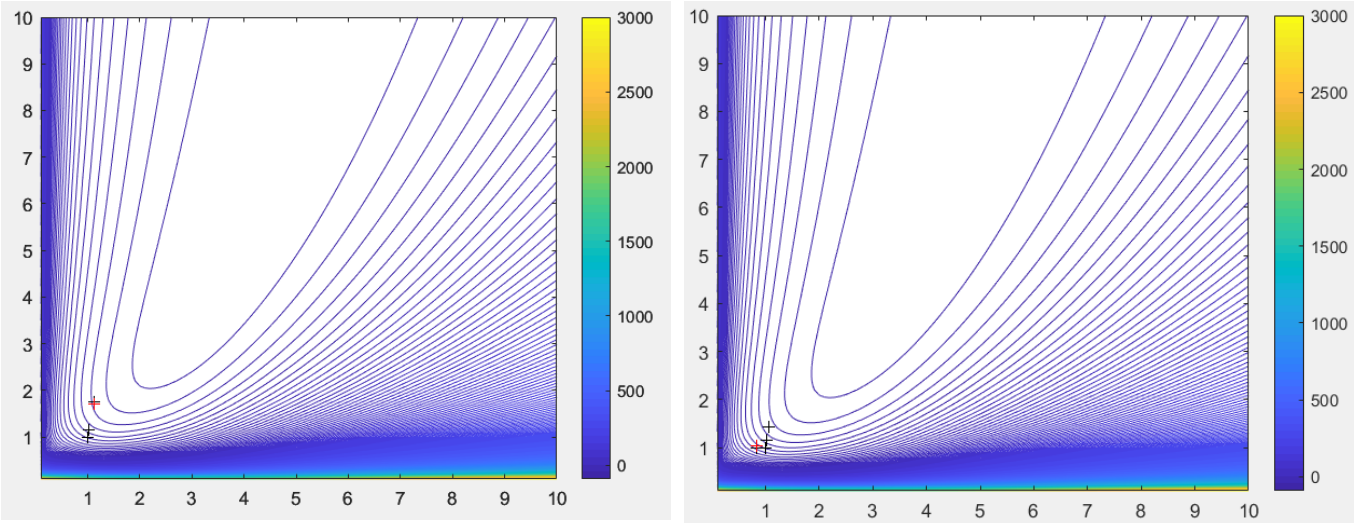


Figure 19. The NLML for the robot model test case, where left - GP using the FR-PR algorithm, right - GP using steepest descent

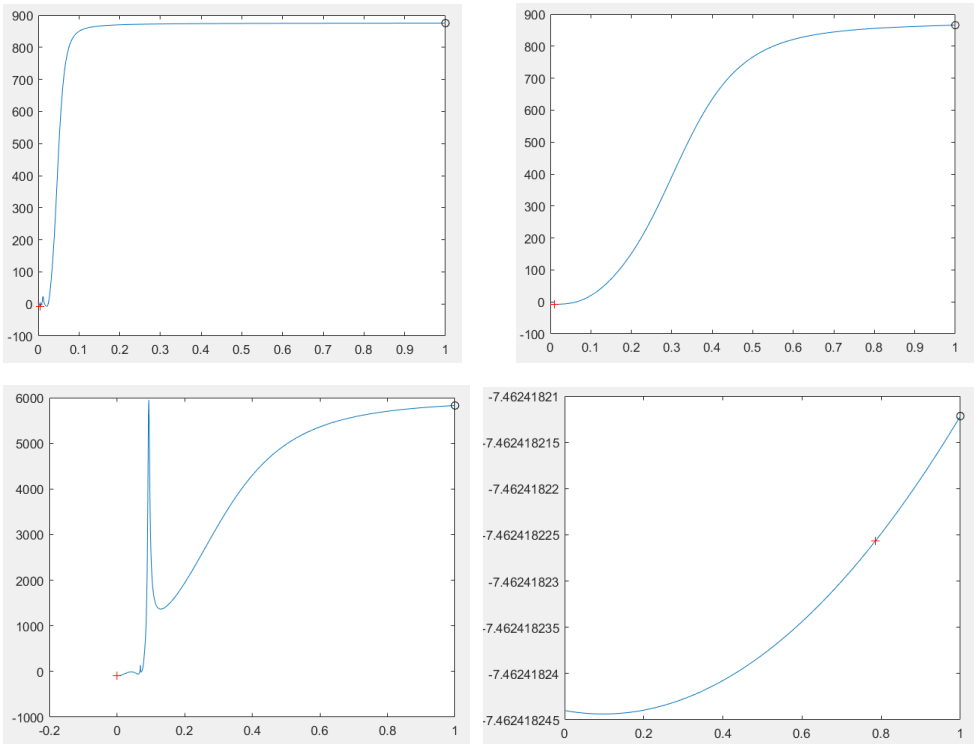


Figure 20. Graphical examples of step search iterations. Black circles meaning the starting points, and the red cross being the found minima. Notice the failure to find the extremum on graph on the lower right.

5. Discussion

The simple test case performed as expected and, when compared to the existing software, produced comparable results. The line search method is fast and find the NLML minima well.

The causes for the failure in the multi-dimensional problem are not obvious. There have been made several attempts of finding any inaccuracies between the theory and the implementation, but the problem remained. This might point to another weakness of the model – the identification trajectory. This trajectory is suspected to not excite the model sufficiently, causing the lack of correlation between the input-output pairs.

Another cause might be the assumption of the independency of the torques. The torques have been inferred separately as functions of all angles and angle velocities. Multiple output GPs have been studied in the past and yield promising results [10].

The step search graphs of the hyperparameter optimization problem, contain interesting behavior, pointing to the NLML or its derivatives being faulty. After some search steps have been found and “taken”, the step search graphs become monotonically increasing at zero in the positive step direction. This suggests that the optimum has been reached, when it clearly has not.

6. Future Work

In order to achieve more accurate predictions, one could use the gradient information at the training points. The information on whether the slope is increasing or decreasing on the test points can dramatically improve the accuracy of the predictions. This technique yields great results in [11].

When the explained data is low, one might suspect the faulty choice of the hyperparameters. While one can use the combination kernels designed as a sum or product of several other simpler kernels, the *curse of dimensionality* becomes evident [12]. With several choices of the hyperparameters, the error of choosing "wrong" hyperparameters, increases. An automated technique of choosing and discarding the increasing complexity priors, deserves further research.

Implementation of cross-validation-based hyperparameter optimizers, such as LOO-CV, sometimes prioritize the complexity and the data-fit differently than the marginal likelihood. It may therefore produce totally different hyperparameters and results, which is very interesting and deserves more research. Implementing other types of likelihoods, non-Gaussian.

Gaussian Processes have regretfully been overseen in the machine learning in the recent years. The main cause of that is the expensive computations needed for inference and hyperparameter learning for larger data sets. That makes GP a poor choice for on-line control tasks. The costs of both inference and learning tend to scale badly with the matrix inversion of an $n \times n$ matrix alone costing $O(n^3)$ time. There have been proposed some remedies in [13].

Almost all following methods use sparse matrices and approximates that greatly speed up the inference and learning processes. One idea is to approximate non-parametric kernels in "dual space": Random Kitchen Sinks [14], Fastfood [15], À la carte [16]. Another technique is using inducing points methods, where the cost for predictions can be reduced to $O(m^2 n)$, where $m \ll n$. For instance, replacing the exact covariance function with an approximation using Subset of Regression (SoR), allows faster computations with the cost of underestimating the uncertainty. Another example is Fully Independent Training Conditional (FITC) which replaces the approximate diagonal of SoR with the diagonal of the true covariance function. [17] introduced Kernel Interpolation for Scalable Structured Gaussian Processes, or KISS-GP, which claims to achieve staggering $O(n)$ for both time and storage costs for GP inference. This method exploits the structure in the covariance function and use Toeplitz and Kronecker algebra, as in the works with Massively Scalable GP in [18].

7. References

[1] O. Egeland and J. T. Gravdahl, Modeling and simulation for automatic control, Trondheim: Marine Cybernetics, 2002, pp. 315, 320, 322, 323.

[2] B. Armstrong, "On Finding Exciting Trajectories for Identification Experiments Involving Systems with Nonlinear Dynamics," 1 December 1989. [Online]. Available: https://journals.sagepub.com/doi/10.1177/027836498900800603.

[3] C. K. I. W. C. E. Rasmussen, Gaussian Processes for Machine Learning, Massachusetts: the MIT press, 2006, pp. 13, 106, 114,.

[4] S. Polamuri, "Five most popular similarity measures, implementation in python," 11 April 2015. [Online]. Available: http://dataaspirant.com/2015/04/11/five-most-popular-similarity-measures-implementation-in-python/.

[5] J. Nocedal and S. J. Wright, Numerical Optimization, 2nd ed., T. V. Mikosch, S. I. Resnick and S. M. Robinson, Eds., Evanston: Springer, 2006.

[6] G. Wahba, "Spline Models for Observational Data," 1990. [Online]. Available: https://epubs.siam.org/doi/book/10.1137/1.9781611970128?mobileUi=0.

[7] J. R. Shewchuk, "An Introduction to," 4 August 1994. [Online]. Available: https://www.cs.cmu.edu/~quake-papers/painless-conjugate-gradient.pdf.

[8] I. Murray, "Introduction to Gaussian Processes, page 34," 2008. [Online]. Available: https://www.cs.toronto.edu/~hinton/csc2515/notes/gp_slides_fall08.pdf.

[9] C. E. Rasmussen and C. Williams, "Documentation for GPML Matlab Code version 4.2," 11 Novermber 2018. [Online]. Available: http://www.gaussianprocess.org/gpml/code/matlab/doc/.

[10] A. M. Álvarez, "Multiple-output Gaussian processes," The University of Sheffield, [Online]. Available: http://gpss.cc/gpss17/slides/multipleOutputGPs.pdf.

[11] E. Solak, R. Murray-Smith, W. E. Leithead, D. J. Leith and C. E. Rasmussen, "Derivative Observations in Gaussian Process Models of Dynamic Systems," 2003. [Online]. Available: https://papers.nips.cc/paper/2287-derivative-observations-in-gaussian-process-models-of-dynamic-systems.

[12] B. Shetty, "Curse of Dimensionality," 16 January 2019. [Online]. Available: https://towardsdatascience.com/curse-of-dimensionality-2092410f3d27.

[13] A. G. Wilson, "Probabilistic Graphical Models," Carnegie Mellon University, 1 April 2015. [Online]. Available: https://www.cs.cmu.edu/~epxing/Class/10708-15/slides/andrewgp2.pdf.

[14] A. Rahimi, "Random Kitchen Sinks: Replacing Optimization with Randomization in Learning," Intel Labs Berkeley, 24 July 2009. [Online].

[15] Q. V. Le, T. Sarlos and A. J. Smola, "Fastfood: Approximate Kernel Expansions in Loglinear Time," Cornell University, 13 August 2014. [Online]. Available: https://arxiv.org/abs/1408.3060.

[16] Z. Yang, A. J. Smola, L. Song and A. G. Wilson, "A la Carte - Learning Fast Kernels," Cornell University, 19 December 2014. [Online]. Available: https://arxiv.org/abs/1412.6493.

[17] A. G. Wilson and H. Nickisch, "Kernel Interpolation for Scalable Structured Gaussian Processes (KISS-GP)," Cornell University, 3 March 2015. [Online]. Available:

https://arxiv.org/abs/1503.01057.

[18] A. G. Wilson, C. Dann and H. Nickisch, "Thoughts on Massively Scalable Gaussian Processes," Cornell University, 5 November 2015. [Online]. Available: https://arxiv.org/abs/1511.01870.

[19] R. N. Jazar, Theory of Applied Robotics. Kinematics, Dynamics and Control, 2nd ed., Springer US, 2010, p. 883.

[20] D. R. Hessmer, "Kinematics for Lynxmotion Robot Arm," October 2009. [Online].

[21] "Gaussian Process - Wikipedia," [Online]. Available: https://en.wikipedia.org/wiki/Gaussian_process.

[22] N. I. o. S. a. Technology, "Engineering Statistics Handbook," [Online]. Available: https://www.itl.nist.gov/div898/handbook/pmc/section4/pmc442.htm.

[23] N. A. C. Cressie, Statistics for Spatial Data, Wiley, 2015.

[24] J. K. P. Y. S. K. M. S. K. F. C. Samy Youssef, "Probabilistic Selection of Ship-Ship Collision Scenarios," ASME 2013, June 2013. [Online]. Available: https://www.researchgate.net/publication/267607095_Probabilistic_Selection_of_Ship-Ship_Collision_Scenarios.

[25] K. R. GmbH, "KUKA LBR iiwa Broschure - PDF, page 30," [Online]. Available: https://www.kuka.com/en-de/services/downloads?terms=Language:en:1;product_name:LBR%20iiwa%2014%20R820.

[26] C. W. Carl Edward Rasmussen, "Documentation for GPML Matlab Code version 4.2," 11 June 2018. [Online]. Available: http://www.gaussianprocess.org/gpml/code/matlab/doc/.

[27] J. S.-T. A. E. J. K. Nello Cristianini. [Online]. Available: http://papers.nips.cc/paper/1946-on-kernel-target-alignment.pdf.

[28] L. C. M. A. K. H. H. B. L. R. F. Jacob Y. Hesterman, "Maximum-Likelihood Estimation With a Contracting-Grid Search Algorithm," 14 June 2010.

[29] I. L. Andrew V. Knyazev, "Steepest descent and conjugate gradient methods," April 2007. [Online]. Available: https://arxiv.org/pdf/math/0605767.pdf.

[30] Y. B. James Bergstra, "Random serach for hyper-parameter optimization," *Journal of Machine Learning Research 13,* February 2012.

[31] H. L. R. P. A. Jasper Snoek, "Practical Bayesian Optimization of Machine Learning Algorithms," 2012.

[32] G. S. L. Cai, "A smooth robust nonlinear controller for robot manipulators with joint stick-slip friction," Atlanta, 2002.

[33] A. C. Harvey Motulsky, Fitting Models to Biological Data Using Linear and Nonlinear Regression, New York: OXFORD University Press, 2004.

[34] A. G. d. G. M. Z. G. James Hensman, "Scalable Variational Gaussian Process Classification," 2015.

[35] R. M.-S. C. E. R. A. G. J. Kocijan, "Gaussian process model based predictive control," Boston, 2004.

[36] N. d. Freitas, "CPSC540 Gaussian Process," January 2013. [Online]. Available: https://www.cs.ubc.ca/~nando/540-2013/lectures/l6.pdf.

[37] N. Wiener, Extrapolation, Interpolation and Smoothing of Stationary Time Series, MIT Press, 1949.

[38] A. Hopgood, "Artificial Intelligence: hype or reality?," 2003.

[39] N. M. G. Cressie, The origins of kriging, vol. 22: 239, Kluwer Academic Publishers, 1990.

[40] Wikipedia, "Bregman divergence," [Online]. Available: https://en.wikipedia.org/wiki/Bregman_divergence.

[41] H. Gattringer, R. Riepl and M. Neubauer, Artists, *End-effector path during identification process.* [Art]. Hindawi, 2013.

[42] M. M. Zirkohi, "An articulated two-link manipulator," 2013. [Online]. Available: https://www.researchgate.net/publication/257705396_Type-2_Fuzzy_Control_for_a_Flexible-_joint_Robot_Using_Voltage_Control_Strategy.

8. Appendix A

8.1. Derivation of the EOM for 2-link planar manipulator

The derivations are based on [1]. The planar manipulator has kinetic energy:

$$T = \frac{1}{2} m_1 \vec{v}_{c1} \cdot \vec{v}_{c1} + \frac{1}{2} m_2 \vec{v}_{c2} \cdot \vec{v}_{c2} + \frac{1}{2} \vec{w}_1 \cdot \vec{M}_{\frac{1}{c}} \cdot \vec{w}_1 + \frac{1}{2} \vec{w}_2 \cdot \vec{M}_{\frac{2}{c}} \cdot \vec{w}_2$$

This can be written as

$$T = \frac{1}{2} \boldsymbol{m_{11}} \dot{q}_1^2 + \boldsymbol{m_{12}} \dot{q}_1 \dot{q}_2 + \frac{1}{2} \boldsymbol{m_{22}} \dot{q}_2^2$$

where

$$\boldsymbol{m_{11}} = I_{1z} + I_{2z} + m_1 L_{c1}^2 + m_2 (L_1^2 + L_{c2}^2 + 2 L_1 L_{c2} \cos q_2)$$
$$\boldsymbol{m_{12}} = \boldsymbol{m_{21}} = I_{2z} + m_2 L_{c2}^2 + m_2 L_1 L_{c2} \cos q_2$$
$$\boldsymbol{m_{22}} = I_{2z} + m_2 L_{c2}^2$$

are the elements of the inertia matrix and q_1 and q_2 are the angles between the horizontal plane and the robot arm 1 and 2 respectively. The potential energy in the system is given by

$$V = (m_1 g L_{c1} + m_2 g L_1) \sin q_1 + m_2 g L_{c2} \sin(q_1 + q_2)$$

Then, from $\mathcal{L} = T - V$ partial derivatives are found to be

$$\frac{\partial \mathcal{L}}{\partial \dot{q}_1} = \frac{\partial T}{\partial \dot{q}_1} = \boldsymbol{m_{11}} \dot{q}_1 + \boldsymbol{m_{12}} \dot{q}_2$$
$$\frac{\partial \mathcal{L}}{\partial \dot{q}_2} = \frac{\partial T}{\partial \dot{q}_2} = \boldsymbol{m_{12}} \dot{q}_1 + \boldsymbol{m_{22}} \dot{q}_2$$
$$\frac{\partial \mathcal{L}}{\partial q_1} = -\frac{\partial V}{\partial q_1} = -(m_1 L_{c1} + m_2 L_1) g \cos q_1 - m_2 L_{c2} g \cos(q_1 + q_2)$$
$$\frac{\partial \mathcal{L}}{\partial q_2} = \frac{\partial T}{\partial q_2} - \frac{\partial V}{\partial q_2} = \frac{1}{2} \frac{\partial \boldsymbol{m_{11}}}{\partial q_2} \dot{q}_1^2 + \frac{\partial \boldsymbol{m_{12}}}{\partial q_2} \dot{q}_1 \dot{q}_2 - m_2 L_{c2} g \cos(q_1 + q_2)$$

Recalling the chain rule expansion:

$$\frac{dz}{dt} = \frac{\partial z}{\partial x}\frac{\partial x}{\partial t} + \frac{\partial z}{\partial y}\frac{\partial y}{\partial t}$$

the equations of motion are then found to be:

$$\tau_1 = \boldsymbol{m_{11}}\ddot{q}_1 + \boldsymbol{m_{12}}\ddot{q}_2 + \left(\frac{\partial \boldsymbol{m_{11}}}{\partial q_2}\dot{q}_2\right)\dot{q}_1 + \left(\frac{\partial \boldsymbol{m_{12}}}{\partial q_2}\dot{q}_2\right)\dot{q}_2 + \frac{\partial V}{\partial q_1}$$

$$\tau_2 = \boldsymbol{m_{21}}\ddot{q}_1 + \boldsymbol{m_{22}}\ddot{q}_2 + \left(\frac{\partial \boldsymbol{m_{21}}}{\partial q_2}\dot{q}_2\right)\dot{q}_1 - \left(\frac{\boldsymbol{\partial m_{21}}}{\boldsymbol{\partial q_2}}\dot{q}_1\right)\dot{q}_2 - \frac{1}{2}\left(\frac{\partial \boldsymbol{m_{11}}}{\partial q_2}\right)\dot{q}_1^2 + \frac{\partial V}{\partial q_2}$$

$$\begin{aligned}\tau_1 = {} & \left(I_1 + I_2 + m_1 L_{c1}^2 + m_2(L_1^2 + L_{c2}^2 + 2L_1 L_{c2}\cos q_2)\right)\ddot{q}_1 \\ & + (I_2 + m_2 L_{c2}^2 + m_2 L_1 L_{c2}\cos q_2)\ddot{q}_2 - m_2 L_1 L_{c2}\sin q_2\,(2\dot{q}_1\dot{q}_2 + \dot{q}_2^2) \\ & + (m_1 L_{c1} + m_2 L_1)g\cos q_1 + m_2 L_{c2} g\cos(q_1 + q_2) \\ \tau_2 = {} & (I_2 + m_2 L_{c2}^2 + m_2 L_1 L_{c2}\cos q_2)\ddot{q}_1 + (I_2 + m_2 L_{c2}^2)\ddot{q}_2 + (m_2 L_1 L_{c2}\sin q_2)\dot{q}_1^2 \\ & + m_2 L_{c2} g\cos(q_1 + q_2)\end{aligned}$$

These equations can be written in a simplified form:

$$\boldsymbol{M}(\boldsymbol{q})\ddot{\boldsymbol{q}} + \boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) + \boldsymbol{G}(\boldsymbol{q}) = \boldsymbol{\tau}$$

where $\boldsymbol{M}(\boldsymbol{q}) = \boldsymbol{M}^{\mathsf{T}}(\boldsymbol{q})$ is a positive definite matrix of masses $\begin{bmatrix} m_{11} & m_{12} \\ m_{21} & m_{22} \end{bmatrix}$ with elements

$$\begin{aligned} m_{11} &= I_1 + I_2 + m_1 L_{c1}^2 + m_2(L_1^2 + L_{c2}^2 + 2L_1 L_{c2}\cos q_2) \\ m_{12} &= m_{21} = I_2 + m_2(L_{c2}^2 + L_1 L_{c2}\cos q_2) \\ m_{22} &= I_2 + m_2 L_{c2}^2 \end{aligned}$$

$\boldsymbol{G}(\boldsymbol{q})$ is the gradient of the gravity potential $\begin{bmatrix} g_1 \\ g_2 \end{bmatrix}$ with elements

$$\begin{aligned} g_1 &= g\left((m_1 L_{c1} + m_2 L_1)\cos q_1 + m_2 L_{c2}\cos(q_1 + q_2)\right) \\ g_2 &= g m_2 L_{c2}\cos(q_1 + q_2) \end{aligned}$$

The matrix $\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}})$ can be selected to be

$$\boldsymbol{C}(\boldsymbol{q}, \dot{\boldsymbol{q}}) = \{\boldsymbol{c}_{jk}\} = \left\{\sum_{i=1}^{n} \boldsymbol{c}_{ijk}\dot{\boldsymbol{q}}_i\right\}$$

where

$$\boldsymbol{c}_{ijk} \coloneqq \frac{\mathbf{1}}{\mathbf{2}}\left(\frac{\partial \boldsymbol{m}_{jk}}{\partial \boldsymbol{q}_{\boldsymbol{i}}} + \frac{\partial \boldsymbol{m}_{ik}}{\partial \boldsymbol{q}_{j}} + \frac{\partial \boldsymbol{m}_{ij}}{\partial \boldsymbol{q}_{k}}\right)$$

are *the Christoffel* symbols of the first kind. Define $\dot{q_1} = \omega_1$ and $\dot{q_2} = \omega_2$ and express Christoffel symbols as

$$c_{11} = -m_2 L_1 L_{c2} \omega_2 \sin q_2$$
$$c_{12} = (-m_2 L_1 L_{c2} \sin q_2)(\omega_1 + \omega_2)$$
$$c_{21} = m_2 L_1 L_{c2} \omega_1 \sin q_2$$
$$c_{22} = 0$$

■

8.2. Inverse kinematics for 2-link planar manipulator

The derivation of these formulas is borrowed from [19] and [20]. Start with forward kinematics formulas

$$\begin{pmatrix} x_e \\ y_e \end{pmatrix} = \begin{pmatrix} l_1 \cos q_1 + l_2 \cos(q_1 + q_2) \\ l_1 \sin q_1 + l_2 \sin(q_1 + q_2) \end{pmatrix} \quad (1^*)$$

where all variables are according Figure 1 on page 3. Rewrite squares of the end effector position

$$\begin{pmatrix} x_e^2 \\ y_e^2 \end{pmatrix} = \begin{pmatrix} l_1^2 \cos^2 q_1 + l_2^2 \cos^2(q_1 + q_2) + 2 l_1 l_2 \cos q_1 \cos(q_1 + q_2) \\ l_1^2 \sin^2 q_1 + l_2^2 \sin^2(q_1 + q_2) + 2 l_1 l_2 \sin q_1 \sin(q_1 + q_2) \end{pmatrix}$$

Use Pythagorean identity

$$a\sin^2 \theta + a\cos^2 \theta = a^2$$

and rewrite

$$x_e^2 + y_e^2 = l_1^2 + l_2^2 + 2 l_1 l_2 [\cos q_1 \cos(q_1 + q_2) + \sin q_1 \sin(q_1 + q_2)]$$

Use the following identities

$$\sin(a \pm b) = \sin a \cos b \pm \cos a \sin b$$
$$\cos(a \pm b) = \cos a \cos b \mp \sin a \sin b$$

to prove

$$\begin{aligned} x_e^2 + y_e^2 &= l_1^2 + l_2^2 \\ &\quad + 2l_1 l_2[\cos q_1 (\cos q_1 \cos q_2 - \sin q_1 \sin q_2) \\ &\quad + \sin q_1 (\sin q_1 \cos q_2 + \cos q_1 \sin q_2)] \\ &= l_1^2 + l_2^2 + 2l_1 l_2[\cos^2 q_1 \cos q_2 + \sin^2 q_1 \cos q_2] \\ &= l_1^2 + l_2^2 + 2l_1 l_2 \cos q_2 \end{aligned}$$

From this follow

$$q_2 = \cos^{-1} \frac{x_e^2 + y_e^2 - l_1^2 - l_2^2}{2l_1 l_2}$$

Since arcsin and arccos are inaccurate for small angles, use the atan2 function:

$$\begin{aligned} q_2 &= \text{atan2}(\sin q_2 , \cos q_2) \\ &= \text{atan2}\left(\pm\sqrt{1 - \cos^2 q_2} , \cos q_2\right) \\ &= \text{atan2}\left(\pm\sqrt{1 - \left(\frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1 l_2}\right)^2} , \frac{x^2 + y^2 - l_1^2 - l_2^2}{2l_1 l_2}\right) \end{aligned}$$

Next, use (1^*) to rewrite x and y:

$$\begin{aligned} x &= k_1 \cos q_1 - k_2 \sin q_1 \\ y &= k_1 \sin q_1 + k_2 \cos q_1 \end{aligned}$$

where $k_1 = l_1 + l_2 \cos q_2$ and $k_2 = l_2 \sin q_2$.

Now, use the following:

$$\begin{aligned} r &= \sqrt{k_1^2 + k_2^2} \\ \gamma &= \text{atan2}(k_2, k_1) \end{aligned}$$

This gives

$$\begin{aligned} k_1 &= r \cos \gamma \\ k_2 &= r \sin \gamma \end{aligned}$$

which leads to

$$x = r\cos(\gamma + q_1)$$
$$y = r\sin(\gamma + q_1)$$

Finally apply atan2 function to find q_1:

$$\gamma + q_1 \text{atan2}\left(\frac{y}{r}, \frac{y}{r}\right) = \text{atan2}(y, x)$$
$$q_1 = \text{atan2}(y, x) - \text{atan2}(k_2, k_1)$$

where $k_1 = l_1 + l_2\cos q_2$ and $k_2 = l_2 \sin q_2$.

The final solution contains $\pm$ sign which can be thought of as *elbow-up* and *elbow-down* solutions. Plus-sign yields the *elbow-up* solution, while the minus yields the opposite.

$$q(x_e, y_e) = \begin{bmatrix} \text{atan2}(y_e, x_e) \pm \text{atan}\,2\left(l_2 \sin q_2\,,\ (l_1 + l_2\cos q_2)\right) \\ \text{atan2}\left(\pm\sqrt{1 - \left(\frac{x_e^2 + y_e^2 - l_1^2 - l_2^2}{2l_1 l_2}\right)^2}\,, \frac{x_e^2 + y_e^2 - l_1^2 - l_2^2}{2l_1 l_2}\right) \end{bmatrix}$$

■

8.3. Finite difference

Finite difference is a numerical method of approximating derivatives, using the differences of known function values. There exist mainly three types differences:

- Forward difference: $f'(x) = \lim_{h\to 0} \frac{f(x+h) - f(x)}{h}$ with the error estimate $O(h)$
- Backward difference: $f'(x) = \lim_{h\to 0} \frac{f(x) - f(x-h)}{h}$ with the error estimate $O(h)$
- Central difference: $f'(x) = \lim_{h\to 0} \frac{f\left(x+\frac{1}{2}h\right) - f\left(x-\frac{1}{2}h\right)}{h}$ with the error estimate $O(h^2)$

where h is a "small" constant.

For multi-dimensional functions, the forward difference is:

$$f_{x_i}(\boldsymbol{x}) \approx \frac{f(x_i + h, \boldsymbol{x}\backslash x_i) - f(\boldsymbol{x})}{h}, \text{ where } \boldsymbol{x} \in \{x_1, x_2, \ldots, x_i\}$$

Forward and backward difference yield near identical results and are cheapest to compute, whereas the central difference is the approximation that yields the most accurate result at the cost of double the computing power of one of the other methods.

8.4. Partial kernel derivatives

The use of the partial derivatives of kernels is the minimization of the negative log marginal likelihood function using a gradient based method. Given an arbitrary kernel function of the form $k(x_i, x_j|\theta) = k(x_i, x_j|\sigma_f, l)$, find the first order partial derivatives with respect to θ or (σ_f, l). All the derivations below start with the partial derivatives with respect to the noise-free signal constant σ_f, following the partial derivatives with respect to the characteristic length-scale l. The derivations where x has more than 1 dimension, l_k will be used, where k is the dimension of x the kernel is being differentiated along.

Squared Exponential

The most common and widely used kernel

$$\frac{\partial k_{SE}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 \exp\overbrace{\left[-\frac{1}{2}\frac{(x_i - x_j)^2}{l^2}\right]}^{f(\boldsymbol{x},\boldsymbol{l})}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 f(\boldsymbol{x},\boldsymbol{l})}{\partial \sigma_f} = 2\sigma_f f(\boldsymbol{x},\boldsymbol{l}) =$$

$$= 2\sigma_f \exp\left[-\frac{1}{2}\frac{(x_i - x_j)^2}{l^2}\right] = \underline{\underline{\frac{2}{\sigma_f} k_{SE}}}$$

$$\frac{\partial k_{SE}}{\partial l} = \frac{\partial \sigma_f^2 \exp\overbrace{\left[-\frac{1}{2}\frac{(x_i - x_j)^2}{l^2}\right]}^{f(\boldsymbol{x},l)}}{\partial l} = \sigma_f^2 \frac{\partial \exp\big(f(\boldsymbol{x},l)\big)}{\partial l} =$$

$$= \sigma_f^2 \exp\big(f(\boldsymbol{x},l)\big)\frac{\partial f(\boldsymbol{x},l)}{\partial l_k} = \underbrace{\sigma_f^2 \exp\big(f(\boldsymbol{x},l)\big)}_{k_{SE}}\left(-\frac{1}{2}\cdot -2\frac{(x_i - x_j)^2}{l^3}\right) =$$

$$= \underline{\underline{\frac{(x_i - x_j)^2}{l^3} k_{SE}}}$$

■

Squared Exponential with ARD

SE kernel with the property of automatically choosing the relevant features in the inference, called automatic relevance determination (ARD). The derivatives are nearly identical to the SE kernel and follow the same procedure.

$$\frac{\partial k_{SE-ard}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 \overbrace{\exp\left[-\frac{1}{2}\sum_{m=1}^{d}\frac{\left(x_{im}-x_{jm}\right)^2}{l_m^2}\right]}^{f(\boldsymbol{x},\boldsymbol{l})}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 f(\boldsymbol{x},\boldsymbol{l})}{\partial \sigma_f} = 2\sigma_f f(\boldsymbol{x},\boldsymbol{l}) =$$

$$= \frac{2}{\sigma_f}\underbrace{\sigma_f^2 \exp\left[-\frac{1}{2}\sum_{m=1}^{d}\frac{\left(x_{im}-x_{jm}\right)^2}{l_m^2}\right]}_{k_{SE-ard}} = \underline{\underline{\frac{2}{\sigma_f}k_{SE-ard}}}$$

$$\frac{\partial k_{SE-ard}}{\partial l_k} = \frac{\partial \sigma_f^2 \overbrace{\exp\left[-\frac{1}{2}\sum_{m=1}^{d}\frac{\left(x_{im}-x_{jm}\right)^2}{l_m^2}\right]}^{f(\boldsymbol{x},\boldsymbol{l})}}{\partial l_k} = \sigma_f^2 \frac{\partial \exp\big(f(\boldsymbol{x},\boldsymbol{l})\big)}{\partial l_k} =$$

$$= \sigma_f^2 \exp\big(f(\boldsymbol{x},\boldsymbol{l})\big)\frac{\partial f(\boldsymbol{x},\boldsymbol{l})}{\partial l_k} = \underbrace{\sigma_f^2 \exp\big(f(\boldsymbol{x},\boldsymbol{l})\big)}_{k_{SE-ard}}\left(-\frac{1}{2}\cdot -2\frac{\left(x_{ik}-x_{jk}\right)^2}{l_k^3}\right) =$$

$$= \underline{\underline{\frac{\left(x_{ik}-x_{jk}\right)^2}{l_k^3}k_{SE-ard}}}$$

The last derivative can also be derived using the exponent rule and differentiating every term. The finals results of both methods yield identical results.

■

Matérn 3/2

Matérn class functions have the following expression:

$$k_{Matern} = \frac{2^{(1-\nu)}}{\Gamma(\nu)}\left(\frac{\sqrt{2\nu}\left(x_i-x_j\right)}{l}\right)^{\nu} K_\nu\left(\frac{\sqrt{2\nu}\left(x_i-x_j\right)}{l}\right)$$

where two prominent cases are obtained by setting $\nu = \frac{3}{2}$ and $\frac{5}{2}$. The behavior differs for

different ν's. The kernel is equal to the squared exponential for $\nu = \infty$. The kernel is differentiable k times for $\nu > k$.

$$\frac{\partial k_{M32}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 \left(1 + \frac{\sqrt{3}(x_i - x_j)}{l}\right) \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right)}{\partial \sigma_f} =$$

$$= 2\sigma_f \left(1 + \frac{\sqrt{3}(x_i - x_j)}{l}\right) \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right) = \underline{\underline{\frac{2}{\sigma_f} k_{M32}}}$$

$$\frac{\partial k_{M32}}{\partial l} = \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right) \frac{\partial \sigma_f^2 \left(1 + \frac{\sqrt{3}(x_i - x_j)}{l}\right)}{\partial l}$$

$$+ \sigma_f^2 \left(1 + \frac{\sqrt{3}(x_i - x_j)}{l}\right) \frac{\partial \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right)}{\partial l} =$$

$$= \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right)\left(-\sigma_f^2 \frac{\sqrt{3}(x_i - x_j)}{l^2}\right)$$

$$+ \underbrace{\sigma_f^2 \left(1 + \frac{\sqrt{3}(x_i - x_j)}{l}\right) \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right)}_{k_{M32}} \left(\frac{\sqrt{3}(x_i - x_j)}{l^2}\right) =$$

$$= \underline{\underline{\frac{\sqrt{3(x_i - x_j)}}{l^2} k_{M32} \left[1 - \frac{l}{l + \sqrt{3(x_i - x_j)}}\right]}}$$

Alternative version, which can be obtained by keeping the exponent, does not involve the kernel itself:

$$\frac{\partial k_{M32}}{\partial l} = \sigma_f^2 \exp\left(-\frac{\sqrt{3}(x_i - x_j)}{l}\right) \left[\frac{l^2 + (l + 2)\sqrt{3}(x_i - x_j)}{l^2}\right]$$

■

Matern 5/2

$$\frac{\partial k_{M52}}{\partial \sigma_f} = \frac{\partial \sigma_f^2 \left(1 + \frac{\sqrt{5}(x_i - x_j)}{l} + \frac{5(x_i - x_j)^2}{3l^2}\right) \exp\left(-\frac{\sqrt{5}(x_i - x_j)}{l}\right)}{\partial \sigma_f} =$$

$$= \underline{\underline{\frac{2}{\sigma_f} k_{M52}}}$$

$$\frac{\partial k_{M52}}{\partial l} = \sigma_f^2 \exp\left(-\frac{\sqrt{5}(x_i - x_j)}{l}\right)\left(-\frac{\sqrt{5}(x_i - x_j)}{l^2} - \frac{10(x_i - x_j)^2}{3l^3}\right)$$

$$+ \sigma_f^2 \left(1 + \frac{\sqrt{5}(x_i - x_j)}{l} + \frac{5(x_i - x_j)^2}{3l^2}\right) \exp\left(-\frac{\sqrt{5}(x_i - x_j)}{l}\right)\left(\frac{\sqrt{5}(x_i - x_j)}{l^2}\right)$$

$$= \sigma_f^2 \exp\left(-\frac{\sqrt{5}(x_i - x_j)}{l}\right)\left[-\frac{10(x_i - x_j)^2}{3l^3} + \frac{5(x_i - x_j)}{l^3} + \frac{5\sqrt{5}(x_i - x_j)^3}{3l^4}\right]$$

$$= \underline{\underline{\sigma_f^2 \frac{5(x_i - x_j)^2}{3l^3}\left(\frac{\sqrt{5}(x_i - x_j)}{l} + 1\right) \exp\left(-\frac{\sqrt{5}(x_i - x_j)}{l}\right)}}$$

■

There exists a solution involving k_{M52} just like for k_{M32}:

$$\frac{\partial k_{M52}}{\partial l} = k_{M52}\left[-\frac{10(x_i - x_j)^2}{3l^3} \cdot \frac{1}{1 + \frac{\sqrt{5}(x_i - x_j)}{l} + \frac{5(x_i - x_j)^2}{3l^2}}\right]$$

$$= \underline{\underline{k_{M52}\left(-\frac{10(x_i - x_j)^2}{3l^3 + 3\sqrt{5}(x_i - x_j)l^2 + 5(x_i - x_j)^2 l}\right)}}$$

■

Periodic

Periodic kernels are useful when the behavior of the system is known to contain oscillations. Parameter p determines the period of the oscillations.

$$k_{Per} = \sigma_f^2 \exp\left(-\frac{2}{l^2}\sin^2\left(\pi \frac{x_i - x_j}{p}\right)\right)$$

The derivatives of this kernel are shown below:

$$\frac{\partial k_{Per}}{\partial \sigma_f} = \underline{\underline{\frac{2}{\sigma_f} k_{Per}}}$$

$$\frac{\partial k_{Per}}{\partial l} = \sigma_f^2 \exp\left(-\frac{2}{l^2}\sin^2\left(\pi\frac{x_i - x_j}{p}\right)\right)\left(\frac{4}{l^3}\sin^2\left(\pi\frac{x_i - x_j}{p}\right)\right) = \underline{\underline{k_{Per}\left(\frac{4}{l^3}\sin^2\left(\pi\frac{x_i - x_j}{p}\right)\right)}}$$

$$\frac{\partial k_{Per}}{\partial p} = \sigma_f^2 \exp\left(-\frac{2}{l^2}\sin^2\left(\pi\frac{x_i - x_j}{p}\right)\right)\left(-\frac{2}{l^2}\cdot -\frac{\pi(x_i - x_j)\sin\left(\frac{2\pi(x_i - x_j)}{p}\right)}{p^2}\right)$$

$$= \underline{\underline{k_{Per}\left(\frac{2\pi(x_i - x_j)\sin\left(\frac{2\pi(x_i - x_j)}{p}\right)}{l^2 p^2}\right)}}$$

■

9. Appendix B

Robot manipulator code

```
function output = Manipulator()
%% Robot manipulator
%%%
% This code uses refers to the paper "Nonlinear optimization for
% hyperparameter computation in Gaussian Processes machine learning" by
% Vilius Ciuzelis.
%
% Date: 1/9/2018
% Last edit: 6/6/2019
% Author: Vilius Ciuzelis
%%%

% INITIALIZE
sandbox = 0;
h = 0.01;                                          % integration step length
time = 50;                                               % simulation length
constants = getConstants(sandbox);                  % Define model constants
trajectory = getTrajectory( h, time, constants );        % Define trajectory

% MAIN
data = simulate( h, trajectory, constants, sandbox );     % Main simulation

% PRESENT
hFig = figure();
plotTraj( constants, trajectory, hFig );            % Present the trajectory
animate( data, constants, hFig );     % Present the results in an animation
plotErrDyn( data );                                  % Present error dynamics

output = struct(    'q_1', data.q(1,:), ...             % Structure the data
                    'q_2', data.q(2,:), ...
                    'q_1_dot', data.omega(1,:), ...
                    'q_2_dot', data.omega(2,:), ...
                    'u_1', data.tau(1,:), ...
                    'u_2', data.tau(2,:));

%% Simulation
function res = simulate( h, traj, constants, sandbox )
    % See II Background theory -> Robot model

    % INITIALIZE
    [ q, omega, omega_dot ] = getInitialStates;
    [ kp, kd ] = getGains( constants, sandbox );
    f_coeff = getFCoeff(sandbox);
    storage = getStorage( length(traj) );

    x_e = traj(:,1);
    y_e = traj(:,2);

    for i=1:1:length(traj)
        % -------------------------SENSE-----------------------------------
        omega = omega + h * omega_dot;
        q = q + h * omega;
        [ M, C, G ] = getModel( constants, q, omega );
        % -------------------------PLAN------------------------------------
```

```
        q_r = inverseKinematics( constants, x_e(i), y_e(i) );
        tau = kd*(-omega) + kp*( q_r-q ) - f_coeff*omega;
        % --------------------------ACT-----------------------------------
        omega_dot = M\( tau-C*omega-G );
        % -------------------------STORE VALUES---------------------------
        storage.q(:,i) = q;
        storage.omega(:,i) = omega;
        storage.omega_dot(:,i) = omega_dot;
        storage.tau(:,i) = tau;
        storage.q_r(:,i) = q_r;
        [ storage.l1_pos(:,i), storage.l2_pos(:,i), storage.ee_pos(:,i) ] =
...
            forwardKinematics(constants,q);
        storage.error(:,i) = storage.q_r(:,i)-q;           % error dynamics
    end

    res = struct(   'q', storage.q,...                % Structure the output
                    'omega', storage.omega,...
                    'omega_dot', storage.omega_dot,...
                    'tau', storage.tau,...
                    'ee_pos', storage.ee_pos,...
                    'l1_pos', storage.l1_pos,...
                    'l2_pos', storage.l2_pos,...
                    'error', storage.error,...
                    'q_r', storage.q_r);
end

%% Storage function
function res = getStorage(count)

    zero = deal( zeros(2, count) );
    res = struct(   'q', zero, ...
                    'omega', zero, ...
                    'omega_dot', zero,...
                    'tau', zero, ...
                    'ee', zero,...
                    'l1', zero,...
                    'l2', zero,...
                    'error', zero,...
                    'qr', zero);
end

%% Storage of constants
function res = getConstants(sandbox)
    if sandbox                                    % Simple model for debugging
        l_1 = 1;
        l_2 = 0.7;
        m_1 = 1;
        m_2 = 1;
    else                          % Parameters based on KUKA LBR iiwa 14 R280
        l_1 = 42;
        l_2 = 52.6;
        m_1 = 9.6156;
        m_2 = 12.0424;
    end

    res = struct(   'l_1', l_1,...                          % length link 1
                    'l_c1', ( l_1/2 ), ...
                    'l_2', l_2,...                          % length link 2
```

```
                    'l_c2', ( l_2/2 ), ...
                    'm_1', m_1, ...                             % mass link 1
                    'm_2', m_2, ...                             % mass link 2
                    'g', 9.81,...                       % gravitational constant
                    'I_1', ( m_1*l_1^2/12 ),...           % Inertia for link 1
                    'I_2', ( m_2*l_2^2/12 ));             % Inertia for link 2
end

%% Manipulator model
function [ M, C, G ] = getModel( const, q, omega )
    % See Appendix A: Derivation of the EOM for 2-link planar manipulator
    l_1 = const.l_1;
    l_c2 = const.l_c2;
    m_1 = const.m_1;
    m_2 = const.m_2;

    M_11 = const.I_1 + const.I_2 + m_1*const.l_c1^2 + ...
           m_2*( l_1^2 + l_c2^2 + 2*l_1*l_c2^2 + 2*l_1*l_c2*cos(q(2)) );
    M_12 = const.I_2 + m_2*( l_c2^2 + l_1*l_c2*cos(q(2)) );
    M_21 = M_12;
    M_22 = const.I_2 + m_2*l_c2^2;
    M =[    M_11    M_12;
            M_21    M_22 ];

    C_11 = -m_2*l_1*l_c2*sin(q(2))*omega(2);
    C_12 = ( -m_2*l_1*l_c2*sin(q(2)) )*(omega(1)+omega(2));
    C_21 = m_2*l_1*l_c2*sin(q(2))*omega(1);
    C_22 = 0;
    C = [   C_11    C_12;
            C_21    C_22];

    G_1 = const.g*( ( m_1*const.l_c1 + m_2*l_1 )*cos(q(1)) + ...
           m_2*l_c2*cos( q(1)+q(2) ) );
    G_2 = const.g*m_2*l_c2*cos( q(1)+q(2) );
    G = [   G_1;
            G_2 ];
end

%% Friction coefficients
function res = getFCoeff(sandbox)
    if sandbox
        res = [ 2 0;
                0 1 ];
    else
        res = [ 10000   0;
                0       5000];
    end
end

%% Initial states
function [ q, omega, omega_dot ] = getInitialStates
    q = [ -1; pi/2 ];
    omega = [ 0; 0 ];
    omega_dot = [ 0; 0 ];
end

%% Forward kinematics
function [ link1, link2, endEffector ] = forwardKinematics( const, q )
    % Computes the x and y coordinates of starting points of the links
```

```
    % See II Background theory -> Kinematics
    link1 = [   0;
                0 ];
    link2 = [   const.l_1*cos(q(1));
                const.l_1*sin(q(1)) ];
    endEffector = [ link2(1) + const.l_2*cos(q(1)+q(2));
                    link2(2) + const.l_2*sin(q(1)+q(2))];
end

%% Inverse kinematics
function res = inverseKinematics( const, x, y )
    % Computes the joint angles as a function of x and y
    % See II Background theory -> Kinematics
    l_1 = const.l_1;
    l_2 = const.l_2;

    arg1 = (x^2+y^2-l_1^2-l_2^2)/(2*l_1*l_2);

    q_2 = atan2(sqrt(1-(arg1)^2), arg1);
    q_1 = atan2( y,x ) - atan2( l_2*sin(q_2), (l_1+l_2*cos(q_2)) );

    res = [ q_1; q_2 ];
end

%% Controller gains
function [ kp, kd ] = getGains( const, sandbox )
    % Controller parameters
    if sandbox
        k_d1 = 50;
        k_p1 = 500;
        k_d2 = 20;
        k_p2 = 200;
    else
        k_d1 = 30000*const.m_1*const.l_1/2;
        k_p1 = 100000*const.m_1*const.l_1;
        k_d2 = 1000*const.m_2*const.l_2;
        k_p2 = 50000*const.m_2*const.l_2/4;
    end
    kd = [  k_d1    0;
            0       k_d2 ];
    kp = [  k_p1    0;
            0       k_p2 ];
end

%% Identification trajectory
function [ traj, count ] = getTrajectory( h, steps, const )
    t = ( 0:h:steps-h+1 )';                                          % time
    count = length(t);
    center = [ const.l_1 0 ];
    radius = 1/2*const.l_2;
    theta = t*( 2*pi/t(end) );
    points = center + radius*[ cos(theta) sin(theta) ];

    traj = points;
end

%% Trajectory plotter
function plotTraj( const, traj, Hfig )
    l_1 = const.l_1;
```

```
    l_2 = const.l_2;

    [q,~,~] = getInitialStates;
    subplot(2,2,1)
    plot( traj(:,1), traj(:,2) );
    grid on;
    title( "Planned e.e. trajectory in the reachable workspace" );
    axis([ -(l_1+l_2) (l_1+l_2) -(l_1+l_2) (l_1+l_2) ]);
    hold on;
    plot([ 0 cos(q(1))*l_1],[0 sin(q(1))*l_1 ]);
    hold on;
    plot([ cos(q(1))*l_1 cos(q(1))*l_1 + cos(q(1) + q(2))*l_2 ],...
        [ sin(q(1))*l_1 sin(q(1))*l_1 + sin(q(1) + q(2))*l_2 ]);
    hold off;
end

%% Animation
function animate( data, const, hFig )
    subplot( 2,2,2 );
    if nargin == 2
        hFig = figure();
    end
    d = 100;                                            % frames per second
    j=1:d:length(data.q);

    ee_pos = data.ee_pos;
    l2_pos = data.l2_pos;

    for i=1:length(j)-1
        hold off
        plot( ee_pos(1,1:j(i)),ee_pos(2,1:j(i)), "-" );
        hold on;
        plot(   [l2_pos(1,j(i)) ee_pos(1, j(i))],...
                [l2_pos(2, j(i)) ee_pos(2, j(i))],'o',...
                [data.l1_pos(1) l2_pos(1,j(i))],...             % first arm
                [data.l1_pos(2) l2_pos(2,j(i))],'k',...
                [l2_pos(1,j(i)) ee_pos(1, j(i))],...          % second arm
                [l2_pos(2,j(i)) ee_pos(2, j(i))],'k')
        hold on;

        title( 'Motion of the robot' )
        xlabel('x')
        ylabel('y')
        axis( [ -const.l_1 - const.l_2, const.l_1 + const.l_2,...
                -const.l_1 - const.l_2, const.l_1 + const.l_2 ] );
        grid on;
        hold on;
        drawnow;
    end
end

%% Error dynamics plotter
function plotErrDyn( data )
    steps = 200;
    subplot(2,2,[3,4] );
    plot( 1:steps, data.error(:,1:steps) );
    txt = sprintf( "Error dynamics in the first %d simulation steps", steps
);
    title(txt);
```

```
    legend( "Error in q_1", "Error in q_2" );
end
end
```

Gaussian Processes inference code

```
%% GP Vilius Ciuzelis

run startup.m;                                              % For gpml_randn()
input = Manipulator();                                    % Generate the model

%% Globals
global sigma_f sigma_n l gamma
sigma_f = 0.9;                % standard deviation of the noise-free signal
sigma_n = 0.09;                            % standard deviation of the noise
l = 1;                                                          % Length-scale
gamma = 2;                  % For use in gammaExp cov. func. Value must be 2
rng( 'default' );                                          % For repeatability

[ X, xs, y, Y] = getInput( '1' );          % input:'1' - 1-dim sin curve
                                                        % '2' - 4-dim robot arm
[ x, y ] = standardize( X', y' );      % Standardize the inputs and outputs
covFunc = 'SE-ARD';
hyp = struct( 'M', ones(size(X,2),1), 'sf', sigma_f , 'sn', sigma_n);
%% OPTIMIZE
wolfe = struct('verbose', 1, ...                show intermediate plots if 1
                             'c_1', 10^-4,...
                             'c_2', 10^-1,...      0.9 for loose line search
                             'itermax', 100,...
                             'jmax', 100);

params = struct('itermax', 100, ...
                'gradientType', 'anal', ...                 'anal', 'approx'
                'precond', '', ...
                'beta', 'FR-PR', ...       'FR', 'PR', 'PR+', 'FR-PR', 'SD'
                'verbose', 1,...
                'v', 0.1, ...            Conjugacy requirement, typical 0.1
                'wolfe', wolfe);

tic;
res_struct = optimize( covFunc, x, y, [ ones(size(x,1),1); sigma_f; sigma_n
]', params);
toc;
res = res_struct.hyp;

%% ----------------------GP REGRESSION-------------------------------------
hyp = struct( 'M', res( 1:end-2 ), 'sf', res( end-1 ) , 'sn', res( end ));

% Perform GP
[ mu, variance, covariance ] = getGP( x', xs, y', covFunc, hyp );
figure;

plot(mu); hold on;
plot(y); legend('Predicted mean \mu', 'Input y');
% MSE = mean(mu-Z).^2;
% SMSE = MSE/std(Y);
if 1 plotGP( covFunc, mu, variance, x', xs, y' ); end
%% ------------------------------PRIOR-------------------------------------
```

```
prior = getFPrior( covFunc, hyp, xs, 10 );
if 1 plotPrior( prior, xs ); end
%% ----------------------------POSTERIOR----------------------------------
posterior = getFPost( mu, covariance, 10 );
if 1 plotPost( posterior, mu, variance, x', xs, y ); end %#ok<*SEPEX>

% ----------------------Plotter functions start--------------------------
%% Plot prior
function plotPrior( prior, xs )
    %%%
    % arg prior attributes:
    %       mean
    %       variance
    %       samples
    %%%

    hyp = evalin( 'base', 'hyp' );
    figure(); subplot( 2,2,2 ); hold on;

    plot( xs, prior.samples );
    mu = plot( xs, prior.mu, 'r--' );
    sigma = plot( xs, prior.mu+2*sqrt(prior.sigma), 'k--' );
    plot( xs, prior.mu-2*sqrt(prior.sigma) ,'k--' );
    legend( [ mu, sigma ] , 'Mean', '95% Confidence interval' );
    txt = sprintf('Samples from the prior distribution');
    title(txt);

    subplot( 2,2,1 );
    plot( xs, prior.random_functions );
    txt = sprintf( "%d random function samples drawn from the normal
distribution", size(prior.random_functions, 2) );
    title( txt );
    hold on;

    subplot( 2,2,3 );
    norm = normpdf( xs, 0, 1 );
    plot( xs, norm, 'LineWidth', 2 ); hold on;
    histogram( normalize(prior.random_functions), 'Normalization', 'pdf',
'BinMethod', 'auto');
    title( "Histogram over normal samples" );
    legend( "Normal distribution" );
    hold off;
end
%% Plot GP
function plotGP( covFunc, mu, variance, X, xs, y )

figure();   hold on;
var = plot( xs, mu+2*sqrt(variance), 'k--' ); plot( xs, mu-2*sqrt(variance)
,'k--' );
mean = plot(xs, mu, 'b' );
testpoints = plot( X, y, 'r+' ); hold off;
legend( [ var, mean, testpoints] , '95% Confidence interval', 'Predicted
mean', 'Test points' );
txt = sprintf( 'GP regression using %s', covFunc );
title( txt );
end

%% Plot Posterior
function plotPost( posterior, mu, variance, X, xs, y )
```

```
subplot( 2,2,4 ); hold on;
plot( xs, posterior );
var = plot( xs, mu+2*sqrt(variance), 'k--' ); plot( xs, mu-2*sqrt(variance)
,'k--' );
testpoints = plot( X, y, 'r+' ); hold off;
legend( [ var, testpoints ] , '95% Confidence interval', 'Test points' );
txt = sprintf( 'Samples from inference' );
title( txt );
end

%% Covariance function
function res = kernel( covariance_function, X, Y, hyp )
    % CONVERT TO X: D x N, Y: d x M
    x = X';
    y = Y';
    [D, N] = size(x);
    [d, M] = size(y);

    % PREALLOC
    res = zeros( N, M );

    switch covariance_function
        case 'SE-ARD'
            prodl = prod( hyp.M )^2;
            for row=1:N
                    for col=1:M
                        res( row,col ) = hyp.sf^2*exp( -0.5*sum((x(:,row)-
y(:,col)).^2/prodl) );
                    end
            end
    end
end
%% Prior
function prior = getFPrior( covFunc, hyp, X_star, n )
    % Generate n samples from the prior distribution.
    % x ~ N(mu, K), where mu is the mean and K is the
    % covariance matrix


    Ns = length( X_star );
    KXX = kernel( covFunc, X_star, X_star, hyp );
    L = chol( KXX + 1e-6*eye(Ns), 'lower' );
    u = randn( Ns , n );                    % n random Gaussian distributions
    samples = L * u;                                              % + 0 mean
    variance = diag( KXX );

    prior = struct( 'mu', zeros(Ns,1), 'sigma', variance, ...
        'samples', samples, 'random_functions', u );
end

%% GP regression
function [ mean, var, cov ] = getGP( X, X_star, y, covFunc, hyp )
    % This function uses algorithm 2.1 from Rasmussen&Williams, 2006

    n = length( X );
    KXX = kernel( covFunc, X,X, hyp ) + hyp.sn^2*eye(n) ;
    K_star = kernel( covFunc, X_star, X, hyp );
    K_star_star = kernel( covFunc, X_star, X_star, hyp );
```

```
    L = chol( KXX , 'lower' );
    Lk = L\K_star';
    invKy = L'\(L\y);

    mean = K_star * invKy;
    cov = K_star_star - Lk' * Lk;
    var = diag( cov ) + hyp.sn^2 * ones( length( cov ), 1 );   % noisy data
end

%% Posterior
function posterior = getFPost( mu, K, n )
    % Generate n samples from the posterior distribution.
    % x ~ N(mu, K), where mu is the mean and K is the
    % covariance matrix, ref p. 201 R&W
    L = chol( K + 10e-6*eye( length(mu) ), 'lower' );
    posterior = mu + L * randn( length(mu), n );
end

%% Squared Exponential covariance function (2.20) R&W 2006
function res = SE(arg1, arg2, hyp)
    res = hyp.sf^2*exp(-0.5*(hyp.l)\(sum((arg1-arg2).^2))) + ...
        hyp.sn^2*eq(arg1, arg2);
end

%% MacKay covariance function, (4.31) R&W 2006
function res = MacKay(arg1, arg2, hyp)
    r = arg1-arg2;
    res = exp(-2*inv(hyp.l^2)*sin(r/2).^2);
end

%% Matérn v=3/2 covariance function, (4.17) R&W 2006
function res = Matern32(arg1, arg2)
    hyp = getHyp;
    r = arg1-arg2;
    res = (1+sqrt(3)*r*inv(hyp.l))*exp(-sqrt(3)*r*inv(hyp.l));
end

%% Matérn v=5/2 covariance function, (4.17) R&W 2006
function res = Matern52(arg1, arg2)
    hyp = getHyp;
    r = arg1-arg2;
    res = (1+sqrt(5)*r*inv(hyp.l)+5*r^2*inv(3*hyp.l^2))*exp(-
sqrt(5)*r*inv(hyp.l));
end

%% Gamma-exponential covariance function, (4.18) R&W 2006
function res = gammaExp(arg1, arg2)
    hyp = getHyp;
    r = arg1-arg2;
    res = exp(-(r/hyp.l)^hyp.gamma);
end

%% Exponential covariance function
function res = Exp(arg1, arg2)
    hyp = getHyp;
    r = arg1-arg2;
    res = exp(-r*inv(hyp.l));
end
```

```
%% Input fetcher
function [ X, xs, Y, y ] = getInput(dataset)
    switch dataset
        case '1'
            %     from http://www.gaussianprocess.org/gpml/code/matlab/doc/
            n = 20;
            X = gpml_randn(0.8, n, 1);                  % 20 training inputs
            xs = linspace(-3, 3, 61)';                      % 20 test inputs
            Y = sin(3*X);
%             Z = sin(3*xs);
            y =  Y + 0.1*gpml_randn(0.9, n, 1); % 20 noisy training targets

        case '2'
            s = 1;      % starting index
            e = 200;    % ending index
            sp = 2;    % spacing index
            output = evalin('base', 'input');
            X = [output.q_1(s:sp:e)', ...                    % Training data
                output.q_2(s:sp:e)', ...
                output.q_1_dot(s:sp:e)', ...
                output.q_2_dot(s:sp:e)'];
            Y = output.u_1(s:sp:e)';                         % True function
            y = Y + 0.05*gpml_randn(0.05, length(Y), 1);       % Target data
            xs = linspace(1, length(y), length(y))';           % test inputs
        case '3'
            % Values from the uniform distribution on the interval (a, b)
            X1 = 100.*rand(100,1);
            X2 = 100.*rand(100,1);
            X = [X1 X2];
            Y = sin(pi*X1)+cos(2*pi*X2);
            y =  Y + 0.1*gpml_randn(0.9, 100, 1);
            xs = linspace(-20, 120, 200);
    end
end

%% Input plotter
function plotInput(val)
    if val
        y = evalin('base', 'y');
        X = evalin('base', 'X');
        figure();
        plot(X, y, '+'); hold on;
        plot(linspace(-2,2,100), sin(3*linspace(-2,2,100)));
        legend('Training points', 'True function');
        xlabel('input, index'); ylabel('output, y');
    end
end

%% Standardize
function varargout = standardize(varargin)
    %%%                                              x - mean(x)
    % Standard score using formula:     x_new = -----------
    %                                               std(x)
    %%%

    % PREALLOC
    varargout = cell(size(varargin));

    for i=1:nargin
```

```
        [D, N] = size(varargin{i});
        for j=1:D
            varargout{i}(j, :) = ...
                (varargin{i}(j, :) - mean(varargin{i}(j,
:)))/std(varargin{i}(j, :));
        end
    end
end
```

Optimization algorithm

```
function res = optimize(covFunc, inputX, inputY, inputTheta, params)
%%%
% Hyperparameter optimization using the negative marginal likelihood method
% Input:
%   inputX - (D x N) training input vector
%   inputY - (d x N) training target vector
%   theta = struct('l', theta(1:end-2), 'sf', theta(end-1), 'sn',
theta(end));
%%%

    % INITIALIZE
    X = inputX;
    Y = inputY;
    theta = inputTheta(1:end-1)';
    sn = inputTheta(end);
    [D, N] = size(X);
    [d, M] = size(Y);
    if(N~=M) error('Check the length of the training vectors X and Y'); end
    verboseON = params.verbose;

    % -------------PLOTTING AREA--------------
    if verboseON
        L1 = linspace(0.1, 10, 100);
        L2 = linspace(0.1, 10, 100);

        len = length(L1);
        MLgrid = zeros(len);
        for i=1:len
            for j=1:len
                MLgrid(i, j ) = NLML([L1(i), L2(j)]);
            end
        end
        fig5 = figure(5);
        ax5 = axes('Parent', fig5);
        contour(ax5, L1, L2, MLgrid', 500);
        hold on;
%     figure; imshow(mat2gray(MLgrid));
    %---------------------------------------

        outputNames = ...
            '| ITER |    FUN VAL    |                          X VAL
|  DATA FIT   | CMPX PEN | NORM CNST |\n';
        outputLine = ...

'|=============================================================================
=========================================|\n';
        outputHeader = strcat('\n\n\n',outputLine,outputNames,outputLine);
    end
```

```
    if strcmp(covFunc, 'SE-ARD')
        if verboseON
            fprintf('--> Hyperparameter optimization: optimizer = NCG,
kernel = %s \n', covFunc);
        end
        %-------------------------NCG START-----------------------------%

        % DEFINE
        getF = @(x) NLML(x);
        getDf = @(x) dNLML(x);
        itermax = params.itermax;

        % PREALLOC
        x = zeros(D+1, itermax+1);
        df = zeros(D+1, itermax+1);
        alpha = zeros(1, itermax+1);
        p = zeros(D+1, itermax+1);
        beta = zeros(1, itermax+1);

        % INITIALIZE
        abort_counter = 0;
        abort_flag = 0;
        success_flag = 0;
        x(:, 1) = theta;                                        % Given x0
        df(:, 1) = getDf(x(:, 1));                  % Evaluate df0 = df(x0)
        p(:, 1) = -df(:, 1);                                % Set p0 = -df0
        k = 1;                                                      % k = 0
        old_fk = getF(x(:, k));

        % MAIN NCG LOOP
        while 1
            if verboseON
                plot(ax5, x(1, k), x(2, k), 'k+');

                % Print to command view
                if mod(k-1, 20)==0 fprintf(outputHeader); end
                [funval, datafit, cmpxpen, normcnst] = NLML(x(:, k));
                outputData = ...
                    getOutputData(k,funval,x(:,k), datafit, cmpxpen,
normcnst);
                fprintf(outputData);
            end

            % Line search
            if ( k>1)
                old_fk = getF(x(:, k-1));
            end
            alpha(:, k) = ...                                % Compute ak
                wolfe(getF, getDf, p(:,k), x(:, k), params.wolfe);

            % Update hyperparameters
            x(:,k+1)= x(:,k) + alpha(:,k)*p(:,k);  % set xk+1 = xk + ak*pk;

            % Update gradients
            df(:, k+1) = getDf(x(:, k+1));               % Evaluate dfk+1;

            % Update search direction
            beta(:, k+1) = ...  % betaFRk+1 = df'k+1*dfk+1/df'k*dfk (5.41a)
```

```
                getBeta(df(:, k+1), df(:, k), p(:, k), params.beta);
            p(:, k+1) = ...          % pk+1 = -dfk+1 + betaFRk+1*pk (5.41b)
                -df(:, k+1)+ beta(:, k+1)*p(:, k);
            k = k + 1;                                    % k = k + 1 (5.41c)

            % Check conjugacy
            gradients_not_conjugate = (p(:, k+1)'*p(:, k)) <= 0;
            if gradients_not_conjugate
                p(:, k+1) = -df(:, k+1);
            end

            % ABORT CONDITIONS
            if (k >= itermax)
                break;
            end
            if (alpha(:, k-1) < sqrt(eps))
                abort_counter = abort_counter + 1;
            else
                abort_counter = 0;
            end
            if abort_counter >= 5
                abort_flag = 1;
                break;
            end
            success_flag = 1;
        end                                                % end (while)
        %----------------------------NCG END-------------------------------%
    end

    if verboseON
        plot(ax5, x(1, k), x(2, k), 'r+');

        if abort_flag
            fprintf('\t\t Aborted due to lack of progress\n');
        elseif success_flag
            fprintf('\t\t Terminated successfully\n');
        end
    end

    % Truncate the vectors
    x = x( :, 1:k );
    df = df( :, 1:k );
    alpha = alpha( 1:k );
    p = p( :, 1:k );
    beta = beta( 1:k );

    res = struct('hyp' , [x(:, k); sn], 'NLML', NLML(x(:, k)));
% ------------------------- HELPER FUNCTIONS ----------------------------%
    function beta = getBeta(r_new, r_old, pk, type)
        if (strcmp(params.beta, 'SD'))    % check for steepest descent
            beta = 0;
            return;
        end
        switch type
            case 'FR'
                beta = (r_new'*r_new)/(r_old'*r_old);
            case 'PR'
                beta = r_new'*(r_new-r_old)/(r_old'*r_old);
            case 'PR+'
```

```
            beta = max(r_new'*(r_new-r_old)/(r_old'*r_old), 0);
        case 'FR-PR'
            betaFR = (r_new'*r_new)/(r_old'*r_old);
            betaPR = r_new'*(r_new-r_old)/(r_old'*r_old);
            if (betaPR < -betaFR)
                beta = -betaFR;
            elseif (abs(betaPR)<=betaFR)
                beta = betaPR;
            elseif betaPR > betaFR
                beta = betaFR;
            end
        case 'HS'
            beta = r_new'*(r_new-r_old)/(r_new-r_old)'*pk;
        otherwise
            warning('No step search method selected. Beware!');
    end
end

function res = dNLML(x)
    %%%
    % Partial derivatives of the Negative Log Marginal Likelihood with
    % respect to all hyperparameters, ref eq. (5.9) R&W 2006
    % x     - (1 x D+1) hyperparameter list
    % type  - 'analytical' (default), 'approximation'
    %%%

    % PREALLOC
    res = zeros(size(x));
    thetak = x;
    l = x(1:end-1);
    sf = x(end);

    switch params.gradientType
        case 'anal'
            Ky = kernel(thetak, X, X)+sn^2*eye(N);                  % Ky
            L = chol(Ky, 'lower');
            invKy = L'\(L\Y');                                      % Ky^-1y
            val = (invKy*invKy'-inv(L*L'));          % alpha*alpha'-K^-1

            % COMPUTE   d(ML)/d(l) and d(ML)/d(sf)
            for dim=1:length(l)
                dKydl = ( ((X(dim, :)-X(dim,:)').^2 )/l(dim)^3).* Ky;
                res(dim) = -0.5*trace(val*dKydl);
            end
            dKydsf = ( 2/sf ) * Ky;
            res(end) = -0.5*trace(val*dKydsf);

        case 'approx'
            delta = sqrt(eps);
            for dim=1:length(l)       % forward finite difference
                theta_delta = thetak;
                theta_delta(dim) = l(dim)+delta;
                res(dim) = (NLML(theta_delta)-NLML(thetak))/delta;
            end
            theta_delta = thetak;
            theta_delta(end) = thetak(end)+delta;
            res(end) = (NLML(theta_delta)-NLML(thetak))/delta;
    end
end
```

```
    % Squared Exponential Automatic Relevance Determination (ARD) kernel
    function res = kernel(hyp, X, Y)
        %%% Checked and optimized
        % hyp - 1 x D+1 vector consisting of
        %               1 x D characteristic length scales l, and
        %               1 x 1 noise-free signal standard deviation -sigma_n
        % X -   D x N training data
        % Y -   D x N training data
        %%%
        lProduct = prod(hyp(1:end-1));
        sf2 = hyp(end)^2;
        exp_arg = zeros(length(X));
        for dim=1:size(X,1)
            exp_arg = exp_arg + (X(dim, :) - Y(dim,:)').^2;
        end
        res = sf2*exp( -0.5*exp_arg/lProduct^2 );

    end

    function [res, data_fit, compl_pen, norm_const] = NLML(theta)
        %%%
        % Negative Log Marginal Likelihood
        %   1                 1             1
        %   -*y'*(Ky^-1)*y + -*log(|Ky|) + -*log(2pi)
        %   2                 2             2
        %%%

        y = Y';
        Ky = kernel(theta, X, X)+sn^2*eye(N);
        try
            L = chol(Ky, 'lower');
        catch ME
            switch ME.identifier
                case 'MATLAB:posdef'
                    error('Ky not positive definite. Check your theta');
            end
        end
        invKy = L'\(L\y);
        data_fit = 0.5*y'*invKy;
        % Complexity penalty is the same as 1/2*log(det(Ky)). This
        % implementation is 3-4 times faster
        compl_pen = sum(log(diag(L)));
        norm_const = 0.5*N*log(2*pi);
        res = data_fit+compl_pen+norm_const;
    end

    function res = getOutputData(iter, funval, xval, datafit, cmpxpen,
normcnst)

        % ITER
        xval = xval';
        val1 = num2str(iter,4);
        for it = 1:(5-(length(val1)))
            val1 = [val1 ' '];
        end
        val1 = ['| ' val1 '|'];

        % FUN VAL
```

```
        val2 = num2str(funval,6);
        for it = 1:(13-(length(val2)))
            val2 = [val2 ' '];
        end
        val2 = [' ' val2 ' |'];

        % X VAL
        val3 = num2str(xval,3);
        while contains(val3, '  ')
            val3 = strrep(val3, '  ', ' ');
        end
        for it = 1:(53-(length(val3)))
            val3 = [val3 ' '];
        end
        val3 = [' ' val3 ' |'];

        % DATA FIT
        val4 = num2str(datafit,3);
        for it = 1:(10-(length(val4)))
            val4 = [val4 ' '];
        end
        val4 = [' ' val4 ' |'];

        % CMPX PEN
        val5 = num2str(cmpxpen,3);
        for it = 1:(8-(length(val5)))
            val5 = [val5 ' '];
        end
        val5 = [' ' val5 ' |'];

        % NORM CNST
        val6 = num2str(normcnst,3);
        for it = 1:(9-(length(val6)))
            val6 = [val6 ' '];
        end
        val6 = [' ' val6 ' |'];

        res = strcat(val1, val2, val3, val4, val5, val6, '\n');
    end
end
```

Wolfe conditions algorithm

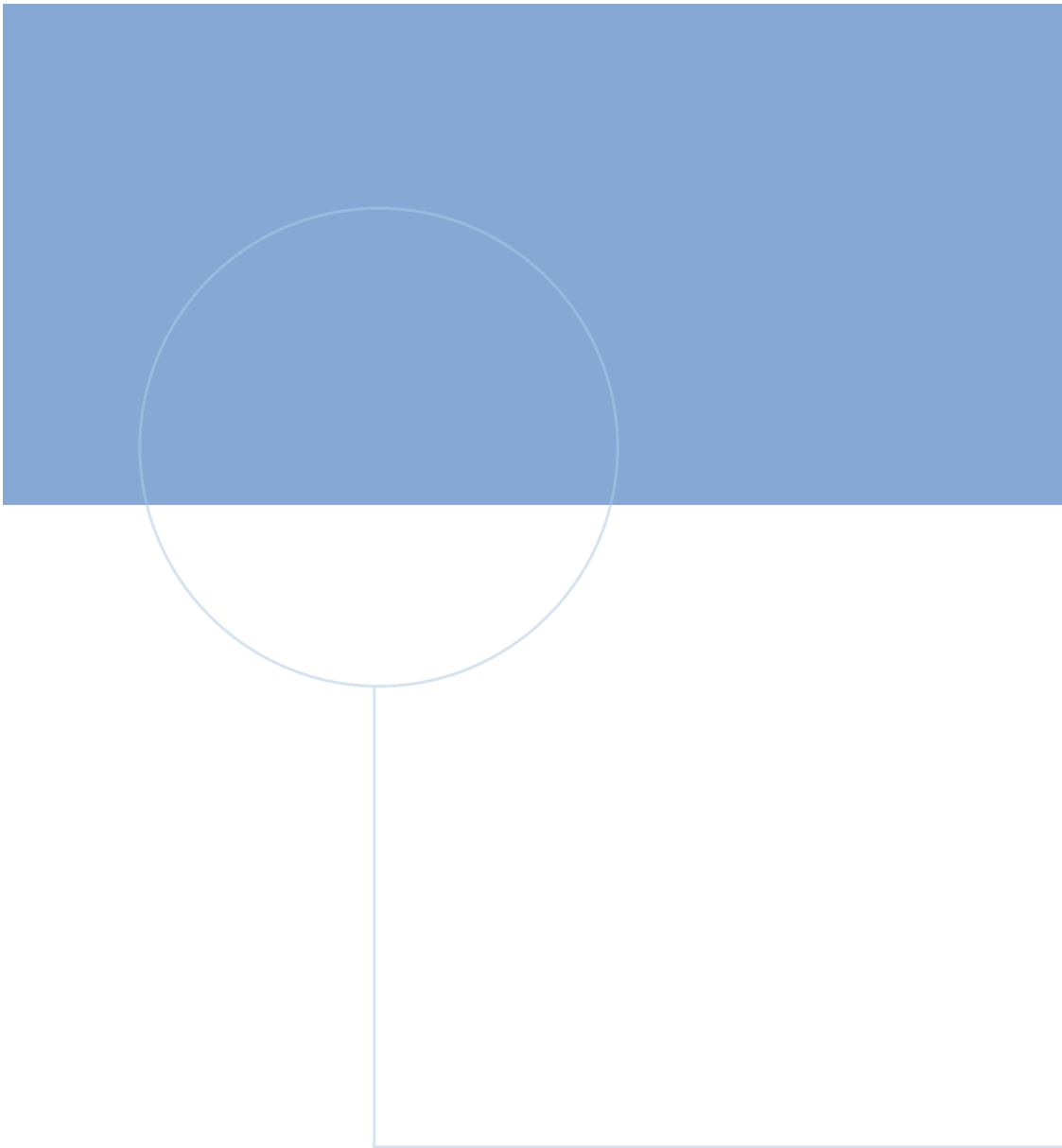
```
function res = wolfe(f, df, pk, xk, params)
%%%
% Wolfe line search algorithm, ref. p. 60 Nocedal 2006
% Search for alpha - D x 1
% given:    xk, ak, pk, f, df
%%%

% DEFINE
phi = @(a) f( xk + a*pk );
dphi = @(a) ( phi(a+sqrt(eps))-phi(a) )/ ( sqrt(eps) );
dphi2 = @(a) df( xk + a*pk )'*pk;
verbose = params.verbose;

% INITIALIZE
phi_0 = phi(0);
```

```
dphi_0 = dphi(0);
alpha_max = 1;
% (3.60) NOCEDAL
% alphas = [ min(1, 0.01*( 2*(f(xk)-fk_minus1)/dphi_0 )); alpha_max ];

alphas = [ 0; alpha_max ];              % [ previous step, current step ]
i = 2;
function_value_increase = 0;
% ------------------------------------------------------------DRAWING BOARD
if verbose
    L1 = linspace(0.0001, alpha_max, 1000);

    len = length(L1);
    l = zeros(len,1);
    for k=1:len
        l(k) = phi(L1(k));
    end
    fig = figure();
    ax = axes('Parent', fig);
    plot(ax, L1, l);
    hold on;
end
%-------------------------------------------------------------------------

% MAIN LOOP
while 1
    try
        if verbose
            plot(ax, alphas(2), phi(alphas(2)), 'ok');
        end

        val = phi(alphas(2));                       % evaluate phi(alpha_i)

        % Sufficient decrease condition
        not_sufficient_decrease = val > (phi_0 +
params.c_1*alphas(2)*dphi_0);
        if i > 2
            function_value_increase = (val >= phi(alphas(1)));
        end
        if not_sufficient_decrease || ( function_value_increase )
            % alpha chosen is too big. Find smaller alpha!
            res = zoom(alphas(1), alphas(2));
            if verbose
                plot(ax, res, phi(res), 'r+');
            end
            break;
        end
        % Sufficient decrease condition passed. Let's take a look at the
        % curvature condition next!
        val = dphi(alphas(2));                     % evaluate phi'(alpha_i)

        % Curvature condition
        sufficient_curvature = abs(val)<= -params.c_2*dphi_0;
        if sufficient_curvature
           % All conditions met. Keep alpha_i!
            res = alphas(2);
            if verbose
                plot(ax, res, phi(res), 'r+');
            end
```

```
            break;
        end

        % Curvature condition violated!
        if val >= 0
            % Alpha too big! Find smaller alpha!
            res = zoom(alphas(2), alphas(1));
            if verbose
                plot(ax, res, phi(res), 'r+');
            end
            break;
        end
        alphas(1) = alphas(2);
        alphas(2) = interpolate(alphas(2), alpha_max);

        i = i+1;

        % EXIT CONDITION
        if (i > params.itermax)
            if verbose
                warning('Reached max iterations');
            end
            break;
        end
    catch ME
        error('Line search has encountered an error and needs to close');
    end
end

% ------------------------   HELPER FUNCTIONS   ----------------------------
    function res = zoom(a, b)
        %%%
        % Zoom algorithm, ref. p. 61 Nocedal 2006
        %%%

        % INITIALIZE
        if a >= b
            alpha_high = a;
            alpha_low = b;
        else
            alpha_low = a;
            alpha_high = b;
        end
        j = 1;
        jmax = params.jmax;
        try
            while j < jmax
                % Interpolate from alpha_low to alpha_high
                alpha_j = interpolate(alpha_low, alpha_high);

                FUNC_VAL_BIG_STEP = phi(alpha_j);
                FUNC_VAL_SMALL_STEP = phi(alpha_low);
                FUNC_VAL_TINY_STEP = (phi_0+params.c_1*alpha_j*dphi_0);
                % IF FUN VAL increases with BIG step OR the TINY step
                if (FUNC_VAL_BIG_STEP > FUNC_VAL_TINY_STEP) ||...
                        (FUNC_VAL_BIG_STEP >= FUNC_VAL_SMALL_STEP)
                    % THEN update the highest allowable step size
                    alpha_high = alpha_j;
                else
```

```
                    % Sufficient decrease achieved!
                    val = dphi(alpha_j);
                    if abs(val) <= -params.c_2*dphi_0
                        % Curvature condition is also OK! BREAK!
                        res = alpha_j;
                        break;
                    end
                    if (val*(alpha_high-alpha_low)) >= 0
                        alpha_high = alpha_low;
                    end
                    alpha_low = alpha_j;
                end
                j = j+1;
            end
        catch
            error('zoom(a, b) has encountered an error and needs to
close');
        end
%         if (phi(alpha_j) > phi_0)    % if no improvement was made, DONT
MOVE
%             res = 0;
%         else
%             res = alpha_j;
%         end
        res = alpha_j;
    end % zoom

    function res = interpolate(x1, x2)
        %%%
        % Interpolation algorithm, ref. p. 57-59 Nocedal 2006
        % Arguments:
        %   type
        %   interval to interpolate [a, b]
        %%%

        % INITIALIZE
        error_flag = 0;
        f_x1 = phi(x1);
        f_x2 = phi(x2);
        g_x1 = dphi(x1);
        g_x2 = dphi(x2);

        if x1==x2                                   % No need to interpolate
            res = x1;
            return;
        end

        if      g_x1 == 0
            res = x1;
            return;
        elseif  g_x2 == 0
            res = x2;
            return;
        end

        % MAIN
        try
            d1 = g_x1 + g_x2 - 3*(f_x1-f_x2)/(x1-x2);
            d2 = sign( x2-x1 )*sqrt( d1^2-g_x1*g_x2 );
```

```
            res = x2 - (x2-x1)*( (g_x2+d2-d1)/(g_x2-g_x1+2*d2) );
        catch
            error_flag = 1;
        end

        if ~isreal(res)
            if verbose
                warning('Interpolation yields imaginary results');
            end
            res = x1;
        end

        if error_flag
            error('interpolate(x1, x2) encountered an error and needs to
close');
        end
    end
end % wolfe
```



NTNU
Norwegian University of
Science and Technology