# Cautious MPC-based control with Machine Learning

**Helge-Andre Langåker**

Master of Science in Cybernetics and Robotics

| | |
|---|---|
| Submission date: | August 2018 |
| Supervisor: | Lars Imsland, ITK |
| Co-supervisor: | Eric Bradford, ITK |

Norwegian University of Science and Technology
Department of Engineering Cybernetics

# MSC THESIS DESCRIPTION SHEET

| | |
|---|---|
| **Name:** | Helge-André Langåker |
| **Department:** | Engineering Cybernetics |
| **Thesis title (Norwegian):** | Forsiktig MPC-basert Regulering med Maskinlæring |
| **Thesis title (English):** | Cautious MPC-based Control with Machine Learning |

**Thesis Description:** There is an increasing interest in control engineering to apply techniques from machine learning. One promising direction involves the application of Gaussian process (GP) models, which provide a flexible, nonparametric approach to modelling nonlinear systems, Ebden (2015). One major advantage of GPs, compared to other nonlinear regression methods such as artificial neural networks, is the ability to give not only accurate predictions, but also a measure of uncertainty for each prediction, Rasmussen and Williams (2006). It has been shown that GPs are an effective approach to identify nonlinear models in Kocijan et al. (2004), and to account for unmodeled effects in Klenske et al. (2016) in the context of model predictive control (MPC). The aim of this project is the implementation of a GP-based model predictive controller, which will identify a disturbance model in a cautious fashion, similar to Murray-Smith, Sbarbaro Murray-Smith et al. (2003).

The following items should be considered:

1. Perform a literature study on GPs in MPC.
2. Develop and implement a model predictive control algorithm involving GPs.
3. Explore effectiveness and robustness of proposed approach on a suitable test case study.

| | |
|---|---|
| **Start date:** | 2018-01-11 |
| **End date:** | 2018-08-13 |

| | |
|---|---|
| **Thesis performed at:** | Department of Engineering Cybernetics, NTNU |
| **Supervisor:** | Lars Imsland, Dept. of Eng. Cybernetics, NTNU |
| **Co-supervisor:** | Eric Bradford, Dept. of Eng. Cybernetics, NTNU |

# Sammendrag

Bruk av Gaussisk prosess som en ikke-parametrisk regresjonsmodell, sammen med modell prediktiv kontroll har de siste årene vist lovende resultater ved å utnytte den forventede usikkerheten som følger GP. Ved å bruke usikkerheten i begrensningene kan vi ta hensyn til regresjonsavvik direkte i MPC-begrensningene. Denne oppgaven har studert effektiviteten av å bruke GP med MPC-begrensninger, med særlig fokus på stabiliteten av prediksjonene. Vi har sett på to forskjellige systemer, et stabilt system med fire tanker med langsom dynamikk, og en kjøretøymodell med hindringsunngåelse ved hjelp av en hybrid-GP-modell. Dette viste den interessante egenskapen at nøyaktigheten av prediksjonen er bestemt av stabilitetsegenskapene i systemet vi ønsker å kontrollere. Vi var i stand til å holde tanksystemet stabilt uten problemer, mens hybrid-GP i bil-systemet led av divergens og eksponensiell vekst i usikkerhet, selv om regresjonsvalideringen viste at hybrid-GP skulle hatt bedre nøyaktighet. Multi-trinns prediksjon i et marginalt stabilt eller ustabilt system, kan resultere i divergens i prediksjonen på grunn av forsvinnende små regresjonsfeil. Dette tatt i betraktning, så viser det at GP viser seg til å være lovende innen adaptiv kontroll som en effektiv metode for tilpasning til systemendringer, hvor usikkerheten gir et godt estimat på feilen til prediksjonen.

# Abstract

Using Gaussian processes as a nonparametric regression model together with model predictive control has the recent years showed promising results by utilizing the expected uncertainty that follows the GP. By utilizing the uncertainty in the constraints we are able to take into account regression deviation directly in the MPC constraints. This thesis has studied the effectiveness of using the GP with MPC constraints, with the special focus on the stability of the predictions. We have looked at two different systems, a stable four-tank system with slow dynamics, and a vehicle model with obstacle avoidance using a hybrid-GP model. This showed the interesting property that the accuracy of the prediction is determined by the stability properties in the system we like to control. We were able to keep the tank system stable without any problems, while the hybrid-GP in the car system suffered from divergence and exponential growth in uncertainty, even though the regression validation showed that the hybrid-GP should had better accuracy. Multi-step prediction in a marginally stable or unstable systems, can result in divergence in the predictions due to minuscule errors. Taken this into account, the GP show great promise in the field of adaptive control as an effective method of adapting to system changes, where the uncertainty can give a good estimate of the prediction error.

# Preface

This thesis is submitted in partial fulfillment for the Master of Science degree at the Norwegian University of Science and Technology. It has been a steep learning curve beginning this project, but it has also been a great opportunity to learn as much as I have during this project in a wast range of topic. It has been a great pleasure to be able to dive into all the parts that make up this thesis.

I have to thank my fellow students for all the good discussions and moral boosters along the way. A great thanks goes to my supervisor Lars Imsland and co-supervisor Eric Bradford, especially to Eric who has given hours and hours for discussions and guidance along the way, provided help with implementing exact moment matching, and given examples on how to optimize a Gaussian Process and set up integrators in CasADi.

*Helge-André Langåker*
*Trondheim, August 2018*

# Table of Contents

# List of Tables

# List of Figures

# Nomenclature

## Abbreviations

| | |
|---|---|
| GP | Gaussian Process |
| MPC | Model Predictive Control |
| LQR | Linear Quadratic Regulator |
| QP | Quadratic Programming |
| SE | Squared Exponential |
| ARD | Automatic Relevance Determination |
| MLE | Maximum Likelihood Estimate |
| MAP | Maximum a posteriori probability estimate |
| DARE | Discrete-time Algebraic Riccati Equation |
| MSE | Mean Squared Error |
| SMSE | Standardized Mean Squared Error |
| MNLP | Mean Negative Log Probability |
| DAE | Differential-Algebraic Equation |
| ODE | Ordinary Differential Equation |
| BDF | Backward Differentiation Formulas |
| NLL | Negative Log Marginal Likelihood |
| AD | Algorithmic Differentiation |
| RK4 | Explicit Runga-Kutta 4 |
| SLSQP | Sequential Least SQuares Programming |
| CG | Conjugate Gradient |
| CoG | Center of Gravity |
| IPOPT | Interior Point OPtimizer |

# Notation

| | |
|---|---|
| $k(\mathbf{x}, \mathbf{x}')$ | Covariance function |
| $K$ | $n \times n$ Covariance Gram matrix $K(X, X)$ |
| $k_*$ | Vector of covariance evaluated between training and test cases $K(X, x_*)$ |
| $\|K\|$ | Determinant of $K$ |
| $\nabla_\xi$ | Partial derivative w.r.t $\xi$ |
| $\mathbb{V}[f_*]$ | Variance of $f_*$ |
| $\mathbb{E}[f_*]$ | Expected value of $f_*$ |
| $\sim$ | distribution according to; example $x \sim \mathcal{N}(\mu, \sigma^2)$ |
| $\\|\xi\\|_M^2$ | Squared Euclidean norm weighted by $M$, e.g. $\xi^T M \xi$ |
| tr($M$) | Trace of matrix $M$, i.e. sum of diagonal elements in $M$ |
| diag($M$) | Vector formed by the diagonal of $M$ |
| diag($\xi$) | Diagonal matrix formed by the vector $\xi$ |
| $\mu_i^x$ | Mean of variable $x$ at time $i$ |
| $[x]_i$ | The i-th element of a vector $x$ |
| $[M]_{ij}$ | The ij-th element of a matrix $M$ |
| $[M]_{\cdot i}, [M]_{i \cdot}$ | Respectively the i-th column and row of a matrix $M$ |
| $Q$ | State penalty matrix |
| $P$ | Solution to DARE and infinite horizon terminal cost |
| $R$ | Input penalty matrix |
| $S$ | Input change penalty martrix |
| $L$ | Cholesky decomposition $A = LL^T$ |

Chapter 1

# Introduction

Gaussian Processes has long been a standard method in probabilistic machine learning, with e.g. text books like Murphy and Bach (2012) and Rasmussen and Williams (2006) giving a comprehensive introduction in the field. This method as also long been used in the field of geostatistics under the name of Krigin for interpolating points, Rasmussen and Williams (2006). One of the interesting aspects of GPs is that it is nonparametrc, avoiding the need of a prespecified finite-dimensional model class, and give a measure of the uncertainty in each prediction. Using Bayesian inference enable us not only to get a probability score for one step estimations, but give a measure on the propagated uncertainty in multi-step predictions. With this property we are able to utilize the uncertainty to give cautious bound in the constraints, where we have a measure of the probable state-space region enclosing the predicted GP. The GP can then be used as the predictive model in model predictive control, as shown in Murray-Smith et al. (2003) and Kocijan et al. (2004). The aim of this project is then to investigate this further by building on the works of Girard et al. (2003) and Hewing and Zeilinger (2017), by evaluating the effectiveness of both the predictions and the use of the uncertainty for cautious control in a novel model predictive control implementation.

## 1.1 Previous work

Gaussian processes has been shown to be a flexible way of estimating nonlinear dynamic models, where Williams and Rasmussen (1996) show that the GPs have the advantage over other liner regression methods like neural networks with that we know the probability distribution of the regression. Girard et al. (2002) take this one step further by using approximation methods to get an estimate of the distribution of the multi-step prediction. By utilizing the propagated uncertainty Kocijan et al. (2004) show how the variance can be used in MPC constraint. Klenske et al. (2016) show how Gaussian process based predictive control can give a correction of periodic model errors by estimating unmodeled effects using the GP with online learning of hyper-parameters. Hewing and Zeilinger (2017) use the same principle and give the foundation for learning unmodeled effects using the GP together with a first principle model to get cautious model predictive control. In Hewing et al. (2017) this is taken one step further using sparse approximation of GP, by estimating model errors in nonlinar systems, to get real-time performance estimating model error in a miniature-car MPC problem. Other noteworthy works is Boedecker et al. (2014) who use sparse GP with together with a Linear Quadratic Regulator to get real-time performance with the GP. Deisenroth and Rasmussen (2011) use online GP model learning and reinforcement control learning. Another interesting work is by Chowdhary et al. (2015) who use GPs with Model Reference Adaptive Control (MRAC).

## 1.2 Contributions

The goal of this thesis is to display the effectiveness of using GP together with MPC. To do this we explore the effectiveness and robustness on two different nonlinear dynamic systems. The first is a simple inherently stable system, without constraints in the MPC. The other is an unstable vehicle model with obstacle avoidance requiring more robustness in the model prediction, with both state and path constraints that will be active in the prediction horizon of the MPC. In this a special focus has been on the stability of the predictions, and how this affect the uncertainty. As a byproduct there was developed a python framework GP-MPC for simulating and controlling nonlinear dynamic systems using Gaussian Process and model predictive control. Built to be a flexible tool for experimentation with machine learning and model predictive control.

## 1.3 Structure of this thesis

This thesis is divided into 8 chapters and 5 appendices as given below

- chapter 1 - Introduction
- chapter 2 - Introduction to Gaussian process theory.
- chapter 3 - Background theory for Model Predictive Control.
- chapter 4 - Heuristics on how to best learn a dynamic model.
- chapter 5 - Model learning and control of a four-tank system.
- chapter 6 - Model learning and obstacle avoidance control of a vehicle model.
- chapter 7 - Discussion
- chapter 8 - Conclusion and future work.
- Appendix A - Mathematics that give faster and more robust computation.
- Appendix B - Information about the GP-MPC implementation and its dependencies.
- Appendix C - Documentation for the GP-MPC framework.
- Appendix D - Code examples on how to use GP-MPC with model learning and MPC control.
- Apendix E - Source code for GP-MPC.

Chapter 2

# Theory: Gaussian Process

In the field of machine learning we have the concept of supervised learning, as a method of learning input-output relations from empirical training data, to predict the output of new unseen data. This may be classification of handwritten numbers in an image, by using the pixels in an image as input to retrieve the correct number label. It could also be a regression problem where we which to learn the dynamics of a robot arm, or predict future stock prices using previous data. These types of machine learning problems has been increasingly more popular in recent years as more computing power lead the way for deep artificial neural networks. In another family branch in probabilistic machine learning we have the Gaussian process as a probabilistic framework that similarly tackle both classification and regression problems. The key difference between these methods are that while neural networks in simple terms can be viewed as linear regression with weights, Gaussian Process on the other hand instead use kernel functions to interpolate between given training data, exploiting the probability of Gaussian distributed points. Having a framework with Bayesian inference give not only the option of getting an uncertainty score for one step predictions, but also the ability to propagate the uncertainty in multi-step predictions, as shown in Girard et al. (2002). This chapter will look at the basics of GP regression with Rasmussen and Williams (2006) as the main reference, and expand this to multi-step predictions with uncertain inputs using Hewing and Zeilinger (2017) and Girard et al. (2002).

## 2.1 Definition

As defined by Rawlings et al. (2017) in definition 2.1 the GP is a generalization of the Gaussian probability distribution, as a collection of random variables with a joint Gaussian distribution. While a multivariate probability distribution describes random vector variables, a stochastic process describes the properties of functions. We can think of an arbitrary function as an infinite long vector where each entry in the vector specifying the function values $f(x)$ with input $x$. Calculating the output using an infinite object is not especially desirable, but the pleasant property of the Gaussian process is that the inference using a subsection with a finite amount of point will give the same answer as in the case of the full space with infinite amount of points.

**Definition 2.1.** A Gaussian process is a collection of random variables, any finite number of which have a joint Gaussian distribution. Rasmussen and Williams (2006)

In this chapter we will use the following notation with a set of training data defined as

$$\mathcal{D} := \{\mathbf{X} := [\mathbf{x}_1, \dots, \mathbf{x}_n]^T, \mathbf{y} := [y_1, \dots, y_n]^T\}$$

where $\mathbf{x}_i$ is an input vector, and the output given by $y_i = f(\mathbf{x}_i) + w_i$ with $f : \mathbb{R}^D \to \mathbb{R}$ and Gaussian noise $w_i \sim \mathcal{N}(0, \sigma_w^2)$. The GP is then specified by the mean function $m(\mathbf{x})$ and the covariance kernel $k(\mathbf{x}, \mathbf{x}')$ for a function $f(\mathbf{x})$

$$m(\mathbf{x}) = \mathbb{E}[f(\mathbf{x})], \tag{2.1}$$

$$k(\mathbf{x}, \mathbf{x}') = \mathbb{E}[(f(\mathbf{x}) - m(\mathbf{x}))(f(\mathbf{x}') - m(\mathbf{x}'))] \tag{2.2}$$

The Gaussian process can then be written as

$$f(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')) \tag{2.3}$$

The fact that the Gaussian Process is defined as a collection of random variables with a joit distribution also implies a consistency requirement. This property means that if the GP specifies $(y_1, y_2) \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, it must also specify $y_1 \sim \mathcal{N}(\mu_1, \Sigma_{11})$, where $\Sigma_{11}$ is a sub-matrix of $\Sigma$.

## 2.2 Bayesian Inference

In the next section we can begin by using the definition from Gelman et al. (2013) where Bayesian inference can be defined as the process of fitting a probability model to a given set of data $\mathcal{D}$, and summarize the result with a probability distribution in the unknown function $f(\mathbf{x})$. In such we will use the terms prior and posterior inference, as the inference with and without data.

### 2.2.1 Prior

In the introduction it was given a simplistic explanation of GP as a way of interpolating data, using training data directly. It is true that the training data is used directly, but to be able to have good interpolation it is also necessary to have a good prior. The prior inference is a guess of the system without any training points, or so far away from any training data that the prior is the best guess. The priors we will look at is the mean and covariance functions in (2.1), with a set of hyper-parameters that can fit any model. Most often the mean function $m(\mathbf{x})$ is chosen to be zero, and instead let the kernel pick up the features. It can however be an advantage to include a mean function if there is little training data or you have prior knowledge of the shape of the dynamics. If the system has a linear behaviour a linear mean function would be able to give a good prior guess of the system. The most important prior is regardless the choice of covariance function, where

Rasmussen and Williams (2006) specify a range of different covariance functions $k(\mathbf{x},\mathbf{x}')$ that can be used, and how these can be combined to best fit the dynamics of the desired system. By far the most common is the squared exponential (SE) kernel, together with the special case of Squared Exponential Automatic Relevance Determination (SEard), with a length scale for each input to the GP. The SE covariance function between pairs of random variables is given as

$$\operatorname{cov}(f(\mathbf{x}_p), f(\mathbf{x}_q)) = k(\mathbf{x}_p,\mathbf{x}_q) = \exp\left(-\frac{1}{2}|\,\mathbf{x}_p - \mathbf{x}_q\,|^2\right), \qquad \mathbf{x}_p,\mathbf{x}_q \in \mathbb{R}^D \tag{2.4}$$

For this specific covariance function we see that the covariance is close to one when the inputs are close, and decrease exponentially as the distance between the inputs increase. The prior covariance is between the two points is then defined as 1. This can again be generalized as

$$k(\mathbf{x}_p,\mathbf{x}_q) = \sigma_f^2 \exp\left(-\frac{1}{2}(\mathbf{x}_p - \mathbf{x}_q)M(\mathbf{x}_p - \mathbf{x}_q)\right) \tag{2.5}$$

with $\boldsymbol{\theta} = (\{M\}, \sigma_f^2, \sigma_n^2)^T$ as the vector containing the hyper-parameters, where $\{M\}$ give the set of parameters in the symmetric matrix $M$. The SE kernel can again take many shapes with many different choices for the matrix $M$, where Rasmussen and Williams (2006) give a selection of the most popular choices

$$M_1 = \operatorname{diag}([\ell_1^2, \dots, \ell_D^2]), \qquad M_2 = \ell^{-1} I, \qquad M_3 = \Lambda\Lambda^T + \operatorname{diag}(\boldsymbol{\ell})^{-2} \tag{2.6}$$

where $\boldsymbol{\ell}$ is a vector with positive values, and $\Lambda$ given by a $D \times k$ matrix with $k < D$. Using hyper-parameters in the SE kernel with distance measure $M_1$, the $\boldsymbol{\ell}$ interprets as the characteristic length scales. Determining how far along an axis in input-space before the function values become uncorrelated. This particular distance measure is called Automatic Relevance Determination, Neal (1996), as the inverse of the length-scale determine how relevant a signal is. With a high length-scale value, the covariance will be independent of the corresponding input. This give the property that irrelevant inputs have no effect on the inference. Examples of this is shown by Williams and Rasmussen (1996). A high length-scale then means that there is a low weight on the training data, as $\ell_d \to \infty$ for a dimension $d$, the function $f$ would vary less and less on the input $x_d$, which would be irrelevant. The opposite case is if the length scale for a given dimension is close to zero, giving a high weight on the mentioned dimension.

In general terms we can view the distance measure as the square Mahalanobis distance, Murphy and Bach (2012), where the $M$ matrix can be interpreted as the covariance in a Gaussian multivariate distribution

$$\mathcal{N}(\mathbf{x}\,|\,\boldsymbol{\mu},\Sigma) = \frac{1}{(2\pi)^{D/2}|\Sigma|^{1/2}} \exp\left[-\frac{1}{2}(\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})\right] \tag{2.7}$$

with the square Mahalanobis distance defined as $r^2 = (\mathbf{x}-\boldsymbol{\mu})^T\Sigma^{-1}(\mathbf{x}-\boldsymbol{\mu})$. The Mahalanobis distance give the surface of which the Gaussian probability density is constant. The probability is high close to the mean, and decays as the input strays further away. If we view the mean $\boldsymbol{\mu}$ as the training data, we would expect to find new data $\mathbf{x}$ within the square

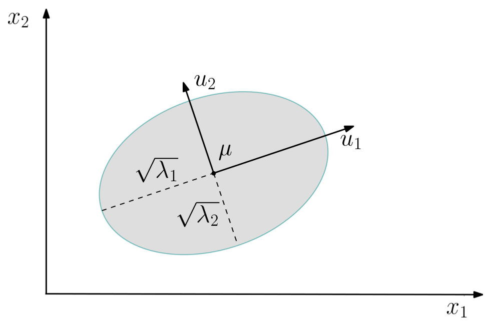


**Figure 2.1:** The Mahalanobis distance shown as a ellipse give the surface of which the probability density for a Gaussian in a two-dimensional space $(x_1, x_2)$ is constant. The axis on the ellipse are defined by the eigenvectors $\mathbf{u}_i$ and eigenvalues $\lambda_i$ of the covariance matrix $\Sigma$. Based on Figure 2.7 from Bishop (2006)

Mahalanobis distance, an ellipsoidal-shaped with principal axes given by the eigenvectors and the corresponding eigenvalues of the covariance matrix, as shown in Figure 2.1. With a diagonal matrix like with $M_1$, the eigenvalues are given as each diagonal element, while the eigenvectors are unit vectors with the same direction as the input vectors. This means that depending on the choice of length scales, we can adjust how far and in which direction from the training data in the input space the expected predicted values would fall.

### 2.2.2 Posterior

With the prior we have an estimate of the model with only the hyper-parameters as the basis. The prior functions are important building blocks in the GP, but it is of limited value without additional data. In Figure 2.2 we see a comparison of the difference between a prior and posterior distribution representing a sine curve. The prior has a zero mean and variance of $\sigma_f^2 = 0.5$, while the posterior has an additional 9 data points. In the prior we only know that the sine curve would be within the 99%-percentile from the zero mean, visualized by ten random samples using the prior distribution. In the posterior we use the same random samples, but by using the posterior distribution we also know that the signal should be close to the points we already know. If a new point lay close to a known point there is a high probability that a new point lay close to the spline between known points, while if there is the new point too far away the information is reduced to the prior zero mean and variance. This is visualized in Figure 2.3 where we see how the uncertainty grows when there is a long distance between known points. This also brings out the importance of having a set of known points that are able to represent the whole state space of interest. Since the GP only add splines between known point and the prior mean, there is no number of data points that could give a good model estimation if there isn't any points close to the operating region.

To further see the how the different hyper-parameters affect the model estimation, we will again use the posterior on a simple scalar sine curve. The free hyper-parameters in a one dimensional square exponential covariance function for a noisy signal $y$ is given by the signal variance $\sigma_f^2$, length-scale $\ell$, and noise variance $\sigma_n^2$.

$$k_y(x_p, x_q) = \sigma_f^2 \text{exp}\left(-\frac{1}{2}\frac{(x_p - x_q)^2}{\ell^2}\right) + \sigma_n^2 \delta_{pq} \tag{2.8}$$

The noise variance parameter let us avoid over-fitting the model on the noise, where we

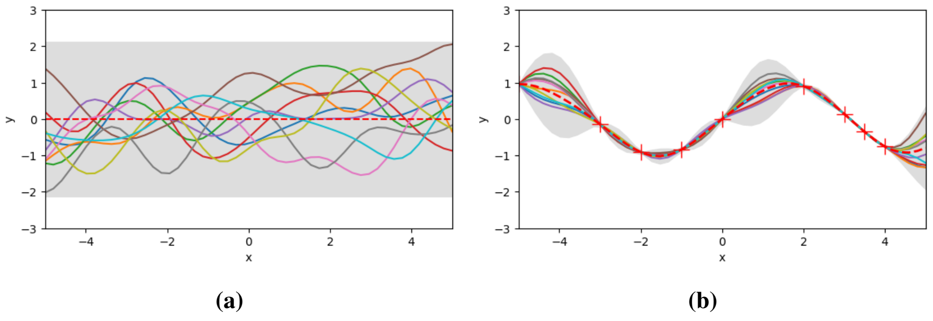


**Figure 2.2:** Ten random samples with the prior distributions in a), and the posterior in b). The prior use a zero mean function given by the read dashed line, and a signal variance of 0.5 represented as the 99-percentile in the grey area. The posterior use the training data to give a good estimate of the dynamics in the signal through the kernel.

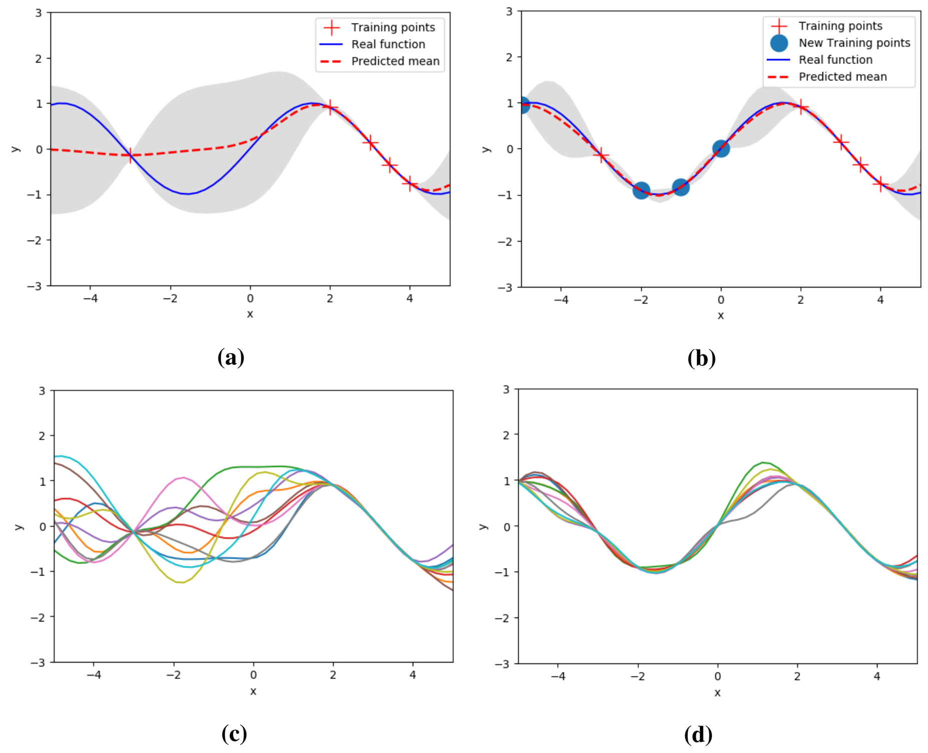


**Figure 2.3:** Adding new data. In the left column we have 5 initial data points with a high uncertainty between the points. When four new points is added in the right column we see quite good fit of the sine curve. The top row show the mean prediction with the 99-percentile, while the bottom row show the posterior distribution using random signals.

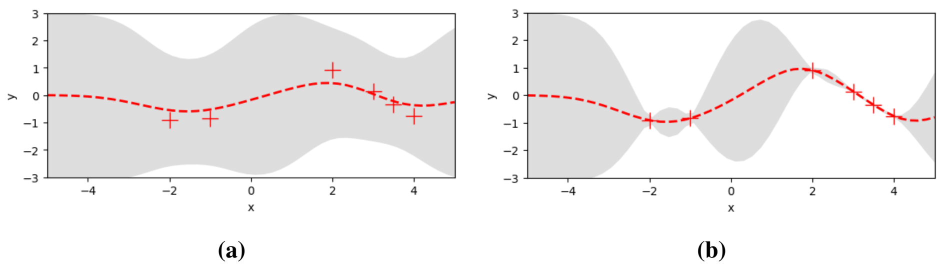


**Figure 2.4:** Adjusting the noise variance, with length scale and signal variance fixed at values $\ell^2 = 1$, $\sigma_f^2 = 1$. In a) we assume that the data is noisy and set the noise variance to $\sigma_n^2 = 1$, while in b) be assume that there is no noise and set $\sigma_n^2 = 0$.

assume the noise is Gaussian distributed, such that the Gaussian process mean acts as a smoothing filter. In Figure 2.4 is is shown one example where it assumed that there is lot of noise in the data, with a noise variance of $\sigma_n^2 = 1$, and one example where the noise is assumed to zero. In the case with the zero noise, we see that the mean fit all the data points perfectly with zero posterior variance at the training points. In the case of assumed noisy data, we are more uncertain where the actual signal is, leading to a high posterior variance at the training points, while the mean try to fit the simplest path without having to go through all the training points.

The most important hyper-parameters are the length-scales as they shape the spline between the points. A low length-scale will give a high weight on the known points with an aggressive and steep behaviour, while larger length-scales focus on a larger region around the known points. This is visualized in Figure 2.5, and as discussed previously with the analog to the Mahalanobis distance we can use the length scale to decide how far and in what direction from the training data we want the predictions to fall. With values close to zero we have a very small region of interest before the prediction is reduces to the prior, while a larger length-scales put less weight in the known data and has a larger region in which we would expect to find new data.

The signal variance $\sigma_f^2$ will not affect the mean prediction, as we can see from equation (2.5) that this is independent from the training data. The signal variance is however quite important when we want to know the certainty of the prediction. Far from the training data, the prediction reduces to the prior where the mean is defined as the given prior mean function, and the variance to the signal variance. In Figure 2.6 we see two cases from the same system as above further away from the training data. With a signal variance of $\sigma_f^2 = 0.01$ the uncertainty is too naive far away from training data, where the sine wave oscillate between $-1$ and $1$ while the prior is a zero mean with the variance reduced to the signal variance. In Figure 2.6b the variance is a lot more conservative, with a signal variance of $\sigma_f^2 = 0.5$. The prior has now a lot higher variance, with the interpretation that we don't know where the signal is, other than that it is within the 99-percentile from the zero-mean.

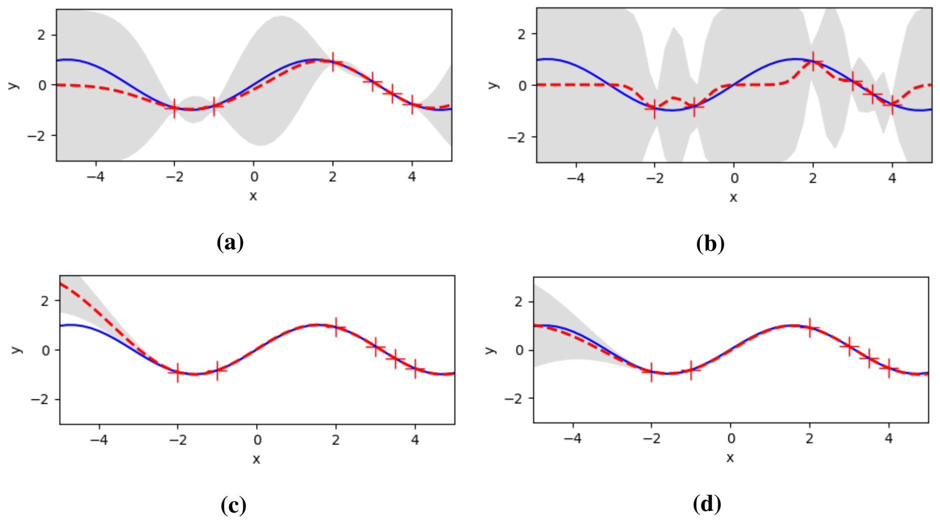


**Figure 2.5:** Adjusting the length scale, with fixed signal variance $\sigma_f^2 = 1$. The baseline is set in a) with $\ell^2 = 1$. In b) we try a low value of $\ell^2 = 0.1$, resulting in an on-off behaviour with a too high weight on the data points. The estimated signal is either at the known points or at the prior mean. In c) we try a higher value with $\ell^2 = 10$, resulting in a less aggressive behaviour with larges splines. In d) we reduce the value a litle to better fit the actual sine curve $\ell^2 = 7$.

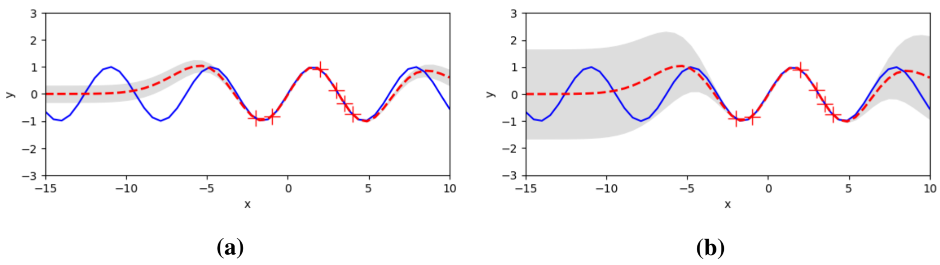


**Figure 2.6:** Adjusting the signal variance. In a) we set the signal variance $\sigma_f^2 = 0.01$, resulting in a naive and saturated posterior uncertainty that don't represent reality. In b) we increase the variance up to $\sigma_f^2 = 0.5$. We now see that the real signal is captured within the 99-percentile of the posterior distribution.

### 2.2.3 Learning Hyper-parameters

In the previous section we looked at how different hyper-parameter values affected the posterior prediction by manually setting the values. This is a rather cumbersome even for small systems like the toy example we have looked at until now. Since the GP is non-parametric we have marginalized out all parameters weights $\mathbf{w}$, and are left with a level 2 Bayesian inference. To be able to optimize the hyper-parameters we will then utilize Maximum Likelihood Estimate (MLE) with the log marginal likelihood as evidence, given by Rasmussen and Williams (2006) as

$$
\begin{aligned}
\log p(\mathbf{y}\,|\boldsymbol{X},\boldsymbol{\theta}) &= \log\int p(\mathbf{y}\,|\mathrm{w},X,\boldsymbol{\theta})\,\mathrm{d}\mathbf{w} \\
&= -\frac{1}{2}(\mathbf{y}-m(\boldsymbol{X}))^T(\boldsymbol{K}_y+\sigma_\epsilon^2\boldsymbol{I})(\mathbf{y}-m(\boldsymbol{X})) \\
&\quad -\frac{1}{2}\log|\boldsymbol{K}_y+\sigma_\epsilon^2\boldsymbol{I}|-\frac{n}{2}\log(2\pi)
\end{aligned}
\tag{2.9}
$$

whit the covariance Gram matrix $K_y = k(\mathbf{X},\mathbf{X})+\sigma_n^2 I$ and mean function $m(\mathbf{X})$. If we simplify to use a zero-mean function to shorten the expression , each of the following terms have interpretative roles.

$$
\log p(\mathbf{y}\,|\boldsymbol{X},\boldsymbol{\theta}) = \underbrace{-\frac{1}{2}\mathbf{y}^T(\boldsymbol{K}_\theta+\sigma_\epsilon^2\boldsymbol{I})\,\mathbf{y}}_{\text{Data fit}} - \underbrace{\frac{1}{2}\log|\boldsymbol{K}_\theta+\sigma_\epsilon^2\boldsymbol{I}|}_{\text{Complexity term}} - \underbrace{\frac{n}{2}\log(2\pi)}_{\text{Normalisation constant}}
\tag{2.10}
$$

The only term that use observations is the Data-fit term which decrease with the length-scales as the model get less flexible. The complexity term penalize complex models as the term increase with the lenth-scales, as the model become less complex with larger length-scales. The last term is a normalization constant.

If we now use the negative log marginal likelihood (NLL), we can use any nonlinear solvers, e.g. conjugate gradient (CG), to minimize the value subject to the hyper-parameters. In optimizing the hyper-parameters is is important to note that the NLL is non-convex. To lessen the risk of local minima it is possible to use a multi-start process with different initial guesses and choose the hyper-parameters with the lowest NLL as the optimal solution.

$$
\theta^* = \mathrm{argmin}(-\log p(\mathbf{y}\,|\boldsymbol{X},\boldsymbol{\theta}))
\tag{2.11}
$$

With small amounts of data is often difficult to find a global minima, and as a result the optimal hyper parameters for the model. To account for this we can use the log marginal posterior instead of the log marginal likelihood by adding a hyper-prior $p(\boldsymbol{\theta})$ to equation (2.10) and use Maximum á posteriori probability estimate (MAP),Murphy and Bach (2012), instead of maximum likelihood estimate (MLE). By adding a hyper-prior we would add an additional cost such that the prior are constrained within the region set by the priors.

As a small side note it is worth mentioning that Cao et al. (2017) propose to minimize the Mean Squared Error in the hyper-parameter learning instead of using the negative log marginal likelihood (NLL). Since the NLL is nonconvex there is a high probability to end up in a local minima or at a plateau, while MSE is a simple convex function. This means there will be examples where MSE give a better validation score than NLL, as shown in Cao et al. (2017).

### 2.2.4 Overfitting

Compared to many other regression methods, e.g. neural networks, the hyper-parameters in Gaussian Process regression are in general less prone to overfitting. But this is still important to note that overfitting is a problem, where the model is fine tuned to the training data and do not generalize to unseen data. This is especially the case when using an ARD covariance kernel. This method is good for estimating nonlinearities for different states, where the covariance function automatically determine which inputs are relevant, but this also means that there is a higher chance of overfitting the model by adding unwanted nonlinearities. If overfitting is suspected a simple test is then to replace the ARD kernel with a non-ARD covariance function and see if the validation score increase. An review of this effect is presented in Cawley and Talbot (2010) and Cawley and Talbot (2007).

## 2.3 Prediction with Deterministic Inputs

For prediction of a noise free model, we will use $\mathbf{f}_*$ to represent the prediction using unseen data $\mathbf{x}_*$ with a zero mean function and the distribution

$$\mathbf{f}_* \sim \mathcal{N}(m(\mathbf{x}_*), K(\boldsymbol{x}_*, \boldsymbol{x}_*)) \tag{2.12}$$

We can then combine this with the distribution of the GP with known observations.

$$\begin{bmatrix} \mathbf{f} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left( m(\mathbf{x}_*), \begin{bmatrix} K(\boldsymbol{X}, \boldsymbol{X}) & K(\boldsymbol{X}, \mathbf{x}_*) \\ K(\mathbf{x}_*, \boldsymbol{X}) & K(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix} \right) \tag{2.13}$$

This can again be generalized with prediction using noisy observations by adding the noise variance to the covariance

$$\operatorname{cov}(\boldsymbol{y}) = K(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 I \tag{2.14}$$

with the new similar distribution

$$\begin{bmatrix} \boldsymbol{y} \\ \mathbf{f}_* \end{bmatrix} \sim \mathcal{N}\left( m(\mathbf{x}_*), \begin{bmatrix} K(\boldsymbol{X}, \boldsymbol{X}) + \sigma_n^2 & K(\boldsymbol{X}, \mathbf{x}_*) \\ K(\mathbf{x}_*, \boldsymbol{X}) & k(\mathbf{x}_*, \mathbf{x}_*) \end{bmatrix} \right) \tag{2.15}$$

The mean and variance prediction for unseen data $\mathbf{x}_*$ is then given by Rasmussen and Williams (2006) as

$$\bar{f}_* = m(\mathbf{x}_*) + \mathbf{k}_*^T (K + \sigma_n^2 I)^{-1} (\mathbf{y} - m(\boldsymbol{X})), \tag{2.16}$$

$$\mathbb{V}(f_*) = k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{k}_*^T (K + \sigma_n^2 I)^{-1} \mathbf{k}_* \tag{2.17}$$

with the covariance function $\mathbf{k}_* = k(\boldsymbol{X}, \mathbf{x}_*)$ and Gram matrix $K = k(\boldsymbol{X}, \boldsymbol{X})$.

### 2.3.1 Computational complexity

Computing the inverse covariance matrix $K_y^{-1} = (K + \sigma_n^2)^{-1}$ is numerically unstable. The recommended algorithm is to use cholesky decomposition ($K_y = LL^T$) of the matrix instead, as shown in algorithm 1, Rasmussen and Williams (2006). To speed up the computation it is possible to pre-compute both the cholesky $L$ and the linear term $\alpha$, as both of these are the most costly operations.

**Algorithm 1:** Gaussian Process prediction, Rasmussen and Williams (2006)

```
    Input: $X$ (inputs), $\mathbf{y}$ (targets), $k$ (covariance function), $\sigma_n^2$ (noise level),
                                                                        $\mathbf{x}_*$ (test input)
    $L := \text{cholesky}(K + \sigma_n^2 I)$
    $\boldsymbol{\alpha} := L^T \setminus (L \setminus \mathbf{y})$
    $\bar{f}_* := \mathbf{k}_*^T \boldsymbol{\alpha}$
    $\mathbf{v} := L \setminus \mathbf{k}_*$
    $\mathbb{V}[f_*] := k(\mathbf{x}_*, \mathbf{x}_*) - \mathbf{v}^T \mathbf{v}$
    return $\bar{f}_*$ (mean), $\mathbb{V}[f*]$ (variance)
```

## 2.4 Prediction with Uncertain Inputs

Until now, we have asumed that the input to the GP is deterministic, while the output is Gaussian distributed. This is often true in the case of one step predictions, but if we want to predict several steps forward, we have to use the the stochastic output as input in the next prediction. If we assume a Gaussian distributed input $\mathbf{x}_* \sim \mathcal{N}(\boldsymbol{\mu}, \Sigma)$, this gives the exact predictive distribution

$$p(f(\mathbf{x}_*)|\,\boldsymbol{\mu}, \Sigma) = \int p(f(\mathbf{x}_*)|\,\mathbf{x}_*) p(\mathbf{x}_* \mid \boldsymbol{\mu}, \Sigma)\, \mathrm{d}\mathbf{x}_* \tag{2.18}$$

In general this distribution is non-Gaussian and therefore cannot be computed analytically since the Gaussian distribution input is mapped through a nonlinear function. Instead we will use different methods of approximating this distribution, to be able to propagate the uncertainty. In the following section we will switch notation to that of Hewing and Zeilinger (2017), and use the $z_i = [x_i, u_i] \in \mathbb{R}^{n_z}$, where $n_z = n_x + n_u$, as input to the GP function $d(z_i)$ at time step $i$, with the distribution $d(z_i) \sim \mathcal{N}(\mu^d(z), \Sigma^d(z))$. The mean vector has the states $\mu^d = [\mu_1^d, \dots, \mu_{n_d}^d]$ and covariance matrix $\Sigma^d = \text{diag}(\Sigma_1^d, \dots, \Sigma_{n_d}^d)$, where the mean $\mu_a^d$ and variance $\Sigma_a^d$ are given by equation (2.16).

Instead of using the GP to predict a single state, we will expand to a generalized state space model. As in Hewing and Zeilinger (2017), we will consider control of a dynamical systems of the discrete-time form

$$x_{i+1} = Ax_i + Bu_i + B^d(g(x_i, u_i) + w_i) \tag{2.19}$$

where $g(x_i, u_i)$ is an unknown nonlinear function with uncorrelated process noise $w_i \sim \mathcal{N}(0, \Sigma^w)$. The linear part is usually easy to find, so this part will use the GP function

$d(z_i)$ to estimate the nonlinear function $g(x_i, u_i)$ assumed to lie in the subspace spanned by $B^d$. If we assume an ancillary feedback controller $u_i = K\mu_i^x + v_i$, the distribution of the full state space model is given by

$$\begin{aligned}\begin{bmatrix} x_i \\ u_i \\ d_i \end{bmatrix} &\sim \mathcal{N}(\mu_i, \Sigma_i) = \mathcal{N}\left(\begin{bmatrix} \mu_i^z \\ \mu_i^d \end{bmatrix}, \begin{bmatrix} \Sigma_i^z & \Sigma_i^{zd} \\ S & \Sigma_i^d \end{bmatrix}\right) \\ &= \mathcal{N}\left(\begin{bmatrix} \mu_i^x \\ K\mu_i^x + v_i \\ \mu_i^d \end{bmatrix}, \begin{bmatrix} \Sigma_i^x & \Sigma_i^x K_i^T & \Sigma_i^{xd} \\ K_i\Sigma_i^x & K_i\Sigma_i^x K_i^T & \Sigma_i^{ud} \\ \Sigma_i^{xd^T} & \Sigma_i^{ud^T} & \Sigma_i^d \end{bmatrix}\right)\end{aligned} \tag{2.20}$$

The update equations for the full system can then be found based on linear transformations of Gaussian distributions

$$\mu_{i+1}^x = [A \quad B \quad B^d]\mu_i, \tag{2.21a}$$

$$\Sigma_{i+1}^x = [A \quad B \quad B^d]\Sigma_i[A \quad B \quad B^d]^T \tag{2.21b}$$

## 2.4.1 Approximate Uncertainty Propagation

To be able to define $\mu_i^d, \Sigma_i^d$ and $\Sigma_i^{zd}$ we have to use an approximation of the posterior GP with Gaussian input in equation (2.18). This section will give a short overview, while further details are referred to Deisenroth and Rasmussen (2011), Girard et al. (2002), and Deisenroth (2010). A further comparison of the different methods can also be found in Deisenroth et al. (2015).

**Mean Equivalence Approximation**

The first one is a cheap approach where the uncertainty of the input variable $\mu_i^z = [\mu_i^x, u_i]$ is neglected. This is a computationally cheap method, where the multiplication of the matrices is the most expensive operation giving a one step prediction complexity of $\mathcal{O}(n_d M^2)$. Girard et al. (2002) show that this can lead to poor approximations, as it neglects the accumulation of uncertainty over the prediction horizon.

$$\mu_i^d = \mu^d(\mu_i^z), \tag{2.22a}$$

$$\Sigma_i^d = \Sigma^d(\mu_i^z), \tag{2.22b}$$

$$\Sigma_i^{zd} = 0 \tag{2.22c}$$

**Taylor Approximation**

A first-order Taylor approximation can be applied to $\mu^d$ and $\Sigma^d$ to give the following predicted mean, variance, and covariance. This has a slightly higher complexity of $\mathcal{O}(n_d n_z M^2)$, as it also takes into account the gradient of the posterior mean and variance of the input. Higher order approximations can be done, e.g. second order in Girard et al. (2002), but at the expense of computational cost.

$$\mu_i^d = \mu^d(\mu_i^z), \tag{2.23a}$$

$$\Sigma_i^d = \Sigma^d(\mu_i^d) + \nabla\mu^d(\mu_i^z)\Sigma_i^z(\nabla\mu^d(\mu_i^z))^T, \tag{2.23b}$$

$$\Sigma_i^{zd} = \Sigma_i^z(\nabla\mu^d(\mu_i^z))^T \tag{2.23c}$$

**Exact Moment Matching**

If we assume a zero prior mean function and SEard covariance kernel, it is shown by Girard et al. (2003) that the mean and covariance can be analytically computed by using the first and second moments of the posterior distribution. Since this method match exactly the first and second moments of the posterior distribution. The computational complexity of this method is by Deisenroth (2010) given as $\mathcal{O}(n_d^2 n_z M^2)$. The equations used for the implementation in this thesis can be found in Deisenroth and Rasmussen (2011).

$$\mu_i^d = \mathbb{E}[d(\mathcal{N}(\mu_i^z, \Sigma_i^z))], \tag{2.24a}$$

$$\Sigma_i^d = \mathrm{var}(d(\mathcal{N}(\mu_i^z, \Sigma_i^z))), \tag{2.24b}$$

$$\Sigma_i^{zd} = \mathrm{cov}(\mathcal{N}(\mu_i^z, \Sigma_i^z), d(\mathcal{N}(\mu_i^z), \Sigma_i^z)) \tag{2.24c}$$

In all the approximations, the computational complexity is influenced directly by the input and output dimensions, and the number of training point. this limits the use of model predictive control with Gaussian process to relatively small and slow systems, Kocijan et al. (2004); Maciejowski and Yang (2013). This can however be overcome if the nonlinear dynamics only depend on a subset of states, such that the GP can be evaluated on this subset. For Mean Equivalence and Taylor Approximation, this can easily be seen from (2.22) and (2.23). Evaluating Exact Moment Matching on a subsets of inputs is given by Hewing and Zeilinger (2017).

## 2.5 Sparse Approximation

To account for the increasing computational complexity given by the training data, it is possible to use inducing point to get a sparse approximation of the GP. Sparse approximation has the advantage of potentially quite large reduction in computational cost. The downside is that it depends on getting a set of inducing points that can essentially represent the entire original input space. In the context of MPC this can be partially overcome by using the reasonable assumption that prediction horizon lay within a local region and a known trajectory. This means that inducing points can be chosen locally within this region at each sampling time. Numerous different approximation methods are presented by Rasmussen and Williams (2006), but we will only look closer at Fully Independent Training Conditional (FITC) from Snelson and Ghahramani (2006) and use the notation in Hewing et al. (2017). With the selection of inducing inputs $\mathbf{z}_{ind}$ and shorthand notation $Q^a_{\xi\tilde{\xi}} := K^a_{\xi\mathbf{z}_{ind}}(K^a_{\mathbf{z}_{ind}\mathbf{z}_{ind}})^{-1}K^a_{\mathbf{z}_{ind}\tilde{\xi}}$ the approximate posterior distribution for a state $a$

and input $z$ is given as

$$\tilde{\mu}_a^d(z) = Q_{z\mathbf{z}}^a(Q_{\mathbf{zz}}^a + \Lambda)^{-1}[\mathbf{y}_{\cdot,a}], \tag{2.25a}$$
$$\tilde{\Sigma}_a^d(z) = K_{zz}^a - Q_{z\mathbf{z}}^a(Q_{\mathbf{zz}}^a + \Lambda)^{-1}Q_{\mathbf{z}z} \tag{2.25b}$$

where $\Lambda$ given as a diagonal matrix with the diagonal from $K_{\mathbf{zz}}^a - Q_{\mathbf{zz}}^a + I\sigma_a^2$. Since several matrices can be precomputed, the resulting complexity is independent of the original number of data points. By using $\tilde{M}$ inducing points the computational cost is reduced from $\mathcal{O}(n_d n_z M)$ and $\mathcal{O}(n_d n_z^2)$ for predictive mean and variance respectively to $\mathcal{O}(n_d n_z \tilde{M})$ and $\mathcal{O}(n_d n_z \tilde{M}^2)$.

## 2.6 Validation

### 2.6.1 Validation score

When we have optimized a model it is important to be able to evaluate the quality of the predictions. The simplest and intuitive choice is using the mean square error (MSE)

$$\text{MSE} = \frac{1}{N}\sum_{i=0}^{N}(y_{*,i} - \bar{f}(\mathbf{x}_{*,i})^2) \tag{2.26}$$

where we compute the residual loss between the the predicted value and a test point. As this method is sensitive to the overall scale of the target values, a good practice is to normalize the data and instead use standardized mean squared error (SMSE)

$$\text{SMSE} = \frac{\text{MSE}}{\sigma_y} \tag{2.27}$$

where $\sigma_y$ is the standard deviation in the training data. A low score would then indicate an accurate model, while a higher score would give the opposite.

In addition it is possible to use the predictive distribution to compute the log loss as the negative log probability of the target under the model

$$-\text{log}p(y_*|\mathcal{D}, \mathbf{x}_*) = \frac{1}{2}\text{log}(2\pi\sigma_*^2) + \frac{(y_* - \bar{f}(\mathbf{x}_*))^2}{2\sigma_*^2} \tag{2.28}$$

where $\sigma_*^2 = \mathbb{V}(f_*) + \sigma_n^2$. To get the an estimate of the negative log probability over the whole training set, we can use the mean negative log probability (MNLP)

$$\text{MNLP} = \frac{1}{N}\sum_{i=0}^{N}\frac{1}{2}\text{log}(2\pi\sigma_{*,i}^2) + \frac{(y_{*,i} - \bar{f}(\mathbf{x}_{*,i}))^2}{2\sigma_{*,i}^2} \tag{2.29}$$

A high negative score would with this method equal an accurate model.

### 2.6.2 Validation data

Having methods of evaluating a score is all well and good, but an equally important part of validation is the test data we use to validate the model with. If we used the same data as the training data, we would normally always get a perfect score, and missing the point of validation. That means there are two different strategies for choosing test data. The first and best option is to use a completely different data set than the training data, and use this to validate the hyper-parameters. If it is difficult to get enough test data it is possible to use cross-validation on the training data by using $k$-fold validation. The training data is then divided into $k$ number of subsets, such that one set is left out for validation while the rest $k-1$ sets are used in the optimization. This procedure is then repeated $k$ times with different subsets.

# Chapter 3

# Theory: Model Predictive Control

In control theory, the go-to controller is normally a simple PID controller. Often just for the simple reason that is simple to implement and analyze. And have an intuitive explanation how it works, making it easy to tune. So why bother with an advanced control algorithm like Model Predictive Control? The general motivation behind MPC is the systematic way constraints are handled. Using knowledge of the system dynamics with MPC, we are able to handle long dead time in feedback measurements, or even the lack of measurements using an open loop MPC.

Model predictive control can be used on both continuous and discrete time system, while only the discrete time methods will be used in this thesis. This chapter will also only take a closer look at nonlinear MPC, meaning that all mentions of MPC in this chapter refers to nonlinear MPC. In the case of linear MPC, the same theory, with the difference that linear systems in general are convex, while nonlinear are generally non-convex. This is further discussed in section 3.2.2. One assumption that will be used throughout this chapter is that the all states can be measured directly. This is often not the case, but as this can be accounted for using state estimation, the general theory in this chapter will apply. The main reference in this chapter is the comprehensive MPC book from Rawlings et al. (2017).

## 3.1 Linear Quadratic Regulator

With a linear system $(A, B)$, we wish to find a stabilizing gain $K$ such that $x$ converge to zero as time limit infinity. Using a quadratic cost function $J$ with positive definite penalty matrices $Q$ and $R$, the infinite horizon linear quadratic regulator can be viewed as a linear optimization problem, where the goal is to minimize the cost subject to the input $u$, finding the optimal gain $K$.

$$\min_{u} \quad J = \frac{1}{2}\sum_{k=0}^{\infty} x_k^T Q x_k + u_k^T R u_k \tag{3.1a}$$

$$s.t. \quad x_{k+1} = Ax_k + Bu_k \tag{3.1b}$$

This then leads to the next lemma by Rawlings et al. (2017), where an infinite horizon LQR is convergent for any system that is controllable with positive definite cost matrices.

**Lemma 3.1.** *LQR Convergence. For (A,B) controllable, the infinite horizon LQR with* $Q, R \succ 0$ *gives a convergent closed-loop system*

$$x_{i+1} = Ax + BK_{\infty}$$

Another property of LQR is that the optimal solution of equation (3.1) can be solved analytically by solving the discrete-time algebraic Riccati equation (DARE). The optimal solution use

$$u(x) = Kx, \quad V(x) = \frac{1}{2}x^T P x \tag{3.2}$$

where $K$ is the linear gain, $V(x)$ is the terminal cost function and $P$ the solution to the discrete-time algebraic Riccati equation

$$\begin{aligned} K &= -(B^T P B + R)^{-1} B^T P A \\ P &= Q + A^T P A - A^T P B (B^T P B + R)^{-1} B^T P A \end{aligned} \tag{3.3}$$

Rawlings et al. (2017) shows that for $(A, B)$ controllable and $Q, R \succ 0$, there exists a positive definite solution to the DARE, where the eigenvalues of $(A + BK)$ are asymptotically stable for the corresponding linear gain $K$. This means that the LQR is an optimal controller for the linear system $(A, B)$.

## 3.2 Model Predictive Control

While the LQR is optimal for a linear system, in the case for nonlinear systems $f(x, u)$ we would have to approximate the dynamics with linearization around an operating point. If we stray to far from the operating point, there is no guarantee that the system remains stable. In addition LQR also assumes that we have an unconstrained problem to be able to solve the DARE.

To account for both nonlinear systems and constrained problems, we can generalize equation (3.1) into the minimization problem in equation (3.6) with respect to the decision variables $x$ and $u$. Similar as with the LQR cost we can choose the quadratic stage cost function

$$l(x_i, u_i, \Delta u_i) = x_i^T Q x_i + u_i^T R u_i + \Delta u_i^T S \Delta u_i \tag{3.4}$$

with $Q, R, S \succ 0$, and a Lyapunov function as the terminal cost

$$V_f = x_N^T P x_N \tag{3.5}$$

for a finite prediction horizon of $N$ steps. The optional cost for change in inputs $\Delta u$ can be beneficial to reduce strain on actuators, but not necessary for stability. The control inputs in the prediction horizon $\mathbf{u} = [u_0, u_1, \ldots, u_{N-1}]$ and predicted state trajectories $\mathbf{x} = [x_0, x_1, \ldots, x_N]$ are all constrained in the respective sets $\mathcal{U}$ and $\mathcal{X}$ with the terminal set $\mathbb{X}_f$. The added constraints could lead to an infeasible solution, but as long as a solution is feasible with $x_N \in \mathbb{X}_f$, the system is stable. Further discussion about existence of solution and stability is referred to Rawlings et al. (2017).

$$\begin{aligned}
\min_{\mathbf{x},\mathbf{u}} \quad & V_N = \sum_{k=0}^{N-1} l(x_k, u_k, \Delta u_k) + V_f(x_N) && && \text{(3.6a)}\\
\text{subject to} \quad & x_0, u_{i-1} \quad \text{given} && && \text{(3.6b)}\\
& x_{i+1} = f(x_k, u_k), && i = 0, 1, \ldots, N-1 && \text{(3.6c)}\\
& x_i \in \mathcal{X}, && i = 0, 1, \ldots, N-1 && \text{(3.6d)}\\
& u_i \in \mathcal{U}, && i = 0, 1, \ldots, N-1 && \text{(3.6e)}\\
& x_N \in \mathbb{X}_f && && \text{(3.6f)}
\end{aligned}$$

The above discussion looked into how the optimization problem differs between LQR and MPC. The real difference is however that LQR find an optimal solution to a fixed time window, using a single stabilizing gain for the whole time window, while MPC optimize in a receding time window where the solution is recalculated at each time step, using only the first optimal input as the next control input. MPC often use a smaller window compared to the infinite horizon LQR as optimizing over an infinite horizon is not really tractable. This lead to a possible sub-optimal solution, but has the benefit that it can handle hard constraints in real-time. Because of this finite horizon we are not able to guarantee nominal stability similar to LQR, but has instead a systematic way of handling constraints at each time step. This leads us to the nonlinear MPC algorithm 2, visualized in Figure 3.1. For each time step we get the current state $x_i$ and use this as the initial state in the optimization problem in (3.6). The first control input $u_i^*$ from the optimal solution is then applied as the next control move. This is then repeated throughout the receding control window. This chapter only discuss nonlinear MPC with state feedback, but it is worth mentioning that the difference from linear MPC algorithm is that the nonlinear model (3.6d) is replaced by a linear model. Linear MPC can then be solved using quadratic programming (QP), while NMPC is non-convex and has to be solved using a nonlinear solver. Using state feedback may be a strict requirement, but this step can easily be replaced by output measurements and the added step of state estimation.

### 3.2.1 Dual mode MPC

Infinite horizon LQR guarantee stability for any unconstrained linear problem, while the finite MPC algorithm can stabilize an arbitrary constrained nonlinear problem. The difference to note is that LQR stabilize the system in the infinite horizon, while MPC has a finite control horizon. In dual mode MPC both of these are combined, where the MPC horizon $N$ represent the degrees of freedom for constraint handling and deviation from the

**Algorithm 2:** Nonlinear MPC with state feedback, Foss and Heirung (2016)

```
1: for $t = 1, 2, \dots$ do
2:     Get current state $x_i$.
3:     Solve the optimization problem in (3.6) on the prediction horizon from
       $i$ to $i + N$ with $x_i$ as the initial condition.
4:     Apply the first control move $u_i$ from the solution above.
5: end for
```

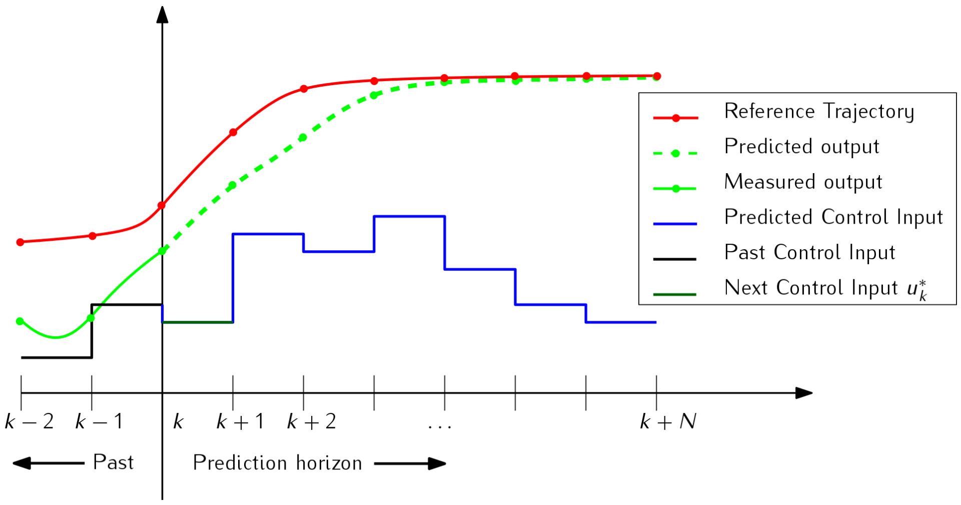


**Figure 3.1:** Model predictive control with prediction and control horizon the same length. The first input $(u_i)$ of the solution is applied as the next control move.

linear system with the optimal input $v_i$, while the LQR feedback $Kx_i$ is used as an ancillary feedback controller. The solution of the DARE, $P$, can then be used as the penalty matrix in the terminal cost function $V_f$ (3.5). After the control horizon in the MPC, the LQR remains as the only control input. Adding feedback instead of using pure open loop predictions have the advantage that we reduce the uncertainty in the predictions. More details is given by Rawlings et al. (1994).

$$u_i = Kx_i + v_i, \qquad i = 0, \dots, N-1 \tag{3.7a}$$
$$u_i = Kx_i, \qquad i \geq N \tag{3.7b}$$

### 3.2.2 Convexity

In optimization the concept of convexity is fundamental. To reach a global minimum it is a requirement that the optimization problem is convex. A convex problem is usually easy to compute, while a nonconvex problem may end up in an infeasible solution, even if a locally feasible point exist. The term convex is used for both sets and functions. A set $S \in \mathbb{R}^n$ is defined by Wright and Nocedal (2006) as convex if it is possible to connect two arbitrary points in $S$ with a straight line segment that lies entirely inside the set, as shown in Figure 3.2. The convexity requirement is then that for all $\lambda \in [0, 1]$, the point $\lambda\mathbf{y} + (1 - \lambda)\mathbf{x}$ has to be contained in $S$ for any two points $\mathbf{x}$ and $\mathbf{y}$ in $S$. A function $f$ is defined as a convex function if its domain $S$ is a convex set, and if for any two points $\mathbf{x}$ and $\mathbf{y}$ in $S$, the following is satisfied:

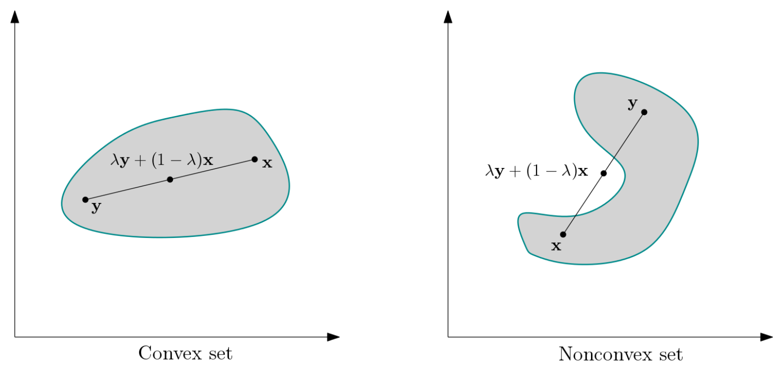
$$f(\lambda\mathbf{x} + (1-\lambda)\mathbf{y}) \leq \lambda f(\mathbf{x}) + (1-\lambda) f(\mathbf{y}), \quad \forall\, \lambda \in [0, 1] \tag{3.8}$$

The objective function can usually be designed to be convex, e.g. a quadratic cost $l(x, u) = x^T Qx + u^T Ru$, with positive definite matrices $Q$ and $R$. The terminal set $\mathbb{X}_f$ can also normally be chosen to be convex, for example the ellipsoid $\mathbb{X}_f = \{x \mid x^T Px \leq 1\}$ for a positive definite matrix $P$. If you are lucky the constraints $\mathcal{X}$ may also be convex, such as a box set on $x_i$ and $u_i$. The system dynamics on the other hand may not always describe a convex set. They are convex if the system model describe a linear system on the form $f(x_k, u_k) = Ax_k + Bu_k + c$, but are generally nonconvex in a nonlinear system.

The assumption is then that linear MPC implies convex linear MPC, while nonlinear usually implies a nonconvex MPC problem, where the convexity is lost due to the nonlinear dynamics. The major problem with nonconvex problems is the case of local minima, where the stability theory require a global minima, leading to sub-optimal MPC. A further discussion on how to handle this is referred to Rawlings et al. (2017).

## 3.3 Stochastic MPC

Until now, we have looked at deterministic systems where we are able to predict the exact behaviour of the system. Noise and inaccurate system models with known disturbance can be accounted for by robust and adaptive methods, while in the case of random disturbance with known probability distributions we have to look to stochastic MPC. In this case the



$\lambda\mathbf{y} + (1-\lambda)\mathbf{x}$
$\mathbf{x}$
$\mathbf{y}$
Convex set
$\mathbf{y}$
$\lambda\mathbf{y} + (1-\lambda)\mathbf{x}$
$\mathbf{x}$
Nonconvex set

**Figure 3.2:** Convex and nonconvex sets

uncertainty is not necessarily bounded within a constrained set, which e.g. is the case of a Gaussian distribution as there exist a probability that any Gaussian distributed function lay anywhere between plus and minus infinity with a given arbitrary mean. Since the random disturbance may be unbounded, it may not be possible to satisfy the constraints $x \in \mathcal{X}, u \in \mathcal{U}$, especially the terminal constraint $x_N \in \mathbb{X}_f$. To account for this Rawlings et al. (2017) propose two different strategies, either replace hard constraints on the form $\phi(x) \in \mathcal{X}$ with average constraints on the form $E[\phi(x) \in \mathcal{X}]$, where $\phi(x)$ represent the state $x$ with a random disturbance. Or in the second case, replace the constraint by a probabilistic constraint on the form

$$P(\phi(x) \in \mathcal{X}) \geq 1 - \epsilon_x \tag{3.9}$$

where $\epsilon \in (0, 1)$ is the probability of violation. To account for rapidly growing uncertainty in the predictions, a recommended approach (Rawlings et al. (2017)) is to use dual mode MPC with an ancillary feedback controller. The LQR feedback will counteract the growing uncertainty in the predictions, while the MPC algorithm optimize over the deviation from the linear feedback control law.

$$\min_{\mathbf{x},\mathbf{u}} \quad \bar{V}_N = \sum_{i=0}^{N-1} \bar{l}(x_i, u_i) + \bar{V}_f(x_N) \tag{3.10a}$$

$$s.t. \quad x_0 \quad \text{given} \tag{3.10b}$$

$$x_{i+1} = f(x_i, u_i) \tag{3.10c}$$

$$u_i = Kx_i + v_i \tag{3.10d}$$

$$P(\phi(x_i) \in \mathcal{X}) \geq 1 - \epsilon_x \tag{3.10e}$$

$$P(u(\phi(x_i)) \in \mathcal{U}) \geq 1 - \epsilon_u \tag{3.10f}$$

$$P(\phi(x_N) \in \mathbb{X}_f) \geq 1 - \epsilon_x \tag{3.10g}$$

for $i = 0, 1, \ldots, N-1$. It should be noted that the stabilizing condition from deterministic MPC with the use of a local Lyapunov function $V_f$ and terminal constraint set $\mathbb{X}_f$ cannot easily be extended to stochastic MPC. Without these conditions the resulting control law $\kappa_f$ from the optimal control problem in (3.10) using a finite horizon $N$ is neither optimal or stabilizing. A further discussion about stability using a local Lyapunov function with unbounded disturbance is referred to Kouvaritakis and Cannon (2016) and Rawlings et al. (2017).

### 3.3.1 Cost function

The constraints in equation (3.6) has in equation (3.10) been replaced by the probabilistic constraints, where we will allow a small probability $\epsilon$ of transgression in the constrains. The other part that has changed is that the cost function has been replaced by the expected value of the cost function.

$$\bar{V}_N(\mathbf{x}, \mathbf{u}) = \mathbb{E}[V_N(\mathbf{x}, \mathbf{u})] \tag{3.11}$$

For a system with Gaussian distributed uncertainty this expected value can be computed analytically for both quadratic cost and saturating cost. We will use the following notation to represent the new cost

$$\bar{V}_f(x_N = c_{(\cdot)}(x_N, P), \tag{3.12a}$$

$$\bar{l}(x_i, u_i) = c_{(\cdot)}(x_i, Q) + c_{(\cdot)}(u_i, R) \tag{3.12b}$$

where $c_{(\cdot)}$ is given as the general cost function.

#### Expected Value of Quadratic Cost

The most common cost function used with MPC, with weighted quadratic cost on state and inputs.

$$c_q(\xi, M) = \mathbb{E}[\|\xi\|_M^2] = \left\|\mu^\xi\right\|_M^2 + \mathrm{tr}(M\Sigma^\xi) \tag{3.13}$$

where $\mu^\xi$ is the mean of a Gaussian distributed vector $\xi$ with a covariance $\Sigma^\xi$. This function is convex in $\mu^\xi$ and $\Sigma^\xi$ as long as $M$ is positive semi-definite. Hewing and Zeilinger (2017)

#### Expected Value of Saturating Cost

For systems with high uncertainty, such as a GP model with little data, having saturation on the cost has been observed by Deisenroth (2010) to benefit exploitation. A generalized saturation cost has been formulated by Hewing and Zeilinger (2017)

$$\begin{aligned} c_s(\xi, M) &= \mathbb{E}[1 - \exp(-\|\xi\|_M^2)] \\ &= 1 - |I + 2\Sigma^\xi M|^{-\frac{1}{2}} \exp(-\left\|\mu^\xi\right\|_S) \end{aligned} \tag{3.14a}$$

$$S := M(I + 2\Sigma^\xi M)^{-1} \tag{3.14b}$$

### 3.3.2 Chance constraints

The probabilistic constraints in equation (3.10) result in general to an intractable optimization problem. For a Gaussian Process with polytopic input and state constraint on the form

$$\mathcal{X} := x \in \{\mathbb{R}^n | H^x x \leq b^x\} \tag{3.15a}$$
$$\mathcal{U} := u \in \{\mathbb{R}^{n_u} | H^u u \leq b^u\} \tag{3.15b}$$

we can however make approximations where Hewing and Zeilinger (2017) show how the joint chance constraints in the shape

$$P(\mathcal{N}(H^x \mu_i^x, H^x \Sigma_i^x (H^x)^T) < b^x) > 1 - \epsilon^x \tag{3.16}$$

can be reduced to conservative individual chance constraints using Boole's inequality:

$$\begin{aligned} &P(\mathcal{N}(H^x \mu_i^x, H^x \Sigma_i^x (H^x)^T) < b^x) \\ &\leq \sum_{j=1}^{p} P(\mathcal{N}([H^x]_{j\cdot} \mu_i^x), [H^x]_{j\cdot}^T) < [b^x]_j) \end{aligned} \tag{3.17}$$

It can further be shown trough equivalence that the individual probabilities can be expressed in terms of the mean $\mu^x$ and covariance $\Sigma^x$

$$\begin{aligned} &P(\mathcal{N}(H^x \mu_i^x, [H^x]_{j\cdot} \Sigma_i^x ([H^x]_{j\cdot})^T) < [b^x]_j) \geq 1 - \epsilon_j^x \\ &\Leftrightarrow [H^x]_{j\cdot} \mu_i^x + r_j^x \sqrt{[H^x]_{j\cdot} \Sigma_i^x [H^x]_{j\cdot}^T} \leq [b^x]_j \end{aligned} \tag{3.18}$$

where $r_j^x$ represent the quantile function $\Phi^{-1}(1 - \epsilon^x)$ of the standard normal distribution. Similarly the same result can be shown for the input chance constrains

$$[H^u]_{j\cdot} (K \mu_i^x + v_i) + r_j^u \sqrt{[H^u]_{j\cdot} K \Sigma_i^x K^T [H^u]_{j\cdot}^T} \leq [b^u]_j \tag{3.19}$$

In the case of Mean Equivalent or Taylor Approximation, Hewing and Zeilinger (2017) also show how it is possible to decompose the covariance matrix $\Sigma_i^x = L_k L_k^T$ using Cholesky decomposition, such that the chance constraints can be expressed as

$$[H^x]_{j\cdot} \mu_i^x + r_j^x \, \| [H^x]_{j\cdot} L_i^x \|_2 \leq [b^x]_j, \tag{3.20a}$$
$$[H^u]_{j\cdot} (K \mu_i^x + v_i) + r_j^u \, \| [H^u]_{j\cdot} K L_i^x \|_2 \leq [b^u]_j, \tag{3.20b}$$

For probabilities of violations under 50%, i.e. $r_j^x, r_j^u > 0$, these formulations are convex second-order cone constraints.

Chapter 4

# Learning the Dynamic Model

In chapter 2 we got an introduction on the theory behind Gaussian Process, and how the model is affected by different hyper-parameters and number of data points. The model has good accuracy close to sampled data points, while further away will reduce the model to the prior. More points will increase the chance of an unseen point having a neighboring data point, and thus the desire of having as many points as possible. The naive approach is then to brute force and sample all you can get of random data points and hope for the best. This chapter will look closer at why this may not the best suited method. We will look at different sampling strategies, and how the stability properties of the dynamic model itself may play a part in the considerations when sampling data for the model.

## 4.1 Acquisition of new Observations

The method we will look at is based on the idea that we need a good spread of data points for a good enough representation of the real dynamics. This leads us to design of experiments, Sacks et al. (1989), where we use methods for cherry-picking the samples we need to get the necessary representation. One of such methods is Latin Hyper-cube, where the goal is to maximize the minimum distance between points. An example of a Latin Hyper-cube distribution with a two dimensional problem is given in Figure 4.1 with 50 points distributed in the plane between $x_1$ and $x_2$. In Figure 4.2 we try to estimate the same sine curve as in chapter 2 with different observation distributions. All four distributions use 10 points to estimate the same signal and the same hyper-parameters, with one random distribution, two distributions with manually selected best and worst case points, and one with Latin Hyper-cube. Using a random uniform distribution would be a good initial guess on how to sample the observations, but as seen in the example in Figure 4.2a, the random sampling resulted in clustering of points, representing only a small section of the signal. Using a deterministic uniform selection instead avoids possible clustering. Since we are estimating a stationary signal, a uniform can give a quite good result as seen in Figure 4.2d with the observations chosen at the minima and maxima of the sine curve. On the other hand, if the uniform selection is slightly offset to only sample the midpoints at zero, the

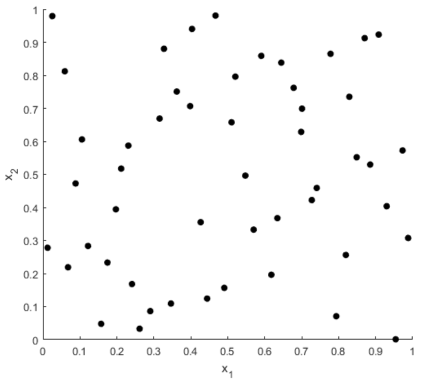


**Figure 4.1:** Latin Hypercube with 50 samples distributed maximizing the minimum distance between points in a two dimensional problem.

resulting prediction is not surprising only zeros as seen in Figure 4.2b. Manually selecting points is not desirable, and not really tractable in higher dimensions, so even if figure 4.2c is not the optimal solution, it result in an adequate prediction, where Latin Hyper-cube give a random distribution without clustering points.

Latin Hyper-cube guarantee that there is an even spread of data. This may look good, but the actual goal is to get the best possible representation of the real system. Disadvantages to take into account is then that you may end up sampling points that are unnecessary. Certain sections in the hyper-cube representing fast dynamics can also end up having too few samples, with points wasted on sections with slower dynamics. It is tempting to just increase the number of observations, but that will also increase the computation cost. Given a system with $N$ states, we would need to distribute enough points to fill an $N$ dimensional hyper-cube, resulting in $x^N$ number of points.

## 4.2 Stability of prediction

Earlier in section 2.2.3 we learned that that the accuracy of the model prediction is determined by the number of observations, and how well the hyper-parameters fit the data. This was further explored in section 4.1 where also the spread in the data had a huge impact on the prediction. This section will look closer at another factor that may affect the accuracy. Section B.2.1 looked at the stability of different integrator methods. The stability of these methods are determined by the stability properties of the dynamic system and the accuracy of the integrator. The same should hold for the GP, where a slight model error could make the prediction unstable.

To explore this we will look at a simple dynamic model, and only change one parameter to adjust the stability properties. The Van der Pol equation (4.1) represent an unstable system with a limit cycle, where the parameter $\mu$ determine the eigenvalues, and thus the

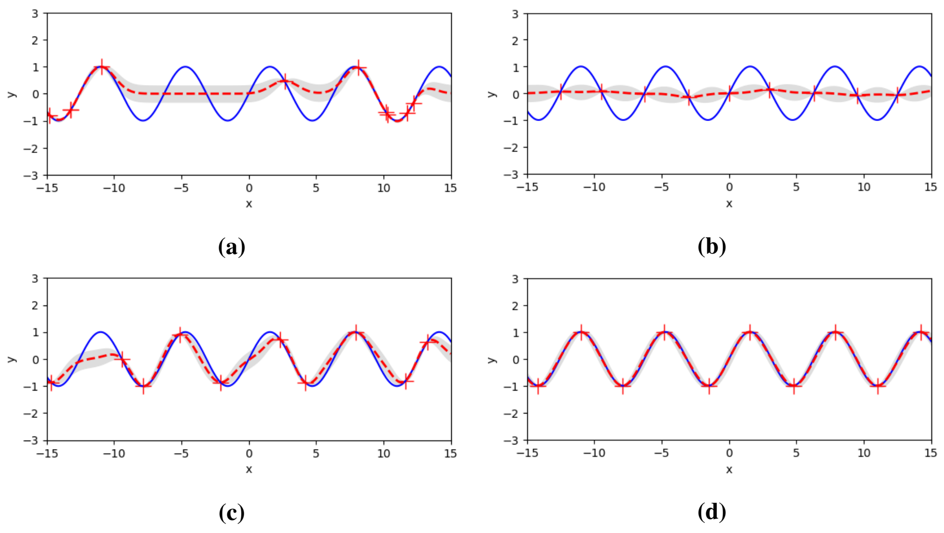


**Figure 4.2:** Result of using different distributions of better or worse observations. All have the same hyper-parameters and 10 data points. a) has a random uniform distribution, b) has a worst case uniform distribution, c) has data points distributed using Latin Hypercube design, while d) has observations chosen manually to cover the min/max of both the inputs and outputs.

stability properties.

$$\dot{x} = y, \tag{4.1a}$$

$$\dot{y} = -x + \mu(1 - x^2)y \tag{4.1b}$$

To test the stability of the GP in an open loop system we will look into three cases where we adjust the parameter $\mu$. Van der Pol is interesting in the case where it has a limit cycle, such that any solution that starts on the limit cycle will stay on it for all time Jordan and Smith (2007). In Figure 4.3 we see a stability analysis of the system, where the solution will converge towards the limit cycle with an arbitrary initial point if the parameter $\mu = 2$. If the parameter sign is changed to a negative, the whole system is reversed, such that the system behaves as if the time $t$ was reversed. The former limit cycle is then changed to the limit between diverging, or converging towards origin which is a degenerated stable node. With $\mu = 0$ the system has pure imaginary eigenvalues and behaving as a centre. Without the damping function it is a simple harmonic oscillator where all the energy is conserved. It is the same system with equality point a the centre, where the only difference is that the stability properties of the system changes by adjusting the parameter $\mu$.

In Figure 4.4 these three systems are estimated using a Gaussian process. The GP is trained using 40 data points, sampled from Latin Hyper-cube Design Sacks et al. (1989), at a sampling time of $0.01s$. The simulated data points is added Gaussian noise with a variance of $10^{-6}$. The testing set is designed the same way, also with 40 data points. All GP models have a prediction horizon of 2000 steps using Mean Equivalence, neglecting the propagated uncertainty.

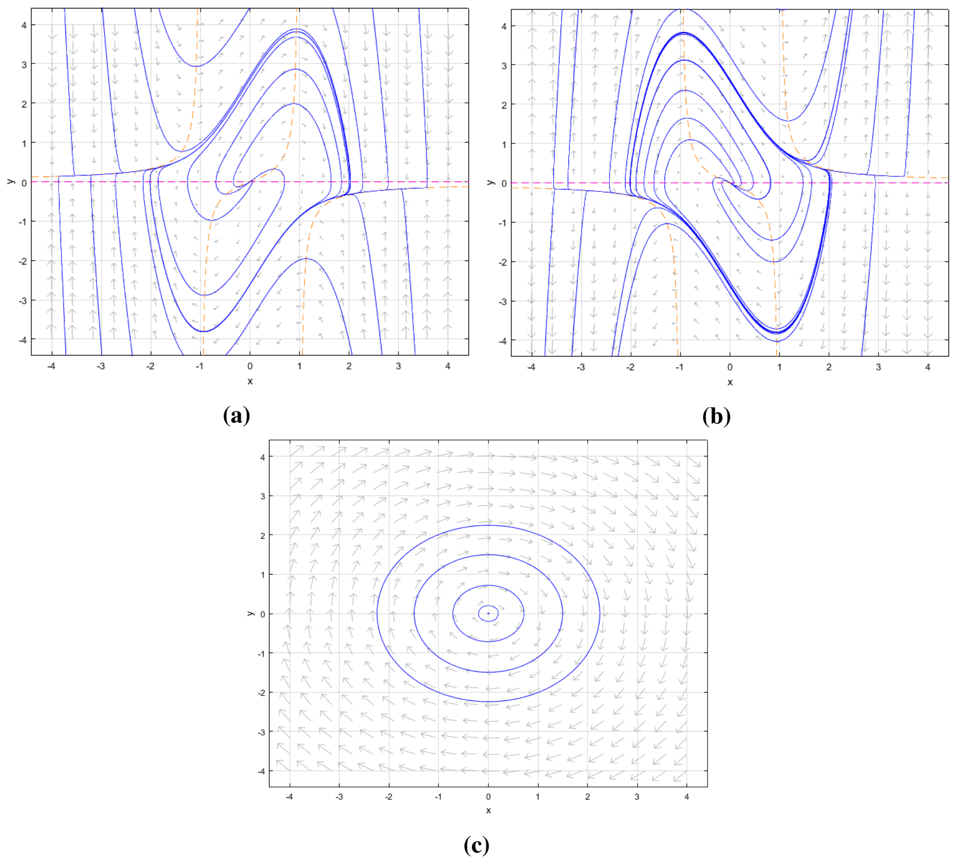
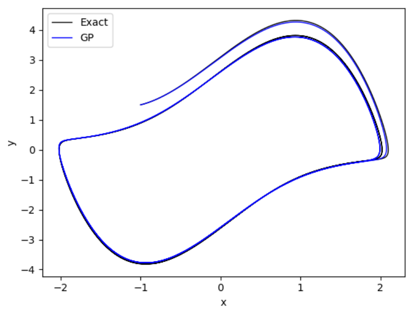


**Figure 4.3:** Stability analysis with the phase plots showing the vector fields of the Van der Pol equation (4.1). a) $\mu = 2$, b) $\mu = -2$, and c) $\mu = 0$

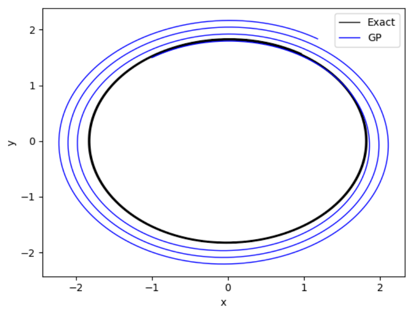
The GP predicts the limit cycle in Figure 4.4a quite well, without any noticeable deviation. In 4.4a the GP is struggling more, with a clear deviation from the real solution. In Figure 4.4b we would expect to see the GP in a constant circle, but because of small perturbations from the real system it diverge instead. The GP was robust when estimating the limit cycle, but struggles more with the other two systems.

This example highlights the problem of only looking at the SMSE as validation. The score is well suited to trow away results that don't fit the model, but with a system that is close to the stability region, even small errors less than the added noise can cause small perturbations that will make the prediction unstable. It is also worth noting that the prediction horizon of 2000 steps is an exaggeration in the context of model predictive control. The horizon was chosen to show the dynamics of the stability region, and also to show the strength of the GP when used on a system that don't diverge as in the case of Figure 4.4a.

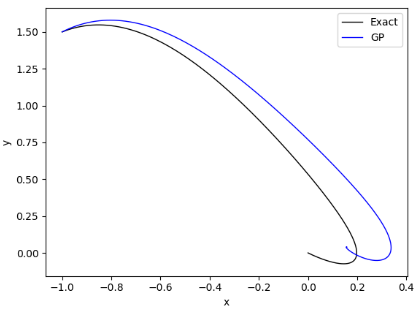
How much each GP would diverge from the real system is dependant on the training data along the path from the initial point to the stable equilibrium or limit cycle. More data



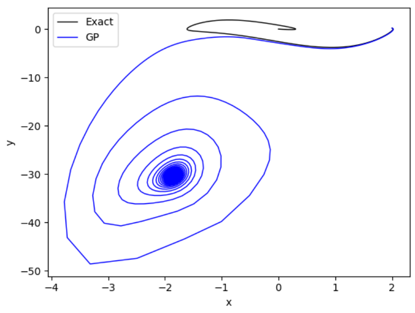
**(a)**



**(b)**



**(c)**



**(d)**

**Figure 4.4:** Van der Pol equation (4.1), with initial point $x_0 = [-0.5, 0.5]^T$. In a) $\mu = 2$, b) $\mu = 0$, c) $\mu = -2$ and d) $\mu = -2$ where the initial point was chosen to be at the border of the stability region. In all the cases the standardized mean squared error is of the order of $10^{-5}$ or less, tested against 40 unseen samples. All GP models have a prediction horizon of 2000 steps using Mean Equivalence, neglecting the propagated uncertainty.

**Table 4.1:** Standardized mean squared error for different parameter values, corresponding to the models in Figure 4.4.

| | | $x$ | $y$ |
|---|---|---|---|
| a) | $\mu = 2$ | 0.000000 | 0.000001 |
| b) | $\mu = 0$ | 0.000000 | 0.000000 |
| c) | $\mu = -2$ | 0.000003 | 0.000010 |
| d) | $\mu = -2$ | 0.000000 | 0.000010 |

points in the prior results in a closer fit. The plots in Figure 4.4 represents a typical result.

In the Figure 4.4d the initial point was chosen at the border of the stability region, seen in Figure 4.3b. A small deviation from the real model in the order of $10^{-5}$ is enough that the GP leaves the stability region and diverge. This figure also highlights the effect where the GP has different stability properties than the real system. While the van der Pol equation has only one equality point in the origin, the GP has a stable spiral close to $(-2, -30)$.

It is interesting to note that the model used in Figure 4.4b has a slightly lower SMSE score than the model in Figure 4.4a, but still performs worse. Note also that both c) and d) have worse SMSE score than a) and b) with the same number of training data, where it seems it is easier to fit a stable system.

## 4.3 Online Learning

In the previous sections we have looked at how to acquire data points that best represent our model using Latin Hypercube with the principle that we need a good spread in the data. The challenge of using this method is that we are required to actually sample the output from the optimal states given either by the Latin Hypercube. This works fine in computer simulations, but is most often not a tractable option for real life measurements as it would require us to be able to have complete control over every possible state. If this where the case, using a Gaussian Process in the control loop would be a little unnecessary. This means that we instead have to measure data and then find the most suitable points that can represent the model. In this section present three different strategies for online learning and adaption.

#### Learn model

The first method is the simplest where the strategy is to add measurement data to the training set and optimize the hyper-parameters based on all collected data, validate the model based on a different set of measurement data, and repeat until a satisfactory model is gained. It is possible to add all measurement data to the training data collection, but this quickly leads to a prohibitive model size for use in control methods. If this naive approach doesn't give a satisfactory validation result within a maximum number of points, another approach is to use Bayesian Optimization, Brochu et al. (2010), where each new point is chosen based on the acquisition function by maximizing the expected utility using state-space subset consisting of the collection of new measurement data, testing each new point to find the points with highest utility. Another simple approach is to just use exploration by using the points with the highest predicted variance. As re-computing the hyper-parameters with an increasing number of data points quickly reach a computation cost that is prohibitive for real-time control, this methods is best suited for offline system identification. For adaptation of system changes in real-time the following is more suited.

### Update model

The observant reader would by now have noticed by the equations in chapter 2 that training data is used directly for predicting the mean and variance, as opposed to e.g. an artificial neural network where the training data is used to train weights. We use the training data to optimize the covariance function hyper-parameters, but as discussed earlier, having a representative collection of training data is the most important part of the GP model. Where the hyper-parameters are used to get the optimal interpolation between the sample points. This means that if have optimized the hyper-parameters well enough, we could in theory be able to update out model with new data without having to re-optimize the hyper-parameters. Depending on the size of the model, optimizing hyper-parameters could take several minutes on larger problems, while updating the training data is only a matter of updating the Gram matrix where the heaviest operation is to inverse the Gram matrix. Compared to a nonlinear optimization problem, this is a significantly easier problem, where several methods are available to efficiently update the Gram matrix without having to recompute the entire matrix and its inverse by using low rank updates of the Cholesky decomposition of the Gram matrix. See Appendix A for more details. This has a computational cost that enable us to use the update with real-time control. The new update data can be chosen the same way as with the previous method, either using all data or cherry-picking the optimal points that give the best expected probability of improvement. It is important to note that if the hyper-parameters are not trained on a reasonably representative collection of points, adding new data can actually give a worse performance. Meaning that this method is most suitable to update or replace training data on an existing model with reasonable hyper-parameters.

Example of adding random data as update is showed in Figure 4.5. Similar to the previous example the goal is to estimate the dynamics in the Van der Pol equation (4.1). The difference in this example is that we only use 5 samples to optimize the hyper-parameters and use an additional 50 samples to update the training data without updating the hyper-parameters. Using only 5 samples give a quite bad fit, as observed in Figure 4.5a, as there is not enough data to represent the model. While adding 50 new samples and updating only the Gram matrix give a more reasonable fit in 4.5b, even though the hyper-parameters where trained on a quite small data set. This show that it is reasonable to only update the training data without re-optimizing the hyper-parameters. With the remark that it is important that the hyper-parameters are trained on a set that reasonably represent the system. Using only 5 samples is in general too few to represent this type of system, so it was only by chance that the hyper-parameters where well fitted in Figure 4.5a. Using a different set of random data again give a quite different result in Figure 4.5d where the hyper-parameters was trained on a sub-optimal set of sample points in Figure 4.5c.

So far we have covered how to update the model to fit an existing system. If however the system changes due to parameter perturbations, e.g. the center of gravity in a car model has changed, it would not be advisable just update the existing data with new measurements. A GP model can only fit one system at a time, but as long as the perturbations are not to large it is possible to use the same hyper-parameters, while all the training data can be replaced with new measurements. In Figure 4.6 we look at the Van der Pol system again with the perturbation from $\mu = 2$ to $\mu = 5$. From Figure 4.5d it is obvious that we cannot use data sampled with $\mu = 2$ to predict a system where $\mu = 5$, they are two

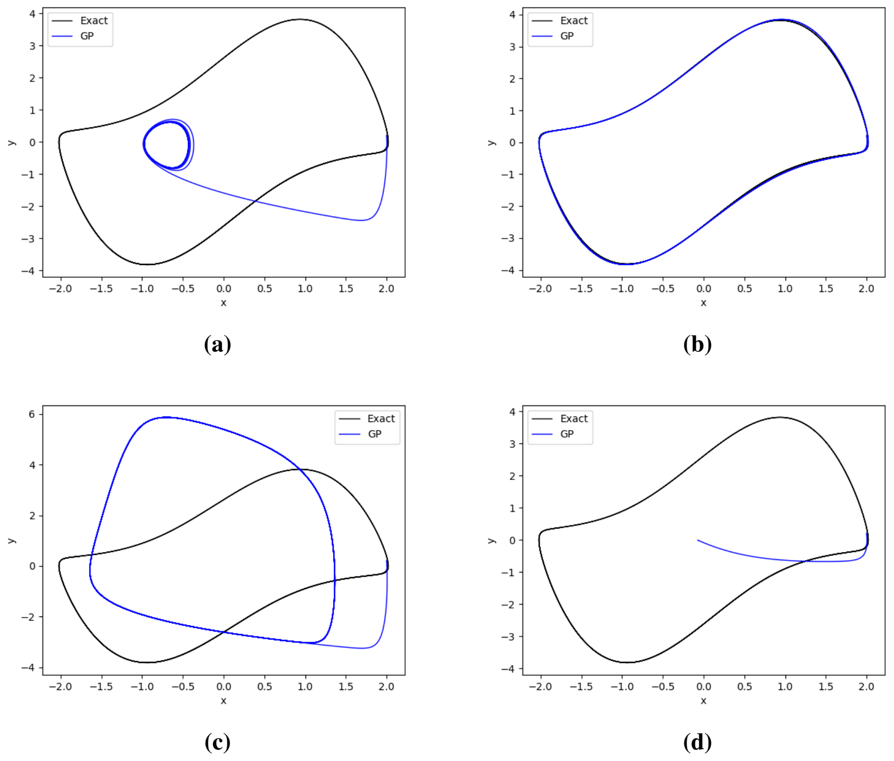
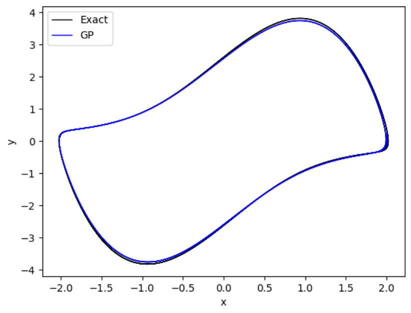


**Figure 4.5:** The GP model in a) has optimized hyper-parameters using 5 samples from the Van der Pol equation in (4.1), while b) use the same hyper-parameters as in a) and update the Gram matrix with 50 new samples. The same is repeated in c) where the hyper-parameters are optimized on a different set of 5 samples, while d) use the hyper-parameters from c) and updating the Gram matrix with 50 new samples, giving a quite different result.

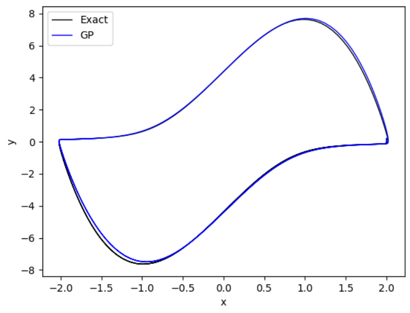
different systems with different dynamics. Using 40 samples on a system with $\mu = 2$ to train the hyper-parameters give a good result in Figure 4.6a as we try to predict the same system. If we then replace all the training data with 40 new measurements from a system with $\mu = 5$ in Figure 4.6a, we are able to re-use the hyper-parameters and still give a good result.

## Adapt model

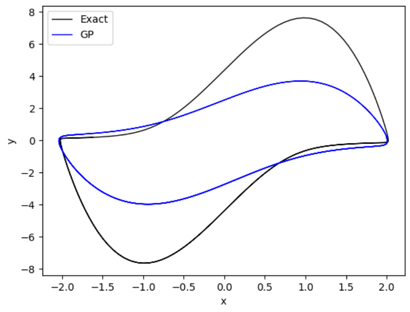
In complex systems with high nonlinearity, there pose a problem with finding a collection of points that give a good enough representation of the system, at the same time that it is possible to compute the predictions in real time. Increasing the number of sample points increase the accuracy of the model, but at a computational burden that can be prohibitive. This leads us to the use of sparse GP models, where a small collection of inducing points can be used to represent the model. Finding such points can be difficult, but Hewing



**(a)**



**(b)**



**(c)**

**Figure 4.6:** If the system has small perturbations from the original system it is possible to replace all the training data without re-optimizing the hyper-parameters. This example use the Van der Pol equation again with $\mu = 2$ in a) and $\mu = 5$ in b). The hyper-parameters are trained on 40 samples in a), where b) replace these samples with 40 new points. As a comparison the GP model in c) has training data where $\mu = 2$, while the simulated system has $\mu = 5$.

et al. (2017) claim that any random collection of measurements are valid locally at each sampling time. The idea is that the area of interest at each sampling time lay close to a known trajectory in the stat-action space. This let us train the hyper-parameters on a large collection of data, while the online mean predictions and model adaption only us a small subset consisting of the inducing points. In Hewing et al. (2017) they used 300 measurement points to estimate the error from the dynamic model, while only 10 inducing points was chosen at each sampling time, based on the first 10 points in the MPC trajectory solution.

Chapter 5

# Control of a Four-Tank System

This section will begin to look closer at how to use a Gaussian Process together with MPC. As a case study we will use a well-known example in MPC literature with a system consisting of four connected water tanks, where the purpose is to control the water level in all the tanks. This is a system with relatively slow nonlinear dynamics with a limited and known range, offering good conditions for training a GP model. This chapter will then look at how we can use a GP model to estimate the tank system, and use this as prediction to control the water level with a model predictive controller.

## 5.1 Tank System Model

This system is a widely used example of MPC application, where we will use the system formulation and parameters from Raff et al. (2006). The system consists of four tanks, two valves, and two pumps that are interconnected as shown in Figure 5.1. Both valves are three way valves, where we will assume that the valve openings are constant, with parameters given as $\gamma_1 = \gamma_2 = 0.4$. Tank one and four are supplied with water from pump 1, while pump 2 supply tank 2 and three. Tank three and four then have outlets that fill up respectively tank one and two, which again have outlets to a sink. The system equations are then given by Raff et al. (2006) as

$$\dot{x}_1 = -\frac{a_1}{A_1}\sqrt{2gx_1} + \frac{a_3}{A_1}\sqrt{2gx_3} + \frac{\gamma_1}{A_1}u_1 \tag{5.1a}$$

$$\dot{x}_2 = -\frac{a_2}{A_2}\sqrt{2gx_2} + \frac{a_4}{A_2}\sqrt{2gx_4} + \frac{\gamma_2}{A_2}u_2 \tag{5.1b}$$

$$\dot{x}_3 = -\frac{a_3}{A_3}\sqrt{2gx_3} + \frac{(1-\gamma_2)}{A_3}u_2 \tag{5.1c}$$

$$\dot{x}_3 = -\frac{a_4}{A_4}\sqrt{2gx_4} + \frac{(1-\gamma_1)}{A_4}u_1 \tag{5.1d}$$

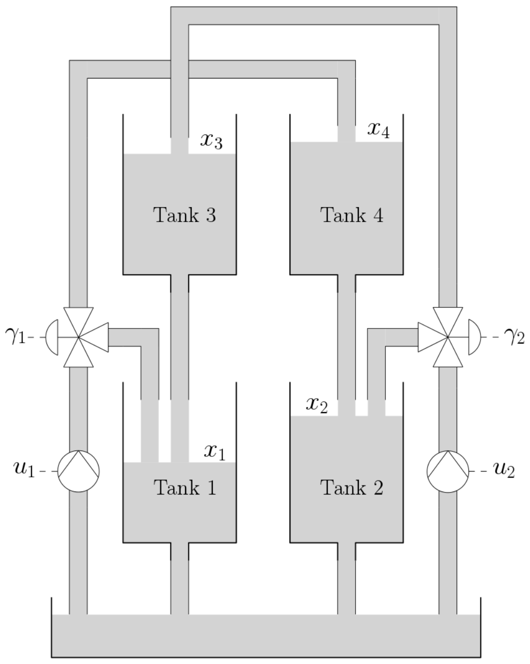


**Figure 5.1:** Tank system based on Figure 2 in Raff et al. (2006). The valve opening $\gamma_1$ and $\gamma_1$ is set at a fixed value, while the only actuation is from the pumps $u_1$ and $u_2$.

where $g$ is the gravity, $A_i$ is the cross-section of tank $i$, and $a_i$ is the cross-section of the outlet hole of tank $i$, with parameters given in table 5.1.

**Table 5.1:** Model parameters from Raff et al. (2006)

| | $A_i$ | $a_i$ |
|---|---|---|
| $i = 1$ | 50.27 $cm^2$ | 0.233 $cm^2$ |
| $i = 2$ | 50.27 $cm^2$ | 0.233 $cm^2$ |
| $i = 3$ | 28.27 $cm^2$ | 0.127 $cm^2$ |
| $i = 4$ | 28.27 $cm^2$ | 0.127 $cm^2$ |

## 5.2 Learning Dynamics

To estimate the system model we will use a single GP model for all the tanks as a single system. The full GP model will then have four states and two control inputs, analog to (5.1). The model will use a zero mean function, and a SEard covariance kernel, optimized using 60 data points sampled using Latin Hyper-cube with a sampling time of $\Delta = 3$ seconds. Measurement noise with a Gaussian distribution of $\mathcal{N}(0, \text{diag}([10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}]))$

is added to the samples. Validation of the model is also done using Latin Hyper-cube with 100 samples.

To propagate the uncertainty we will use first order Taylor Approximation given by equation (2.23), such that for each time step we estimate both the mean $\mu^x_{i+1}$ and the state covariance $\Sigma^x_{i+1}$, given the input mean $\mu^z_i = [\mu^x_i; u_i]$ and input covariance

$$\Sigma^z_i = \begin{bmatrix} \Sigma^x_i & \Sigma^x_i K^T \\ K\Sigma^x_i & K\Sigma^x_i K^T \end{bmatrix} \tag{5.2}$$

where $K$ is the linear gain in the ancillary LQR feedback controller.

## 5.3 MPC formulation

The MPC formulation in equation (3.10) written in terms of propagated uncertain states and inputs. The numerical optimization use simultaneous multiple shooting such that the predicted means $\mu^x$ and covariance $\Sigma^x$ is minimized together with the optimal inputs $v$.

$$\begin{aligned}
\min_{v,\mu^x,\Sigma^x} \quad & V_f(\mu^x_N - x_{\text{ref}}, \Sigma^x_N) + \sum_{i=0}^{N-1} l(\mu^x_i - x_{\text{ref}}, \Sigma^x_i, v_i) && \text{(5.3a)}\\
s.t. \quad & \mu^x_0 = x_0 && \text{(5.3b)}\\
& \Sigma^x_0 = \mathbf{0} && \text{(5.3c)}\\
& \mu^x_{i+1}, \Sigma^x_{i+1} \quad \text{acc. to (2.23)} && \text{(5.3d)}\\
& u_i = K\mu^x_i + v_i && \text{(5.3e)}\\
& P(x_i \in \mathcal{X}) \geq 1 - \epsilon \quad \text{acc. to (3.20)} && \text{(5.3f)}\\
& P(u_i \in \mathcal{U}) \geq 1 - \epsilon \quad \text{acc. to (3.20)} && \text{(5.3g)}\\
& && \text{(5.3h)}
\end{aligned}$$

Both the stage cost $l(\cdot)$ and terminal cost $V_f(\cdot)$ use the expected value of quadratic cost (3.12). The constraint sets $\mathcal{X}$ and $\mathcal{U}$ are given by the input and state limits in table 5.2. The linear gain $K$ for the LQR feedback is calculated using (3.3) with the linearized GP model using the initial state $x_0$ as operating point and the same cost matrices $Q$ and $R$ as the MPC problem. The terminal cost $P$ is given as the solution of the DARE (3.3). Both prediction and control horizon is set at 30 steps.

## 5.4 Simulation

To simulate the experiment we will use the GP-MPC framework developed for this thesis. The underlying backbone is CasADi, Andersson et al. (2018) for automatic Algorithmic differentiation, together with CVODES, Hindmarsh et al. (2005), for simulation, IPOPT, Wächter and Biegler (2006), as the nonlinear solver in the optimal control problem, and ma27 (HSL) as the linear solver. The prediction horizon is 30 steps, where the prediction is done using GP Taylor Approximation, and RK4 for comparison. The system is then simulated for 250 seconds, equivalent to 80 control moves. The parameters for the control

**Table 5.2:** Control problem parameters.

| Param | Value | Units | |
|---|---|---|---|
| $Q$ | diag($[10, 10, 1, 1]$) | - | State penalty matrix |
| $R$ | diag($[10^{-3}, 10^{-3}]$) | - | Input penalty matrix |
| $S$ | diag($[10^{-2}, 10^{-2}]$) | - | Change in input penalty matrix |
| Cov($w$) | diag($[10^{-5}, 10^{-5}, 10^{-5}, 10^{-5}]$) | - | Noise covariance matrix |
| $\epsilon$ | 0.05 | - | Probability of violation |
| $\Delta t$ | 3 | s | Sampling time |
| $H_c$ | 30 | - | Number of steps in control horizon |
| $H_p$ | 30 | - | Number of steps in prediction horizon |
| $x_0$ | $[8.0, 10.0, 8.0, 19.0]$ | cm | Initial state |
| $x_{\text{ref}}$ | $[14.0, 14.0, 14.2, 21.3]$ | cm | Desired set point |
| $x_{\text{min}}$ | $[7.5, 7.5, 3.5, 4.5]$ | cm | Lower tank limit |
| $x_{\text{max}}$ | $[28., 28., 28., 28.]$ | cm | Upper tank limit |
| $u_{\text{min}}$ | $[10.0, 10.0]$ | ml/s | Minimum input |
| $u_{\text{max}}$ | $[60.0, 60.0]$ | ml/s | Maximum input |

problem is listed in table 5.2. All simulations are done on a laptop from 2011, with i5 processor and 8GB of memory.

## 5.5 Results

### Model Learning

The hyper-parameters optimized for the GP model used is given in table 5.3, with the validation score in table 5.4. The estimated noise is slightly to naive, except in the case of Tank 1 where the noise variance is matched exactly to the real measurement noise. Note also how the length-scales fit the different dimensions according to equation (5.1). $\dot{x}_1$ is dependent on $x_1, u_1, x_3$, which again depend on $u_2$, which is reflected on the length-scales. A lower value mean that that input dimension has a higher weight on in the prediction of state. In this case the hyper-parameters had a lower bound at 10. This should have been lower, but since the validation score gave a good result, ths models was chosen. The validation is based on 100 random samples from Latin Hypercube, where the result is given by table 5.4. For validation score we use both standardized mean square error (SMSE) and mean negative log probability (MNLP) as two different measures of model quality. Both give the indication that tank 1 has the worst accuracy, while the other tanks has relatively similar scores. The uncertainty is also too naive where the 95% confidence interval don't enclose the actual signal in tank 4. In the open loop prediction in Figure 5.2 we that in this combination of state and inputs case tank four has the highest model deviation. The inputs are fixed at $u = [45, 45]$ with the initial state $x_0 = [8.0, 10.0, 8.0, 19.0]$, with an open loop prediction horizon of 30 steps. In all the predictions both Expected Moment matching (EM) and Taylor Approximation (TA) performs equally with overlap on both

**Table 5.3:** Hyper-parameters fitted for the tank model using 60 normalized samples.

| | $\sigma_f^2$ | $\ell_{x_1}$ | $\ell_{x_2}$ | $\ell_{x_3}$ | $\ell_{x_4}$ | $\ell_{u_1}$ | $\ell_{u_2}$ | $\sigma_n^2$ |
|---|---|---|---|---|---|---|---|---|
| $x_1$ | 4.60 | 10.0 | 29.81 | 14.58 | 1878 | 26.02 | 27.15 | $1.09 \cdot 10^{-5}$ |
| $x_2$ | 6.88 | 36.10 | 10.0 | 44.31 | 10.0 | 24.13 | 65.36 | $6.77 \cdot 10^{-6}$ |
| $x_3$ | 4.97 | 28.58 | 41.87 | 10.0 | 26.94 | 24.44 | 26.25 | $6.71 \cdot 10^{-6}$ |
| $x_4$ | 4.97 | 29.30 | 21.87 | 31.49 | 10.0 | 30.5 | 41.65 | $6.29 \cdot 10^{-6}$ |

**Table 5.4:** Validation of the hyper-parameters in table 5.3, using 100 randomized samples produced by Latin Hypercube design.

| | SMSE | MNLP |
|---|---|---|
| $x_1$ | $5.7 \cdot 10^{-5}$ | $-3.606$ |
| $x_2$ | $1.8 \cdot 10^{-5}$ | $-4.074$ |
| $x_3$ | $2.4 \cdot 10^{-5}$ | $-4.131$ |
| $x_4$ | $2.0 \cdot 10^{-5}$ | $-4.010$ |

the mean and variance, while Mean Equivalence (ME) have no propagation of uncertainty, with the result that it is not able to catch the propagated model error in the predictions.

In Figure 5.3 we see the eigenvalue plots for both the real discrete system and the GP model. To find the eigenvalues we linearized both the discrete model equations and the GP model using $x_0 = [8., 10., 8., 19.]$ and $u_0 = [45, 45]$ as the linearized operating point. With this we are able to get an estimate of the stability properties, where the open loop system is stable if all eigenvalues is enclosed by the unit circle and unstable otherwise. This is also another way validating the model, where similar eigenvalues would indicate similar dynamics. In this case we see that both the linearized discrete model and the linearized GP model have eigenvalues that are close, where the GP model has eigenvalues that are slightly shifted along the real axis, resulting in one destabilizing pole.

### Control

In Figure 5.4 and 5.5 we see the result of solving the MPC problem in 80 iterations, using respectively RK4 and the GP model. If we use RK4 as the baseline we see that the GP model performs quite well with similar results as RK4. In the penalty matrix $Q$ we put a greater cost on tank one and two, which is observed in both the RK4 and GP model. There is an overshoot in both the upper tanks in the beginning, that again can fill the lower tanks. In Figure 5.4 we see that the first optimal state trajectory is followed closely. With negligible noise levels this is to be expected, as there is no new information to be gained in in future states if the first iteration gave the true optimal solution with an accurate prediction. In both RK4 and GP there is about 1mm deviation from the set point in all the tanks, where both model have some oscillation around the set point. The GP model has as shown earlier a small model error that we can recognize in Figure 5.5. Tank two and three follow the read model closely, while there is some deviation from the first prediction in tank one and four from the measured levels. The deviation from thre true

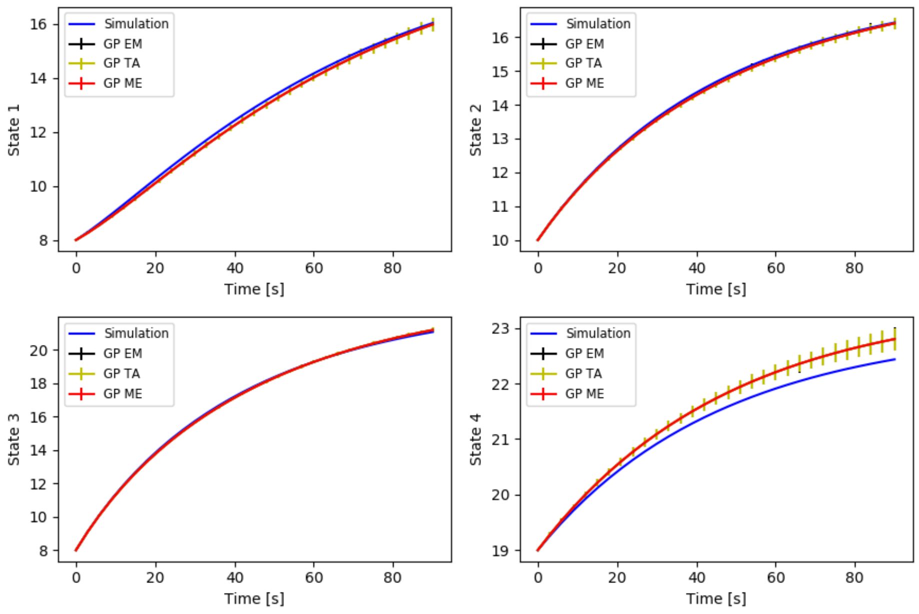


**Figure 5.2:** Open loop prediction of GP model, comparing Expected Moments (EM), Taylor Expansion (TA) and Mean Equivalence (ME) for uncertainty propagation. The prediction horizon is 30 steps with a fixed input $u = [45, 45]$ and initial state $x_0 = [8.0, 10.0, 8.0, 19.0]$. The vertical bar represent the 95% confidence interval of the uncertainty. Both EM and TA give similar results with increased uncertainty, while ME has no uncertainty propagation

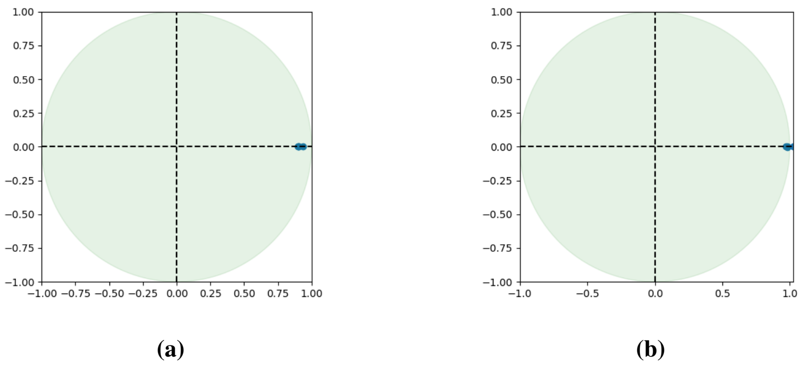


**Figure 5.3:** Eigenvalue plot for the linearized discretized dynamic model in a) and the linearized GP model in b). Both use $x_0 = [8., 10., 8., 19.]$ and $u_0 = [45, 45]$ as the linearized operating point. Both models have similar eigenvalues with the exception that the GP model has one destabilizing pole.

model in all the tanks is covered by the 95% confidence interval.

### 5.5.1 Computational cost

Both predictions with RK4 and GP relatively similar predictions performance, so it is also relevant to look at the computational cost of each of these. An explicit Runga-Kutta solver is a cheap way of integrating the system, with a worst case of 376ms in the first iteration, and an average of 25ms with the use of warm start. The GP model is more computationally expensive with a worst case of 38 seconds in the first iteration and an average of around one second later.

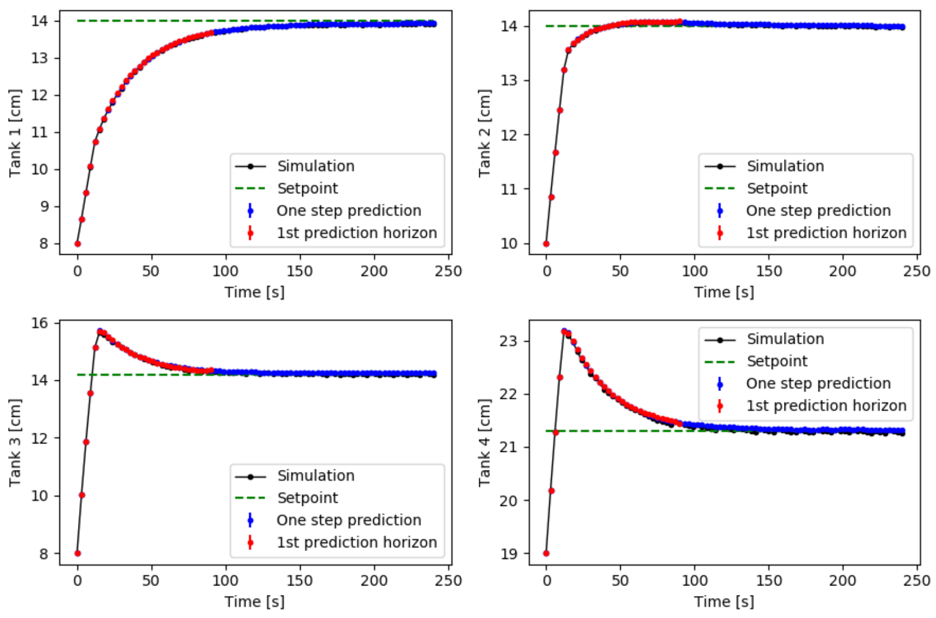
## 5.6 Discussion

**Model Learning**

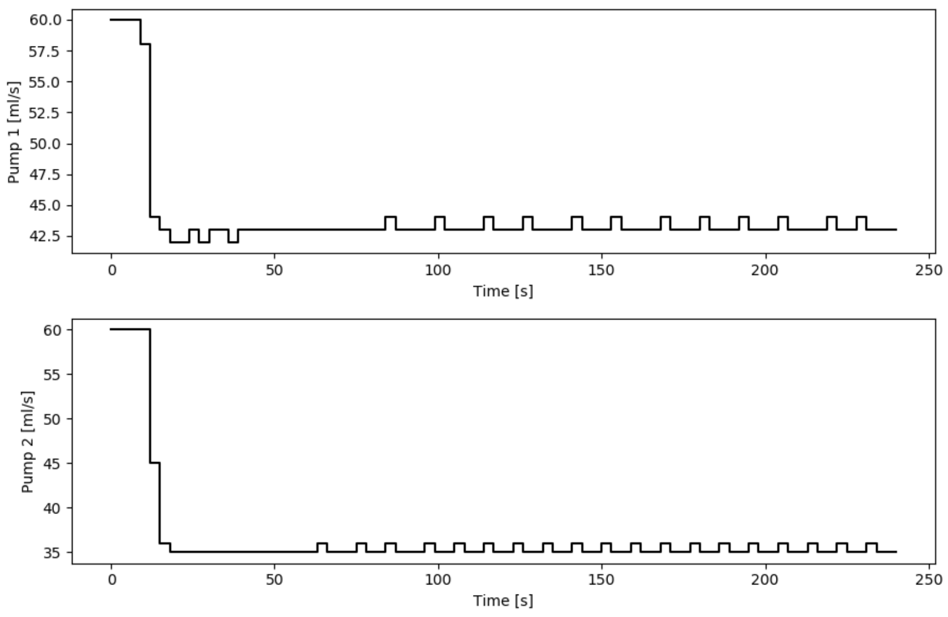
From the result it is interesting to note that a good model validation is only as good as the test data used. In the model used in this chapter tank 1 had the worst validation score, but from the open loop prediction we see that tank 4 has more deviation error than the other tanks. This show how important it is to have a data set of test data that accurately represent the the whole state-input space. Since the computational cost of the Gaussian model increase with the number of states and inputs, it could be beneficial to see how we can reduce the model. It is then interesting to note from Figure 5.1 that tank one and three is decoupled from tank two and four, where only the inputs are common. Instead of one model with with four states and two inputs, we could potentially reduce this to two models with two states and two inputs. This means that we reduce from a six dimensional state-input space down to only four dimensions, a substantial reduction in terms of how much data we need. A single state system only need to fill up point along the one dimension, while with two states we would need to fill up a two dimensional space, three dimensional space for three space and so on. This means that there can be quite the difference between using the same number of data point on a GP model with six inputs, and one with four inputs. We can see from the hyper-parameters that the length-scales mostly reflect the dependencies in equation (5.1). The cases that are off tell us that these are not the optimal hyper-parameters, even if they give a decent fit. The dependencies between the states are mostly accurate where a length scale less than 10 indicate a relation, while a higher number indicate states that are not important for the prediction. For the inputs this is not as clear as the inputs are directly or indirectly connected to all the states.

**Control**

With a relatively accurate GP model, the important question to ask is whether it is suited control purposes, and if the ability to propagate uncertainty can give a cautious control. With the RK4 integrator as a baseline for prediction we see that a simple GP model, with relatively little training data is able to follow closely to real system. The small deviation is not as important since only the first control input is used for each iteration. The consequence is however that longer predictions could destabilize the system, where each control iteration would have to compensate for the prediction errors. In this example we

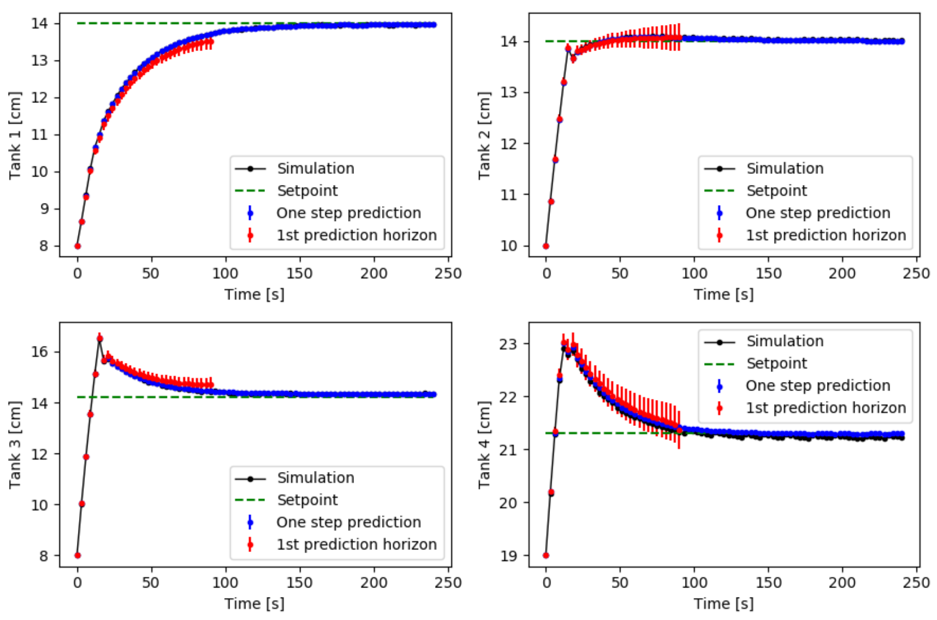


**(a)** Simulated states

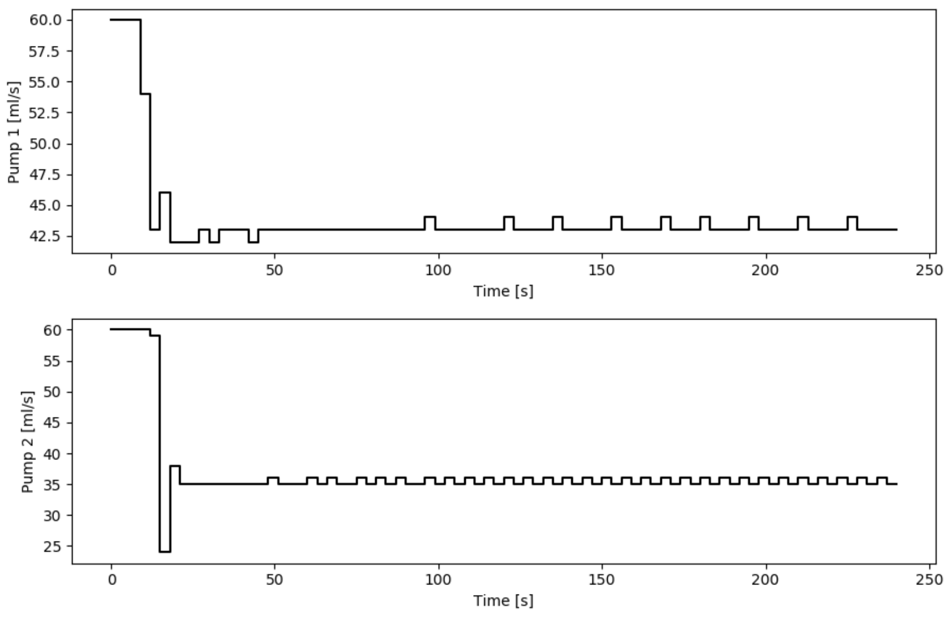


**(b)** Control inputs

**Figure 5.4:** MPC simulation for 250 seconds using LQR feedback, and RK4 for prediction. The red line show the first prediction horizon while the blue line show the first predicted step for each iteration.



**(a)** Simulated states



**(b)** Control inputs

**Figure 5.5:** MPC simulation for 250 seconds using LQR feedback, and TA GP for prediction. The red line show the first prediction horizon with 95% confidence interval. The blue line show the first predicted step for each iteration.

can see that tank three and four overshoot from the set point due the higher cost in the lower tanks. This is also the strength of using MPC that we are able to predict future states to optimize the pump actuation by taking into account that the upper tanks also will fill up the lower tanks. Without this consideration it would be difficult to reach the set point in all the tanks. In the RK4 model this works seamlessly with smooth changes in the control inputs. With the model error in tank one, with a higher measured level compared to the predicted level, the MPC controller has to act by lowering the expected actuation in pump 2, to avoid overfilling tank one and three. This give a more aggressive control behaviour with rapid drop in actuation from pump two compared to the RK4 model. We have put a very low cost in control input change for this problem to allow for this type of actuation. If such rapid changes would strain the pump, it would be advisable to increase the penalty $S$. Even though an imperfect model give more aggressive control it is clear that the GP model works as an alternative for first principle dynamic models. In the terms of investigating if the uncertainty lead to a more cautious control, this example can not say for certain as no state constraints are active, but it seems promising that the real signal is within the 95% confidence interval of the predicted mean. The chance constraints would then make sure that the tank don't overflow by underestimating the water level. The controller would then always err on the safe side, such that we would not necessary reach the desired set point if we are close to the constraints, and avoid overflowing the tank.

Chapter 6

# Vehicle Obstacle Avoidance

In the field of vehicle control there are two major concerns. The first is the balance between using simple computationally cheap models that may violate system constraints, while using an accurate nonlinear model could be too computationally demanding for real-time application. The other concern is the presence of noise and modelling errors that are sources of uncertainty in the state predictions that might prevent the vehicle from keeping the desired path. To handle the first issue there are real-time MPC strategies that could be used, Diehl (2014). One of the more promising is the use of hierarchical MPC where a simple model is optimized on a long horizon to avoid sudden obstacles, such that the more complex model can be warm started with a feasable trajectory, as done in Gao et al. (2010). Handling the uncertainties is a more challenging problem, but if the noise is assumed to be bounded it is possible to use methods like tube-based robust nonlinear MPC showed in Gao et al. (2014) or estimate the noise and model error with a Gaussian Process model as presented in Hewing et al. (2017). This field of GP use show the real potential using GP models with control, leading the way for adaptive control. In this chapter we will look at exactly how effective a Gaussian Process is to estimate not just the noise and model error, but all the dynamics in the car model.

## 6.1 Car model

This thesis use the same model as Gao et al. (2014), where the car is modelled as the bicycle model in Figure 6.1 where each wheel represent two merged wheels. In this model the dynamic equations are functions of the forces in the lateral and longitudinal directions of the vehicle $F_x*$ and $F_y*$, and the momentum $M_z = I_z\ddot{\psi}$ exerted on the center of gravity.

$$
\begin{aligned}
m\ddot{x} &= m\dot{y}\dot{\psi} + 2F_{xf} + 2F_{xr}, && \text{(6.1a)}\\
m\ddot{y} &= -m\dot{x}\dot{\psi} + 2F_{yf} + 2F_{yr}, && \text{(6.1b)}\\
I_z\ddot{\psi} &= 2l_f F_{yf} - 2l_r F_{yr} && \text{(6.1c)}\\
\dot{e}_\psi &= \dot{\psi} - \dot{\psi}_d, && \text{(6.1d)}\\
\dot{e}_y &= \dot{y}\cos(e_\psi) + \dot{x}\sin(e_\psi), && \text{(6.1e)}\\
\dot{s} &= \dot{x}\cos(e_\psi) - \dot{y}\sin(e_\psi) && \text{(6.1f)}
\end{aligned}
$$

In these equations $\dot{x}$ and $\dot{x}$ denotes the longitudinal and lateral velocity of the vehicle, while $\dot{\psi}$ give the yaw rate around the vehicle's center of gravity. The deviation error from the desired path $e_\psi$ and $e_y$ is respectively given by the vehicle orientation and lateral position relative to the road aligned coordinate frame. $\psi_d$ denote the angle of the tangent to the road centerline in the origin frame, and $s$ give the distance traveled as the vehicle longitudinal position along the desired road path. $F_{yf}$ and $F_{yr}$ represent front and rear tire forces along the vehicle lateral axis, while $F_{xf}$ and $F_{xr}$ give the forces acting along the longitudinal axis.

**Assumption 1.** *Assuming that the vehicle will drive on a horizontal plane, where the pitch and roll components of the vehicle can be neglected.*

**Assumption 2.** *Assuming that the vertical load $F_z$ is distributed evenly between the front and rear tires, such that the normal forces $F_{z*}$ are assumed constant and given by the steady state weight distribution of the vehicle.*

$$
F_{zf} = \frac{l_r mg}{2(l_r + l_f)}, \quad F_{zr} = \frac{l_f mg}{2(l_r + l_f)} \tag{6.2}
$$

The friction forces are modelled as

$$
\begin{aligned}
F_{x*} &= F_{l*}\cos(\delta_*) - F_{c*}\sin(\delta_*), && \text{(6.3a)}\\
F_{y*} &= F_{l*}\sin(\delta_*) + F_{c*}\cos(\delta_*), \quad * \in \{f, r\} && \text{(6.3b)}
\end{aligned}
$$

where $*$ represent $f$ or $r$ for front and rear tire, and $\delta_*$ the steering angle in the wheel.

**Assumption 3.** *Assuming that only the steering angle at the front wheels can be controlled, giving $\delta_f = \delta$ and $\delta_r = 0$.*

The tire forces $F_{l*}$ and $F_{c*}$ respectively in the longitudinal and lateral direction relative to the tires are given by the MAGIC formulas by Pacejka (2012). They are a highly non-linear semi-empirical functions determined by the tire slip angle $\alpha_*$, slip ratio $\sigma_*$, normal forces $F_{z*}$ and friction coefficient between the tire and road $\mu_*$.

$$
\begin{aligned}
F_{l*} &= f_l(\alpha_*, \sigma_*, F_{z*}, \mu_*) && \text{(6.4a)}\\
F_{c*} &= f_c(\alpha_*, \sigma_*, F_{z*}, \mu_*) && \text{(6.4b)}
\end{aligned}
$$

The full generalized MAGIC formula reads

$$
Y(X) = D\sin[C\arctan\{B(X+S_h) - E(B(X+S_h) - \arctan B(X+S_h))\}] + S_v \tag{6.5}
$$

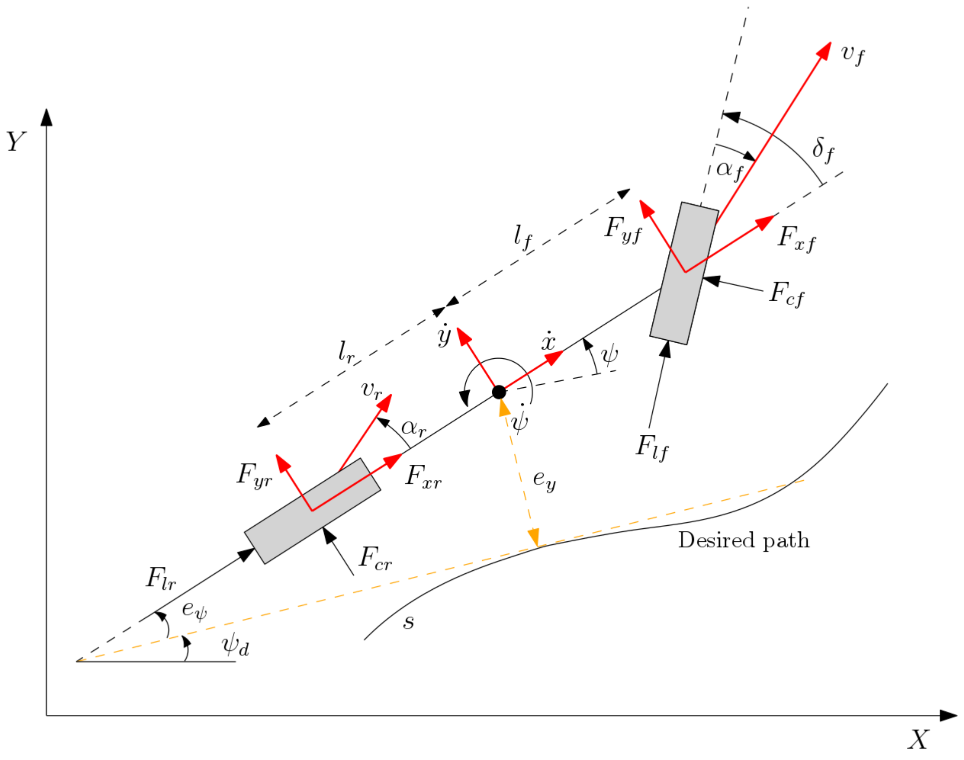


**Figure 6.1:** Car model simplified as a bicycle model in the horizontal plane. The vehicle move along a desired path where $s$ denotes the position along this path. The deviation from the path in the lateral direction relative to the path is given by $e_y$, while the deviation in the orientation of the vehicle relative relative to the path is given by $e_\psi$.

where $Y$ represent either $F_{l*}$ or $F_{c*}$, while $X$ is given either as the slip angle $\alpha_*$ or the slip ratio $\sigma_*$. The parameters are the stiffness factor $B$, shape factor $C$, peak value $D$, curvature factor $E$, horizontal shift $S_h$, and vertical shift $S_v$. It is not desirable to have this level of non-linearity in the optimal control problem, so to ease the computation the following section will instead use a linearized approximation.

The slip ratio is a ratio of how much the tires are slipping in the longitudinal direction of the tire, and is given as

$$\sigma_* = \begin{cases} \frac{r\omega}{v_l} - 1, & \text{if } v_l > r\omega, \quad v_l \neq 0 \text{ for breaking} \\ 1 - \frac{v_l}{r\omega}, & \text{if } v_l r < \omega, \quad \omega \neq 0 \text{ for driving} \end{cases} \tag{6.6}$$

where $r$ is the effective radius of the tire, $\omega$ the angular velocity of the tire, and $v_l$ the longitudinal velocity of the vehicle. This ratio will however be assumed negligible in the approximated tire force. The slip angle is a measure of the difference between the steering angle $\delta$ and the actual direction of the vehicle, and is given by Yoon et al. (2009) as

$$\alpha_f = \tan^{-1}\left(\frac{\dot{y} + l_f\dot{\psi}}{\dot{x}}\right) - \delta, \tag{6.7a}$$

$$\alpha_r = \tan^{-1}\left(\frac{\dot{y} - l_r\dot{\psi}}{\dot{x}}\right) \tag{6.7b}$$

**Assumption 4.** *The side slip angle $\alpha_*$ is assumed to be small $\alpha_* \ll 1rad$, where $\dot{y} \ll \dot{x}$. The same assumption can be made for the vehicle orientation error $e_\psi \ll 1$ giving the approximations sin($e_\psi$) $\simeq e_\psi$ and cos($e_\psi$) $\simeq 1rad$. Yoon et al. (2009) show that any loss of accuracy against the full nonlinear model is negligible.*

With the small angle assumption, the slip angles in equation (6.7) can be approximated as

$$\alpha_f = \frac{\dot{y} + l_f\dot{\psi}}{\dot{x}} - \delta, \tag{6.8a}$$

$$\alpha_r = \frac{\dot{y} - l_r\dot{\psi}}{\dot{x}} \tag{6.8b}$$

With the ability to control both the torque and angle in the front wheel we are able to control all forces in the full circle around the wheel, controlling both the slip angle and slip ratio. This means we have the choice of using a lower level force controller and using the front wheel forces $F_{xf}$ and $F_{yf}$ directly as inputs to the model in the optimal control problem, as done in Gao et al. (2014). This is however not an option for the rear tires due to the fixed wheel angle, where we are only able to control the longitudinal breaking/throttle force $F_{lr}$, with a linearized tire model. To keep it simple this thesis will also use the same approximation for the front wheel. It is assumed that simple low level controllers can control the breaking and throttle in $F_{l*}$, and steering angle $\delta$. The linearization of the Pacejka model in (6.5) is then given by Gao et al. (2014) as

$$F_{l*} = \beta_*\mu F_{zr}, \tag{6.9a}$$

$$F_{c*} = C_*\alpha_*, \quad * \in \{f, r\} \tag{6.9b}$$

where Gao et al. (2014) show that the approximation is valid for small values of $\alpha_*$. The longitudinal tire forces are linear with the breaking and throttle ratio $\beta_*$, where $\beta_* = -1$ correspond to full throttle and $\beta_* = -1$ max breaking. With moderate breaking, Gao et al. (2014) show that the side force $F_{c*}$ has low sensitivity with respect to $\beta_*$, thus the linear gain will be constricted to $\beta_* \in [-0.5, 0.5]$. The constant parameter $C_*$ is the cornering stiffness, a measure of how resistant the tire is to deform while the vehicle corners.

The resulting full model where the approximated tire forces are inserted in (6.1) is presented in equation (6.10). In these equations the first three states represent the dynamics of the vehicle, while the last three give the kinematic relationship.

$$
\begin{aligned}
\ddot{x} &= \frac{1}{m}\Big(m\dot{y}\dot{\psi} + 2\beta_f\mu F_{zf} + 2\delta^2 C_f \\
&\quad - 2\delta C_f\frac{(\dot{y} + l_f\dot{\psi})}{\dot{x}} + 2\beta_r\mu F_{zr}\Big) && (6.10a)\\
\ddot{y} &= \frac{1}{m}\Big(-m\dot{x}\dot{\psi} + 2\delta\beta_f\mu F_{zf} + 2C_f\frac{(\dot{y} + l_f\dot{\psi})}{\dot{x}} \\
&\quad - 2\delta C_f + 2\delta C_r\frac{(\dot{y} - l_r\dot{\psi})}{\dot{x}}\Big) && (6.10b)\\
\ddot{\psi} &= \frac{1}{I_z}\Big(2l_f\delta\beta_f\mu F_{zf} + 2l_f C_f\frac{(\dot{y} + l_f\dot{\psi})}{\dot{x}} \\
&\quad - 2l_f\delta C_f - 2l_r C_r\frac{(\dot{y} - l_r\dot{\psi})}{\dot{x}}\Big) && (6.10c)\\
\dot{e}_\psi &= \dot{\psi} - \dot{\psi}_d && (6.10d)\\
\dot{e}_y &= \dot{x}\sin(e_\psi) + \dot{y}\cos(e_\psi) && (6.10e)\\
\dot{s} &= \dot{x}\cos(e_\psi) - \dot{y}\sin(e_\psi) && (6.10f)
\end{aligned}
$$

We want to keep the dimensions of the GP model as low as possible, to reduce computational cost. This means we simplify our model with the following assumption, resulting in two instead of three inputs.

**Assumption 5.** *Both rear and front tires break and apply force at the same time, such that* $\beta = \beta_r = \beta_f$.

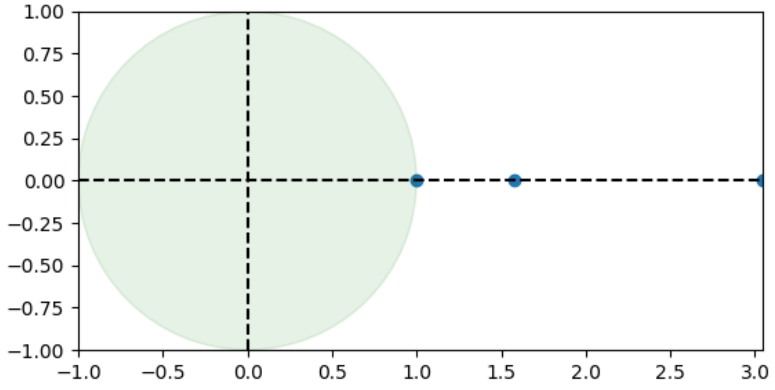
Our model is then given by the states $\xi = [\dot{x}, \dot{y}, \dot{\psi}, e_\psi, e_y, s]^T$, with inputs $u = [\beta, \delta]^T$.

## 6.2 Model learning

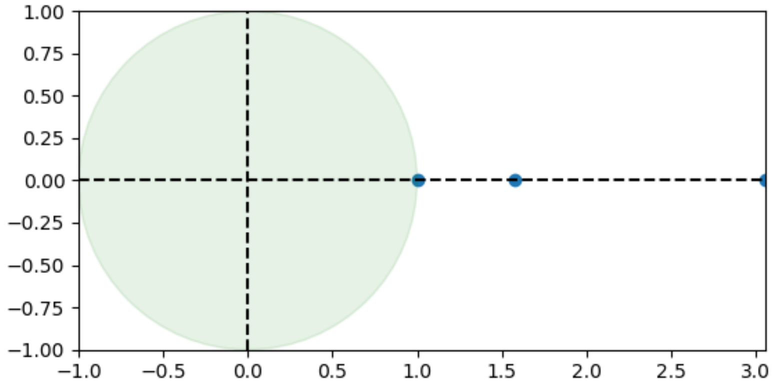
Having six states, of which three are integrating states that could potentially give any real value pose a problem when learning the GP model. In simple terms the GP model is only a form of interpolating between known training data to get an estimate of unseen points. The farther away from the training data we get, the less accurate the prediction is, as discussed earlier in chapter 2. This means that we are not able to properly train our model unless

**Table 6.1:** Car parameters from Gao et al. (2014),

| Param. | Value | Units | |
|---|---|---|---|
| $g$ | 9.81 | m/s$^2$ | Gravity |
| $m$ | 2050 | kg | Vehicle mass |
| $I_z$ | 3344 | kgm$^2$ | Yaw inertia |
| $C_r$ | 65000 | N/rad | Rear tire cornering stiffness |
| $C_f$ | 65000 | N/rad | Front tire cornering stiffness |
| $\mu$ | 0.5 | - | Tire friction coefficient |
| $l$ | 4.0 | m | Vehicle length |
| $l_f$ | 2.0 | m | Distance from CoG to the front tire |
| $l_r$ | 2.0 | m | Distance from CoG to the rear tire |



(a)



(b)

**Figure 6.2:** Eigenvalue plot for the linearized discretized dynamic model in a) and the linearized GP model in b). Both use $\mu_0^d = [13.89, 0, 0]$ and $u_0 = [0, 0]$ as the linearized operating point. Both have practically identical eigenvalues, one marginally stable and two destabilizing points.

we constrain our model to a very limited area. The workaround in this thesis is to use a hybrid model where the GP model estimate the dynamics within a known operating region, while the rest of the states are given directly by kinematic relations, and is not necessary to be estimated. These can then be integrated by a standard integrator, e.g. RK4, using the estimated dynamic states as inputs.

$$\mu^d_{i+1} = d(\mu^d_i, u_i), \tag{6.11a}$$
$$\mu^\eta_{i+1} = \eta(\mu^\eta_i, \mu^d_i) \tag{6.11b}$$

The dynamic states are then given by $\mu^d_i$, with the kinematic states $\mu^\eta_i$, and the new combined hybrid state vector given as $\mu^\xi_i = [\mu^d_i; \mu^\eta_i]^T$. The mean of the dynamic states $\mu^d_{i+1}$ is computed using first-order Taylor approximation (2.23), while the kinematic states $\mu^\eta_{i+1}$ are computed using equations (6.10d)-(6.10f). By linear transformations of Gaussian distributions, the update equation for the covariance of the full system is given as

$$\Sigma^\xi_{i+1} = \begin{bmatrix} \Sigma^d_{i+1} & 0 \\ 0 & B\Sigma^d_{i+1}B^T \end{bmatrix} \tag{6.12}$$

where $B = \nabla_d \eta(\mu^\eta_i, \mu^d_i)$, and $\Sigma^d_{i+1}$ is given by equation (2.23). The input covariance matrix to the GP function (2.23) is given similarly as

$$\Sigma^z_i = \begin{bmatrix} \Sigma^d_i & \Sigma^d_i K^T \\ K\Sigma^d_i & K\Sigma^d_i K^T \end{bmatrix} \tag{6.13}$$

where $K$ is the linear gain in the ancillary feedback controller. To train the GP model we will use 200 samples from the Latin Hypercube limited to the range of the dynamic states and input boundaries given in table 6.2. The model is then validated using with 500 random samples.

## 6.3 Safety constraints

The main objective of the controller is to make sure that the vehicle maintain stability and follow the road while avoiding obstacles. To achieve this we will use the same strategy as Gao et al. (2014) of using state constraints.

### Road boundary

The center of gravity of the vehicle is constrained within the road bounds by using road boundary constraints that take into account the vehicle width.

$$e_{y_{\text{min}}} \leq e_y \leq e_{y_{\text{max}}} \tag{6.14}$$

where $e_y$ is the lateral distance relative to the road from the road center line, and $e_{y_{\text{min}}}$, $e_{y_{\text{max}}}$ are the distance to the road boundaries from the center line, accounted for the car width.

### Obstacle avoidance

We will assume all obstacles can be written as ellipsoidal in the form

$$\left(\frac{(s-s_{obs})}{a_{\text{obs}}}\right)^2+\left(\frac{(e_y-y_{\text{obs}})}{b_{obs}}\right)^2\leq 1 \tag{6.15}$$

where $s_{\text{obs}}$ is the distance to the obstacle in the longitudinal direction from the car, while $y_{\text{obs}}$ is the distance in the lateral direction. The obstacle constraint can then be written as

$$\left(\frac{(s-s_{\text{obs}})}{a_{\text{obs}}+l_f}\right)^2+\left(\frac{(e_y-y_{\text{obs}})}{b_{obs}+l_w}\right)^2\geq 1 \tag{6.16}$$

such that the vehicle movement is constrained outside of the ellipsoidal obstacle. To take into account the shape of the vehicle the elliptical shape is enlarged with the vehicle length $l_f$ and width $l_w$ from the vehicle CoG. The length of the ellipsoidal is also further increased to ensure a smooth turning radius.

### Slip Constraints

To allow for an easy operation of the vehicle the slip angle will be constrained to the linear region of the tire forces. This will ensure that the car actually follow the direction set by the steering angle and avoid approximation errors due to the linearization of the tire forces.

$$\alpha_{\text{min}}\leq\alpha_*\leq\alpha_{\text{max}},\quad *\in\{f,r\} \tag{6.17}$$

The constraints (6.14), (6.16) and (6.17) can then be written in shorthand as

$$h(\xi,u)\leq\mathbf{0} \tag{6.18}$$

To ease the feasibility of the optimal control problem we will relax the constraints by using slack variables $\varepsilon$ to let us use soft constraints. This enables the controller to violate the constraints, but increase the chance of finding a solution.

$$h(\xi,u)\leq\varepsilon \tag{6.19}$$

## 6.4 MPC formulation

By utilizing the hybrid GP model in the predictions we are able to use the following optimal control formulation. As previously mentioned earlier, an ancillary LQR feedback controller will be used with the same cost matrices $Q$ and $R$ as the MPC problem. The terminal constraint set is chosen to be $\mathbb{X}_f=\{\xi\mid\xi^TP\xi\leq 10\}$, where $P$ is the infinite horizon cost matrix in (3.3), the same matrix used in the terminal cost $V_f$. To take into account the stochastic property of the predicted states we will constrain the states to be within the expected value of the terminal set, using the expected value of a Gaussian distributed quadratic function.

$$\min_{v,\mu^\xi,\Sigma^\xi} \quad V_f(\mu_N^\xi - \xi_{\text{ref}}, \Sigma_N^\xi) + \sum_{i=0}^{N-1} l(\mu_i^\xi - \xi_{\text{ref}}, \Sigma_i^\xi, v_i) + \|\Delta u_i\|_S^2 + \lambda^T \varepsilon_i \tag{6.20a}$$

$$s.t. \quad \mu_0^\xi = \xi_k \tag{6.20b}$$

$$\Sigma_0^\xi = 0 \tag{6.20c}$$

$$\mu_{i+1}^\xi, \Sigma_{i+1}^\xi \quad \text{acc. to (6.11), (6.12)} \tag{6.20d}$$

$$\mu_{i+1}^d, \Sigma_{i+1}^d \quad \text{acc. to (2.23)} \tag{6.20e}$$

$$u_i = K\mu_i^\xi + v_i \tag{6.20f}$$

$$h(\mu_i^\xi, u_i) \leq \varepsilon_i \tag{6.20g}$$

$$P(\xi_i \in \mathcal{X}) \geq 1 - \epsilon \quad \text{acc. to (3.20)} \tag{6.20h}$$

$$P(u_i \in \mathcal{U}) \geq 1 - \epsilon \quad \text{acc. to (3.20)} \tag{6.20i}$$

$$||\mu_N^\xi||_P + \text{tr}(P\Sigma_N^\xi) \leq 10 \tag{6.20j}$$

The expected value of the terminal cost $V_N(\cdot)$ and the stage cost $l(\cdot)$ is given by equation (3.12) using quadratic cost functions. The optimal control law is then given as $\kappa(\xi_k) = K\xi_k + v_0^*$.

## 6.5 Simulation

As with the tank model we will use GP-MPC to simulate the experiment. CasADi, Andersson et al. (2018) is used for automatic Algorithmic differentiation, together with CVODES, Hindmarsh et al. (2005), for simulation, IPOPT, Wächter and Biegler (2006), as the non-linear solver in the optimal control problem, and ma27 (HSL) as the linear solver. The prediction horizon is 17 steps, where the prediction is done using hybrid GP model, where Taylor Approximation predicts the dynamics while a RK4 integrator give the kinematic states. For comparison we will use an RK4 integrator for predicting the whole system. The system is then simulated for 12.5 seconds, equivalent to 250 control moves. The car will start in the initial state of driving 50km/h along the $X$ axis, while all other states are zero. The obstacle avoidance see just the closest obstacle and is able to see obstacles within a radius of 40m. The parameters for the control problem is listed in table 5.2. All simulations are done on a laptop from 2011, with i5 processor and 8GB of memory.

## 6.6 Results

### Model learning

For the hybrid-GP model we use the three dynamic states to generate 200 training samples using Latin Hypercube, and 500 samples for validation. The hyper-parameters optimized for the resulting GP model used is then given by table 5.3, with the validation score in table 5.4. In Figure 6.2 we see the eigenvalue plot of both the linearized discrete dynamic states

**Table 6.2:** Control problem parameters.

| Param | Value | Units | |
|---|---|---|---|
| $Q$ | diag$(10^{-3}, 5.0, 1.0,$ $0.1, 10^{-10}, 1.0)$ | - | State penalty matrix |
| $R$ | diag$(0.1, 1)$ | - | Input penalty matrix |
| $S$ | diag$(1, 10)$ | - | Change in input penalty matrix |
| Cov$(w)$ | diag$(10^{-5}, 10^{-8}, 10^{-8},$ $10^{-8}, 10^{-5}, 10^{-5})$ | - | Noise covariance matrix |
| $\lambda$ | $500$ | - | Slack variable penalty |
| $\epsilon$ | $0.05$ | - | Probability of violation |
| $\Delta t$ | $50$ | ms | Sampling time |
| $H_c$ | $17$ | - | Number of steps in control horizon |
| $H_p$ | $17$ | - | Number of steps in prediction horizon |
| $\xi_0$ | $[13.89, 0, 0, 0, 0, 0]$ | - | Initial state |
| $\xi_{\text{ref}}$ | $[13.89, 0, 0, 0, 100, 0]$ | - | Desired set point |
| $\xi_{\text{min}}$ | $[10.0, -0.5, -0.2, -0.3, 0.0, -10.0]$ | - | Lower state boundary |
| $\xi_{\text{min}}$ | $[30.0, 0.5, 0.2, 0.3, 0.0, 10.0]$ | - | Upper state boundary |
| $\alpha_{\text{min}}$ | $-4$ | deg | Minimum slip angle |
| $\alpha_{\text{max}}$ | $4$ | deg | Maximum slip angle |
| $u_{\text{min}}$ | $[-0.5, -2]$ | - | Minimum input |
| $u_{\text{max}}$ | $[0.5, 2]$ | - | Maximum input |
| $e_{y_{\text{min}}}$ | $-3.5$ | m | Lower road boundary |
| $e_{y_{\text{max}}}$ | $3.5$ | m | Top road boundary |

**Table 6.3:** Validation of GP model using 500 randomized samples using Latin Hyper-cube.

| | SMSE | MNLP |
|---|---|---|
| $\dot{x}$ | $4 \cdot 10^{-6}$ | $-4.015$ |
| $\dot{y}$ | $0 \cdot 10^{-6}$ | $-7.027$ |
| $\dot{\psi}$ | $3 \cdot 10^{-6}$ | $-6.047$ |

**Table 6.4:** Hyper-parameters for GP model.

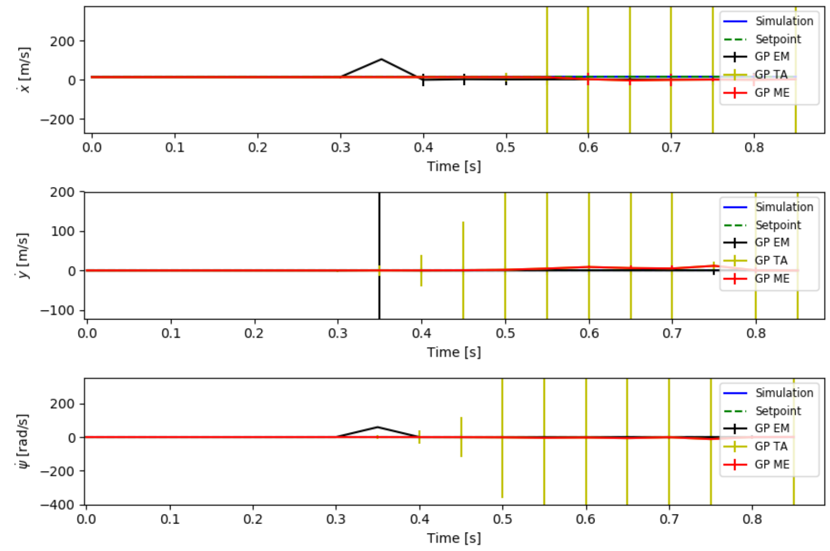
| State | $\sigma_f$ | $\ell_0$ | $\ell_1$ | $\ell_2$ | $\ell_3$ | $\ell_4$ | $\sigma_n$ |
|---|---|---|---|---|---|---|---|
| $\dot{x}$ | $266.86$ | $11.47$ | $7.66$ | $1.75$ | $3.02$ | $1.73$ | $2.19 \cdot 10^{-6}$ |
| $\dot{y}$ | $15.18$ | $8.80$ | $4.31$ | $3.89$ | $6.69$ | $5.68$ | $3.06 \cdot 10^{-8}$ |
| $\dot{\psi}$ | $18.28$ | $9.13$ | $4.89$ | $2.06$ | $6.28$ | $1.80$ | $6.67 \cdot 10^{-8}$ |

and the linearized GP model, with $\mu_0^d = [13.89, 0, 0]$ and $u_0 = [0, 0]$ as the linearized operating point. Both models have similar eigenvalues, with practically identical eigenvalues, one marginally stable and two destabilizing points. In 6.3 the GP model use Expected moments (EM), Taylor Approximation (TA), and Mean Equivalence (ME) compared against the simulated system, both unforced open loop and closed loop with a LQR feedback controller, with a prediction horizon of 17 steps. Even though the validation indicated a good model, the open loop prediction results in an exponential growth in the uncertainty and mean predictions that are way off. The mean don't diverge from the simulated states, but has an error bias that is far outside the safety region, equal to the car spinning in circles. To reduce the uncertainty we introduce feedback with the LQR controller, giving a substantial better result where all states and uncertainty are within the safety constraints.

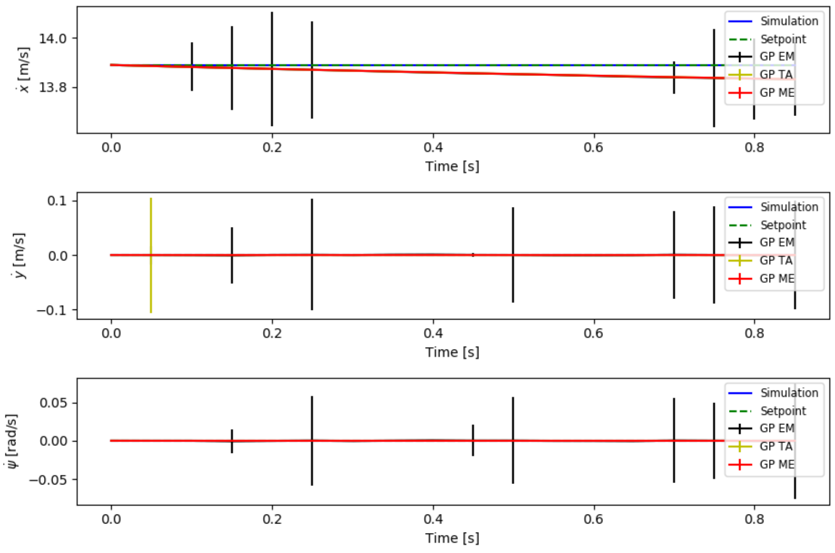
### Control

To test the GP effectiveness in a MPC setting we will again use the RK4 integrator as a baseline comparison, with and without terminal constraints. In Figure 6.4 we see the car avoiding obstacles using RK4 and no terminal constraints. In this case the MPC solver can't find any feasable solutions after the first obstacle, resulting in the instability that breaks the CVODES simulator after around six seconds, with the yaw rate growing at an exponential rate. In Figure 6.5 we again use RK4, but turn on the terminal constraints. We now see a far better performance, where the car avoids all the obstacles, follow the path and keep to the road middle line except when avoiding obstacles. We can also see from the optimal control inputs that the driving is quite aggressive with saturation in both steering and throttle. To explain the instability using the RK4 method we replay the optimal control inputs from 6.5 again to compare the open-loop simulation between RK4 and an implicit method in Figure 6.6. Here it is clear that RK4 is numerically unstable in the car predictions, where the MPC controller struggle to compensate and regain stability.

The vehicle path avoiding obstacles in the RK4 simulations together with the hybrid-GP is given in Figure 6.7. Here we see that the slow control due to the infeasable solutions using RK4 without terminal constraints, resulting in the simulated breaking after around

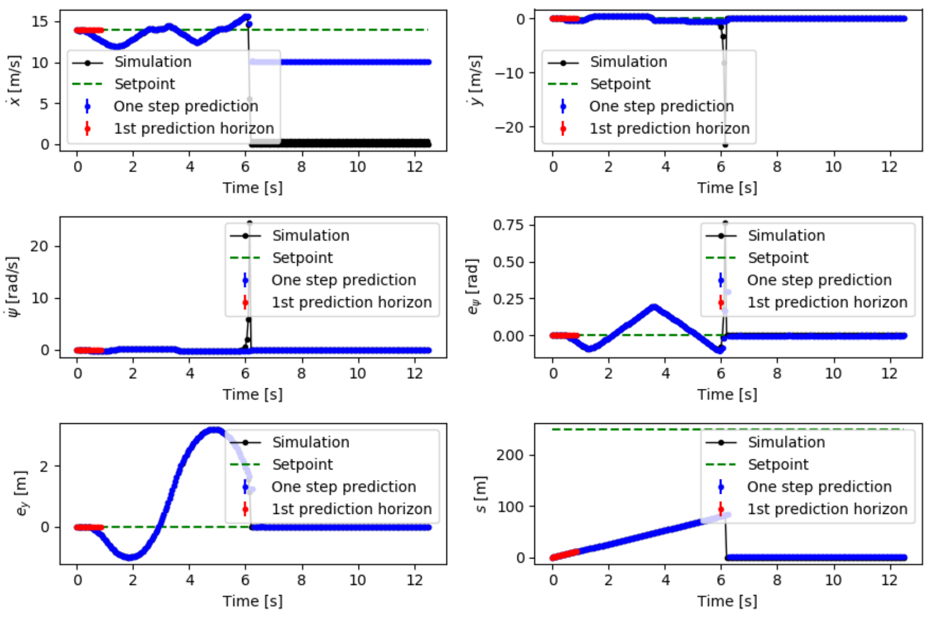


**(a)** Open loop GP prediction

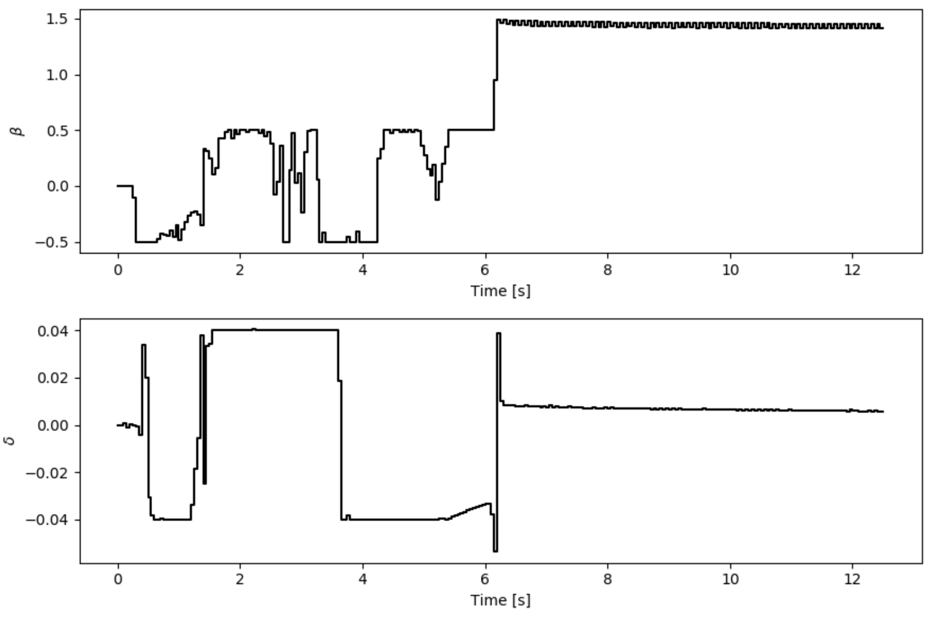


**(b)** Closed loop GP prediction using LQR feedback.

**Figure 6.3:** Comparison between open loop and closed loop with the GP model predicting 17 steps. In the open loop the GP model is a unforced system with initial state at $\xi = [13.89, 0, 0]$. The uncertainty in the open loop predictions quickly explode, while in the closed loop the LQR feedback controller reduce the uncertainty and keep the system stable.

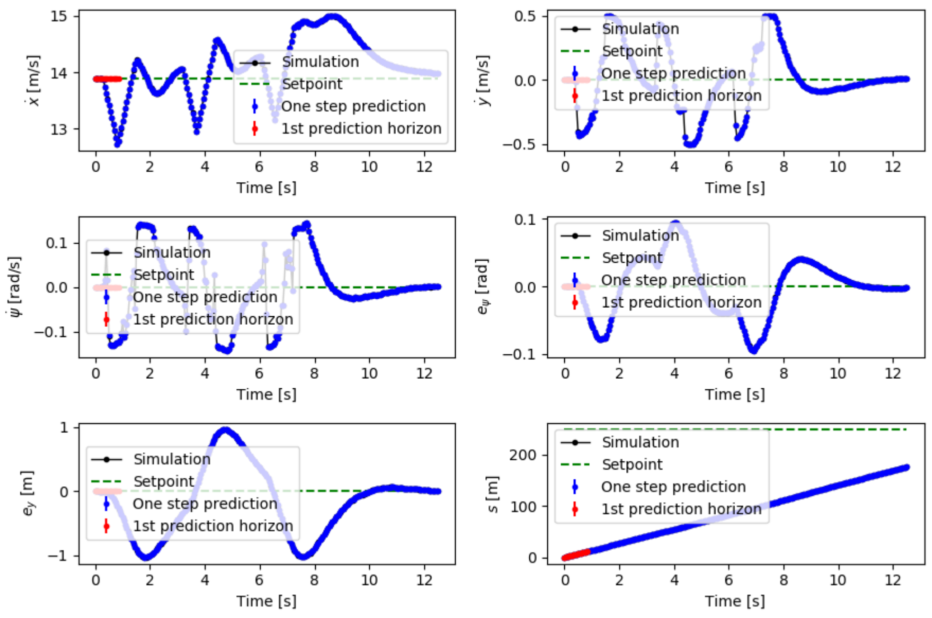


**(a)** Simulated states

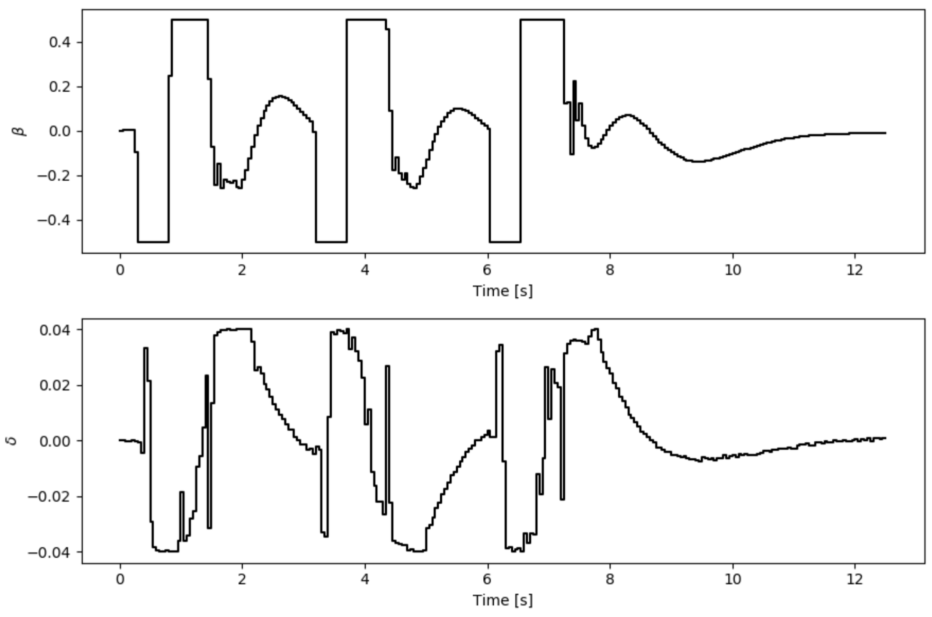


**(b)** Optimal control inputs

**Figure 6.4:** RK4 MPC with noise and no terminal constraint.



**(a)** Simulated states



**(b)** Optimal control inputs

**Figure 6.5:** RK4 MPC with noise and terminal constraint.

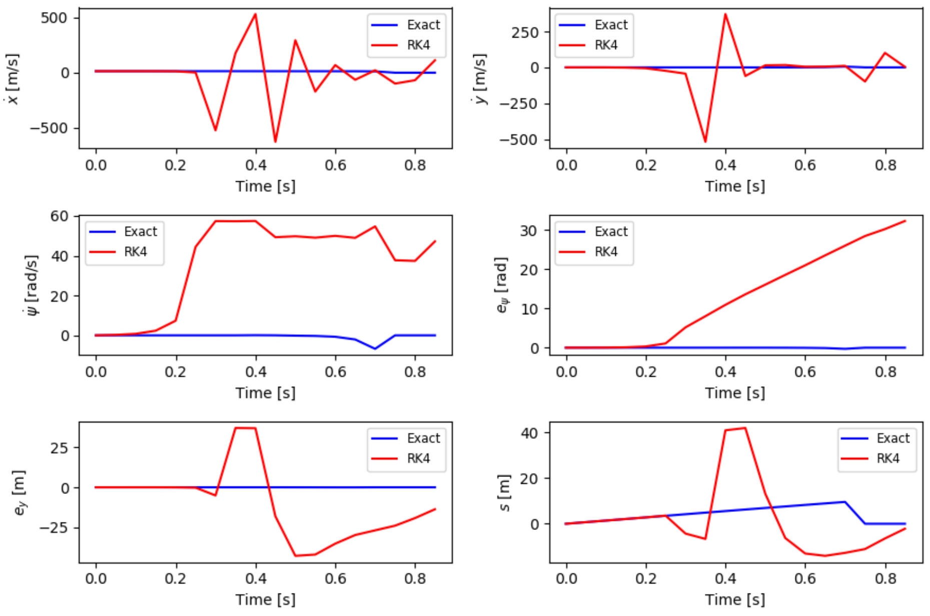


**Figure 6.6:** The 17 first control inputs from Figure6.7b with RK4 as integrator are replayed with both RK4 and an exact implicit integrator. The exact simulation behaves as expected while the RK4 is unstable. This is also not unexpected as the eigenvalues of the model, as seen in Figure 6.2, is outside of the stability region of the RK4 method.

80 meters. In Figure 6.7b we see that the car is able to avoid all obstacles, and is able to drive longer than the hybrid-GP simulations due to the aggressive driving, which results in a small constraint violation in the second obstacle. Both hybrid-GP simulations with and without terminal constraints are qualitatively identical, where the terminal constraint doesn't seem to have an effect. The car is able to avoid all the obstacles without violating any constraints.

We were not able to get a GP model that returned a feasable solution in the MPC problem, due to the high variance, even with the use of feedback. Instead we use a deterministic hybrid-GP model, where the covariance is defined as zero. This enables us to solve the MPC problem without terminal constraints in Figure 6.8, and with terminal constraints in Figure 6.9. As shown in the car path in Figure 6.7, both of these problems give qualitatively similar control input solutions, with practically identical resulting state simulations. The car is able to avoid all the obstacles, and follow the road path, without violating any constraint. Compared to the RK4 solutions, the d-hybrid-GP solution give more smooth transitions in the control inputs, avoiding saturation of the control inputs.

### Computational cost

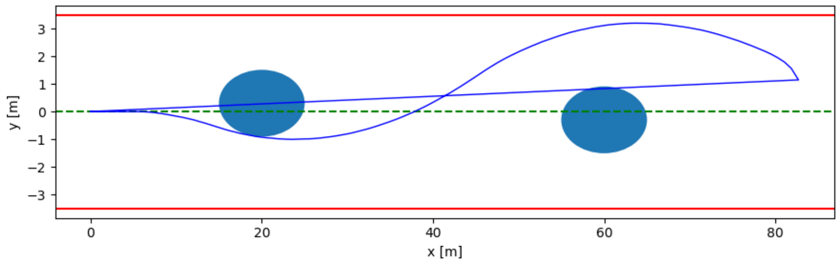
For the RK4 with terminal constraints the computation time is around 500ms when avoiding obstacles, and down to around 50ms when driving on a straight line. The deterministic hybrid-GP use around one second or more to avoid obstacles, and runs within 100ms when driving a straight line, and 38 seconds in the first iteration without warm-start. Both use more than one second if the solver is unable to find a feasable solution.

## 6.7 Discussion

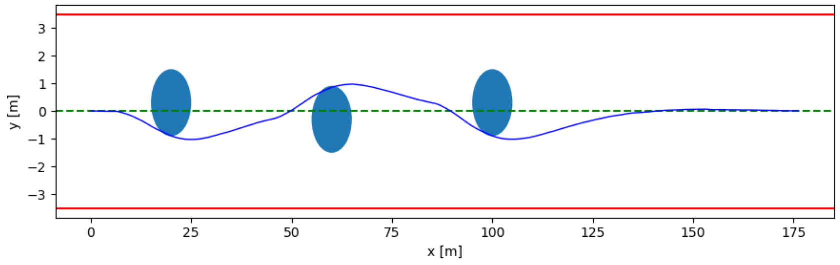
### Vehicle Model

It is not really necessary to simplify the nonlinear model, which was only done in this thesis to use the same parameters as in Gao et al. (2014), where the simplification where done to reduce computational cost in the optimal control problem. This simplification only makes sense if the goal was to use the approximated system in the MPC controller, not so much if we want an accurate model to learn the car dynamics from. However since the goal was never to use this model on an actual car, using the approximated model give a simpler implementation. An interesting continuation would be to comparison the GP model trained on the full nonlinear model with the simplified mechanical model presented in this thesis, where the goal would be to control the full nonlinear model with the presence of noise. In theory, the GP model would be more robust and the computational cost would be about the same as when trained on the simplified model. The question to investigate would then be to see if the added non-linearity to the model would require more training data, and how much more data are needed.

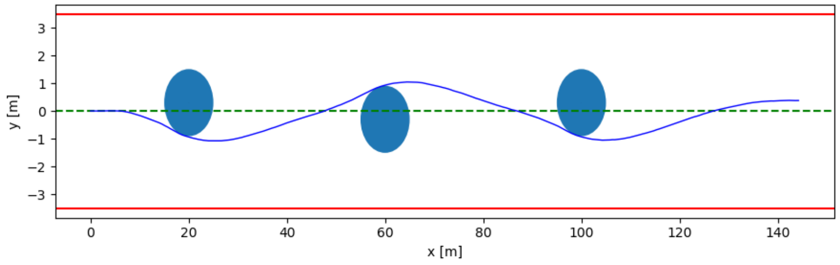
One other point is that the obstacle constraints used in this thesis is far from optimal, and chosen by their easy implementation. The dimensions of the car is indirectly taken into account by incorporating them in the elliptic constraint. An alternative way is to use the parallax information from the vehicle about the detected obstacles, as done in Yoon et al. (2009).



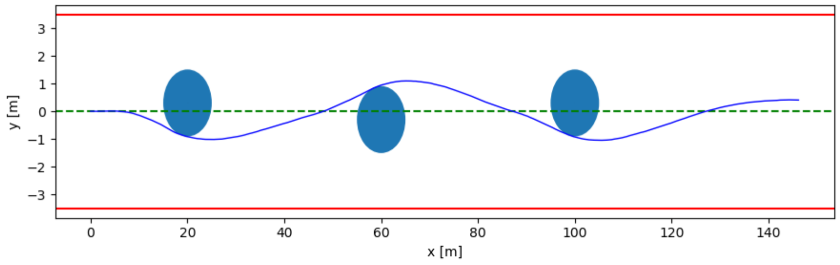
**(a)** RK4 simulation with no terminal constraints used.



**(b)** RK4 simulation with terminal constraints.

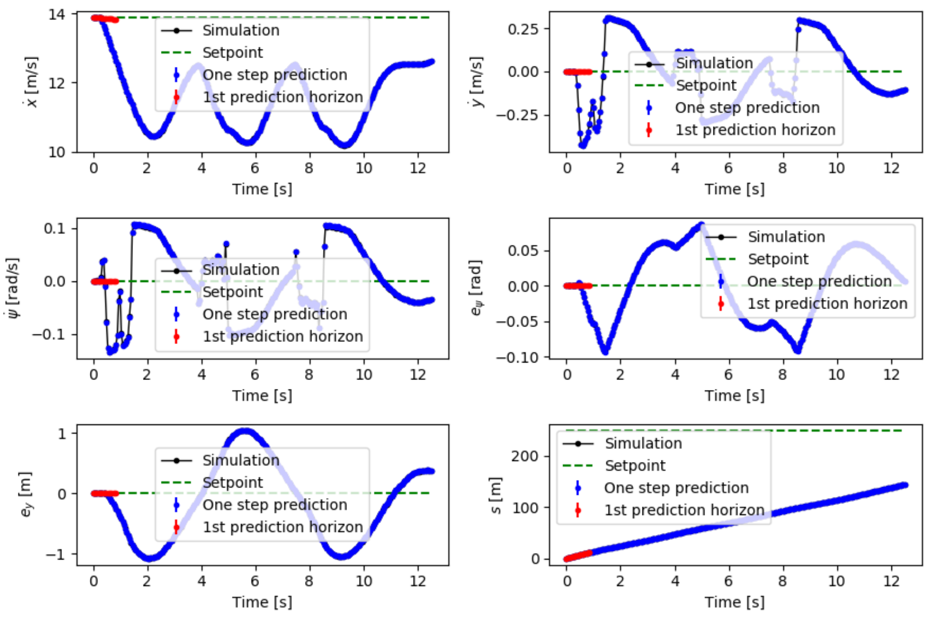


**(c)** Hybrid-GP without terminal constraints.

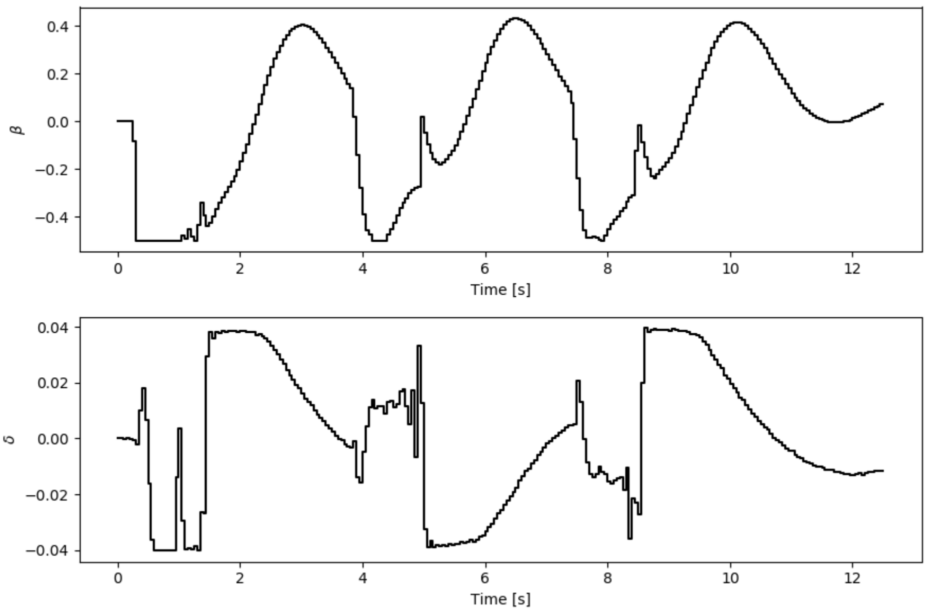


**(d)** Hybrid-GP with terminal constraints.

**Figure 6.7:** Comparison of RK4 and deterministic hybrid-GP with and without terminal constraints. The vehicle drive along the road path, constrained within the road path, and try to avoid three obstacles along the way. The simulated feedback measurements are added Gaussian noise.

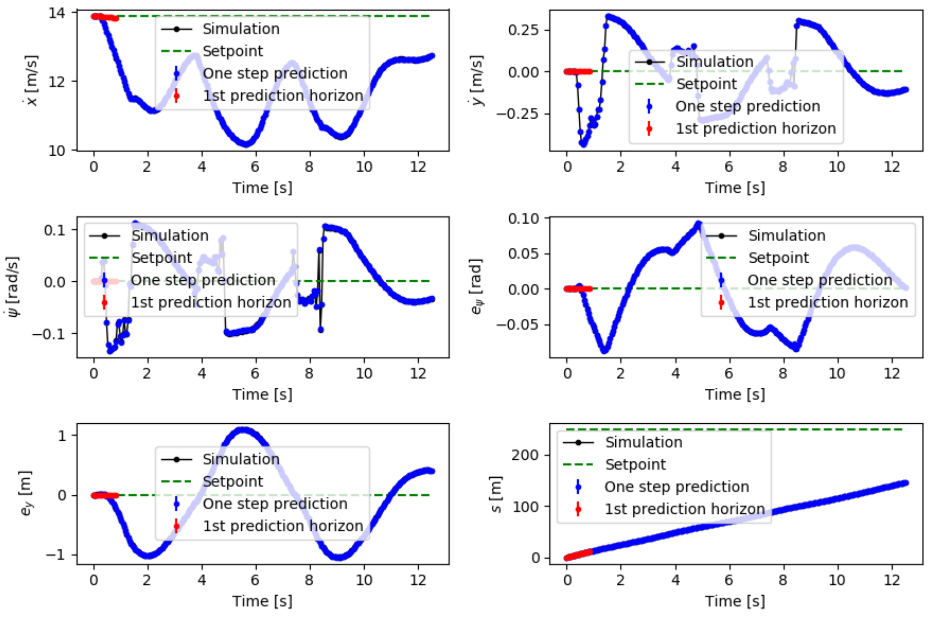


**(a)** Simulation of states.

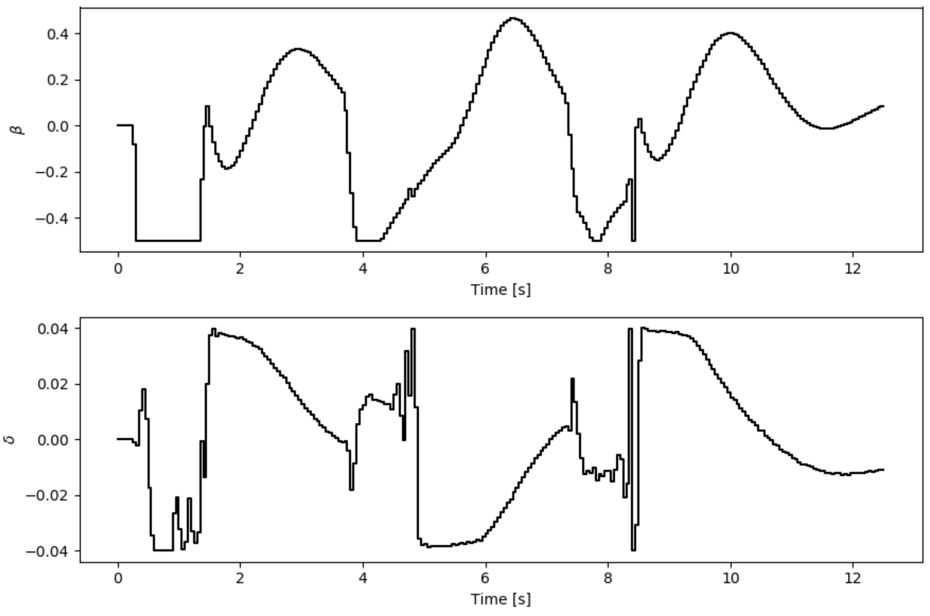


**(b)** Optimal control inputs

**Figure 6.8:** Deterministic hybrid-GP MPC without terminal constraint.



**(a)** Simulation of states.



**(b)** Optimal control inputs

**Figure 6.9:** Deterministic hybrid-GP MPC with terminal constraint.

### Model Learning

Using 200 samples gave a very good validation score, where all the estimated states have quite good validation with both SMSE and MNLP compared to the tank model. But as this chapter has presented, a good validation score is not enough to have a robust prediction. If the system we try to estimate is unstable, or borderline unstable, a slight deviation from the real model could lead to divergence in the predictions.

One of the major issues in this project was getting the GP model of the car accurate enough to be able to run the MPC solver. With the hybrid-GP model we were able to get a very good validation score, but due to the exponential growth of the variance, using the model in a MPC problem was infeasable. Adding a LQR feedback controller reduced the uncertainty substantially when the prediction was close to the operating point, but was not enough to be able to find a feasable solution in the MPC problem. To be able to use a GP model we had to define the covariance as zero, giving deterministic hybrid-GP instead. Using this we were able to get good results, except for the fact that we unable to utilize the covariance.

In Hewing et al. (2017) they used 300 samples to estimate just the modelling error from the simplified vehicle equations. Pointing to that using 200 samples like we do is not enough. The relatively low number is because of hardware constraints, as a larger number is more difficult to optimize and especially to use with the MPC problem. The other difference from Hewing et al. (2017) is that we use an optimal data spread with Latin Hypercube, while they only used random measurement data, resulting in the need for more data to be able to represent to whole state space. If we would use a larger model, we need to use sparse approximation with a small set of inducing points in the MPC problem. In the case of Hewing et al. (2017), they use a set of 10 inducing points instead of the full model with 300 samples, picked from the previous optimal MPC trajectory. Using the full GP model would be intractable for larger models due to the computational cost.

### Chance constraints

The observant reader might have noticed that the safety constraints do not utilize the variance in the predictions. This was mostly just a simplification to neglect this part, but a naive solution could be to use a three standard deviation distance in the constraints to be able to be within the 99% confidence interval, similar to the approach in the other chance state constraints. The same go for some of the state constraints that was not really necessary for stability. These were mostly added to tighten the state-space to confine the nonlinear solver to a set of states that are within the stability region.

### Simulation Stability

Using a RK4 integrator in an open-loop unstable system is not the best idea, where the the eigenvalues of the system is far outside of the stability region in the explicit Runga-Kutta method, given by (B.2). We are however still able to use this method instead of needing to turn to implicit methods because the MPC controller handles all constraints, even though is struggles more when the integrator diverge from the real system, as it need to do large changes in teh control inputs at each iteration to compensate for the prediction error. The GP-MPC framework also supports using CVODES/IDAS as the discretization method inside the MPC problem, as an alternative to RK4, but the current implementations has problems where the gradient of the integrator returns NaN values, and is slower than the GP model in the cases it is able to run. This thesis then only focus on using the RK4 as a comparison to the GP model, even though it is unstable in this particular case. Due to the slip angle constrain there is a possibility if a singularity if the longitudinal velocity $\dot{x}$ is zero, which is a possibility when using a interior point method. To avoid to much trouble in the optimization a simple fix in this thesis is to just constrain the velocity to be strictly positive. This could be handles better if we used the reasonable assumption that the slip angle to be zero at low velocities, but since the simulation only use high velocities, this fix was neglected.

### System Stability

As a stabilizing condition, the terminal constraints help with making sure that the state trajectories reach a terminal region in each iteration. For the RK4 method this was clear where the terminal constraint made sure that the prediction didn't diverge due to the instability. Since the hybrid-Gp don't diverge when using LQR feedback, the addition of a terminal constraint did not add much value since it was already stable. In general this show that a terminal constraint help with stabilization of a system. The drawback of using this is that it increase the chance of infeasable solutions, when the MPC solver can not find a solution that bring the state trajectory to the terminal region within the control horizon. A longer prediction horizon would ensure better stability with more time to react to obstacles, and more time to reach the terminal region, but the drawback of using long horizons is the increased computational cost. It doesn't matter if we find a feasable solution if we can't actuate the control within the sampling interval.

Even though we were able to get a stable solution using a GP model, we were not able to utilize the propagated uncertainty to get cautious control. Vehicle control is the type of problem that have a need for cautious control, as small deviations due to model error or noise can lead to instability. Having an estimate of the uncertainty directly from the predictions would ensure that the states are kept within the stability region. The alternative is to use methods from robust MPC like tube-based methods that rely on that we know the bounds of the noise.

Chapter 7

# Discussion

## 7.1 Prediction Stability

One of the things we have observed in this thesis is that even though the car has smaller state-input space than the tank, more data points and better validation score, it still has a substantially worse multi-step predication performance. As discussed in section 4.2 with the Van der Pol system, the validation score is not enough to give an indication on how well the prediction will turn out. The validation score is only a regression evaluation, and can not say anything about the stability of the prediction. If the model is open-loop stable, there should not be any problems as long as the GP model is also stable. If on the other hand the open-loop system is unstable, a slight error could lead to divergence in the prediction. This is a property that the GP model share with other simulation methods. In appendix B.2 there is a discussion on the numerical stability of explicit integrators like Runga-Kutta. In the car example we saw that the GP had better prediction stability than the RK4, given by the fact that the vehicle system has eigenvalues outside of the stability region of RK4. This then means that we cannot use a standard regression validation alone, and instead look to the stability analysis of simulation methods to learn more about how we can find sensitivity analysis methods suitable for Gaussian Process models.

## 7.2 Computational cost

One point that has not been the focus of this thesis is the computational cost of using a GP model over a cheaper RK4 model. The focus of this thesis has mainly been to investigate how effective the GPs are to estimate nonlinear dynamics, and how we can use utilize the uncertainty. This question can however not be neglected completely. If the solver cannot find a feasable solution within the sample interval, we would not be able to use the result. In both cases with the tank system and the car we observed a quite high computation time, a factor of ten higher or more than RK4 in both cases. It should be noted that RK4 alone would not be a realistic comparison, as the GP model would not be a direct alternative to first principle models. The real value lay in being able to to adapt the model online, either using the GP as the full model, or using the GP to estimate model errors and noise. For direct comparison we would need to compare with state of the art system identification methods or adaptive controllers. In addition to the fact that more stable implicit solvers are computationally more expensive than RK4.

It should also be noted that even tough an effort has been made to optimize the code for speed, using warm start, optimizing the math for numerical stability and speed, and picked solver options and solvers that are optimal for the problems, the main goal of the GP-MPC has been to create a flexible framework to ease experimentation. At the cost of performance.

## 7.3 Stability

The condition for stability with the use of MPC is the that the terminal state is constrained within an invariant set. Since GP models in theory have unbounded uncertainty distributions, such set is not tractable so we have to consider the probability that the system will converge within the terminal set. In this thesis we have used the uncertainty propagation with the GP model to constrain the model within the 95th percentile from the predicted mean.

This method will in theory give us a probabilistic condition for stability, but there are a few issues that have to be discussed. The most important one is that there is by no means a guarantee that the actual system lays within the 2 sigma bound from the mean. If the GP has to little data or has a bad fitting, the predicted variance would not necessarily give a correct estimate of the distance between the real value and the predicted mean. A bad fitting of the hyper-parameters could give saturation of the predicted variance that is too small, while training data consisting of points that don't represent the system would give an unrealistic estimate of the variance. Both of these conditions are exemplified in Figure 4.2, where ten sample points can give an accurate estimate of both the mean and the uncertainty if the data is chosen wisely, or predict a constant zero mean where the 95th percentile of the predicted variance is not even close to enclose the real system in the worst case scenario. This problem should be accounted for be satisfactory model validation, but as discussed earlier it is important to note that there is no guarantee that the validation is good enough. As with the trained model, the validation is only as good as the data it has, in addition to the stability region of the prediction.

On the opposite site there could be a problem with unrealistic high uncertainty, with growing uncertainty in long predictions, even though the prediction stay close to the real value. This is problematic with the use of terminal constrained set as it become difficult to find a feasable solution. Removing the terminal set would also remove the stability condition, and only relying on the cost functions, while keeping the set could cause infeasable solutions and almost guarantee instability if a re-initialization is not able to bring a feasable solution within reasonable time. This has the same fundamental problem as above where there is not enough training data, or the data does not represent the actual system well enough.

## 7.4 System Identification and Online Adaption

Even though system identification is the main objective for using Gaussian Process with control, this thesis has barely looked into this topic. System identification present us with the option of using measurement data to better estimate the our model, which if done online can be implemented as an adaptive controller. This thesis has presented use cases where first principle models work better, as a way of investigating the effectiveness of the PG prediction in MPC. The strength of using GP in control is when we can use it on systems where we either do not know the first principle models, or it is difficult to gen an accurate model based on coefficients that change or is difficult to find. Hewing and Zeilinger (2017) use this method to estimate the model error while Chowdhary et al. (2015) use GP together with Model reference adaptive control (MRAC) to account for changing model parameters.

Chapter 8

# Conclusion and Future Work

## 8.1 Conclusion

In previous works like Kocijan et al. (2004) and Hewing and Zeilinger (2017) we have seen that Gaussian Process has the interesting property that we can utilize the uncertainty for cautious model predictive control. This thesis has then been an extension of this to investigate the effectiveness og GPs in control. By using two different use cases, a slow stable tank system against a marginally stable vehicle model with fast dynamics we have observed that we are able to stabilize and meet the constraints without any violation. That we are able to propagate the uncertainty seems promising where the uncertainty can compensate for the model error in the constraints. What we have observed in the this thesis is however that just using the uncertainty is by no means a guaranty that the constraints are not going to be violated. In this thesis we have discussed three points. The two first are that it is necesary to have an accurate model, with well fitted hyper-parameters t be able to trust the uncertainty. With sub-optimal hyper-parameters, and a sub-optimal representation of the system in the training data, we risk both a naive variance that don't cover the model error within the 95% confidence interval, or 99% for that matter, and the opposite case where the uncertainty grows out of bounds, far beyond the actual model error. Meaning that it is not possible to meet the chance constraints due to the unrealistic high variance. A badly fitted model is often due to lack in model validation. and is the topic of the third point. Normal validation scores for GPs are meant to validate a regression model for one step predictions. In multi-step predictions we also have to take into account the stability of the system we want to estimate. In systems that are chaotic we know that any small perturbations can lead to instability. The same must also be considered even for marginally stable systems, as even a minuscule model error can propagate and change the stability properties, resulting in divergence. Similar to well known numerical methods like explicit Runga-Kutta, it is essential that we have methods for sensitivity analysis. Without this there is always the danger of instability in the predictions, and as a result instability in the system we want to control.

Given these considerations it is still clear that Gaussian processes show great potential in the field of control. Problems that have been discussed can be handled by a large enough representation of data, while using sparse GP with inducing point can take care of the computational cost. This thesis has looked into offline learning of the whole system, but the real value lays by using the GPs online in an adaptive manner. Gaussian process is a powerful method for regression, and can be utilized both with adaptive, robust control by learning the whole model like in this thesis, or as a way of estimating noise and model errors in the system.

## 8.2 Future work

This thesis has given an introduction to the use of Gaussian processes with model predictive control. One interesting continuation of this project would be to implement sparse GPs with a greater number of data to see if it is possible to get a cautious control of the vehicle model. A path that could give a great deal of value would be in the sensitivity analysis of GP prediction, to have a real measure of the numerical stability in the prediction. As mentioned previously, the power of using GPs with control is not to replace first principle models with a machine learning model. The potential is in the combination, where the GPs can compensate for inaccurate, or changing model parameters, where the estimated uncertainty can give a cautious method with both robust and adaptive control. Researching this field further would be very interesting.

# Bibliography

Andersson, J. A. E., Gillis, J., Horn, G., Rawlings, J. B., and Diehl, M. (In Press, 2018). CasADi – A software framework for nonlinear optimization and optimal control. *Mathematical Programming Computation*.

Bishop, C. (2006). *Pattern Recognition and Machine Learning*. Springer.

Boedecker, J., Springenberg, J. T., Wülfing, J., and Riedmiller, M. (2014). Approximate real-time optimal control based on sparse gaussian process models. In *Adaptive Dynamic Programming and Reinforcement Learning (ADPRL), 2014 IEEE Symposium on*, pages 1–8. IEEE.

Brochu, E., Cora, V. M., and De Freitas, N. (2010). A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*.

Cao, G., Lai, E. M.-K., and Alam, F. (2017). Gaussian process model predictive control of an unmanned quadrotor. *Journal of Intelligent & Robotic Systems*, 88(1):147–162.

Cawley, G. C. and Talbot, N. L. (2007). Preventing over-fitting during model selection via bayesian regularisation of the hyper-parameters. *Journal of Machine Learning Research*, 8(Apr):841–861.

Cawley, G. C. and Talbot, N. L. (2010). On over-fitting in model selection and subsequent selection bias in performance evaluation. *Journal of Machine Learning Research*, 11(Jul):2079–2107.

Chowdhary, G., Kingravi, H. A., How, J. P., Vela, P. A., et al. (2015). Bayesian nonparametric adaptive control using gaussian processes. *IEEE Trans. Neural Netw. Learning Syst.*, 26(3):537–550.

Deisenroth, M. and Rasmussen, C. E. (2011). Pilco: A model-based and data-efficient approach to policy search. In *Proceedings of the 28th International Conference on machine learning (ICML-11)*, pages 465–472.

Deisenroth, M. P. (2010). *Efficient reinforcement learning using Gaussian processes*, volume 9. KIT Scientific Publishing.

Deisenroth, M. P., Fox, D., and Rasmussen, C. E. (2015). Gaussian processes for data-efficient learning in robotics and control. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 37(2):408–423.

Diehl, M. (2014). Lecture notes on optimal control and estimation.

Ebden, M. (2015). Gaussian processes: A quick introduction. *arXiv preprint arXiv:1505.02965*.

Egeland, O. and Gravdahl, J. T. (2002). *Modeling and Simulation for Automatic Control*. Marine Cybernetics Trondheim, Norway.

Foss, B. and Heirung, T. A. N. (2016). Merging optimization and control. *Lecture Notes*.

Gao, Y., Gray, A., Tseng, H. E., and Borrelli, F. (2014). A tube-based robust nonlinear predictive control approach to semiautonomous ground vehicles. *Vehicle System Dynamics*, 52(6):802–823.

Gao, Y., Lin, T., Borrelli, F., Tseng, E., and Hrovat, D. (2010). Predictive control of autonomous ground vehicles with obstacle avoidance on slippery roads. In *ASME 2010 dynamic systems and control conference*, pages 265–272. American Society of Mechanical Engineers.

Gelman, A., Carlin, J. B., Stern, H. S., Dunson, D. B., Vehtari, A., and Rubin, D. B. (2013). *Bayesian data analysis*. CRC press.

Girard, A., Rasmussen, C. E., Candela, J. Q., and Murray-Smith, R. (2003). Gaussian process priors with uncertain inputs application to multiple-step ahead time series forecasting. In *Advances in neural information processing systems*, pages 545–552.

Girard, A., Rasmussen, C. E., and Murray-Smith, R. (2002). Gaussian process priors with uncertain inputs: Multiple-step ahead prediction. *Univ. Glasgow, Glasgow, Technical Report TR-2002-119*.

Hairer, E., Nørsett, S. P., and Wanner, G. (1993). *Solving ordinary differential equations I - Nonstiff Problems*. Springer Series in Computational Mathematics. Springer, 2nd edition.

Hairer, E. and Wanner, G. (1996). *Solving ordinary differential equations II - Stiff and Differential-Algebraic Problems*. Springer Series in Computational Mathematics. Springer, 2nd edition.

Hewing, L., Liniger, A., and Zeilinger, M. N. (2017). Cautious nmpc with gaussian process dynamics for miniature race cars. *arXiv preprint arXiv:1711.06586*.

Hewing, L. and Zeilinger, M. N. (2017). Cautious model predictive control using gaussian process regression. *arXiv preprint arXiv:1705.10702*.

Hindmarsh, A. C., Brown, P. N., Grant, K. E., Lee, S. L., Serban, R., Shumaker, D. E., and Woodward, C. S. (2005). SUNDIALS: Suite of nonlinear and differential/algebraic equation solvers. *ACM Transactions on Mathematical Software (TOMS)*, 31(3):363–396.

HSL. A collection of fortran codes for large scale scientific computation. `http://www.hsl.rl.ac.uk/`.

Jordan, D. and Smith, P. (2007). *Nonlinear Ordinary Differential Equations: An introduction for Scientists and Engineers*. Oxford University Press.

Klenske, E. D., Zeilinger, M. N., Schölkopf, B., and Hennig, P. (2016). Gaussian process-based predictive control for periodic error correction. *IEEE Transactions on Control Systems Technology*, 24(1):110–121.

Kocijan, J., Murray-Smith, R., Rasmussen, C. E., and Girard, A. (2004). Gaussian process model based predictive control. In *American Control Conference, 2004. Proceedings of the 2004*, volume 3, pages 2214–2219. IEEE.

Kouvaritakis, B. and Cannon, M. (2016). *Model predictive control*. Springer.

Lay, D. C. (2012). *Linear Algebra and Its Applications*. Addison-Wesley, 4th edition.

Maciejowski, J. M. and Yang, X. (2013). Fault tolerant control using gaussian processes and model predictive control. In *Control and Fault-Tolerant Systems (SysTol), 2013 Conference on*, pages 1–12. IEEE.

Murphy, K. and Bach, F. (2012). *Machine Learning: A Probabilistic Perspective*. Adaptive computation and machine learning. MIT Press.

Murray-Smith, R., Sbarbaro, D., Rasmussen, C. E., and Girard, A. (2003). Adaptive, cautious, predictive control with gaussian process priors. *IFAC Proceedings Volumes*, 36(16):1155–1160.

Neal, R. M. (1996). *Bayesian Learning for Neural Networks*. Springer.

Nguyen-Tuong, D., Seeger, M., and Peters, J. (2009). Model learning with local gaussian process regression. *Advanced Robotics*, 23(15):2015–2034.

Pacejka, H. B. (2012). *Tyre and vehicle dynamics*. Elsevier/BH, 3rd edition.

Press, W. H., Teukolsky, S. A., Vetterling, W. T., and Flannery, B. P. (1992). *Numerical recipies in C*, volume 2. Cambridge university press Cambridge.

Raff, T., Huber, S., Nagy, Z. K., and Allgower, F. (2006). Nonlinear model predictive control of a four tank system: An experimental stability study. In *Control Applications, 2006. CCA'06. IEEE International Conference on*, pages 237–242. IEEE.

Rasmussen, C. E. and Williams, C. K. I. (2006). *Gaussian Processes for Machine Learning*. Adaptive computation and machine learning. MIT Press.

Rawlings, J., Mayne, D. Q., and Diehl, M. M. (2017). *Model Predictive Control: Theory and Design*. Nob Hill Pub.

Rawlings, J., Meadows, E., and Muske, K. (1994). Nonlinear model predictive control: A tutorial and survey. *IFAC Proceedings Volumes*, 27(2):185 – 197. IFAC Symposium on Advanced Control of Chemical Processes, Kyoto, Japan, 25-27 May 1994.

Sacks, J., Welch, W. J., Mitchell, T. J., and Wynn, H. P. (1989). Design and analysis of computer experiments. *Statistical science*, pages 409–423.

Snelson, E. and Ghahramani, Z. (2006). Sparse gaussian processes using pseudo-inputs. In *Advances in neural information processing systems*, pages 1257–1264.

Strang, G. (2006). *Linear Algebra and Its Applications*. Thomson, Brooks/Cole, 4th edition.

Wächter, A. and Biegler, L. T. (2006). On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming. *Mathematical Programming*, 106(1):25–57.

Williams, C. K. I. and Rasmussen, C. E. (1996). Gaussian processes for regression. In *Advances in neural information processing systems*, pages 514–520.

Wright, S. and Nocedal, J. (2006). Numerical optimization. *Springer Science*.

Yoon, Y., Shin, J., Kim, H. J., Park, Y., and Sastry, S. (2009). Model-predictive active steering and obstacle avoidance for autonomous ground vehicles. *Control Engineering Practice*, 17(7):741–750.

Appendix A

# Linear Algebra

This chapter will contain mathematics that are nontrivial and mathematical tricks that may speed up the computation or ensure better numerical stability,

**Matrix Inversion Lemma**

If we have done the effort of obtaining the inverse matrix $\mathbf{A}^{-1}$ of a square $n \times n$ matrix $\mathbf{A}$, it would be beneficial to be able to avoid recomputing the whole inverse matrix if we are only want a small change in $\mathbf{A}$, e.g. one element $a_{ij}$ or a column. To do this efficiently we can use the matrix inversion lemma, also known as Woodbury, Sherman and Morrison formula Press et al. (1992) if our change is on the form

$$\mathbf{A} \rightarrow (\mathbf{A} + \mathbf{U}\mathbf{W}\mathbf{V}^T) \tag{A.1}$$

where $\mathbf{U}$ and $\mathbf{V}$ both have the shape $n \times m$, $W$ is a $n \times n$ matrix. For low rank pertubations ($m < n$) to $\mathbf{A}$, considerable speed up may be achieved. The lemma then states the following

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

**Cholesky factorization**

With a positive definite matrix $A$, the factorization is given as $A = LL^T$, where $L$ is a lower triangular matrix with all positive elements. Lay (2012).

**QR decomposition**

Given a $m \times n$ matrix $A$ with independent columns, the factorization is given as $A = QR$. $R$ is an upper triangular matrix, while $Q$ contain the orthogonalized columns from $A$. Strang (2006).

### Gram Matrix

Given a set of vectors $V = \{\mathbf{v}_1, \ldots, \mathbf{v}_n\}$, the Gram matrix $G$ is the matrix of all possible inner products of $V$ with each element given as

$$g_{ij} = \mathbf{v}_i^T \mathbf{v}_j \tag{A.3}$$

where $i$ and $j$ denotes the row and column respectively. The Gram matrix of a $m \times n$ matrix $A$ is given as $G = A^T A$, where all the elements in the Gram matrix are the inner products of the columns of A. Properties of the Gram matrix is that it is positive definite and symmetric. Lay (2012).

### Determinant computation using QR Decomposition

To speed up the determinant computation of the covariance matrix we can take advantage of the fact the the matrix is symmetric and positive definite, using the QR decomposition, $K = QR$. This means we can reduce the computation of the determinant to a rather cheap computational problem.

$$\det(K) = \det(Q)\det(R) \tag{A.4}$$

The resulting $Q$ matrix is orthogonal, so $\det(Q) = 1$, and since $R$ is triangular, the determinant is just the multiplication of the diagonal elements. If we want that log determinant, this can be simplified further with the log determinant given as the trace of the log of $R$

$$\log(\det(K)) = \log(\det(R)) = \text{trace}(\log(R)) \tag{A.5}$$

### Update Cholesky decomposition of the Gram matrix

Instead of recompute the Gram matrix at each update Nguyen-Tuong et al. (2009) instead propose an efficient method of updating the Cholesky decomposition of the Gram matrix $\mathbf{K}$. If $\mathbf{L}$ is the Cholesky decomposition, then the new matrices $\mathbf{L}_{\text{new}}$ and $\mathbf{K}_{\text{new}}$ are obtained by adding additional row and columns

$$\mathbf{L}_{\text{new}} = \begin{bmatrix} \mathbf{L} & \mathbf{0} \\ \mathbf{l}^T & l_* \end{bmatrix}, \quad \mathbf{K}_{\text{new}} = \begin{bmatrix} \mathbf{K} & \mathbf{k}_{\text{new}}^T \\ \mathbf{k}_{\text{new}} & k_{\text{new}} \end{bmatrix} \tag{A.6}$$

where $\mathbf{k}_{\text{new}} = \mathbf{k}(\mathbf{X}, \mathbf{x}_{\text{new}})$ and $k_{\text{new}} = k(\mathbf{x}_{\text{new}}, \mathbf{x}_{\text{new}})$. $\mathbf{l}$ and $l_*$ can then be solved using

$$\mathbf{L}\mathbf{l} = \mathbf{k}_{\text{new}}, \tag{A.7}$$

$$l_* = \sqrt{k_{\text{new}} - \|\mathbf{l}\|^2} \tag{A.8}$$

Nguyen-Tuong et al. (2009) also discuss how to delete old data points when a maximum number of training points is reached using a permutation matrix $\mathbf{R} = \mathbf{I} - (\delta_m - \delta_n)(\delta_m - \delta_n)^T$, where $\delta_i$ is a zero vector with element $i$ equal to 1. New data can then be inserted in the $n$th row, while the $m$th data can be removed. Instead of using the cholesky matrix directly, we can use the adjusted matrix $\mathbf{R}\mathbf{L}_{\text{new}}\mathbf{L}_{\text{new}}^T\mathbf{R}$.

Appendix B

# GP-MPC: Implementation

This chapter give the detials of the implementation of GP-MPC. A framework written as a part of this thesis for using Gaussian Process together with Model Predictive Control for optimal control. The framework has been implemented with the principles of being flexible enough to experiment with different GP methods, optimization of GP models. and using different MPC schemes and constraints.

The GP methods has been implemented using Hewing and Zeilinger (2017) and Deisenroth and Rasmussen (2011) as references while the MPC algorithm is a nonlinear stochastic MPC implementation based on Rawlings et al. (2017), with probabilistic constraints given by Hewing and Zeilinger (2017) . As a backbone in this framework lay CasADi, Andersson et al. (2018), as a symbolic framework for large scale optimization. For simulation this framework support the solvers provided by CasADi and Sundails, Hindmarsh et al. (2005), for both ODEs (CVODES), and DEAs (IDAS). In addition this framework has implemented a simple RK4 method in CasADi for faster computation of the optimal control problem.

As a model in the MPC algorithm it is possible to use an exact integrator from Sundails (CVODES, IDAS), RK4, GP, a hybrid model consisting of a GP estimating the dynamics and RK4 to integrate the kinematic equation based on the dynamic GP model, or a hybrid where the GP model estimates the noise and modeling error, similar to Hewing and Zeilinger (2017).

**Requirements**

- Python $> 3.5$
- CasADi (tested with version 3.4)

## B.1 GP model

The Gaussian Process is implemented with the support for only the SEard covariance function (2.5), and the following mean functions: zero, constant, linear, and polynomial.

Approximations of the uncertainty propagation is implemented using Mean Equivalence, first order Taylor expansion, and Expected Moments.

Optimization of hyper-parameters can be done using interior-point method (IPOPT) with CasADi or Sequential Least SQuares Programming (SLSQP) using SciPy. Both are constrained nonlinear solvers where a uniform prior ($l_i \in (1e^{-2}, 1e^{2})$) on the hyper-parameters is implemented using the constraints. The optimization can also be done without using a prior on the hyper-parameters using conjugate gradient (CG), but this usually give a worse result so long as we have an idea of which prior to use. To avoid negative hyper-parameters, the CG method require the change to optimize over the log-value of the hyper-parameters. CasADi has the advantage that it give the derivatives automatically, but has problems where the symbolic expression of the Hessian grows out of hand when optimizing the GP model, even for relatively small GP models. For 32bit python the GP model is constricted to less than 50 samples, while 64bit struggle with sample sizes larger than 100. SciPy use finite differences to estimate the derivatives and do not suffer from the same problem with exponential memory use, and is the recommended choice.

## B.2 Numerical simulation

The GP-MPC framework supports both DEA and ODE equations using IDAS integrator from Sundails Hindmarsh et al. (2005) . For ODEs it is also implemented an explicit Runga-Kutta 4 (RK4) integrator to be used as a less computational expensive integration method. IDAS and CVODES are implicit methods that can be used as an exact simulation reference, while RK4 can be used to discretize the system model for use in the MPC problem. It is also support for IDAS and CVODES as the discrete model in the MPC problem, where CasADi is able to provide the derivative of these solvers, but while these are more stable it come to the expense of computation cost.

### B.2.1 Explicit Runga-Kutta

A simple discrete integrator where the fourth order method in equation (B.1) is abbreviated to RK4. It is one of the most widely used methods for simulating ordinary differential equations, Rawlings et al. (2017), as it is simple to implement and computationally cheap.

$$\begin{aligned}
\mathbf{k}_1 &= \mathbf{f}(t, \mathbf{x}) \\
\mathbf{k}_2 &= \mathbf{f}(t + h/2, \mathbf{x} + (h/2)\mathbf{k}_1) \\
\mathbf{k}_3 &= \mathbf{f}(t + h/2, \mathbf{x} + (h/2)\mathbf{k}_2) \\
\mathbf{k}_4 &= \mathbf{f}(t + h, \mathbf{x} + h\mathbf{k}_2) \\
\mathbf{\Phi} &= (h/6)\mathbf{k}_1 + (h/3)\mathbf{k}_2 + (h/3)\mathbf{k}_3 + (h/6)\mathbf{k}_4
\end{aligned} \tag{B.1}$$

From Egeland and Gravdahl (2002) we can get that the numerical method is stable if the magnitude of the stability function is less than or equal to one for all the eigenvalues $\lambda_i$

$$|R(h\lambda_i)| \leq 1, \quad i = 1, \ldots, d \tag{B.2}$$

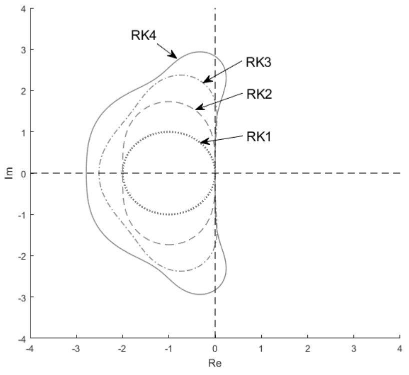


**Figure B.1:** Stability regions for Runga-Kutta methods with a step size of $h = 1$.

of the Jacobian $\boldsymbol{J}$, evaluated at the solution $\boldsymbol{x}_*$

$$\boldsymbol{J} = \left.\frac{\partial \boldsymbol{f}(\boldsymbol{x})}{\partial \boldsymbol{x}^T}\right|_{\boldsymbol{x}=\boldsymbol{x}_*} \tag{B.3}$$

the stability function for a general explicit Runga-Kutta function

$$R(h\lambda) = \det\left[\boldsymbol{I} - \lambda h\left(\boldsymbol{A} - \boldsymbol{1}\boldsymbol{b}^T\right)\right] \tag{B.4}$$

where $\boldsymbol{A}$ and $\boldsymbol{b}$ is the matrices from the Butcher array. For explicit Runga-Kutta methods of order $\sigma$ and stages $p$ less than or equal to four, the stability function is given as the polynomial

$$R(\lambda h) = 1 + h\lambda + \frac{(h\lambda)^2}{2!} + \cdots + \frac{(h\lambda)^p}{p!}, \qquad p = \sigma \leq 4 \tag{B.5}$$

RK4 is a forth order method, giving $\sigma = p = 4$.

The drawback of using Runga Kutta and other explicit methods in general is that they are conditionally stable. Equation (B.5) gives the stability for a certain values of time steps, where an integration method could by this stability definition be stable for systems, and unstable for a different one. To give a more strict stability formulation Egeland and Gravdahl (2002) gives the following definitions for A- and L-stability. A method that is A-stable is then stable for all test systems that are stable. Systems that are A-stable will be stable even if the system dynamics are faster than the time step $h$.

**Definition B.1.** A method is A-stable if $|R(\lambda h)| \leq 1$ for all Re$\lambda \leq 0$.

Even if an A-stable method is stable with fast dynamics it cannot give an accurate estimate of dynamics that are faster than the Nyquist frequency of $\pi/h$, resulting in aliasing. To avoid this it is useful to dampen the fast dynamics, giving the definition of L-stability.

**Definition B.2.** A method is L-stable if it is A-stable and, in addition. if $|R(j\omega h)| \to 0$ when $\omega \to \infty$ for all systems $\dot{y} = \lambda y$ where $\lambda = j\omega$.

### B.2.2 Implicit Methods

According to Egeland and Gravdahl (2002), with explicit RK methods, the time step $h$ can not be chosen such that $h|\lambda_{max}|$ is significantly larger than the unity. This means that no explicit methods can be A-stable. With the explicit methods, the time-step must be selected such that stability is ensured. If the system has a large spread in eigenvalues, the small eigenvalues result in a lot of small time steps to compute both the fast and slow dynamics. This is problematic in terms of time and accuracy. Stiff systems are problems where explicit methods don't work, where there is presence of rapidly damped mode, with time constant small compared to the time scale of the solution.Hairer and Wanner (1996)

To account for this we can use an implicit solver. They are more expensive than the explicit solvers, but are more robust to numerical instability even for relatively large values of time steps. Examples of implicit solvers are implicit Runga-Kutta, or multi-step methods such as Adam-Moulton, or Backward Differentiation Formulas (BDFs). Further reading is referred to Hairer et al. (1993), while their stability analysis on stiff systems is referred to Hairer and Wanner (1996).

### B.2.3 SUNDAILS

The SUNDAILS suite from Hindmarsh et al. (2005), is a suite of equation solvers for nonlinear ODE, DAE and algebraic equations, where the following implicit solvers are supported by the GP-MPC framework.

#### CVODES

CVODES is a part of the SUNDAILS suite, and solves ODE initial value problem in $N$ dimensional space

$$\dot{y} = f(t, y), \quad y(t_0) = y_0 \tag{B.6}$$

with $y \in \mathbb{R}^N$, solving both stiff and non-stiff systems. For non-stiff problems Adams-Moulton formulas is used, while for stiff problems CVODES use Backward Differentiation Formulas (BDFs) in fixed-leading coefficient form.

#### IDAS

Solve differential-algebraic equations (DAE)

$$F(t, y, \dot{y}) = 0, \quad y(t_0) = y_0 \quad \dot{y}(t_0) = \dot{y}_0 \tag{B.7}$$

IDEAS is a part of the SUNDAILS suite from Hindmarsh et al. (2005) and use a variable-order, variable-coefficient BDF in fixed-leading-coefficient form as the integration method.

## B.3 Numerical optimization

### B.3.1 Linear solver

For problems in the form

$$x = A^{-1}y \tag{B.8}$$

The default linear solver in CasADi is the built in direct sparse solver CSparse. For the most cases in this thesis it has been used the plugin for direct sparse linear solver for symmetric systems, ma27 from Harwell Subroutine Library (HSL).

### B.3.2 Nonlinear solver

For constrained nonlinear optimization problems the nonlinear solver used within CasADi is the interior-point method IPOPT from Wächter and Biegler (2006), a primal-dual interior-point algorithm with filter line-search built for large scale optimization. An introduction to interior-point methods is provided by Wright and Nocedal (2006).

### B.3.3 Computing derivatives using CasADi

The framework used in GP-MPC is CasADi from Andersson et al. (2018), a symbolic framework with Algorithmic differentiation (AD). CasADi provide AD on user-defined symbolic expressions, interface to simulation of ODE and IDEs (e.g. Sundails Hindmarsh et al. (2005)) and optimization (e.g. IPOPT Wächter and Biegler (2006)). Every user defined CasADi function passed to a numerical solver automatically provide the necessary derivatives to the solver. For many of the solvers, CasADi also support the derivatives of the solvers themselves. This enables the use of e.g. the CVODES integrator as a constraint in an optimization problem.

### B.3.4 Linearization

Approximation of a nonlinear system with a linear system

$$x_{k+1} = f(x_k, u_k) \tag{B.9a}$$

$$x_{k+1} \approx Ax_k + Bu_k \tag{B.9b}$$

where $f(x, u)$ is linearized with the Jacobian evaluated around a local operating point

$$A = \left.\frac{\partial f(x,u)}{\partial x^T}\right|_{x_0,u_0}, \quad B = \left.\frac{\partial f(x,u)}{\partial u^T}\right|_{x_0,u_0} \tag{B.10}$$

The Jacobian is provided automatically by CasADi, where the linearization is implemented for both the system model (ODE/DEA) and GP model.

## B.4 MPC

The GP-MPC framework has implemented the stochastic MPC problem in (3.10) using the simultaneous approach, multiple shooting, with optional terminal constraint and optional feedback using LQR on the approximated linearized system. For cost function both expected value of quadratic and saturated cost has been implemented. The optimization problem is solved using IPOPT as nonlinear solver Wächter and Biegler (2006) with ma27 as a linear solver HSL. The first and second derivatives are automatically provided by CasADi. For prediction models it is support for RK4, CVODES, GP, hybrid-GP, and deterministic hybrid-GP where the variance is defined as zero.

Appendix C

# GP-MPC: Documentation

This appendix contain all the code used in this thesis, where it all is implemented as the python package gp_mpc. See `https://github.com/helgeanl/GP-MPC` for the most recent updates.

**Requirements**

- Python $>$ 3.5
- CasADi (tested with version 3.4)

## C.1 gp_class

**Available class functions:**

```
GP.__init__(...)
GP.optimize(...)
GP.validate(...)
GP.set_method(...)
GP.predict(...)
GP.get_size()
GP.get_hyper_parameter()
gp.print_hyper_parameters()
GP.covSEard(...)
GP.covar(...)
GP.update_data(...)
GP.covSEard(...)
GP.standardize(...)
GP.normalize(...)
GP.inverse_mean(...)
GP.inverse_variance(...)
GP.discrete_linearize(...)
GP.jacobian(...)
GP.noise_variance()
GP.__to_dict()
GP.save_model(...)
GP.load_model(...)
GP.predict_compare(...)
GP.inverse_variance(...)
```

## C.2 gp_functions

**Available functions:**

```
covSEard(...)
get_mean_function(...)
build_gp(...)
build_TA_cov(...)
gp(...)
gp_taylor_approx(...)
gp_exact_moment(...)
maha(...)
```

## C.3 optimize

**Available functions:**

```
calc_NLL(...) # CasADi
train_gp(...) #CasADi
calc_cov_matrix(...) # Numpy/Scipy
calc_NLL_numpy(...) # Numpy/Scipy
train_gp_numpy(...) # Numpy/Scipy
validate(...)
```

## C.4 model_class

**Available class functions:**

```
Model.__init__(...)
Model.linearize(...)
Model.discrete_linearize(...)
Model.discrete_rk4_linearize(...)
Model.rk4_jacobian_x(...)
Model.rk4_jacobian_u(...)
Model.check_rk4_stability(...)
Model.sampling_time()
Model.size()
Model.integrate(...)
Model.sim(...)
Model.generate_training_data(...)
Model.plot(...)
Model.predict_compare(...)
```

## C.5 mpc_class

**Available class functions:**

```
MPC.__init__(...)
MPC.solve(...)
MPC.__set_cost_function(...)
MPC.__cost_saturation_lf(...)
MPC.__cost_saturation_l(...)
MPC.__cost_l(...)
MPC.__constraint(...)
MPC.__debug(...)
MPC.plot(...)
lqr(...)
plot_eiq(...)
```

Appendix D

# GP-MPC: Code Example

This chapter contain the following examples:

- Use a Gussian Process to estimate the Van der Pol equations.
- Train a GP model on data from a four-tank system and control the system using MPC with the GP as model in the prediction horizon.
- Train GP model on a car model and use this with MPC to avoid obstacles and follow a road path.

## D.1 Van der Pol

```
# -*- coding: utf-8 -*-
"""
Example of predicting the Van der Pol equation with a Gaussian Process

@author: Helge-André Langåker
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from sys import path
path.append(r"./../") # Add gp_mpc pagkage to path

import numpy as np
import casadi as ca
import matplotlib.pyplot as plt
from gp_mpc import Model, GP, MPC, plot_eig, lqr


def plot_van_der_pol():
    """ Plot comparison of GP prediction with exact simulation
        on a 2000 step prediction horizon
    """
    Nt = 2000
    x0 = np.array([2., .201])

    cov = np.zeros((2,2))
    x = np.zeros((Nt,2))
    x_sim = np.zeros((Nt,2))

    x[0] = x0
    x_sim[0] = x0

    gp.set_method('ME')          # Use Mean Equivalence as GP method
    for i in range(Nt-1):
        x_t, cov = gp.predict(x[i], [], cov)
        x[i + 1] = np.array(x_t).flatten()
        x_sim[i+1] = model.integrate(x0=x_sim[i], u=[], p=[])

    plt.figure()
    ax = plt.subplot(111)
    ax.plot(x_sim[:,0], x_sim[:,1], 'k-', linewidth=1.0, label='Exact')
    ax.plot(x[:,0], x[:,1], 'b-', linewidth=1.0, label='GP')
    ax.set_ylabel('y')
    ax.set_xlabel('x')
    plt.legend(loc='best')
    plt.show()


def ode(x, u, z, p):
    """ Van der Pol equation
    """
    mu = 2
    dxdt = [
            x[1],
            -x[0] + mu * (1 - x[0]**2) * x[1]
    ]
    return  ca.vertcat(*dxdt)


""" System Parameters """
dt = .01                        # Sampling time
Nx = 2                          # Number of states
Nu = 0                          # Number of inputs
R_n = np.eye(Nx) * 1e-6         # Covariance matrix of added noise

# Limits in the training data
ulb = []     # No inputs are used
uub = []     # No inputs are used
xlb = [-4., -6.]
xub = [4., 6.]

N = 40             # Number of training data
N_test = 100       # Number of test data

""" Create simulation model and generate training/test data"""
model          = Model(Nx=Nx, Nu=Nu, ode=ode, dt=dt, R=R_n, clip_negative=True)
X, Y           = model.generate_training_data(N, uub, ulb, xub, xlb, noise=True)
X_test, Y_test = model.generate_training_data(N_test, uub, ulb, xub, xlb, noise=True)
```

```
""" Create GP model and optimize hyper-parameters"""
gp = GP(X, Y, mean_func='zero', normalize=True, xlb=xlb, xub=xub, ulb=ulb,
        uub=uub, optimizer_opts=None)
gp.validate(X_test, Y_test)

""" Predict and plot the result """
plot_van_der_pol()
```

## D.2 Control of a Four-Tank System

This script provide an example of using GP-MPC for fitting a GP model to a system consisting of four tanks, and solving the MPC problem for 80 iterations using simulated measurements.

```
# -*- coding: utf-8 -*-
"""
@author: Helge-André Langåker
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

from sys import path
path.append(r"./../") # Add gp_mpc pagkage to path

import numpy as np
import casadi as ca
from gp_mpc import Model, GP, MPC


def ode(x, u, z, p):
    # Model Parameters (Raff, Tobias et al., 2006)
    g = 981
    a1 = 0.233
    a2 = 0.242
    a3 = 0.127
    a4 = 0.127
    A1 = 50.27
    A2 = 50.27
    A3 = 28.27
    A4 = 28.27
    gamma1 = 0.4
    gamma2 = 0.4

    dxdt = [
            (-a1 / A1) * ca.sqrt(2 * g * x[0] + 1e-3) + (a3 / A1 )
                * ca.sqrt(2 * g * x[2] + 1e-3) + (gamma1 / A1) * u[0],
            (-a2 / A2) * ca.sqrt(2 * g * x[1] + 1e-3) + a4 / A2
                * ca.sqrt(2 * g * x[3]+ 1e-3) + (gamma2 / A2) * u[1],
            (-a3 / A3) * ca.sqrt(2 * g * x[2] + 1e-3) + (1 - gamma2) / A3 * u[1],
                (-a4 / A4) * ca.sqrt(2 * g * x[3] + 1e-3) + (1 - gamma1) / A4 * u[0]
    ]

    return  ca.vertcat(*dxdt)
```

```
""" System parameters """
dt = 3.0
Nx = 4
Nu = 2
R = np.eye(Nx) * 1e-5 # Noise covariance

""" Limits in the training data """
ulb = [0., 0.]
uub = [60., 60.]
xlb = [.0, .0, .0, .0]
xub = [30., 30., 30., 30.]

N = 60          # Number of training data
N_test = 100    # Number of test data

""" Create Simulation Model """
model          = Model(Nx=Nx, Nu=Nu, ode=ode, dt=dt, R=R, clip_negative=True)
X, Y           = model.generate_training_data(N, uub, ulb, xub, xlb, noise=True)
X_test, Y_test = model.generate_training_data(N_test, uub, ulb, xub, xlb, noise=True)


if 1:
    """ Create GP model and optimize hyper-parameters on training data """
    gp = GP(X, Y, mean_func='zero', normalize=True, xlb=xlb, xub=xub, ulb=ulb,
            uub=uub)
    gp.save_model('models/gp_tank')
else:
    """ Or Load Example Model"""
    gp = GP.load_model('models/gp_tank_example')
gp.validate(X_test, Y_test)
gp.print_hyper_parameters()

""" Limits in the MPC problem """
ulb = [10., 10.]
uub = [60., 60.]
xlb = [7.5, 7.5, 3.5, 4.5]
xub = [28., 28., 28., 28.]

""" Initial state, input and set point  """
x_sp = np.array([14.0, 14.0, 14.2, 21.3])
x0 = np.array([8., 10., 8., 19.])
u0 = np.array([45, 45])

""" Penalty matrices """
Q = np.diag([20, 20, 10, 10])   # State penalty
R = np.diag([1e-3, 1e-3])       # Input penalty
S = np.diag([.01, .01])         # Input change penalty

""" Options to pass to the MPC solver """
solver_opts = {
        #    'ipopt.linear_solver' : 'ma27',    # Plugin solver from HSL
            'ipopt.max_cpu_time' : 30,
            'expand' : True,
}
```

```
""" Build MPC solver """
mpc = MPC(horizon=30*dt, gp=gp, model=model,
          gp_method='TA',
          ulb=ulb, uub=uub, xlb=xlb, xub=xub, Q=Q, R=R, S=S,
          terminal_constraint=None, costFunc='quad', feedback=True,
          solver_opts=solver_opts, discrete_method='gp',
          inequality_constraints=None
          )

""" Solve and plot the MPC solution, simulating 80 iterations """
x, u = mpc.solve(x0, u0=u0, sim_time=80*dt, x_sp=x_sp, debug=False, noise=True)
mpc.plot(xnames=['Tank 1 [cm]', 'Tank 2 [cm]','Tank 3 [cm]','Tank 4 [cm]'],
        unames=['Pump 1 [ml/s]', 'Pump 2 [ml/s]'])
```

## D.3 Vehicle Obstacle Avoidance

This script provide an example of using GP-MPC for training a GP model on a bicycle car model, predicting open/closed loop, building MPC constraints, and solve the MPC problem for 50 iterations using simulated measurements.

```
# -*- coding: utf-8 -*-
"""
@author: Helge-André Langåker
"""
from sys import path
path.append(r"./../") # Add gp_mpc pagkage to path

import numpy as np
import casadi as ca
import scipy.linalg
import matplotlib.pyplot as plt
from matplotlib.patches import Ellipse, Rectangle
from gp_mpc import Model, GP, MPC, plot_eig, lqr



def ode(x, u, z, p):
    """ Full Bicycle Model
    """
    # Model Parameters (Gao et al., 2014)
    g   = 9.18                          # Gravity [m/s^2]
    m   = 2050                          # Vehicle mass [kg]
    Iz  = 3344                          # Yaw inertia [kg*m^2]
    Cr   = 65000                        # Tyre corning stiffness [N/rad]
    Cf   = 65000                        # Tyre corning stiffness [N/rad]
    mu  = 0.5                           # Tyre friction coefficient
    l   = 4.0                           # Vehicle length
    lf  = 2.0                           # Distance from CG to the front tyre
    lr  = l - lf                        # Distance from CG to the rear tyre
    Fzf = lr * m * g / (2 * l)          # Vertical load on front wheels
    Fzr = lf * m * g / (2 * l)          # Vertical load on rear wheels
    eps = 1e-20                         # Small epsilon to avoid dividing by zero

    dxdt = [
                1/m * (m*x[1]*x[2] + 2*mu*Fzf*u[0] + 2*Cf*u[1]**2
                    - 2*Cf*u[1] * (x[1] + lf*x[2]) / (x[0] + eps) + 2*mu*Fzr*u[0]),
                1/m * (-m*x[0]*x[2] + 2*mu*Fzf*u[1]*u[0]
                    + 2*Cf*(x[1] + lf*x[2]) / (x[0] + eps) - 2*Cf*u[1]
                    + 2*Cr*(x[1] - lf*x[2]) / (x[0] + eps)),
                1/Iz * (2*lf*mu*Fzf*u[0]*u[1] + 2*lf*Cf*(x[1] + lf*x[2]) / (x[0] + eps)
                    - 2*lf*Cf*u[1] - 2*lr*Cr*(x[1] - lf*x[2]) / (x[0] + eps)),
                x[2],
                x[0]*ca.cos(x[3]) - x[1]*ca.sin(x[3]),
                x[0]*ca.sin(x[3]) + x[1]*ca.cos(x[3])
            ]
    return  ca.vertcat(*dxdt)


def ode_gp(x, u, z, p):
    """ Dynamic equation of Bicycle model for use with GP model
    """
    # Model Parameters (Gao et al., 2014)
    g   = 9.18                          # Gravity [m/s^2]
    m   = 2050                          # Vehicle mass [kg]
    Iz  = 3344                          # Yaw inertia [kg*m^2]
    Cr   = 65000                         # Tyre corning stiffness [N/rad]
    Cf   = 65000                        # Tyre corning stiffness [N/rad]
    mu  = 0.5                           # Tyre friction coefficient
    l   = 4.0                           # Vehicle length
    lf  = 2.0                           # Distance from CG to the front tyre
    lr  = l - lf                        # Distance from CG to the rear tyre
    Fzf = lr * m * g / (2 * l)          # Vertical load on front wheels
    Fzr = lf * m * g / (2 * l)          # Vertical load on rear wheels
    eps = 1e-10                          # Small epsilon to avoid dividing by zero

    dxdt = [
                1/m * (m*x[1]*x[2] + 2*mu*Fzf*u[0] + 2*Cf*u[1]**2
                    - 2*Cf*u[1] * (x[1] + lf*x[2]) / (x[0] + eps) + 2*mu*Fzr*u[0]),
                1/m * (-m*x[0]*x[2] + 2*mu*Fzf*u[1]*u[0]
                    + 2*Cf*(x[1] + lf*x[2]) / (x[0] + eps) - 2*Cf*u[1]
                    + 2*Cr*(x[1] - lf*x[2]) / (x[0] + eps)),
                1/Iz * (2*lf*mu*Fzf*u[0]*u[1] + 2*lf*Cf*(x[1] + lf*x[2]) / (x[0] + eps)
                    - 2*lf*Cf*u[1] - 2*lr*Cr*(x[1] - lf*x[2]) / (x[0] + eps)),
            ]
    return  ca.vertcat(*dxdt)


def ode_hybrid(x, u, z, p):
    """ Kinematic equations of Bicycle model for use with hybrid model
    """
    dxdt = [
                u[2],
                u[0]*ca.cos(x[0]) - u[1]*ca.sin(x[0]),
                u[0]*ca.sin(x[0]) + u[1]*ca.cos(x[0])
            ]
    return  ca.vertcat(*dxdt)


def constraint_parameters(x):
    """ Constraint parameters to send to the solver at each iteration
    """
    car_pos = x[4:]
    dist = np.sqrt((car_pos[0] - obs[:,0])**2
                   + (car_pos[1] - obs[:, 1])**2 )
    if min(dist) > 40:
        return np.hstack([car_pos * 1000, [0,0]])

    return obs[np.argmin(dist)]


def inequality_constraints(x, covar, u, eps, par):
    """ Inequality constraints to send to the MPC builder
    """
    con_ineq = []
    con_ineq_lb = []
    con_ineq_ub = []

    """ Slip angle constraint """
    dx_s = ca.SX.sym('dx')
    dy_s = ca.SX.sym('dy')
    dpsi_s = ca.SX.sym('dpsi')
    delta_f_s = ca.SX.sym('delta_f')
    lf   = 2.0
    lr   = 2.0

    slip_f = ca.Function('slip_f', [dx_s, dy_s, dpsi_s, delta_f_s],
                         [(dy_s + lf*dpsi_s)/(dx_s + 1e-6)    - delta_f_s])
    slip_r = ca.Function('slip_r', [dx_s, dy_s, dpsi_s],
                         [(dy_s - lr*dpsi_s)/(dx_s + 1e-6)])

    con_ineq.append(slip_f(x[0], x[1], x[2], u[1]) - slip_max - eps[0])
    con_ineq_ub.append(0)
    con_ineq_lb.append(-np.inf)

    con_ineq.append(slip_min - slip_f(x[0], x[1], x[2], u[1]) - eps[0])
    con_ineq_ub.append(0)
    con_ineq_lb.append(-np.inf)

    con_ineq.append(slip_r(x[0], x[1], x[2]) - slip_max - eps[0])
    con_ineq_ub.append(0)
    con_ineq_lb.append(-np.inf)

    con_ineq.append(slip_min - slip_r(x[0], x[1], x[2]) - eps[0])
    con_ineq_ub.append(0)
    con_ineq_lb.append(-np.inf)

    """ Add road boundry constraints """
    con_ineq.append(x[5] - eps[1])
    con_ineq_ub.append(road_bound)
    con_ineq_lb.append(-road_bound)

    """ Obstacle avoidance """
    Xcg_s = ca.SX.sym('Xcg')
    Ycg_s = ca.SX.sym('Ycg')
    obs_s = ca.SX.sym('obs', 4)
    ellipse = ca.Function('ellipse', [Xcg_s, Ycg_s, obs_s],
                          [ ((Xcg_s - obs_s[0]) / (obs_s[2] + car_length))**2
                           + ((Ycg_s - obs_s[1]) / (obs_s[3] + car_width))**2] )
    con_ineq.append(1 - ellipse(x[4], x[5], par) - eps[2])
    con_ineq_ub.append(0)
    con_ineq_lb.append(-np.inf)

    cons = dict(con_ineq=con_ineq,
                con_ineq_lb=con_ineq_lb,
                con_ineq_ub=con_ineq_ub
            )
    return cons


""" Dynamic Model options"""
dt = 0.05
Nx = 3
Nu = 2
R_n = np.diag([1e-5, 1e-8, 1e-8])

""" Training data options """
N = 200             # Number of training data
N_test = 500        # Number of validation data

normalize = False   # Option to normalize data in GP model

""" Limits in the training data """
ulb = [-.5, -.04]
uub = [.5, .04]
xlb = [10.0, -.6, -.2]
xub = [30.0, .6, .2]

""" Create simulation model """
model_gp = Model(Nx=Nx, Nu=Nu, ode=ode_gp, dt=dt, R=R_n)

""" Generate training and test data  """
X, Y     = model_gp.generate_training_data(N, uub, ulb, xub, xlb, noise=True)
X_test, Y_test = model_gp.generate_training_data(N_test, uub, ulb, xub, xlb, noise=False)

""" Options for hyper-parameter optimization """
solver_opts = {}
solver_opts['ipopt.linear_solver'] = 'ma27' # Faster plugin solver than default
solver_opts['expand']= False                # Choise between SX or MX graph

if 0:
    """ Create GP model estimating dynamics from car model """
    gp = GP(X, Y, ulb=ulb, uub=uub, optimizer_opts=solver_opts, normalize=normalize)
    SMSE, MNLP = gp.validate(X_test, Y_test)
    gp.save_model('models/gp_car')
else:
    """ Load example model """
    gp = GP.load_model('models/gp_car_example')

""" Predict GP open/closed loop """
# Test data
x0 = np.array([13.89, 0.0, 0.0])
x_sp = np.array([13.89, 0., 0.001])
u0 = [0.0, 0.0]
cov0 = np.eye(Nx+Nu)
```

```
t = np.linspace(0,20*dt, 20)
u_i = np.sin(0.01*t) * 0
u_test = np.vstack([0.5*u_i, 0.02*u_i]).T

# Penalty matrices for LQR
Q = np.diag([.1, 10., 50.])
R = np.diag([.1, 1])

# Name of states for use with plotting
xnames = [r'$\dot{x}$', r'$\dot{y}$', r'$\dot{\psi}$']

# Predict and plot open loop GP using fixed inputs
gp.predict_compare(x0, u_test, model_gp, feedback=False, x_ref=x_sp, Q=Q, R=R,
                   methods = ['TA','ME'], num_cols=1, xnames=xnames)
# Predict and plot closed loop GP using LQR feedback
gp.predict_compare(x0, u_test, model_gp, feedback=True, x_ref=x_sp, Q=Q, R=R,
                   methods = ['TA', 'ME'], num_cols=1, xnames=xnames)


""" Create hybrid model with state integrator """
Nx = 6
R_n = np.diag([1e-5, 1e-8, 1e-8, 1e-8, 1e-5, 1e-5])
model_hybrid   = Model(Nx=3, Nu=3, ode=ode_hybrid, dt=dt, R=R_n)
model          = Model(Nx=Nx, Nu=Nu, ode=ode, dt=dt, R=R_n)

""" Options for MPC solver"""
solver_opts = {}
#solver_opts['ipopt.linear_solver'] = 'ma27' # Plugin solver from HSL
solver_opts['ipopt.max_cpu_time'] = 20
solver_opts['expand']= True
solver_opts['ipopt.expect_infeasible_problem'] = 'yes'

# Constraint parameters
slip_min = -4.0 * np.pi / 180
slip_max = 4.0 * np.pi / 180
road_bound = 2.0
car_width = 1.2
car_length = 5.

# Position and size of eliptical obsticles [x, y, a, b]
obs = np.array([[20, .3, 0.01, 0.01],
               [60, -0.3, .01, .01],
               [100, 0.3, .01, .01],
               ])

# Limits in the MPC problem
ulb = [-.5, -.04]
uub = [.5, .04]
xlb = [10.0, -.5, -.2, -.3, .0, -10]
xub = [30.0, .5, .2, .3, 500, 10]

# Penalty matrices
Q = np.diag([.001, 5., 1., .1, 1e-10, 1])
R = np.diag([.1, 1])
S = np.diag([1, 10])

# Penalty in soft constraint
```

```
lam = 500

# Initial value and set point
x0 = np.array([13.89, 0.0, 0.0, 0.0,.0 , 0.0])
x_sp = np.array([13.89, 0., 0., 0., 100., 0. ])

""" Build MPC object"""
mpc = MPC(horizon=17*dt, model=model,gp=gp, hybrid=model_hybrid,
          discrete_method='hybrid', gp_method='ME',
          ulb=ulb, uub=uub, xlb=xlb, xub=xub, Q=Q, R=R, S=S, lam=lam,
          terminal_constraint=None, costFunc='quad', feedback=True,
          solver_opts=solver_opts,
          inequality_constraints=inequality_constraints, num_con_par=4
          )

""" Simulate measurments and solve MPC problem with constraints for 50 steps"""
x, u = mpc.solve(x0, sim_time=50*dt, x_sp=x_sp, debug=False, noise=True,
                 con_par_func=constraint_parameters)
""" Plot """
mpc.plot()
```

Appendix E

# GP-MPC: Code

This appendix contain all the code used in this thesis, where it all is implemented as the python package gp_mpc. See `https://github.com/helgeanl/GP-MPC` for the most recent updates.

**Requirements**

- Python > 3.5
- CasADi (tested with version 3.4)

## E.1 gp_class

```
# -*- coding: utf-8 -*-
"""
Gaussian Process Model
Copyright (c) 2018, Helge-André Langåker
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import time
import numpy as np
import casadi as ca
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties
from .gp_functions import gp_exact_moment, gp_taylor_approx, gp
from .gp_functions import build_gp, build_TA_cov, get_mean_function
from .optimize import train_gp, train_gp_numpy
from .mpc_class import lqr

class GP:
    def __init__(self, X, Y, mean_func="zero", gp_method="TA",
                 optimizer_opts=None, hyper=None, normalize=True, multistart=1,
                 xlb=None, xub=None, ulb=None, uub=None, meta=None,
                 optimize_nummeric=True):
```

```
        """ Initialize and optimize GP model

        """

        X = np.array(X).copy()
        Y = np.array(Y).copy()
        self.__X = X
        self.__Y = Y
        self.__Ny = Y.shape[1]
        self.__Nx = X.shape[1]
        self.__N = X.shape[0]
        self.__Nu = self.__Nx - self.__Ny

        self.__gp_method = gp_method
        self.__mean_func = mean_func
        self.__normalize = normalize

        if meta is not None:
            self.__meanY = np.array(meta['meanY'])
            self.__stdY  = np.array(meta['stdY'])
            self.__meanZ= np.array(meta['meanZ'])
            self.__stdZ = np.array(meta['stdZ'])
            self.__meanX = np.array(meta['meanX'])
            self.__stdX = np.array(meta['stdX'])
            self.__meanU = np.array(meta['meanU'])
            self.__stdU  = np.array(meta['stdU'])

        """ Optimize hyperparameters """
        if hyper is None:
            self.optimize(X=X, Y=Y, opts=optimizer_opts, mean_func=mean_func,
                          xlb=xlb, xub=xub, ulb=ulb, uub=uub,
                          multistart=multistart, normalize=normalize,
                          optimize_nummeric=optimize_nummeric)
        else:
            self.__hyper   = np.array(hyper['hyper'])
            self.__invK    = np.array(hyper['invK'])
            self.__alpha   = np.array(hyper['alpha'])
            self.__chol    = np.array(hyper['chol'])
            self.__hyper_length_scales    = np.array(hyper['length_scale'])
            self.__hyper_signal_variance = np.array(hyper['signal_var'])
            self.__hyper_noise_variance   = np.array(hyper['noise_var'])
            self.__hyper_mean = np.array(hyper['mean'])

        # Build GP
        self.__mean, self.__var, self.__covar, self.__mean_jac = \
                                build_gp(self.__invK, self.__X, self.__hyper,
                                         self.__alpha, self.__chol)
        self.__TA_covar = build_TA_cov(self.__mean, self.__covar,
                                       self.__mean_jac, self.__Nx, self.__Ny)

        self.set_method(gp_method)


    def optimize(self, X=None, Y=None, opts=None, mean_func='zero',
                 xlb=None, xub=None, ulb=None, uub=None,
                 multistart=1, normalize=True, warm_start=False,
                 optimize_nummeric=True):
```

```
self.__mean_func = mean_func
self.__normalize = normalize

if normalize and X is not None:
    self.__xlb = np.array(xlb)
    self.__xub = np.array(xub)
    self.__ulb = np.array(ulb)
    self.__uub = np.array(uub)
    self.__lb = np.hstack([xlb, ulb])
    self.__ub = np.hstack([xub, uub])
    self.__meanY = np.mean(Y, 0)
    self.__stdY  = np.std(Y, 0)
    self.__meanZ = np.mean(X, 0)
    self.__stdZ  = np.std(X, 0)
    self.__meanX = np.mean(X[:, :self.__Ny], 0)
    self.__stdX  = np.std(X[:, :self.__Ny], 0)
    self.__meanU = np.mean(X[:, self.__Ny:], 0)
    self.__stdU  = np.std(X[:, self.__Ny:], 0)


if X is not None:
    X = np.array(X).copy()
    if normalize and X is not None:
        self.__X = self.standardize(X, self.__meanZ, self.__stdZ)
    else:
        self.__X = X.copy()
else:
    X = self.__X.copy()
if Y is not None:
    Y = np.array(Y).copy()
    if normalize and X is not None:
        self.__Y = self.standardize(Y, self.__meanY, self.__stdY)
    else:
        self.__Y = Y.copy()
else:
    Y = self.__Y.copy()

if warm_start:
    hyp_init = self.__hyper
    lam_x = self.__lam_x
else:
    hyp_init = None
    lam_x = None
if optimize_nummeric:
    opt = train_gp_numpy(self.__X, self.__Y, meanFunc=self.__mean_func,
                       optimizer_opts=opts, multistart=multistart,
                       hyper_init=hyp_init)
else:
    opt = train_gp(self.__X, self.__Y, meanFunc=self.__mean_func,
                       optimizer_opts=opts, multistart=multistart,
                       hyper_init=hyp_init, lam_x0=lam_x)

self.__hyper = opt['hyper']
self.__lam_x = opt['lam_x']
self.__invK   = opt['invK']
self.__alpha = opt['alpha']
self.__chol   = opt['chol']
```

```
        self.__hyper_length_scales   = self.__hyper[:, :self.__Nx]
        self.__hyper_signal_variance = self.__hyper[:, self.__Nx]**2
        self.__hyper_noise_variance  = self.__hyper[:, self.__Nx + 1]**2
        self.__hyper_mean            = self.__hyper[:, (self.__Nx + 1):]


    def validate(self, X_test, Y_test):
        """ Validate GP model with test data
        """

        Y_test = Y_test.copy()
        X_test = X_test.copy()
        if self.__normalize:
            Y_test = self.standardize(Y_test, self.__meanY, self.__stdY)
            X_test = self.standardize(X_test, self.__meanZ, self.__stdZ)

        N, Ny = Y_test.shape
        loss = 0
        NLP = 0

        for i in range(N):
            mean = self.__mean(X_test[i, :])
            var = self.__var(X_test[i, :]) + self.noise_variance()
            loss += (Y_test[i, :] - mean)**2
            NLP += 0.5*np.log(2*np.pi * (var)) + ((Y_test[i, :] - mean)**2)/(2*var)

        loss = loss / N
        SMSE = loss/ np.std(Y_test, 0)
        MNLP = NLP / N

        print('\n________________________________________')
        print('# Validation of GP model ')
        print('----------------------------------------')
        print('* Num training samples: ' + str(self.__N))
        print('* Num test samples: ' + str(N))
        print('----------------------------------------')
        print('* Mean squared error: ')
        for i in range(Ny):
            print('\t- State %d: %f' % (i + 1, loss[i]))
        print('----------------------------------------')
        print('* Standardized mean squared error:')
        for i in range(Ny):
            print('\t* State %d: %f' % (i + 1, SMSE[i]))
        print('----------------------------------------')
        print('* Mean Negative log Probability:')
        for i in range(Ny):
            print('\t* State %d: %f' % (i + 1, MNLP[i]))
        print('----------------------------------------\n')

        self.__SMSE = np.max(SMSE)

        return np.array(SMSE).flatten(), np.array(MNLP).flatten()


    def set_method(self, gp_method='TA'):
        """ Select wich GP function to use
        # Arguments:
            gp_method: Method for propagating uncertainty.
                        'ME': Mean Equivalence (normal GP),
                        'TA': 1st order Tayolor Approximation,
                        'EM': 1st and 2nd Expected Moments,
                        'old_ME': non-optimized ME function,
                        'old_TA': non-optimzed TA function. Use 2nd order
                                    TA, but don't take into account covariance
                                    between states.
        """

        x = ca.MX.sym('x', self.__Ny)
        covar_s = ca.MX.sym('covar', self.__Nx, self.__Nx)
        u = ca.MX.sym('u', self.__Nu)
        self.__gp_method = gp_method

        if gp_method is 'ME':
            self.__predict = ca.Function('gp_mean', [x, u, covar_s],
                                [self.__mean(ca.vertcat(x,u)),
                                 self.__covar(ca.vertcat(x,u))])
        elif gp_method is 'TA':
            self.__predict = ca.Function('gp_taylor', [x, u, covar_s],
                                [self.__mean(ca.vertcat(x,u)),
                                 self.__TA_covar(ca.vertcat(x,u), covar_s)])
        elif gp_method is 'EM':
            self.__predict = ca.Function('gp_exact_moment', [x, u, covar_s],
                                gp_exact_moment(self.__invK, ca.MX(self.__X),
                                        ca.MX(self.__Y), ca.MX(self.__hyper),
                                        ca.vertcat(x, u).T, covar_s))
        elif gp_method is 'old_ME':
            self.__predict = ca.Function('gp_mean', [x, u, covar_s],
                                gp(self.__invK, ca.MX(self.__X), ca.MX(self.__Y),
                                   ca.MX(self.__hyper),
                                   ca.vertcat(x, u).T, meanFunc=self.__mean_func))
        elif gp_method is 'old_TA':
            self.__predict = ca.Function('gp_taylor_approx', [x, u, covar_s],
                                gp_taylor_approx(self.__invK, ca.MX(self.__X),
                                        ca.MX(self.__Y), ca.MX(self.__hyper),
                                        ca.vertcat(x, u).T, covar_s,
                                        meanFunc=self.__mean_func, diag=True))
        else:
            raise NameError('No GP method called: ' + gp_method)

        self.__discrete_jac_x = ca.Function('jac_x', [x, u, covar_s],
                                      [ca.jacobian(self.__predict(x,u, covar_s)[0], x)])
        self.__discrete_jac_u = ca.Function('jac_x', [x, u, covar_s],
                                      [ca.jacobian(self.__predict(x,u,covar_s)[0], u)])


    def predict(self, x, u, cov):
        """ Predict future state

        # Arguments:
            x: State vector (Nx x 1)
            u: Input vector (Nu x 1)
            cov: Covariance matrix of input z=[x, u] (Nx+nu x Nx+Nu)
        """
        if self.__normalize:
            x_s = self.standardize(x, self.__meanX, self.__stdX)
            u_s = self.standardize(u, self.__meanU, self.__stdU)
        else:
            x_s = x
            u_s = u
        mean, cov = self.__predict(x_s, u_s, cov)
        if self.__normalize:
            mean = self.inverse_mean(mean, self.__meanY, self.__stdY)
#            cov = self.inverse_covariance(cov, self.__stdY)
        return mean, cov


    def get_size(self):
        """ Get the size of the GP model

        # Returns:
                N:  Number of training data
                Ny: Number of outputs
                Nu: Number of control inputs
        """
        return self.__N, self.__Ny, self.__Nu


    def get_hyper_parameters(self):
        """ Get hyper-parameters

        # Return:
            hyper: Dictionary containing the hyper-parameters,
                    'length_scale', 'signal_var', 'noise_var', 'mean'
        """
        hyper = dict(
                    length_scale = self.__hyper_length_scales,
                    signal_var = self.__hyper_signal_variance,
                    noise_var = self.__hyper_noise_variance,
                    mean = self.__hyper_mean
                )
        return hyper


    def print_hyper_parameters(self):
        """ Print out all hyperparameters
        """
        print('\n________________________________________')
        print('# Hyper-parameters')
        print('----------------------------------------')
        print('* Num samples:', self.__N)
        print('* Ny:', self.__Ny)
        print('* Nu:', self.__Nu)
        print('* Normalization:', self.__normalize)
        for state in range(self.__Ny):
            print('----------------------------------------')
            print('* Lengthscale: ', state)
            for i in range(self.__Ny + self.__Nu):
                print(('-- l{a}: {l}').format(a=i,l=self.__hyper_length_scales[state, i]))
            print('* Signal variance: ', state)
            print('-- sf2:', self.__hyper_signal_variance[state])
            print('* Noise variance: ', state)
            print('-- sn2:', self.__hyper_noise_variance[state])
        print('----------------------------------------')

    def covSEard(self, X, Z, ell, sf2):
        """ GP Squared Exponential Kernel

        # Arguments:
            X: Input matrix or vector (n_x x D)
            Z: Input matrix or vector (n_z x D)
            ell: Lengthscale vector (D x 1)
            sf2: Signal variance (scalar)

        # Returns:
            k(X,Z): Covariance between X and Z.
        """
        dist = 0

        if X.ndim > 1:
            n1, D = X.shape
        else:
            D = X.shape[0]
            n1 = 1
            X.shape = (n1, D)

        if Z.ndim > 1:
            n2, D2 = Z.shape
        else:
            D2 = Z.shape[0]
            n2 = 1
            Z.shape = (n2, D2)

        if D != D2:
            raise ValueError('Input dimensions are not the same! D_x=' + str(D)
                                 + ', D_z=' + str(D2))
        for i in range(D):
            x1 = X[:, i].reshape(n1, 1)
            x2 = Z[:, i].reshape(n2, 1)
            dist = (np.sum(x1**2, 1).reshape(-1, 1) + np.sum(x2**2, 1) -
                2 * np.dot(x1, x2.T)) / ell[i]**2 + dist
        return sf2 * np.exp(-.5 * dist)


    def covar(self, X_new):
        """ Compute covariance of input data

        # Arguments:
            X_new: Input matrix or vector of size (n x D), with n samples,
                    and D inputs.
        # Returns:
            covar: Covariance between the samples for all the inputs
                    (D x (n x n)).
        """

        if X_new.ndim > 1:
            n, D = X_new.shape
            covar = np.zeros((D,n,n))
        else:
            D = X_new.shape[0]
            n = 1
            X_new.shape = (n, D)
            covar = np.zeros((D,n))

        for output in range(self.__Ny):
            ell = self.__hyper_length_scales[output]
            sf2 = self.__hyper_signal_variance[output]
            L = self.__chol[output]
            ks = self.covSEard(self.__X, X_new, ell, sf2)
            kss = sf2
            v = np.linalg.solve(L, ks)
            covar[output] = kss - v.T @ v
        return covar


    def update_data(self, X_new, Y_new, N_new=None):
        """ Update training data with new observations

        Will update training data with N_new samples, updating the
        cholesky covariance matrix, alpha, inverse covariance, and re-build
        the GP functions with the updated matrices.

        # Arguments:
            X_new: Input matrix with (n x Nx) new observations.
            Y_new: Corresponding measurments (n x Ny) from input X_new.
            N_new: Number of new samples to pick, default to N_new=n.


        # NOTE: NOT working as intended...
        """

        X_new = np.array(X_new).copy()
        Y_new = np.array(Y_new).copy()
        n, D = X_new.shape
        if N_new is None:
            N_new = n

        if self.__normalize:
            Y_new = self.standardize(Y_new, self.__meanY, self.__stdY)
            X_new = self.standardize(X_new, self.__meanZ, self.__stdZ)

        print('\n________________________________________')
        print('# Updating training data with ' + str(N_new) + ' new samples')
        print('----------------------------------------')
        for k in range(N_new):
            """ Explore point with highest combined variance """
            n, D = X_new.shape

            covar = self.covar(X_new)
            covar = np.sum(covar, 0) # Sum covariance of all states
            var = np.diag(covar)

            max_var_index = np.argmin(var)
            x_new = X_new[max_var_index]
            y_new = Y_new[max_var_index]
            X_new = np.delete(X_new, max_var_index, 0)
            Y_new = np.delete(Y_new, max_var_index, 0)

            """ Update matrices """
            N, D = self.__X.shape
            hyper = self.__hyper
            invK  = np.zeros((self.__Ny, N + 1, N + 1))
            alpha = np.zeros((self.__Ny, N + 1))
            chol  = np.zeros((self.__Ny, N + 1, N + 1))
            chol[:, :N, :N] = self.__chol

            for output in range(self.__Ny):
                ell = self.__hyper_length_scales[output]
                sf2 = self.__hyper_signal_variance[output]
                sn2 = self.__hyper_noise_variance[output]
                K_new   = self.covSEard(self.__X, x_new, ell, sf2)
                k_new__ = self.covSEard(x_new, x_new, ell, sf2) + sn2
                L = self.__chol[output]
                l_new = np.linalg.solve(L, K_new)
                l_new__ = np.sqrt(k_new__ - np.linalg.norm(l_new))
                chol[output, N:, :N] = l_new.T
                chol[output, N, N] = l_new__
                invL = np.linalg.solve(chol[output], np.eye(N + 1))
                invK[output, :, :] = np.linalg.solve(chol[output].T, invL)

            self.__X = np.vstack([self.__X, x_new])
            self.__Y = np.vstack([self.__Y, y_new])
            self.__N = self.__X.shape[0]

            for output in range(self.__Ny):
                m = get_mean_function(ca.MX(hyper[output, :]),
                                  self.__X.T, func=self.__mean_func)
                mean = np.array(m(self.__X.T)).reshape((self.__N + 1,))
                alpha[output] = np.linalg.solve(chol[output].T,
                                     np.linalg.solve(chol[output],
                                     self.__Y[:, output] - mean))
            self.__alpha = alpha
            self.__chol = chol
            self.__invK = invK

            # Rebuild GP with the new data
            self.__mean, self.__var, self.__covar, self.__mean_jac = \
                            build_gp(self.__invK, self.__X, self.__hyper,
                                     self.__alpha, self.__chol)
            print('* Update ' + str(k) + ' with new data point ' +str(max_var_index))
        self.__TA_covar = build_TA_cov(self.__mean, self.__covar,
                                      self.__mean_jac, self.__Nx, self.__Ny)
        self.set_method(self.__gp_method)


    def update_data_all(self, X_new, Y_new):
        """ Update training data with all new observations

        Will update training data with all samples, updating the
        cholesky covariance matrix, alpha, inverse covariance, and re-build
        the GP functions with the updated matrices.
        # Arguments:
            X_new: Input matrix with (n x Nx) new observations.
            Y_new: Corresponding measurments (n x Ny) from input X_new.
        """

        X_new = np.array(X_new).copy()
        Y_new = np.array(Y_new).copy()
        n, D = X_new.shape

        N_new = n

        if self.__normalize:
            Y_new = self.standardize(Y_new, self.__meanY, self.__stdY)
            X_new = self.standardize(X_new, self.__meanZ, self.__stdZ)

        print('\n________________________________________')
        print('# Updating training data with ' + str(N_new) + ' new samples')
        print('-----------------------------------------')

        """ Explore point with highest combined variance """
        n, D = X_new.shape

        """ Update matrices """
        self.__X = np.vstack([self.__X, X_new])
        self.__Y = np.vstack([self.__Y, Y_new])
        self.__N = self.__X.shape[0]

        N, D = self.__X.shape
        hyper = self.__hyper

        invK  = np.zeros((self.__Ny, N , N ))
        alpha = np.zeros((self.__Ny, N ))
        chol  = np.zeros((self.__Ny, N , N ))


        for output in range(self.__Ny):
            ell = self.__hyper_length_scales[output]
            sf2 = self.__hyper_signal_variance[output]
            sn2 = self.__hyper_noise_variance[output]
            K_new   = self.covSEard(self.__X, self.__X, ell, sf2)

            K = K_new + sn2 * np.eye(self.__N)
            K = (K + K.T) * 0.5   # Make sure matrix is symmentric
            try:
                L = np.linalg.cholesky(K)
            except np.linalg.LinAlgError:
                print("K matrix is not positive definit, adding jitter!")
                K = K + np.eye(N) * 1e-8
                L = np.linalg.cholesky(K)
            invL = np.linalg.solve(L, np.eye(self.__N))
            invK[output, :, :] = np.linalg.solve(L.T, invL)
            chol[output] = L
            m = get_mean_function(ca.MX(hyper[output, :]), self.__X.T,
                                    func=self.__mean_func)
            mean = np.array(m(self.__X.T)).reshape((self.__N,))
            alpha[output] = np.linalg.solve(L.T,
                                    np.linalg.solve(L, self.__Y[:, output] - mean))
        self.__alpha = alpha
        self.__chol = chol
        self.__invK = invK

        # Rebuild GP with the new data
        self.__mean, self.__var, self.__covar, self.__mean_jac = \
                    build_gp(self.__invK, self.__X, self.__hyper,
                             self.__alpha, self.__chol)

        self.__TA_covar = build_TA_cov(self.__mean, self.__covar,
                               self.__mean_jac, self.__Nx, self.__Ny)
        self.set_method(self.__gp_method)


    def replace_data_all(self, X_new, Y_new):
        """ Replace training data with new observations

        Will replace training data with new samples, replacing the
        cholesky covariance matrix, alpha, inverse covariance, and re-build
        the GP functions with the updated matrices.

        # Arguments:
            X_new: Input matrix with (n x Nx) new observations.
            Y_new: Corresponding measurments (n x Ny) from input X_new.
        """

        X_new = np.array(X_new).copy()
        Y_new = np.array(Y_new).copy()
        n, D = X_new.shape

        N_new = n

        if self.__normalize:
            Y_new = self.standardize(Y_new, self.__meanY, self.__stdY)
            X_new = self.standardize(X_new, self.__meanZ, self.__stdZ)

        print('\n________________________________________')
        print('# Replacing training data with ' + str(N_new) + ' new samples')
        print('----------------------------------------')

        """ Update matrices """
        self.__X = X_new
        self.__Y = Y_new
        self.__N = self.__X.shape[0]

        N, D = self.__X.shape
        hyper = self.__hyper

        invK  = np.zeros((self.__Ny, N , N ))
        alpha = np.zeros((self.__Ny, N ))
        chol  = np.zeros((self.__Ny, N , N ))


        for output in range(self.__Ny):
            ell = self.__hyper_length_scales[output]
            sf2 = self.__hyper_signal_variance[output]
            sn2 = self.__hyper_noise_variance[output]
            K_new   = self.covSEard(self.__X, self.__X, ell, sf2)

            K = K_new + sn2 * np.eye(self.__N)
            K = (K + K.T) * 0.5   # Make sure matrix is symmentric
            try:
                L = np.linalg.cholesky(K)
            except np.linalg.LinAlgError:
                print("K matrix is not positive definit, adding jitter!")
                K = K + np.eye(N) * 1e-8
                L = np.linalg.cholesky(K)
            invL = np.linalg.solve(L, np.eye(self.__N))
            invK[output, :, :] = np.linalg.solve(L.T, invL)
            chol[output] = L
            m = get_mean_function(ca.MX(hyper[output, :]), self.__X.T,
                                        func=self.__mean_func)
            mean = np.array(m(self.__X.T)).reshape((self.__N,))
            alpha[output] = np.linalg.solve(L.T,
                                        np.linalg.solve(L, self.__Y[:, output] - mean))

        self.__alpha = alpha
        self.__chol = chol
        self.__invK = invK

        # Rebuild GP with the new data
        self.__mean, self.__var, self.__covar, self.__mean_jac = \
                            build_gp(self.__invK, self.__X, self.__hyper,
                                        self.__alpha, self.__chol)

        self.__TA_covar = build_TA_cov(self.__mean, self.__covar,
                                        self.__mean_jac, self.__Nx, self.__Ny)
        self.set_method(self.__gp_method)


    def standardize(self, Y, mean, std):
        return (Y - mean) / std

    def normalize(self, u, lb, ub):
        return (u - lb) / (ub - lb)

    def inverse_mean(self, x, mean, std):
        """ Inverse standardization of the mean
        """
        return (x * std) + mean

    def inverse_variance(self, variance):
        """ Inverse standardization of the variance
        """
#         return (covariance[..., np.newaxis] * self.__stdY**2)
        return variance * self.__stdY**2


    def discrete_linearize(self, x0, u0, cov0):
        """ Linearize the GP around the operating point
            x[k+1] = Ax[k] + Bu[k]
        # Arguments:
            x0: State vector
            u0: Input vector
            cov0: Covariance
        """
        if self.__normalize:
            x0 = self.standardize(x0, self.__meanX, self.__stdX)
            u0 = self.standardize(u0, self.__meanU, self.__stdU)
        Ad = np.array(self.__discrete_jac_x(x0, u0, cov0))
        Bd = np.array(self.__discrete_jac_u(x0, u0, cov0))

        return Ad, Bd


    def jacobian(self, x0, u0, cov0):
        """ Jacobian of posterior mean
            J = dmu/dx
        # Arguments:
            x0: State vector
            u0: Input vector
            cov0: Covariance
        """
        return self.__discrete_jac_x(x0, u0, cov0)


    def noise_variance(self):
        """ Get the noise variance
        """
        return self.__hyper_noise_variance


#TODO: Fix this
    def sparse(self, M):
        """ Sparse Gaussian Process
            Use Fully Independent Training Conditional (FITC) to approximate
            the GP distribution and reduce computational complexity.

        # Arguments:
            M: Reduce the model size from N to M.
        """



    def __to_dict(self):
        """ Store model data in a dictionary """

        gp_dict = {}
        gp_dict['X'] = self.__X.tolist()
        gp_dict['Y'] = self.__Y.tolist()
        gp_dict['hyper'] = dict(
                    hyper = self.__hyper.tolist(),
                    invK = self.__invK.tolist(),
                    alpha = self.__alpha.tolist(),
                    chol = self.__chol.tolist(),
                    length_scale = self.__hyper_length_scales.tolist(),
                    signal_var = self.__hyper_signal_variance.tolist(),
                    noise_var = self.__hyper_noise_variance.tolist(),
                    mean = self.__hyper_mean.tolist()
                )
        gp_dict['mean_func'] = self.__mean_func
        gp_dict['normalize'] = self.__normalize
        if self.__normalize:
            gp_dict['xlb'] = self.__xlb.tolist()
            gp_dict['xub'] = self.__xub.tolist()
            gp_dict['ulb'] = self.__ulb.tolist()
            gp_dict['uub'] = self.__uub.tolist()
            gp_dict['meta'] = dict(
                        meanY = self.__meanY.tolist(),
                        stdY = self.__stdY.tolist(),
                        meanZ = self.__meanZ.tolist(),
                        stdZ = self.__stdZ.tolist(),
                        meanX = self.__meanX.tolist(),
                        stdX = self.__stdX.tolist(),
                        meanU = self.__meanU.tolist(),
                        stdU = self.__stdU.tolist()
                )
        return gp_dict


    def save_model(self, filename):
        """ Save model to a json file"""
        import json
        output_dict = self.__to_dict()
        with open(filename + ".json", "w") as outfile:
            json.dump(output_dict, outfile)


    @classmethod
    def load_model(cls, filename):
        """ Create a new model from file"""
        import json
        with open(filename + ".json") as json_data:
            input_dict = json.load(json_data)
        return cls(**input_dict)


    def predict_compare(self, x0, u, model, num_cols=2, xnames=None,
                        title=None, feedback=False, x_ref = None,
                        Q=None, R=None, methods=None):
        """ Predict and compare all GP methods
        """
        # Predict future
        Nx = self.__Nx
        Ny = self.__Ny

        dt = model.sampling_time()
        Nt = np.size(u, 0)
        sim_time = Nt * dt
        initVar = self.__hyper[:,Nx + 1]**2
        if methods is None:
            methods = ['EM', 'TA', 'ME']
        color = ['k', 'y', 'r']
        mean = np.zeros((len(methods), Nt + 1 , Ny))
        var = np.zeros((len(methods), Nt + 1, Ny))
        covar = np.eye(Nx) * 1e-6 # Initial covar input matrix
        if Q is None:
```

```
        Q = np.eye(Ny)
    if R is None:
        R= np.eye(Nx - Ny)

    if x_ref is None and feedback:
        x_ref = np.zeros((Ny))

    if feedback:
        A, B = self.discrete_linearize(x0, u[0], covar)
        K, S, E = lqr(A, B, Q, R)

    for i in range(len(methods)):
        self.set_method(methods[i])
        mean_t = x0
        covar[:Ny, :Ny] = ca.diag(initVar)
        mean[i, 0, :] = x0
        u_t = u[0]

        A, B = self.discrete_linearize(mean_t, u_t, covar)
        K, P, E = lqr(A, B, Q, R)

        for t in range(1, Nt + 1):
            if feedback:
                u_t = K @ (mean_t - x_ref)
            else:
                u_t = u[t-1, :]
            mean_t, covar_x = self.predict(mean_t, u_t, covar)
            mean[i, t, :] = np.array(mean_t).reshape((Ny,))
            var[i, t, :] = np.diag(covar_x)
            if self.__normalize:
                var[i, t, :] = self.inverse_variance(var[i, t, :])

            if feedback:
                covar_u = K @ covar_x @ K.T
                cov_xu = covar_x @ K.T
                covar[Ny:, Ny:] = covar_u
                covar[Ny:, :Ny] = cov_xu.T
                covar[:Ny, Ny:] = cov_xu
            covar[:Ny, :Ny] = covar_x

    #TODO: Fix feedback
    if feedback:
        A, B = model.discrete_linearize(x0, u[0])
        K, P, E = lqr(A, B, Q, R)
        y_sim = np.zeros((Nt + 1 , Ny))
        y_sim[0] = x0
        y_t = x0
        for t in range(1, Nt + 1):
            if 0: #feedback:
                u_t = K @ (y_t - x_ref)
            else:
                u_t = u[t-1, :]
            y_t = model.integrate(x0, u_t, []).flatten()
            y_sim[t] = y_t
    else:
        y_sim = model.sim(x0, u)
        y_sim = np.vstack([x0, y_sim])
    t = np.linspace(0.0, sim_time, Nt + 1)

    if np.any(var < 0):
        var = var.clip(min=0)

    num_rows = int(np.ceil(Ny / num_cols))
    if xnames is None:
        xnames = ['State %d' % (i + 1) for i in range(Ny)]
    if x_ref is not None:
        x_sp = x_ref * np.ones((Nt+1, Ny))

    fontP = FontProperties()
    fontP.set_size('small')
    fig = plt.figure(figsize=(9.0, 6.0))
    for i in range(Ny):
        ax = fig.add_subplot(num_rows, num_cols, i + 1)
        ax.plot(t, y_sim[:, i], 'b-', label='Simulation')
        if x_ref is not None:
            ax.plot(t, x_sp[:, i], color='g', linestyle='--', label='Setpoint')

        for k in range(len(methods)):
            mean_i = mean[k, :, i]
            sd_i = np.sqrt(var[k, :, i])
            ax.errorbar(t, mean_i, yerr=2 * sd_i, color = color[k],
                         label='GP ' + methods[k])
        ax.set_ylabel(xnames[i])
        ax.legend(prop=fontP, loc='best')
        ax.set_xlabel('Time [s]')
#         ax.set_ylim([-20,20])
    if title is not None:
        fig.canvas.set_window_title(title)
    else:
        fig.canvas.set_window_title(('Training data: {x},  Mean Function: {y},  '
                                     'Normalize: {q}, Feedback: {f}'
                                     ).format(x=self.__N, y=self.__mean_func,
                                     q=self.__normalize, f=feedback))
    plt.tight_layout()
    plt.show()
```

## E.2 gp_functions

```
# -*- coding: utf-8 -*-
"""
Gaussian Process functions
Copyright (c) 2018, Helge-André Langåker, Eric Bradford
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import numpy as np
import casadi as ca
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties
```

```
def covSEard(x, z, ell, sf2):
    """ GP squared exponential kernel
        Copyright (c) 2018, Helge-André Langåker
    """
    dist = ca.sum1((x - z)**2 / ell**2)
    return sf2 * ca.SX.exp(-.5 * dist)


def get_mean_function(hyper, X, func='zero'):
    """ Get mean function
        Copyright (c) 2018, Helge-André Langåker

    # Arguments:
        hyper: Matrix with hyperperameters.
        X: Input vector or matrix.
        func: Option for mean function:
                'zero':       m = 0
                'const':      m = a
                'linear':     m(x) = aT*x + b
                'polynomial': m(x) = aT*x^2 + bT*x + c
    # Returns:
         CasADi mean function [m(X, hyper)]
    """

    Nx, N = X.shape
    X_s = ca.SX.sym('x', Nx, N)
    Z_s = ca.MX.sym('x', Nx, N)
    m = ca.SX(N, 1)
    hyp_s = ca.SX.sym('hyper', *hyper.shape)
    if func == 'zero':
        meanF = ca.Function('zero_mean', [X_s, hyp_s], [m])
    elif func == 'const':
        a =  hyp_s[-1]
        for i in range(N):
            m[i] = a
        meanF = ca.Function('const_mean', [X_s, hyp_s], [m])
    elif func == 'linear':
        a = hyp_s[-Nx-1:-1].reshape((1, Nx))
        b = hyp_s[-1]
        for i in range(N):
            m[i] = ca.mtimes(a, X_s[:,i]) + b
        meanF = ca.Function('linear_mean', [X_s, hyp_s], [m])
    elif func == 'polynomial':
        a = hyp_s[-2*Nx-1:-Nx-1].reshape((1,Nx))
        b = hyp_s[-Nx-1:-1].reshape((1,Nx))
        c = hyp_s[-1]
        for i in range(N):
            m[i] = ca.mtimes(a, X_s[:, i]**2) + ca.mtimes(b, X_s[:,i]) + c
        meanF = ca.Function('poly_mean', [X_s, hyp_s], [m])
    else:
        raise NameError('No mean function called: ' + func)

    return ca.Function('mean', [Z_s], [meanF(Z_s, hyper)])
```

```python
def build_gp(invK, X, hyper, alpha, chol, meanFunc='zero'):
    """ Build Gaussian Process function
        Copyright (c) 2018, Helge-André Langåker

    # Arguments
        invK: Array with the inverse covariance matrices of size (Ny x N x N),
            with Ny number of outputs from the GP and N number of training points.
        X: Training data matrix with inputs of size (N x Nx), with Nx number of
            inputs to the GP.
        alpha: Training data matrix with invK time outputs of size (Ny x N).
        hyper: Array with hyperparame|ters [ell_1 .. ell_Nx sf sn].

    # Returns
        mean:     GP mean casadi function [mean(z)]
        var:      GP variance casadi function [var(z)]
        covar:    GP covariance casadi function [covar(z) = diag(var(z))]
        mean_jac: Casadi jacobian of the GP mean function [jac(z)]
    """


    Ny = len(invK)
    X = ca.SX(X)
    N, Nx = ca.SX.size(X)

    mean  = ca.SX.zeros(Ny, 1)
    var   = ca.SX.zeros(Ny, 1)

    # Casadi symbols
    x_s        = ca.SX.sym('x', Nx)
    z_s        = ca.SX.sym('z', Nx)
    m_s        = ca.SX.sym('m')
    ell_s      = ca.SX.sym('ell', Nx)
    sf2_s      = ca.SX.sym('sf2')
    X_s        = ca.SX.sym('X', N, Nx)
    ks_s       = ca.SX.sym('ks', N)
    v_s       = ca.SX.sym('v', N)
    kss_s      = ca.SX.sym('kss')
    alpha_s    = ca.SX.sym('alpha', N)

    covSE = ca.Function('covSE', [x_s, z_s, ell_s, sf2_s],
                          [covSEard(x_s, z_s, ell_s, sf2_s)])

    ks = ca.SX.zeros(N, 1)
    for i in range(N):
        ks[i] = covSE(X_s[i, :], z_s, ell_s, sf2_s)
    ks_func = ca.Function('ks', [X_s, z_s, ell_s, sf2_s], [ks])

    mean_i_func = ca.Function('mean', [ks_s, alpha_s, m_s],
                            [ca.mtimes(ks_s.T, alpha_s) + m_s])

    L_s = ca.SX.sym('L', ca.Sparsity.lower(N))
    v_func = ca.Function('v', [L_s, ks_s], [ca.solve(L_s, ks_s)])

    var_i_func  = ca.Function('var', [v_s, kss_s,],
                            [kss_s - v_s.T @ v_s])

    for output in range(Ny):
        ell       = ca.SX(hyper[output, 0:Nx])
        sf2       = ca.SX(hyper[output, Nx]**2)
        alpha_a   = ca.SX(alpha[output])
        ks        = ks_func(X_s, z_s, ell, sf2)
        v         = v_func(chol[output], ks)
        m = get_mean_function(ca.MX(hyper[output, :]), z_s, func=meanFunc)
        mean[output] = mean_i_func(ks, alpha_a, m(z_s))
        var[output]  = var_i_func(v, sf2)


    mean_temp  = ca.Function('mean_temp', [z_s, X_s], [mean])
    var_temp   = ca.Function('var_temp',  [z_s, X_s], [var])

    mean_func  = ca.Function('mean', [z_s], [mean_temp(z_s, X)])
    covar_func = ca.Function('var',  [z_s], [ca.diag(var_temp(z_s, X))])
    var_func = ca.Function('var',  [z_s], [var_temp(z_s, X)])

    mean_jac_z = ca.Function('mean_jac_z', [z_s],
                                      [ca.jacobian(mean_func(z_s), z_s)])

    return mean_func, var_func, covar_func, mean_jac_z


def build_TA_cov(mean, covar, jac, Nx, Ny):
    """ Build 1st order Taylor approximation of covariance function
        Copyright (c) 2018, Helge-André Langåker

    # Arguments:
        mean: GP mean casadi function [mean(z)]
        covar: GP covariance casadi function [covar(z)]
        jac: Casadi jacobian of the GP mean function [jac(z)]
        Nx: Number of inputs to the GP
        Ny: Number of ouputs from the GP

    # Return:
        cov: Casadi function with the approximated covariance
             function [cov(z, covar_x)].
    """
    cov_z  = ca.SX.sym('cov_z', Nx, Nx)
    z_s    = ca.SX.sym('z', Nx)
    jac_z = jac(z_s)
    cov    = ca.Function('cov', [z_s, cov_z],
                      [covar(z_s) + jac_z @ cov_z @ jac_z.T])

    return cov


def gp(invK, X, Y, hyper, inputmean,  alpha=None, meanFunc='zero', log=False):
    """ Gaussian Process
        Copyright (c) 2018, Helge-André Langåker

    # Arguments
        invK: Array with the inverse covariance matrices of size (Ny x N x N),
            with Ny number of outputs from the GP and N number of training points.
        X: Training data matrix with inputs of size (N x Nx), with Nx number of
            inputs to the GP.
        Y: Training data matrix with outpyts of size (N x Ny).
        hyper: Array with hyperparame|ters [ell_1 .. ell_Nx sf sn].
        inputmean: Input to the GP of size (1 x Nx)

    # Returns
        mean: The estimated mean.
        var: The estimated variance
    """
    if log:
        X = ca.log(X)
        Y = ca.log(Y)
        inputmean = ca.log(inputmean)

    Ny = len(invK)
    N, Nx = ca.MX.size(X)

    mean  = ca.MX.zeros(Ny, 1)
    var  = ca.MX.zeros(Ny, 1)

    # Casadi symbols
    x_s     = ca.SX.sym('x', Nx)
    z_s     = ca.SX.sym('z', Nx)
    ell_s   = ca.SX.sym('ell', Nx)
    sf2_s   = ca.SX.sym('sf2')

    invK_s  = ca.SX.sym('invK', N, N)
    Y_s     = ca.SX.sym('Y', N)
    m_s     = ca.SX.sym('m')
    ks_s    = ca.SX.sym('ks', N)
    kss_s   = ca.SX.sym('kss')
    ksT_invK_s = ca.SX.sym('ksT_invK', 1, N)
    alpha_s = ca.SX.sym('alpha', N)

    covSE = ca.Function('covSE', [x_s, z_s, ell_s, sf2_s],
                          [covSEard(x_s, z_s, ell_s, sf2_s)])

    ksT_invK_func = ca.Function('ksT_invK', [ks_s, invK_s],
                          [ca.mtimes(ks_s.T, invK_s)])

    if alpha is not None:
        mean_func = ca.Function('mean', [ks_s, alpha_s],
                          [ca.mtimes(ks_s.T, alpha_s)])
    else:
        mean_func = ca.Function('mean', [ksT_invK_s, Y_s],
                          [ca.mtimes(ksT_invK_s, Y_s)])

    var_func  = ca.Function('var', [kss_s, ksT_invK_s, ks_s],
                          [kss_s - ca.mtimes(ksT_invK_s, ks_s)])

    for output in range(Ny):
        m = get_mean_function(hyper[output, :], inputmean, func=meanFunc)
        ell = ca.MX(hyper[output, 0:Nx])
        sf2 = ca.MX(hyper[output, Nx]**2)

        kss = covSE(inputmean, inputmean, ell, sf2)
        ks = ca.MX.zeros(N, 1)
        for i in range(N):
            ks[i] = covSE(X[i, :], inputmean, ell, sf2)
        ksT_invK = ksT_invK_func(ks, ca.MX(invK[output]))
        if alpha is not None:
            mean[output] = mean_func(ks, ca.MX(alpha[output]))
        else:
            mean[output] = mean_func(ksT_invK, Y[:, output])
        var[output] = var_func(kss, ks, ksT_invK)

    if log:
        mean = ca.exp(mean)
        var = ca.exp(var)

    covar = ca.diag(var)
    return mean, covar


def gp_taylor_approx(invK, X, Y, hyper, inputmean, inputcovar,
                     meanFunc='zero', diag=False, log=False):
    """ Gaussian Process with Taylor Approximation
        Copyright (c) 2018, Helge-André Langåker

    This uses a first order taylor for the mean evaluation (a normal GP mean),
    and a second order taylor for estimating the variance.

    # Arguments
        invK: Array with the inverse covariance matrices of size (Ny x N x N),
            with Ny number of outputs from the GP and N number of training points.
        X: Training data matrix with inputs of size NxNx, with Nx number of
            inputs to the GP.
        Y: Training data matrix with outpyts of size (N x Ny).
        hyper: Array with hyperparameters [ell_1 .. ell_Nx sf sn].
        inputmean: Mean from the last GP iteration of size (1 x Nx)
        inputvar: Variance from the last GP iteration of size (1 x Ny)

    # Returns
        mean: Array with estimated mean of size (Ny x 1).
        covariance: The estimated covariance matrix with the output variance in the
                    diagonal of size (Ny x Ny).
    """
    if log:
        X = ca.log(X)
        Y = ca.log(Y)
        inputmean = ca.log(inputmean)

    Ny          = len(invK)
    N, Nx       = ca.MX.size(X)
    mean        = ca.MX.zeros(Ny, 1)
    var         = ca.MX.zeros(Nx, 1)
    v           = X - ca.repmat(inputmean, N, 1)
    covar_temp       = ca.MX.zeros(Ny, Ny)

    covariance = ca.MX.zeros(Ny, Ny)
    d_mean      = ca.MX.zeros(Ny, 1)
    dd_var      = ca.MX.zeros(Ny, Ny)


    # Casadi symbols
    x_s     = ca.SX.sym('x', Nx)
    z_s     = ca.SX.sym('z', Nx)
    ell_s   = ca.SX.sym('ell', Nx)
    sf2_s   = ca.SX.sym('sf2')
    covSE   = ca.Function('covSE', [x_s, z_s, ell_s, sf2_s],
                          [covSEard(x_s, z_s, ell_s, sf2_s)])

    for a in range(Ny):
        ell = hyper[a, :Nx]
        w = 1 / ell**2
        sf2 = ca.MX(hyper[a, Nx]**2)
        m = get_mean_function(hyper[a, :], inputmean, func=meanFunc)
        iK = ca.MX(invK[a])
        alpha = ca.mtimes(iK, Y[:, a] - m(inputmean)) + m(inputmean)
        kss = sf2

        ks = ca.MX.zeros(N, 1)
        for i in range(N):
            ks[i] = covSE(X[i, :], inputmean, ell, sf2)

        invKks = ca.mtimes(iK, ks)
        mean[a] = ca.mtimes(ks.T, alpha)
        var[a] = kss - ca.mtimes(ks.T, invKks)
        d_mean[a] = ca.mtimes(ca.transpose(w[a] * v[:, a] * ks), alpha)

        #BUG: This don't take into account the covariance between states
        for d in range(Ny):
            for e in range(Ny):
                dd_var1a = ca.mtimes(ca.transpose(v[:, d] * ks), iK)
                dd_var1b = ca.mtimes(dd_var1a, v[e] * ks)
                dd_var2 = ca.mtimes(ca.transpose(v[d] * v[e] * ks), invKks)
                dd_var[d, e] = -2 * w[d] * w[e] * (dd_var1b + dd_var2)
                if d == e:
                    dd_var[d, e] = dd_var[d, e] + 2 * w[d] * (kss - var[d])

        mean_mat = ca.mtimes(d_mean, d_mean.T)
        covar_temp[0, 0] = inputcovar[a, a]
        covariance[a, a] = var[a] + ca.trace(ca.mtimes(covar_temp, .5
                                             * dd_var + mean_mat))

    return [mean, covariance]


def gp_exact_moment(invK, X, Y, hyper, inputmean, inputcov):
    """ Gaussian Process with Exact Moment Matching
    Copyright (c) 2018, Eric Bradford, Helge-André Langåker

    The first and second moments are used to compute the mean and covariance of the
    posterior distribution with a stochastic input distribution. This assumes a
    zero prior mean function and the squared exponential kernel.

    # Arguments
        invK: Array with the inverse covariance matrices of size (Ny x N x N),
            with Ny number of outputs from the GP and N number of training points.
        X: Training data matrix with inputs of size NxNx, with Nx number of
            inputs to the GP.
        Y: Training data matrix with outpyts of size (N x Ny).
        hyper: Array with hyperparameters [ell_1 .. ell_Nx sf sn].
        inputmean: Mean from the last GP iteration of size (1 x Nx)
        inputcov: Covariance matrix from the last GP iteration of size (Nx x Nx)

    # Returns
        mean: Array of the output mean of size (Ny x 1).
        covariance: Covariance matrix of size (Ny x Ny).
    """

    hyper = ca.log(hyper)
    Ny      = len(invK)
    N, Nx      = ca.MX.size(X)
    mean  = ca.MX.zeros(Ny, 1)
    beta  = ca.MX.zeros(N, Ny)
    log_k = ca.MX.zeros(N, Ny)
    v     = X - ca.repmat(inputmean, N, 1)

    covariance = ca.MX.zeros(Ny, Ny)

    #TODO: Fix that LinsolQr don't work with the extended graph?
    A = ca.SX.sym('A', inputcov.shape)
    [Q, R2] = ca.qr(A)
    determinant = ca.Function('determinant', [A], [ca.exp(ca.trace(ca.log(R2)))])

    for a in range(Ny):
        beta[:, a] = ca.mtimes(invK[a], Y[:, a])
        iLambda    = ca.diag(ca.exp(-2 * hyper[a, :Nx]))
        R  = inputcov + ca.diag(ca.exp(2 * hyper[a, :Nx]))
        iR = ca.mtimes(iLambda, (ca.MX.eye(Nx) - ca.solve((ca.MX.eye(Nx)
                + ca.mtimes(inputcov, iLambda)), (ca.mtimes(inputcov, iLambda)))))
        T  = ca.mtimes(v, iR)
        c  = ca.exp(2 * hyper[a, Nx]) / ca.sqrt(determinant(R)) \
                * ca.exp(ca.sum2(hyper[a, :Nx]))
        q2 = c * ca.exp(-ca.sum2(T * v) * 0.5)
        qb = q2 * beta[:, a]
        mean[a] = ca.sum1(qb)
        t  = ca.repmat(ca.exp(hyper[a, :Nx]), N, 1)
        v1 = v / t
        log_k[:, a] = 2 * hyper[a, Nx] - ca.sum2(v1 * v1) * 0.5

    # covariance with noisy input
    for a in range(Ny):
        ii = v / ca.repmat(ca.exp(2 * hyper[a, :Nx]), N, 1)
        for b in range(a + 1):
            R = ca.mtimes(inputcov, ca.diag(ca.exp(-2 * hyper[a, :Nx])
                + ca.exp(-2 * hyper[b, :Nx]))) + ca.MX.eye(Nx)
            t = 1.0 / ca.sqrt(determinant(R))
            ij = v / ca.repmat(ca.exp(2 * hyper[b, :Nx]), N, 1)
            Q = ca.exp(ca.repmat(log_k[:, a], 1, N)
                + ca.repmat(ca.transpose(log_k[:, b]), N, 1)
                + maha(ii, -ij, ca.solve(R, inputcov * 0.5), N))
            A = ca.mtimes(beta[:, a], ca.transpose(beta[:, b]))
            if b == a:
                A = A - invK[a]
            A = A * Q
            covariance[a, b] = t * ca.sum2(ca.sum1(A))
            covariance[b, a] = covariance[a, b]
        covariance[a, a] = covariance[a, a] + ca.exp(2 * hyper[a, Nx])
    covariance = covariance - ca.mtimes(mean, ca.transpose(mean))

    return [mean, covariance]


def maha(a1, b1, Q1, N):
    """Calculate the Mahalanobis distance
    Copyright (c) 2018, Eric Bradford
    """
    aQ = ca.mtimes(a1, Q1)
    bQ = ca.mtimes(b1, Q1)
    K1  = ca.repmat(ca.sum2(aQ * a1), 1, N) \
            + ca.repmat(ca.transpose(ca.sum2(bQ * b1)), N, 1) \
            - 2 * ca.mtimes(aQ, ca.transpose(b1))
    return K1
```

## E.3 optimize

```
# -*- coding: utf-8 -*-
"""
Optimize hyperparameters for Gaussian Process Model
Copyright (c) 2018, Helge-André Langåker
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import time
import pyDOE
import numpy as np
import casadi as ca
from scipy.spatial import distance
from .gp_functions import get_mean_function, gp, gp_exact_moment
from scipy.optimize import minimize

# -----------------------------------------------------------------------------
# Optimization of hyperperameters as a constrained minimization problem
# -----------------------------------------------------------------------------
def calc_NLL(hyper, X, Y, squaredist, meanFunc='zero', prior=None):
    """ Objective function

    Calculate the negative log likelihood function using Casadi SX symbols.

    # Arguments:
        hyper: Array with hyperparameters [ell_1 .. ell_Nx sf sn], where Nx is the
            number of inputs to the GP.
        X: Training data matrix with inputs of size (N x Nx).
        Y: Training data matrix with outpyts of size (N x Ny),
            with Ny number of outputs.

    # Returns:
        NLL: The negative log likelihood function (scalar)
    """
```

```
N, Nx = ca.MX.size(X)
ell = hyper[:Nx]
sf2 = hyper[Nx]**2
sn2 = hyper[Nx + 1]**2

m = get_mean_function(hyper, X.T, func=meanFunc)

# Calculate covariance matrix
K_s = ca.SX.sym('K_s',N, N)
sqdist = ca.SX.sym('sqd', N, N)
elli = ca.SX.sym('elli')
ki = ca.Function('ki', [sqdist, elli, K_s], [sqdist / elli**2 + K_s])
K1 = ca.MX(N, N)
for i in range(Nx):
    K1 = ki(squaredist[:, (i  * N):(i + 1) * N], ell[i], K1)

sf2_s   = ca.SX.sym('sf2')
exponent   = ca.SX.sym('exp', N, N)
K_exp = ca.Function('K', [exponent, sf2_s], [sf2_s * ca.SX.exp(-.5 * exponent)])
K2 = K_exp(K1, sf2)

K = K2 + sn2 * ca.MX.eye(N)
K = (K + K.T) * 0.5   # Make sure matrix is symmentric

A = ca.SX.sym('A', ca.MX.size(K))
cholesky = ca.Function('cholesky', [A], [ca.chol(A).T])
L = cholesky(K)

B = 2 * ca.sum1(ca.SX.log(ca.diag(A)))
log_determinant = ca.Function('log_det', [A], [B])
log_detK = log_determinant(L)

Y_s = ca.SX.sym('Y', ca.MX.size(Y))
L_s = ca.SX.sym('L', ca.Sparsity.lower(N))
sol = ca.Function('sol', [L_s, Y_s], [ca.solve(L_s, Y_s)])
invLy = sol(L, Y - m(X.T))

invLy_s = ca.SX.sym('invLy', ca.MX.size(invLy))
sol2 = ca.Function('sol2', [L_s, invLy_s], [ca.solve(L_s.T, invLy_s)])
alpha = sol2(L, invLy)

alph = ca.SX.sym('alph', ca.MX.size(alpha))
detK = ca.SX.sym('det')

# Calculate hyperpriors
theta = ca.SX.sym('theta')
mu = ca.SX.sym('mu')
s2 = ca.SX.sym('s2')
prior_gauss = ca.Function('hyp_prior', [theta, mu, s2],
                          [-(theta - mu)**2/(2*s2) - 0.5*ca.log(2*ca.pi*s2)])
log_prior = 0
if prior is not None:
    for i in range(Nx):
        log_prior += prior_gauss(ell[i], prior['ell_mean'], prior['ell_std']**2)
    log_prior += prior_gauss(sf2, prior['sf_mean'], prior['sf_std']**2)
    log_prior += prior_gauss(sn2, prior['sn_mean'], prior['sn_std']**2)
    NLL = ca.Function('NLL', [Y_s, alph, detK],
                      [0.5 * ca.mtimes(Y_s.T, alph) + 0.5 * detK])
    return NLL(Y - m(X.T), alpha, log_detK) + log_prior


def train_gp(X, Y, meanFunc='zero', hyper_init=None, lam_x0=None, log=False,
             multistart=1, optimizer_opts=None):
    """ Train hyperparameters using CasADi/IPOPT

    Maximum likelihood estimation is used to optimize the hyperparameters of
    the Gaussian Process. The optimalization use CasADi to find the gradients
    and use the interior point method IPOPT to find the solution.

    A uniform prior of the hyperparameters are assumed and implemented as
    limits in the optimization problem.

    NOTE: This function use the symbolic framework from CasADi to optimize the
            hyperparameters, where the gradients are found using algorithmic
            differentiation. This gives the exact gradients, but require a lot
            more memory than the nummeric version 'train_gp_numpy' and have a
            quite horrible scaling problem. The memory usage from the symbolic
            gradients tend to explode with the number of observations.

    # Arguments:
        X: Training data matrix with inputs of size (N x Nx),
            where Nx is the number of inputs to the GP.
        Y: Training data matrix with outpyts of size (N x Ny),
            with Ny number of outputs.
        meanFunc: String with the name of the wanted mean function.
            Possible options:
                'zero':       m = 0
                'const':      m = a
                'linear':     m(x) = aT*x + b
                'polynomial': m(x) = xT*diag(a)*x + bT*x + c

    # Return:
        opt: Dictionary with the optimal hyperparameters
                [ell_1 .. ell_Nx sf sn].
    """

    if log:
        X = np.log(X)
        Y = np.log(Y)

    N, Nx = X.shape
    Ny = Y.shape[1]
    # Counting mean function parameters
    if meanFunc == 'zero':
        h_m = 0
    elif meanFunc == 'const':
        h_m = 1
    elif meanFunc == 'linear':
        h_m = Nx + 1
    elif meanFunc == 'polynomial':
        h_m = 2 * Nx + 1
    else:
        raise NameError('No mean function called: ' + meanFunc)

    h_ell   = Nx    # Number of length scales parameters
    h_sf    = 1     # Standard deviation function
    h_sn    = 1     # Standard deviation noise
    num_hyp = h_ell + h_sf + h_sn + h_m
    prior = None
#     prior = dict(
#                 ell_mean = 10,
#                 ell_std = 10,
#                 sf_mean  = 10,
#                 sf_std  = 10,
#                 sn_mean  = 1e-5,
#                 sn_std  = 1e-5
#            )

    # Create solver
    Y_s          = ca.MX.sym('Y', N)
    X_s          = ca.MX.sym('X', N, Nx)
    hyp_s        = ca.MX.sym('hyp', 1, num_hyp)
    squaredist_s = ca.MX.sym('sqdist', N, N * Nx)
    param_s      = ca.horzcat(squaredist_s, Y_s)

    NLL_func = ca.Function('NLL', [hyp_s, X_s, Y_s, squaredist_s],
                           [calc_NLL(hyp_s, X_s, Y_s, squaredist_s,
                                     meanFunc=meanFunc, prior=prior)])
    nlp = {'x': hyp_s, 'f': NLL_func(hyp_s, X, Y_s, squaredist_s), 'p': param_s}

    # NLP solver options
    opts = {}
    opts['expand']              = True
    opts['print_time']          = False
    opts['verbose']             = False
    opts['ipopt.print_level']   = 1
    opts['ipopt.tol']          = 1e-8
    opts['ipopt.mu_strategy'] = 'adaptive'
    if optimizer_opts is not None:
        opts.update(optimizer_opts)

    warm_start = False
    if hyper_init is not None:
        opts['ipopt.warm_start_init_point'] = 'yes'
        warm_start = True
    Solver = ca.nlpsol('Solver', 'ipopt', nlp, opts)

    hyp_opt = np.zeros((Ny, num_hyp))
    lam_x_opt = np.zeros((Ny, num_hyp))
    invK = np.zeros((Ny, N, N))
    alpha = np.zeros((Ny, N))
    chol = np.zeros((Ny, N, N))

    print('\n________________________________________')
    print('# Optimizing hyperparameters (N=%d)' % N )
    print('----------------------------------------')
    for output in range(Ny):
        meanF     = np.mean(Y[:, output])
        lb        = -np.inf * np.ones(num_hyp)
        ub         = np.inf * np.ones(num_hyp)

        lb[:Nx]     = 1e-2
        ub[:Nx]     = 1e2
        lb[Nx]      = 1e-8
        ub[Nx]      = 1e2
        lb[Nx + 1]  = 10**-10
        ub[Nx + 1]  = 10**-2

        if hyper_init is None:
#               hyp_init = pyDOE.lhs(num_hyp, samples=1).flatten()
            hyp_init = np.zeros((num_hyp))
            hyp_init[:Nx] = np.std(X, 0)
            hyp_init[Nx] = np.std(Y[:, output])
            hyp_init[Nx + 1] = 1e-5
#               hyp_init = hyp_init * (ub - lb) + lb
        else:
            hyp_init = hyper_init[output, :]

        if meanFunc is 'const':
            lb[-1] = -1e2
            ub[-1] = 1e2
        elif meanFunc is not 'zero':
            lb[-1] = meanF / 10 -1e-8
            ub[-1] = meanF * 10 + 1e-8
            lb[-h_m:-1] = -1e-2
            ub[-h_m:-1] = 1e-2

        squaredist = np.zeros((N, N * Nx))
        for i in range(Nx):
            d = distance.pdist(X[:, i].reshape(N, 1), 'sqeuclidean')
            squaredist[:, (i * N):(i + 1) * N] = distance.squareform(d)
        param = ca.horzcat(squaredist, Y[:, output])

        obj = np.zeros((multistart, 1))
        hyp_opt_loc = np.zeros((multistart, num_hyp))
        lam_x_opt_loc = np.zeros((multistart, num_hyp))

        for i in range(multistart):
            solve_time = -time.time()
            if warm_start:
                res = Solver(x0=hyp_init, lam_x0=lam_x0[output],
                                   lbx=lb, ubx=ub, p=param)
            else:
                res =  Solver(x0=hyp_init, lbx=lb, ubx=ub, p=param)
            status = Solver.stats()['return_status']
            obj[i]              = res['f']
            hyp_opt_loc[i, :]   = res['x']
            lam_x_opt_loc       = res['lam_x']
            solve_time += time.time()
            print("* State %d: %s - %f s" % (output, status, solve_time))

        # With multistart, get solution with lowest decision function value
        hyp_opt[output, :]   = hyp_opt_loc[np.argmin(obj)]
        lam_x_opt[output, :] = lam_x_opt_loc[np.argmin(obj)]
        ell = hyp_opt[output, :Nx]
        sf2 = hyp_opt[output, Nx]**2
        sn2 = hyp_opt[output, Nx + 1]**2

        # Calculate the inverse covariance matrix
        K = np.zeros((N, N))
        for i in range(Nx):
            K = squaredist[:, (i  * N):(i + 1) * N] / ell[i]**2 + K
        K = sf2 * np.exp(-.5 * K)
        K = K + sn2 * np.eye(N)      # Add noise variance to diagonal
        K = (K + K.T) * 0.5          # Make sure matrix is symmentric
        try:
            L = np.linalg.cholesky(K)
        except np.linalg.LinAlgError:
            print("K matrix is not positive definit, adding jitter!")
            K = K + np.eye(N) * 1e-8
            L = np.linalg.cholesky(K)
        invL = np.linalg.solve(L, np.eye(N))
        invK[output, :, :] = np.linalg.solve(L.T, invL)
        chol[output] = L
        m = get_mean_function(ca.MX(hyp_opt[output, :]), X.T, func=meanFunc)
        mean = np.array(m(X.T)).reshape((N,))
        alpha[output] = np.linalg.solve(L.T, np.linalg.solve(L, Y[:, output] - mean))
    print('----------------------------------------')

    opt = {}
    opt['hyper'] = hyp_opt
    opt['lam_x'] = lam_x_opt
    opt['invK'] = invK
    opt['alpha'] = alpha
    opt['chol'] = chol
    return opt



# -----------------------------------------------------------------------------
# Optimization of hyperperameters using scipy
# -----------------------------------------------------------------------------

def calc_cov_matrix(X, ell, sf2):
    """ Calculate covariance matrix K

        Squared Exponential ARD covariance kernel

    # Arguments:
        X: Training data matrix with inputs of size (N x Nx).
        ell: Vector with length scales of size Nx.
        sf2: Signal variance (scalar)
    """
    dist = 0
    n, D = X.shape
    for i in range(D):
        x = X[:, i].reshape(n, 1)
        dist = (np.sum(x**2, 1).reshape(-1, 1) + np.sum(x**2, 1) -
                2 * np.dot(x, x.T)) / ell[i]**2 + dist
    return sf2 * np.exp(-.5 * dist)
```

```
def calc_NLL_numpy(hyper, X, Y):
    """ Objective function

    Calculate the negative log likelihood function.

    # Arguments:
        hyper: Array with hyperparameters [ell_1 .. ell_Nx sf sn], where Nx is the
                number of inputs to the GP.
        X: Training data matrix with inputs of size (N x Nx).
        Y: Training data matrix with outpyts of size (N x Ny), with Ny number of outputs.

    # Returns:
        NLL: The negative log likelihood function (scalar)
    """

    n, D = X.shape
    ell = hyper[:D]
    sf2 = hyper[D]**2
    lik = hyper[D + 1]**2
    #m   = hyper[D + 2]
    K = calc_cov_matrix(X, ell, sf2)
    K = K + lik * np.eye(n)
    K = (K + K.T) * 0.5   # Make sure matrix is symmentric
    try:
        L = np.linalg.cholesky(K)
    except np.linalg.LinAlgError:
        print("K is not positive definit, adding jitter!")
        K = K + np.eye(n) * 1e-8
        L = np.linalg.cholesky(K)

    logK = 2 * np.sum(np.log(np.abs(np.diag(L))))
    invLy = np.linalg.solve(L, Y)
    alpha = np.linalg.solve(L.T, invLy)
    NLL = 0.5 * np.dot(Y.T, alpha) + 0.5 * logK
    return NLL


def train_gp_numpy(X, Y, meanFunc='zero', hyper_init=None, lam_x0=None, log=False,
                multistart=1, optimizer_opts=None):
    """ Train hyperparameters using scipy / SLSQP

    Maximum likelihood estimation is used to optimize the hyperparameters of
    the Gaussian Process. The optimization use finite differences to estimate
    the gradients and Sequential Least SQuares Programming (SLSQP) to find
    the optimal solution.

    A uniform prior of the hyperparameters are assumed and implemented as
    limits in the optimization problem.

    NOTE: Unlike the casadi version 'train_gp', this function use finite
            differences to estimate the gradients. To get a better result
            and reduce the computation time the explicit gradients should
            be implemented. The gradient equations are given by
            (Rassmussen, 2006).

    NOTE: This version only support a zero-mean function. To enable the use of
            other mean functions, this has to be included in the calculations
            in the 'calc_NLL_numpy' function.

    # Arguments:
        X: Training data matrix with inputs of size (N x Nx),
            where Nx is the number of inputs to the GP.
        Y: Training data matrix with outpyts of size (N x Ny),
            with Ny number of outputs.
        meanFunc: String with the name of the wanted mean function.
            Possible options:
                'zero':       m = 0
                'const':      m = a
                'linear':     m(x) = aT*x + b
                'polynomial': m(x) = xT*diag(a)*x + bT*x + c

    # Return:
        opt: Dictionary with the optimal hyperparameters [ell_1 .. ell_Nx sf sn].
    """
#    if log:
#        X = np.log(X)
#        Y = np.log(Y)

    N, Nx = X.shape
    Ny = Y.shape[1]

    # Counting mean function parameters
    if meanFunc == 'zero':
        h_m = 0
    elif meanFunc == 'const':
        h_m = 1
    elif meanFunc == 'linear':
        h_m = Nx + 1
    elif meanFunc == 'polynomial':
        h_m = 2 * Nx + 1
    else:
        raise NameError('No mean function called: ' + meanFunc)

    h_ell   = Nx    # Number of length scales parameters
    h_sf    = 1     # Standard deviation function
    h_sn    = 1     # Standard deviation noise
    num_hyp = h_ell + h_sf + h_sn + h_m

    options = {'disp': True, 'maxiter': 10000}
    if optimizer_opts is not None:
        options.update(optimizer_opts)


    hyp_opt = np.zeros((Ny, num_hyp))
    invK = np.zeros((Ny, N, N))
    alpha = np.zeros((Ny, N))
    chol = np.zeros((Ny, N, N))

    print('\n________________________________________')
    print('# Optimizing hyperparameters (N=%d)' % N )
    print('----------------------------------------')
    for output in range(Ny):
        meanF     = np.mean(Y[:, output])
        lb        = -np.inf * np.ones(num_hyp)
```

```
ub         = np.inf * np.ones(num_hyp)
lb[:Nx]     = 1-2
ub[:Nx]     = 2e2
lb[Nx]      = 1e-8
ub[Nx]      = 1e2
lb[Nx + 1] = 10**-10
ub[Nx + 1] = 10**-2
bounds = np.hstack((lb.reshape(num_hyp, 1), ub.reshape(num_hyp, 1)))

if hyper_init is None:
    hyp_init = np.zeros((num_hyp))
    hyp_init[:Nx] = np.std(X, 0)
    hyp_init[Nx] = np.std(Y[:, output])
    hyp_init[Nx + 1] = 1e-5
else:
    hyp_init = hyper_init[output, :]

if meanFunc is 'const':
    lb[-1] = -1e2
    ub[-1] = 1e2
elif meanFunc is not 'zero':
    lb[-1] = meanF / 10 -1e-8
    ub[-1] = meanF * 10 + 1e-8
    lb[-h_m:-1] = -1e-2
    ub[-h_m:-1] = 1e-2

obj = np.zeros((multistart, 1))
hyp_opt_loc = np.zeros((multistart, num_hyp))
for i in range(multistart):
    solve_time = -time.time()
    res = minimize(calc_NLL_numpy, hyp_init, args=(X, Y[:, output]),
                   method='SLSQP', options=options, bounds=bounds, tol=1e-12)
    obj[i] = res.fun
    hyp_opt_loc[i, :] = res.x
solve_time += time.time()
print("* State %d:  %f s" % (output, solve_time))

# With multistart, get solution with lowest decision function value
hyp_opt[output, :]   = hyp_opt_loc[np.argmin(obj)]
ell = hyp_opt[output, :Nx]
sf2 = hyp_opt[output, Nx]**2
sn2 = hyp_opt[output, Nx + 1]**2

# Calculate the inverse covariance matrix
K = calc_cov_matrix(X, ell, sf2)
K = K + sn2 * np.eye(N)
K = (K + K.T) * 0.5   # Make sure matrix is symmentric
try:
    L = np.linalg.cholesky(K)
except np.linalg.LinAlgError:
    print("K matrix is not positive definit, adding jitter!")
    K = K + np.eye(N) * 1e-8
    L = np.linalg.cholesky(K)
invL = np.linalg.solve(L, np.eye(N))
invK[output, :, :] = np.linalg.solve(L.T, invL)
chol[output] = L
m = get_mean_function(ca.MX(hyp_opt[output, :]), X.T, func=meanFunc)
```

```
        mean = np.array(m(X.T)).reshape((N,))
        alpha[output] = np.linalg.solve(L.T, np.linalg.solve(L, Y[:, output] - mean))
    print('----------------------------------------')

    opt = {}
    opt['hyper'] = hyp_opt
    opt['lam_x'] = 0 # Warm start not implemented
    opt['invK'] = invK
    opt['alpha'] = alpha
    opt['chol'] = chol
    return opt



# -----------------------------------------------------------------------------
# Validation of model
# -----------------------------------------------------------------------------

def validate(X_test, Y_test, X, Y, invK, hyper, meanFunc, alpha=None):
    """ Validate GP model with new test data
    """
    N, Ny = Y_test.shape
    Nx = np.size(X, 1)
    z_s = ca.MX.sym('z', Nx)

    gp_func = ca.Function('gp', [z_s],
                            gp(invK, ca.MX(X), ca.MX(Y), ca.MX(hyper),
                               z_s, meanFunc=meanFunc, alpha=alpha))
    loss = 0
    NLP = 0

    for i in range(N):
        mean, var = gp_func(X_test[i, :])
        loss += (Y_test[i, :] - mean)**2
        NLP += 0.5*np.log(2*np.pi * (var)) + ((Y_test[i, :] - mean)**2)/(2*var)
        print(NLP)
        print(var)
    loss = loss / N
    SMSE = loss/ np.std(Y_test, 0)
    MNLP = NLP / N


    print('\n________________________________________')
    print('# Validation of GP model ')
    print('----------------------------------------')
    print('* Num training samples: ' + str(np.size(Y, 0)))
    print('* Num test samples: ' + str(N))
    print('----------------------------------------')
    print('* Mean squared error: ')
    for i in range(Ny):
        print('\t- State %d: %f' % (i + 1, loss[i]))
    print('----------------------------------------')
    print('* Standardized mean squared error:')
    for i in range(Ny):
        print('\t* State %d: %f' % (i + 1, SMSE[i]))
    print('----------------------------------------\n')
    print('* Mean Negative log Probability:')
    for i in range(Ny):
        print('\t* State %d: %f' % (i + 1, MNLP[i]))
    print('----------------------------------------\n')
    return SMSE, MNLP


"""-----------------------------------------------------------------------------
# Preprocesing of training data
-----------------------------------------------------------------------------"""


def normalize(X, lb, ub):
    """ Normalize data between 0 and 1
    # Arguments:
        X: Input data (scalar/vector/matrix)
        lb: Lower boundry (scalar/vector)
        ub: Upper boundry (scalar/vector)
    # Return:
        X normalized (scalar/vector/matrix)
    """

    return (X - lb) / (ub - lb)


def normalize_inverse(X_scaled, lb, ub):
    # Scale input and output variables
    # Normalize input data to [0 1]
    return X_scaled * (ub - lb) + lb


def standardize(X_original, meanX, stdX):
    # Scale input and output variables
    return (X_original - meanX) / stdX


def standardize_inverse(X_scaled, meanX, stdX):
    # Scale input and output variables
    return X_scaled * stdX + meanX
```

## E.4 model_class

```
# -*- coding: utf-8 -*-
"""
Dynamic System Model
Copyright (c) 2018, Helge-André Langåker
"""

from __future__ import absolute_import
from __future__ import division
from __future__ import print_function


import pyDOE
import numpy as np
import casadi as ca
import scipy.linalg
```

```python
import matplotlib.pyplot as plt
from matplotlib.font_manager import FontProperties


class Model:
    def __init__(self, Nx, Nu, ode, dt, R=None,
                 alg=None, alg_0=None, Nz=0, Np=0,
                 opt=None, clip_negative=False):
        """ Initialize dynamic model

        # Arguments:
            Nx:   Number of states
            Nu:   Number of inputs
            ode:  ode(x, u, z, p)
            dt:   Sampling time

        # Arguments (optional):
            R:   Noise covariance matrix (Ny, Ny)
            alg: alg(x, z, u)
            alg_0: Initial value of algebraic variables
            Nz:  Number of algebraic states
            Np:  Number of parameters
            opt: Options dict to pass to the IDEAS integrator
            clip_negative: If true, clip negative simulated outputs to zero
        """

        # Create a default noise covariance matrix
        if R is None:
            self.__R = np.eye(self.__Ny) * 1e-3
        else:
            self.__R = R

        self.__dt   = dt
        self.__Nu = Nu
        self.__Nx = Nx
        self.__Nz = Nz
        self.__Np = Np
        self.__clip_negative = clip_negative

        """ Create integrator """
        # Integrator options
        options = {
            "abstol" : 1e-5,
            "reltol" : 1e-9,
            "max_num_steps": 100,
            "tf" : dt,
        }
        if opt is not None:
            options.update(opt)

        x = ca.MX.sym('x', Nx)
        u = ca.MX.sym('u', Nu)
        z = ca.MX.sym('z', Nz)
        p = ca.MX.sym('p', Np)
        par = ca.vertcat(u, p)

        dae = {'x': x, 'ode': ode(x,u,z,p), 'p':par}
        if alg is not None:
            self.__alg0 = ca.Function('alg_0', [x, u],
                                      [alg_0(x, u)])
            dae.update({'z':z, 'alg': alg(x, z, u)})
            self.Integrator = ca.integrator('DEA_Integrator', 'idas', dae, options)
        else:
            self.Integrator = ca.integrator('ODE_Integrator', 'cvodes', dae, options)

        #TODO: Fix discrete DAE model
        if alg is None:
            """ Create discrete RK4 model """
            ode_casadi = ca.Function("ode", [x, u, p], [ode(x,u,z,p)])
            k1 = ode_casadi(x, u, p)
            k2 = ode_casadi(x + dt/2*k1, u, p)
            k3 = ode_casadi(x + dt/2*k2, u, p)
            k4 = ode_casadi(x + dt*k3,u, p)
            xrk4 = x + dt/6*(k1 + 2*k2 + 2*k3 + k4)
            self.rk4 = ca.Function("ode_rk4", [x, u, p], [xrk4])

            # Jacobian of continuous system
            self.__jac_x = ca.Function('jac_x', [x, u, p],
                                       [ca.jacobian(ode_casadi(x,u,p), x)])
            self.__jac_u = ca.Function('jac_x', [x, u, p],
                                       [ca.jacobian(ode_casadi(x,u,p), u)])

            # Jacobian of discrete RK4 system
            self.__discrete_rk4_jac_x = ca.Function('jac_x', [x, u, p],
                                        [ca.jacobian(self.rk4(x,u,p), x)])
            self.__discrete_rk4_jac_u = ca.Function('jac_x', [x, u, p],
                                        [ca.jacobian(self.rk4(x,u,p), u)])

        # Jacobian of exact discretization
        self.__discrete_jac_x = ca.Function('jac_x', [x, u, p],
                                   [ca.jacobian(self.Integrator(x0=x,
                                             p=ca.vertcat(u,p))['xf'], x)])

        self.__discrete_jac_u = ca.Function('jac_u', [x, u, p],
                                    [ca.jacobian(self.Integrator(x0=x,
                                              p=ca.vertcat(u,p))['xf'], u)])


    def linearize(self, x0, u0, p0=[]):
        """ Linearize the continuous system around the operating point
            dx/dt = Ax + Bu
        # Arguments:
            x0: State vector
            u0: Input vector
            p0: Parameter vector (optional)
        """
        A = np.array(self.__jac_x(x0, u0, p0))
        B = np.array(self.__jac_u(x0, u0, p0))
        return A, B


    def discrete_linearize(self, x0, u0, p0=[]):
        """ Linearize the exact discrete system around the operating point
            x[k+1] = Ax[k] + Bu[k]
        # Arguments:
            x0: State vector
            u0: Input vector
            p0: Parameter vector (optional)
        """
        Ad = np.array(self.__discrete_jac_x(x0, u0, p0))
        Bd = np.array(self.__discrete_jac_u(x0, u0, p0))
        return Ad, Bd


    def discrete_rk4_linearize(self, x0, u0, p0=[]):
        """ Linearize the discrete rk4 system around the operating point
            x[k+1] = Ax[k] + Bu[k]
        # Arguments:
            x0: State vector
            u0: Input vector
            p0: Parameter vector (optional)
        """
        Ad = np.array(self.__discrete_rk4_jac_x(x0, u0, p0))
        Bd = np.array(self.__discrete_rk4_jac_u(x0, u0, p0))
        return Ad, Bd


    def rk4_jacobian_x(self, x0, u0, p0=[]):
        """ Return state jacobian evaluated at the operating point
            x[k+1] = Ax[k] + Bu[k]
        # Arguments:
            x0: State vector
            u0: Input vector
            p0: Parameter vector (optional)
        """
        return self.__discrete_rk4_jac_x(x0, u0, p0)


    def rk4_jacobian_u(self, x0, u0, p0=[]):
        """ Return input jacobian evaluated at the operating point
            x[k+1] = Ax[k] + Bu[k]
        # Arguments:
            x0: State vector
            u0: Input vector
            p0: Parameter vector (optional)
        """
        return self.__discrete_rk4_jac_u(x0, u0, p0)


    def check_rk4_stability(self, x0, u0, d=.1, plot=False):
        """ Check if Runga Kutta 4 method is stable around operating point

        # Return True if stable, False if not stable
        """
        A, B = self.linearize(x0, u0, p0=[])
        eigenvalues, eigenvec = scipy.linalg.eig(A)
        h = self.sampling_time()
        for eig in eigenvalues:
            R = 1 + h*eig + (h*eig)**2/2 + (h*eig)**3/6 + (h*eig)**4/24
            if np.abs(R) >= 1:
                return False
```

```
#         if plot:
#             h = d
##             N = 1000;
##             th = np.linspace(0, 2*np.pi, N);
##             r = np.exp(1j*th);
##             f = lambda r: 1 + h*r + (h*r)**2/2 + (h*r)**3/6 + (h*r)**4/24
#             plt.figure()
#             x = np.arange(-3.0, 3.0, 0.01)
#             y = np.arange(-3.0, 3.0, 0.01)
#             X, Y = np.meshgrid(x, y)
#             print(h)
#             z = X + 1j*Y;
#             R = 1 + h*z + (h*z)**2/2 + (h*z)**3/6 + (h*z)**4/24
#             print(R.shape)
#             zlevel4 = abs(R);
#             plt.contour(x,y, zlevel4)
#             plt.show()
        return True


    def sampling_time(self):
        """ Get the sampling time
        """
        return self.__dt


    def size(self):
        """ Get the size of the model

        # Returns:
                Nx: Number of states
                Nu: Number of inputs
                Np: Number of parameters
        """
        return self.__Nx, self.__Nu, self.__Np


    def integrate(self, x0, u, p):
        """ Integrate one time sample dt

        # Arguments:
            x0: Initial state vector
            u: Input vector
            p: Parameter vector
        # Returns:
            x: Numpy array with x at t0 + dt
        """
        par=ca.vertcat(u, p)
        if self.__Nz is not 0:
            z0 = self.__alg0(x0, u)
            out = self.Integrator(x0=x0, p=u, z0=z0)
        else:
            out = self.Integrator(x0=x0, p=par)
        return np.array(out["xf"]).flatten()

#TODO: Fix this or remove
    def set_method(self, method='exact'):
        """ Select wich discrete time method to use """


    def sim(self, x0, u, p=None, noise=False):
        """ Simulate system

        # Arguments:
            x0: Initial state (Nx, 1)
            u: Input matrix with the input for each timestep in the
                simulation horizon (Nt, Nu)
            p: Parameter matrix with the parameters for each timestep
                in the simulation horizon (Nt, Np)
            noise: If True, add gaussian noise using the noise covariance matrix

        # Output:
            Y_sim: Matrix with the simulated outputs (Nt, Ny)
        """

        Nt = np.size(u, 0)

        # Initial state of the system
        x = x0

        # Predefine matrix to collect noisy state outputs
        Y = np.zeros((Nt, self.__Nx))

        for t in range(Nt):
            u_t = u[t, :]    # control input for simulation
            if p is not None:
                p_t = p[t, :]    # parameter at step t
            else:
                p_t = []
            try:
                x = self.integrate(x, u_t, p_t).flatten()
            except RuntimeError:
                print('-----------------------------------------')
                print('** System unstable, simulator crashed **')
                print('** t: %d **' % t)
                print('-----------------------------------------')
                return Y
            Y[t, :] = x

            # Add normal white noise to state outputs
            if noise:
                Y[t, :] += np.random.multivariate_normal(
                                    np.zeros((self.__Nx)), self.__R)

            # Limit values to above 1e-8 to avvoid to avvoid numerical errors
            if self.__clip_negative:
                if np.any(Y < 0):
                    print('Clipping negative values in simulation!')
                    Y = Y.clip(min=1e-6)
        return Y


    def generate_training_data(self, N, uub, ulb, xub, xlb,
                               pub=None, plb=None, noise=True):
```

```
""" Generate training data using latin hypercube design

# Arguments:
    N:   Number of data points to be generated
    uub: Upper input range (Nu,1)
    ulb: Lower input range (Nu,1)
    xub: Upper state range (Ny,1)
    xlb: Lower state range (Ny,1)

# Returns:
    Z: Matrix (N, Nx + Nu) with state x and inputs u at each row
    Y: Matrix (N, Nx) where each row is the state x at time t+dt,
        with the input from the same row in Z at time t.
"""
# Make sure boundry vectors are numpy arrays
uub = np.array(uub)
ulb = np.array(ulb)
xub = np.array(xub)
xlb = np.array(xlb)

# Predefine matrix to collect noisy state outputs
Y = np.zeros((N, self.__Nx))

# Create control input design using a latin hypecube
# Latin hypercube design for unit cube [0,1]^Nu
if self.__Nu > 0:
    U = pyDOE.lhs(self.__Nu, samples=N, criterion='maximin')
     # Scale control inputs to correct range
    for k in range(N):
        U[k, :] = U[k, :] * (uub - ulb) + ulb
else:
    U = []

# Create state input design using a latin hypecube
# Latin hypercube design for unit cube [0,1]^Ny
X = pyDOE.lhs(self.__Nx, samples=N, criterion='maximin')

# Scale state inputs to correct range
for k in range(N):
    X[k, :] = X[k, :] * (xub - xlb) + xlb

# Create parameter matrix
par = pyDOE.lhs(self.__Np, samples=N)
if pub is not None:
    for k in range(N):
        par[k, :] = par[k, :] * (pub - plb) + plb

for i in range(N):
    if self.__Nu > 0:
        u_t = U[i, :]     # control input for simulation
    else:
        u_t = []
    x_t = X[i, :]     # state input for simulation
    p_t = par[i, :]     # parameter input for simulation

    # Simulate system with x_t and u_t inputs for deltat time
    Y[i, :] = self.integrate(x_t, u_t, p_t)
```

```
            # Add normal white noise to state outputs
            if noise:
                Y[i, :] += np.random.multivariate_normal(
                                np.zeros((self.__Nx)), self.__R)

        # Concatenate previous states and inputs to obtain overall input to GP model
        if self.__Nu > 0:
            Z = np.hstack([X, U])
        else:
            Z = X
        return Z, Y


    def plot(self, x0, u, numcols=2):
        """ Simulate and plot model

        # Arguments:
            x0: Initial state
            u: Matrix with inputs for all time steps (Nt, Nu)
            numcols: Number of columns in the plot
        """
        y = self.sim(x0, u, noise=True)
        Nt = np.size(u, 0)
        t = np.linspace(0.0, (Nt - 1)* self.__dt, Nt )
        numrows = int(np.ceil(self.__Nx / numcols))

        fig_x = plt.figure()
        for i in range(self.__Nx):
            ax = fig_x.add_subplot(numrows, numcols, i + 1)
            ax.plot(t, y[:, i], 'b-', marker='.', linewidth=1.0)
            ax.set_ylabel('x_' + str(i + 1))
            ax.set_xlabel('Time')
        fig_x.canvas.set_window_title('Model simulation')
        plt.show()


    def predict_compare(self, x0, u, num_cols=2, xnames=None, title=None,):
        """ Predict and compare dicrete RK4 model and linearized model against
            the exact model.
        """
        # Predict future
        Nx = self.__Nx

        dt = self.sampling_time()
        Nt = np.size(u, 0)
        sim_time = Nt * dt

        # Exact model with no noise
        y_exact = self.sim(x0, u, noise=False)
        y_exact = np.vstack([x0, y_exact])

        # RK4
        y_rk4 = np.zeros((Nt + 1 , Nx))
        y_rk4[0] = x0
        for t in range(Nt):
            y_rk4[t + 1]= np.array(self.rk4(y_rk4[t], u[t-1, :], [])).reshape((Nx,))
        #  Linearized Model of Exact discretization
        Ad, Bd = self.discrete_linearize(x0, u[0])
        y_lin = np.zeros((Nt + 1, Nx))
        y_lin[0] = x0
        for t in range(Nt):
            y_lin[t+1] = Ad @ y_lin[t] + Bd @ u[t]

        #  Linearized Model of RK4 discretization
        Ad, Bd = self.discrete_rk4_linearize(x0, u[0])
        y_rk4_lin = np.zeros((Nt + 1, Nx))
        y_rk4_lin[0] = x0
        for t in range(Nt):
            y_rk4_lin[t+1] = Ad @ y_rk4_lin[t] + Bd @ u[t]

        t = np.linspace(0.0, sim_time, Nt + 1)


        num_rows = int(np.ceil(Nx / num_cols))
        if xnames is None:
            xnames = ['State %d' % (i + 1) for i in range(Nx)]

        fontP = FontProperties()
        fontP.set_size('small')
        fig = plt.figure(figsize=(9.0, 6.0))
        for i in range(Nx):
            ax = fig.add_subplot(num_rows, num_cols, i + 1)
            ax.plot(t, y_exact[:, i], 'b-', label='Exact')
            ax.plot(t, y_rk4[:, i], 'r-', label='RK4')
#             ax.plot(t, y_lin[:, i], 'g--', label='Linearized')
#             ax.plot(t, y_lin[:, i], 'y--', label='Linearized RK4')
            ax.set_ylabel(xnames[i])
            ax.legend(prop=fontP, loc='best')
            ax.set_xlabel('Time [s]')
        if title is not None:
            fig.canvas.set_window_title(title)
        else:
            fig.canvas.set_window_title('Compare approximations of system model')
        plt.tight_layout()
        plt.show()
```

## E.5 mpc_class

```
# -*- coding: utf-8 -*-
"""
Model Predictive Control with Gaussian Process
Copyright (c) 2018, Helge-André Langåker
"""
from __future__ import absolute_import
from __future__ import division
from __future__ import print_function

import time
import numpy as np
import matplotlib.pyplot as plt
import casadi as ca
import casadi.tools as ctools
from scipy.stats import norm
import scipy.linalg


class MPC:
    def __init__(self, horizon, model, gp=None,
                 Q=None, P=None, R=None, S=None, lam=None, lam_state=None,
                 ulb=None, uub=None, xlb=None, xub=None, terminal_constraint=None,
                 feedback=True, percentile=None, gp_method='TA', costFunc='quad',
                 solver_opts=None, discrete_method='gp', inequality_constraints=None,
                 num_con_par=0, hybrid=None, Bd=None, Bf=None
                 ):

        """ Initialize and build the MPC solver

        # Arguments:
            horizon: Prediction horizon with control inputs
            model: System model

        # Optional Argumants:
            gp: GP model
            Q: State penalty matrix, default=diag(1,...,1)
            P: Termial penalty matrix, default=diag(1,...,1)
                if feedback is True, then P is the solution of the DARE,
                discarding this option.
            R: Input penalty matrix, default=diag(1,...,1)*0.01
            S: Input rate of change penalty matrix, default=diag(1,...,1)*0.1
            lam: Slack variable penalty for constraints, defalt=1000
            lam_state: Slack variable penalty for violation of upper/lower
                        state boundy, defalt=None
            ulb: Lower boundry input
            uub: Upper boundry input
            xlb: Lower boundry state
            xub: Upper boundry state
            terminal_constraint: Terminal condition on the state
                    * if None: No terminal constraint is used
                    * if zero: Terminal state is equal to zero
                    * if nonzero: Terminal state is bounded within +/- the constraint
                    * if not None and feedback is True, then the expected value of
                        the Lyapunov function E{x^TPx} < terminal_constraint
                        is used as a terminal constraint.
            feedback: If true, use an LQR feedback function u= Kx + v
            percentile: Measure how far from the contrain that is allowed,
                        P(X in constrained set) > percentile,
                        percentile= 1 - probability of violation,
                        default=0.95
            gp_method: Method of propagating the uncertainty
                    Possible options:
                        'TA': Second order Taylor approximation
                        'ME': Mean equivalent approximation

            costFunc: Cost function to use in the objective
                    'quad': Expected valaue of Quadratic Cost
                    'sat':  Expected value of Saturating cost
            solver_opts: Additional options to pass to the NLP solver
                    e.g.: solver_opts['print_time'] = False
                    solver_opts['ipopt.tol'] = 1e-8
        discrete_method: 'gp' -  Gaussian process model
                         'rk4' - Runga-Kutta 4 Integrator
                         'exact' - CVODES or IDEAS (for ODEs or DEAs)
                         'hybrid' - GP model for dynamic equations, and RK4
                                    for kinematic equations
                         'd_hybrid' - Same as above, without uncertainty
                         'f_hybrid' - GP estimating modelling errors, with
                                      RK4 computing the the actual model
        num_con_par: Number of parameters to pass to the inequality function
        inequality_constraints: Additional inequality constraints
                Use a function with inputs (x, covar, u, eps) and
                that returns a dictionary with inequality constraints and limits.
                    e.g. cons = dict(con_ineq=con_ineq_array,
                                     con_ineq_lb=con_ineq_lb_array,
                                     con_ineq_ub=con_ineq_ub_array
                                )

    # NOTES:
        * Differentiation of Sundails integrators is not supported with SX graph,
          meaning that the solver option 'extend_graph' must be set to False
          to use MX graph instead when using the 'exact' discrete method.
        * At the moment the f_hybrid option is not finished implemented...
    """


    build_solver_time = -time.time()
    dt = model.sampling_time()
    Ny, Nu, Np = model.size()
    Nx = Nu + Ny
    Nt = int(horizon / dt)

    self.__dt = dt
    self.__Nt = Nt
    self.__Ny = Ny
    self.__Nx = Nx
    self.__Nu = Nu
    self.__num_con_par = num_con_par
    self.__model = model
    self.__hybrid = hybrid
    self.__gp = gp
    self.__feedback = feedback
    self.__discrete_method = discrete_method


    """ Default penalty values """
    if P is None:
        P = np.eye(Ny)
    if Q is None:
        Q = np.eye(Ny)
    if R is None:
        R = np.eye(Nu) * 0.01
    if S is None:
        S = np.eye(Nu) * 0.1
    if lam is None:
        lam = 1000
        self.__Q = Q
        self.__P = P
        self.__R = R
        self.__S = S
        self.__Bd = Bd
        self.__Bf = Bf

        if xub is None:
            xub = np.full((Ny), np.inf)
        if xlb is None:
            xlb = np.full((Ny), -np.inf)
        if uub is None:
            uub = np.full((Nu), np.inf)
        if ulb is None:
            ulb = np.full((Nu), -np.inf)

        """ Default percentile probability """
        if percentile is None:
            percentile = 0.95
        quantile_x = np.ones(Ny) * norm.ppf(percentile)
        quantile_u = np.ones(Nu) * norm.ppf(percentile)
        Hx = ca.MX.eye(Ny)
        Hu = ca.MX.eye(Nu)


        """ Create parameter symbols """
        mean_0_s       = ca.MX.sym('mean_0', Ny)
        mean_ref_s     = ca.MX.sym('mean_ref', Ny)
        u_0_s          = ca.MX.sym('u_0', Nu)
        covariance_0_s = ca.MX.sym('covariance_0', Ny * Ny)
        K_s            = ca.MX.sym('K', Nu * Ny)
        P_s            = ca.MX.sym('P', Ny * Ny)
        con_par        = ca.MX.sym('con_par', num_con_par)
        param_s        = ca.vertcat(mean_0_s, mean_ref_s, covariance_0_s,
                                    u_0_s, K_s, P_s, con_par)


        """ Select wich GP function to use """
        if discrete_method is 'gp':
            self.__gp.set_method(gp_method)
#TODO:Fix
        if solver_opts['expand'] is not False and discrete_method is 'exact':
            raise TypeError("Can't use exact discrete system with expanded graph")

        """ Initialize state variance with the GP noise variance """
        if gp is not None:
            #TODO: Cannot use gp variance with hybrid model
            self.__variance_0 = np.full((Ny), 1e-10) #gp.noise_variance()
        else:
            self.__variance_0 = np.full((Ny), 1e-10)


        """ Define which cost function to use """
        self.__set_cost_function(costFunc, mean_ref_s, P_s.reshape((Ny, Ny)))


        """ Feedback function """
        mean_s = ca.MX.sym('mean', Ny)
        v_s = ca.MX.sym('v', Nu)
        if feedback:
            u_func = ca.Function('u', [mean_s, mean_ref_s, v_s, K_s],
                                 [v_s + ca.mtimes(K_s.reshape((Nu, Ny)),
                                 mean_s-mean_ref_s)])
        else:
            u_func = ca.Function('u', [mean_s, mean_ref_s, v_s, K_s], [v_s])
        self.__u_func = u_func


        """ Create variables struct """
        var = ctools.struct_symMX([(
                ctools.entry('mean', shape=(Ny,), repeat=Nt + 1),
                ctools.entry('L', shape=(int((Ny**2 - Ny)/2 + Ny),), repeat=Nt + 1),
                ctools.entry('v', shape=(Nu,), repeat=Nt),
                ctools.entry('eps', shape=(3,), repeat=Nt + 1),
                ctools.entry('eps_state', shape=(Ny,), repeat=Nt + 1),
        )])
        num_slack = 3 #TODO: Make this a little more dynamic...
        num_state_slack = Ny
        self.__var = var
        self.__num_var = var.size

        # Decision variable boundries
        self.__varlb = var(-np.inf)
        self.__varub = var(np.inf)

        """ Adjust hard boundries """
        for t in range(Nt + 1):
            j = Ny
            k = 0
            for i in range(Ny):
                # Lower boundry of diagonal
                self.__varlb['L', t, k] = 0
                k += j
                j -= 1
            self.__varlb['eps', t] = 0
            self.__varlb['eps_state', t] = 0
            if xub is not None:
                self.__varub['mean', t] = xub
            if xlb is not None:
                self.__varlb['mean', t] = xlb
            if lam_state is None:
                self.__varub['eps_state'] = 0


        """ Input covariance matrix """
        if discrete_method is 'hybrid':
            N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
            Nz_gp = Ny_gp + Nu_gp
            covar_d_sx = ca.SX.sym('cov_d', Ny_gp, Ny_gp)
            K_sx = ca.SX.sym('K', Nu, Ny)
            covar_u_func = ca.Function('cov_u', [covar_d_sx, K_sx],
#                                       [K_sx @ covar_d_sx @ K_sx.T])
                                        [ca.SX(Nu, Nu)])
            covar_s = ca.SX(Nz_gp, Nz_gp)
            covar_s[:Ny_gp, :Ny_gp] = covar_d_sx
#            covar_s = ca.blockcat(covar_x_s, cov_xu, cov_xu.T, cov_u)
            covar_func = ca.Function('covar', [covar_d_sx], [covar_s])
        elif discrete_method is 'f_hybrid':
            #TODO: Fix this...
                N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
                Nz_gp = Ny_gp + Nu_gp
    #            covar_x_s = ca.MX.sym('covar_x', Ny_gp, Ny_gp)
                covar_d_sx = ca.SX.sym('cov_d', Ny_gp, Ny_gp)
                K_sx = ca.SX.sym('K', Nu, Ny)
    #
                covar_u_func = ca.Function('cov_u', [covar_d_sx, K_sx],
    #                                       [K_sx @ covar_d_sx @ K_sx.T])
                                           [ca.SX(Nu, Nu)])
    #            cov_xu_func = ca.Function('cov_xu', [covar_x_sx, K_sx],
    #                                      [covar_x_sx @ K_sx.T])
    #            cov_xu = cov_xu_func(covar_x_s, K_s.reshape((Nu, Ny)))
    #            cov_u = covar_u_func(covar_x_s, K_s.reshape((Nu, Ny)))
                covar_s = ca.SX(Nz_gp, Nz_gp)
                covar_s[:Ny_gp, :Ny_gp] = covar_d_sx
    #            covar_s = ca.blockcat(covar_x_s, cov_xu, cov_xu.T, cov_u)
                covar_func = ca.Function('covar', [covar_d_sx], [covar_s])
        else:
            covar_x_s = ca.MX.sym('covar_x', Ny, Ny)
            covar_x_sx = ca.SX.sym('cov_x', Ny, Ny)
            K_sx = ca.SX.sym('K', Nu, Ny)
            covar_u_func = ca.Function('cov_u', [covar_x_sx, K_sx],
                                       [K_sx @ covar_x_sx @ K_sx.T])
            cov_xu_func = ca.Function('cov_xu', [covar_x_sx, K_sx],
                                      [covar_x_sx @ K_sx.T])
            cov_xu = cov_xu_func(covar_x_s, K_s.reshape((Nu, Ny)))
            cov_u = covar_u_func(covar_x_s, K_s.reshape((Nu, Ny)))
            covar_s = ca.blockcat(covar_x_s, cov_xu, cov_xu.T, cov_u)
            covar_func = ca.Function('covar', [covar_x_s], [covar_s])

        """ Hybrid output covariance matrix """
        if discrete_method is 'hybrid':
            N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
            covar_d_sx = ca.SX.sym('covar_d', Ny_gp, Ny_gp)
            covar_x_sx = ca.SX.sym('covar_x', Ny, Ny)
            u_s       = ca.SX.sym('u', Nu)

            cov_x_next_s = ca.SX(Ny, Ny)
            cov_x_next_s[:Ny_gp, :Ny_gp] = covar_d_sx
            #TODO: Missing kinematic states
            covar_x_next_func = ca.Function( 'cov',
                                #[mean_s, u_s, covar_d_sx, covar_x_sx],
                                [covar_d_sx],
                                [cov_x_next_s])

            """ f_hybrid output covariance matrix """
        elif discrete_method is 'f_hybrid':
                N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
    #            Nz_gp = Ny_gp + Nu_gp
                covar_d_sx = ca.SX.sym('covar_d', Ny_gp, Ny_gp)
                covar_x_sx = ca.SX.sym('covar_x', Ny, Ny)
    #            L_x       = ca.SX.sym('L', ca.Sparsity.lower(Ny))
    #            L_d       = ca.SX.sym('L', ca.Sparsity.lower(3))
                mean_s     = ca.SX.sym('mean', Ny)
                u_s        = ca.SX.sym('u', Nu)

    #            A_f     = hybrid.rk4_jacobian_x(mean_s[Ny_gp:], mean_s[:Ny_gp])
    #            B_f     = hybrid.rk4_jacobian_u(mean_s[Ny_gp:], mean_s[:Ny_gp])
    #            C       = ca.horzcat(A_f, B_f)
    #            cov = ca.blocksplit(covar_x_s, Ny_gp, Ny_gp)
    #            cov[-1][-1] = covar_d_sx
    #            cov_i = ca.blockcat(cov)
    #            cov_f   =  C @ cov_i @ C.T
    #            cov[0][0] = cov_f

                cov_x_next_s = ca.SX(Ny, Ny)
                cov_x_next_s[:Ny_gp, :Ny_gp] = covar_d_sx
    #            cov_x_next_s[Ny_gp:, Ny_gp:] =
    #TODO: Pre-solve the GP jacobian using the initial condition in the solve iteration
    #            jac_mean  = ca.SX(Ny_gp, Ny)
    #            jac_mean = self.__gp.jacobian(mean_s[:Ny_gp], u_s, 0)
    #            A = ca.horzcat(jac_f, Bd)
    #            jac = Bf @ jac_f @ Bf.T + Bd @ jac_mean @ Bd.T

    #            temp = jac_mean @ covar_x_s
    #            temp = jac_mean @ L_s
    #            cov_i = ca.SX(Ny + 3, Ny + 3)
    #            cov_i[:Ny,:Ny] = covar_x_s
    #            cov_i[Ny:, Ny:] = covar_d_s
    #            cov_i[Ny:, :Ny] = temp
    #            cov_i[:Ny, Ny:] = temp.T
                #TODO: This is just a new TA implementation... CLEAN UP...
                covar_x_next_func = ca.Function( 'cov',
                                    [mean_s, u_s, covar_d_sx, covar_x_sx],
                                    #TODO: Clean up
                                    #[A @ cov_i @ A.T])
                                    #[Bd @ covar_d_s @ Bd.T + jac @ covar_x_s @ jac.T])
                                    #[ca.blockcat(cov)])
                                    [cov_x_next_s])
                # Cholesky factorization of covariance function
    #            S_x_next_func = ca.Function( 'S_x', [mean_s, u_s, covar_d_s, covar_x_s],
    #                                          [Bd @ covar_d_s + jac @ covar_x_s])



        L_s = ca.SX.sym('L', ca.Sparsity.lower(Ny))
        L_to_cov_func = ca.Function('cov', [L_s], [L_s @ L_s.T])
        covar_x_sx = ca.SX.sym('cov_x', Ny, Ny)
        cholesky = ca.Function('cholesky', [covar_x_sx], [ca.chol(covar_x_sx).T])

        """ Set initial values """
        obj = ca.MX(0)
        con_eq = []
        con_ineq = []
        con_ineq_lb = []
        con_ineq_ub = []
        con_eq.append(var['mean', 0] - mean_0_s)
        L_0_s = ca.MX(ca.Sparsity.lower(Ny), var['L', 0])
        L_init = cholesky(covariance_0_s.reshape((Ny,Ny)))
```

```
con_eq.append(L_0_s.nz[:]- L_init.nz[:])
u_past = u_0_s


""" Build constraints """
for t in range(Nt):
    # Input to GP
    mean_t = var['mean', t]
    u_t = u_func(mean_t, mean_ref_s, var['v', t], K_s)
    L_x = ca.MX(ca.Sparsity.lower(Ny), var['L', t])
    covar_x_t = L_to_cov_func(L_x)

    if discrete_method is 'hybrid':
        N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
        covar_t = covar_func(covar_x_t[:Ny_gp, :Ny_gp])
    elif discrete_method is 'd_hybrid':
        N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
        covar_t = ca.MX(Ny_gp + Nu_gp, Ny_gp + Nu_gp)
    elif discrete_method is 'gp':
        covar_t = covar_func(covar_x_t)
    else:
        covar_t = ca.MX(Nx, Nx)


    """ Select the chosen integrator """
    if discrete_method is 'rk4':
        mean_next_pred = model.rk4(mean_t, u_t,[])
        covar_x_next_pred = ca.MX(Ny, Ny)
    elif discrete_method is 'exact':
        mean_next_pred = model.Integrator(x0=mean_t, p=u_t)['xf']
        covar_x_next_pred = ca.MX(Ny, Ny)
    elif discrete_method is 'd_hybrid':
        # Deterministic hybrid GP model
        N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
        mean_d, covar_d = self.__gp.predict(mean_t[:Ny_gp], u_t, covar_t)
        mean_next_pred = ca.vertcat(mean_d, hybrid.rk4(mean_t[Ny_gp:],
                                    mean_t[:Ny_gp], []))
        covar_x_next_pred = ca.MX(Ny, Ny)
    elif discrete_method is 'hybrid':
        # Hybrid GP model
        N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
        mean_d, covar_d = self.__gp.predict(mean_t[:Ny_gp], u_t, covar_t)
        mean_next_pred = ca.vertcat(mean_d, hybrid.rk4(mean_t[Ny_gp:],
                                            mean_t[:Ny_gp], []))
        #covar_x_next_pred = covar_x_next_func(mean_t, u_t, covar_d,
        #                                      covar_x_t)
        covar_x_next_pred = covar_x_next_func(covar_d )
    elif discrete_method is 'f_hybrid':
        #TODO: Hybrid GP model estimating model error
        N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
        mean_d, covar_d = self.__gp.predict(mean_t[:Ny_gp], u_t, covar_t)
        mean_next_pred = ca.vertcat(mean_d, hybrid.rk4(mean_t[Ny_gp:],
                                            mean_t[:Ny_gp], []))
        covar_x_next_pred = covar_x_next_func(mean_t, u_t, covar_d,
                                              covar_x_t)
    else: # Use GP as default
        mean_next_pred, covar_x_next_pred = self.__gp.predict(mean_t,
                                                                u_t, covar_t)

            """ Continuity constraints """
            mean_next = var['mean', t + 1]
            con_eq.append(mean_next_pred - mean_next )

            L_x_next = ca.MX(ca.Sparsity.lower(Ny), var['L', t + 1])
            covar_x_next = L_to_cov_func(L_x_next).reshape((Ny*Ny,1))
            L_x_next_pred = cholesky(covar_x_next_pred)
            con_eq.append(L_x_next_pred.nz[:] - L_x_next.nz[:])


            """ Chance state constraints """
            cons = self.__constraint(mean_next, L_x_next, Hx, quantile_x, xub,
                                    xlb, var['eps_state',t])
            con_ineq.extend(cons['con'])
            con_ineq_lb.extend(cons['con_lb'])
            con_ineq_ub.extend(cons['con_ub'])

            """ Input constraints """
#            cov_u = covar_u_func(covar_x_t, K_s.reshape((Nu, Ny)))
            cov_u = ca.MX(Nu, Nu)
#            cons = self.__constraint(u_t, cov_u, Hu, quantile_u, uub, ulb)
#            con_ineq.extend(cons['con'])
#            con_ineq_lb.extend(cons['con_lb'])
#            con_ineq_ub.extend(cons['con_ub'])
            if uub is not None:
                con_ineq.append(u_t)
                con_ineq_ub.extend(uub)
                con_ineq_lb.append(np.full((Nu,), -ca.inf))
            if ulb is not None:
                con_ineq.append(u_t)
                con_ineq_ub.append(np.full((Nu,), ca.inf))
                con_ineq_lb.append(ulb)

            """ Add extra constraints """
            if inequality_constraints is not None:
                cons = inequality_constraints(var['mean', t + 1],
                                              covar_x_next,
                                              u_t, var['eps', t], con_par)
                con_ineq.extend(cons['con_ineq'])
                con_ineq_lb.extend(cons['con_ineq_lb'])
                con_ineq_ub.extend(cons['con_ineq_ub'])

            """ Objective function """
            u_delta = u_t - u_past
            obj += self.__l_func(var['mean', t], covar_x_t, u_t, cov_u, u_delta) \
                    + np.full((1, num_slack),lam) @ var['eps', t]
            if lam_state is not None:
                obj += np.full((1,num_state_slack),lam_state) @ var['eps_state', t]
            u_t = u_past
        L_x = ca.MX(ca.Sparsity.lower(Ny), var['L', Nt])
        covar_x_t = L_to_cov_func(L_x)
        obj += self.__lf_func(var['mean', Nt], covar_x_t, P_s.reshape((Ny, Ny))) \
            + np.full((1, num_slack),lam) @ var['eps', Nt]
        if lam_state is not None:
```

```
        obj += np.full((1,num_state_slack),lam_state) @ var['eps_state', Nt]

    num_eq_con = ca.vertcat(*con_eq).size1()
    num_ineq_con = ca.vertcat(*con_ineq).size1()
    con_eq_lb = np.zeros((num_eq_con,))
    con_eq_ub = np.zeros((num_eq_con,))

    """ Terminal contraint """
    if terminal_constraint is not None and not feedback:
        con_ineq.append(var['mean', Nt] - mean_ref_s)
        num_ineq_con += Ny
        con_ineq_lb.append(np.full((Ny,), - terminal_constraint))
        con_ineq_ub.append(np.full((Ny,), terminal_constraint))
    elif terminal_constraint is not None and feedback:
        con_ineq.append(self.__lf_func(var['mean', Nt],
                        covar_x_t, P_s.reshape((Ny, Ny))))
        num_ineq_con += 1
        con_ineq_lb.append(0)
        con_ineq_ub.append(terminal_constraint)
    con = ca.vertcat(*con_eq, *con_ineq)
    self.__conlb = ca.vertcat(con_eq_lb, *con_ineq_lb)
    self.__conub = ca.vertcat(con_eq_ub, *con_ineq_ub)

    """ Build solver object """
    nlp = dict(x=var, f=obj, g=con, p=param_s)
    options = {
        'ipopt.print_level' : 0,
        'ipopt.mu_init' : 0.01,
        'ipopt.tol' : 1e-8,
        'ipopt.warm_start_init_point' : 'yes',
        'ipopt.warm_start_bound_push' : 1e-9,
        'ipopt.warm_start_bound_frac' : 1e-9,
        'ipopt.warm_start_slack_bound_frac' : 1e-9,
        'ipopt.warm_start_slack_bound_push' : 1e-9,
        'ipopt.warm_start_mult_bound_push' : 1e-9,
        'ipopt.mu_strategy' : 'adaptive',
        'print_time' : False,
        'verbose' : False,
        'expand' : True
    }
    if solver_opts is not None:
        options.update(solver_opts)
    self.__solver = ca.nlpsol('mpc_solver', 'ipopt', nlp, options)

    # First prediction used in the NLP, used in plot later
    self.__var_prediction = np.zeros((Nt + 1, Ny))
    self.__mean_prediction = np.zeros((Nt + 1, Ny))
    self.__mean = None

    build_solver_time += time.time()
    print('\n________________________________________')
    print('# Time to build mpc solver: %f sec' % build_solver_time)
    print('# Number of variables: %d' % self.__num_var)
    print('# Number of equality constraints: %d' % num_eq_con)
    print('# Number of inequality constraints: %d' % num_ineq_con)
    print('----------------------------------------')
```

```python
def solve(self, x0, sim_time, x_sp=None, u0=None, debug=False, noise=False,
          con_par_func=None):
    """ Solve the optimal control problem

    # Arguments:
        x0: Initial state vector.
        sim_time: Simulation length.

    # Optional Arguments:
        x_sp: State set point, default is zero.
        u0: Initial input vector.
        debug: If True, print debug information at each solve iteration.
        noise: If True, add gaussian noise to the simulation.
        con_par_func: Function to calculate the parameters to pass to the
                      inequality function, inputs the current state.

    # Returns:
        mean: Simulated output using the optimal control inputs
        u: Optimal control inputs
    """

    Nt = self.__Nt
    Ny = self.__Ny
    Nu = self.__Nu
    dt = self.__dt

    # Initial state
    if u0 is None:
        u0 = np.zeros(Nu)
    if x_sp is None:
        self.__x_sp = np.zeros(Ny)
    else:
        self.__x_sp = x_sp

    self.__Nsim = int(sim_time / dt)

    # Initialize variables
    self.__mean          = np.full((self.__Nsim + 1, Ny), x0)
    self.__mean_pred     = np.full((self.__Nsim + 1, Ny), x0)
    self.__covariance    = np.full((self.__Nsim + 1, Ny, Ny), np.eye(Ny) * 1e-8)
    self.__u             = np.full((self.__Nsim, Nu), u0)

    self.__mean[0]       = x0
    self.__mean_pred[0]  = x0
    #TODO: cannot use variance_0 with a hybrid model
    self.__covariance[0] = np.eye(Ny)*1e-10 #np.diag(self.__variance_0)
    self.__u[0]          = u0

    # Initial guess of the warm start variables
    #TODO: Add option to restart with previous state
    self.__var_init = self.__var(0)

    #TOTO: Add initialization of cov cholesky
    cov0 = self.__covariance[0]
    self.__var_init['L', 0] = cov0[np.tril_indices(Ny)]
```

```
        self.__lam_x0 = np.zeros(self.__num_var)
        self.__lam_g0 = 0

        """ Linearize around operating point and calculate LQR gain matrix """
        if self.__feedback:
            if self.__discrete_method is 'exact':
                A, B = self.__model.discrete_linearize(x0, u0)
            elif self.__discrete_method is 'rk4':
                A, B = self.__model.discrete_rk4_linearize(x0, u0)
            elif self.__discrete_method is 'hybrid':
                N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
                A_f, B_f = self.__hybrid.discrete_rk4_linearize(x0[Ny_gp:], x0[:Ny_gp])
                A_gp, B_gp = self.__gp.discrete_linearize(x0[:Ny_gp],
                                                u0, np.eye(Ny_gp+Nu_gp)*1e-8)
                A = np.zeros((Ny, Ny))
                B = np.zeros((Ny, Nu))
                A[:Ny_gp, :Ny_gp] = A_gp
#                A[Ny_gp:, Ny_gp:] = A_f
#                A[Ny_gp:, :Ny_gp] = B_f
                B[:Ny_gp, :] = B_gp

            elif self.__discrete_method is 'd_hybrid':
                N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
                A_f, B_f = self.__hybrid.discrete_rk4_linearize(x0[Ny_gp:], x0[:Ny_gp])
                A_gp, B_gp = self.__gp.discrete_linearize(x0[:Ny_gp],
                                                u0, np.eye(Ny_gp+Nu_gp)*1e-8)
                A = np.zeros((Ny, Ny))
                B = np.zeros((Ny, Nu))
                A[:Ny_gp, :Ny_gp] = A_gp
#                A[Ny_gp:, Ny_gp:] = A_f
#                A[Ny_gp:, :Ny_gp] = B_f
                B[:Ny_gp, :] = B_gp

#                A = self.__Bf @ A_f @ self.__Bf.T + self.__Bd @ A_gp @ self.__Bd.T
#                B = self.__Bf @ B_f + self.__Bd @ B_gp

            elif self.__discrete_method is 'gp':
                N_gp, Ny_gp, Nu_gp = self.__gp.get_size()
                A, B = self.__gp.discrete_linearize(x0,
                                                u0, np.eye(Ny_gp+Nu_gp)*1e-8)

            K, P, E = lqr(A, B, self.__Q, self.__R)
        else:
            K = np.zeros((Nu, Ny))
            P = self.__P
        self.__K = K

        print('\nSolving MPC with %d step horizon' % Nt)
        for t in range(self.__Nsim):
            solve_time = -time.time()

            # Test if RK4 is stable for given initial state
            if self.__discrete_method is 'rk4':
                if not self.__model.check_rk4_stability(x0,u0):
                    print('-- WARNING: RK4 is not stable! --')

            """ Update Initial values with measurment"""
```

```
self.__var_init['mean', 0]  = self.__mean[t]

# Get constraint parameters
if con_par_func is not None:
    con_par = con_par_func(self.__mean[t, :])
else:
    con_par = []
    if self.__num_con_par > 0:
        raise TypeError(('Number of constraint parameters ({x}) is '
                         'greater than zero, but no parameter '
                         'function is provided.'
                            ).format(x=self.__num_con_par))

param  = ca.vertcat(self.__mean[t, :], self.__x_sp,
                    cov0.flatten(), u0, K.flatten(),
                    P.flatten(), con_par)
args = dict(x0=self.__var_init,
            lbx=self.__varlb,
            ubx=self.__varub,
            lbg=self.__conlb,
            ubg=self.__conub,
            lam_x0=self.__lam_x0,
            lam_g0=self.__lam_g0,
            p=param)

""" Solve nlp"""
sol             = self.__solver(**args)
status          = self.__solver.stats()['return_status']
optvar          = self.__var(sol['x'])
self.__var_init = optvar
self.__lam_x0   = sol['lam_x']
self.__lam_g0   = sol['lam_g']

""" Print status """
solve_time     += time.time()
print("* t=%f: %s - %f sec" % (t * self.__dt, status, solve_time))

if t == 0:
     for i in range(Nt + 1):
         Li = ca.DM(ca.Sparsity.lower(self.__Ny), optvar['L', i])
         cov = Li @ Li.T
         self.__var_prediction[i, :] = np.array(ca.diag(cov)).flatten()
         self.__mean_prediction[i, :] = np.array(optvar['mean', i]).flatten()

v = optvar['v', 0, :]

self.__u[t, :] = np.array(self.__u_func(self.__mean[t, :], self.__x_sp,
                    v, K.flatten())).flatten()
self.__mean_pred[t + 1] = np.array(optvar['mean', 1]).flatten()
L = ca.DM(ca.Sparsity.lower(self.__Ny), optvar['L', 1])
self.__covariance[t + 1] = L @ L.T

if debug:
    self.__debug(t)

""" Simulate the next step """
try:
```

```
                self.__mean[t + 1] = self.__model.sim(self.__mean[t],
                                       self.__u[t].reshape((1, Nu)), noise=noise)
            except RuntimeError:
                print('----------------------------------------')
                print('** System unstable, simulator crashed **')
                print('----------------------------------------')
                return self.__mean, self.__u

            """Initial values for next iteration"""
            x0 = self.__mean[t + 1]
            u0 = self.__u[t]
        return self.__mean, self.__u


    def __set_cost_function(self, costFunc, mean_ref_s, P_s):
        """ Define stage cost and terminal cost
        """

        mean_s = ca.MX.sym('mean', self.__Ny)
        covar_x_s = ca.MX.sym('covar_x', self.__Ny, self.__Ny)
        covar_u_s = ca.MX.sym('covar_u', self.__Nu, self.__Nu)
        u_s = ca.MX.sym('u', self.__Nu)
        delta_u_s = ca.MX.sym('delta_u', self.__Nu)
        Q = ca.MX(self.__Q)
        R = ca.MX(self.__R)
        S = ca.MX(self.__S)

        if costFunc is 'quad':
            self.__l_func = ca.Function('l', [mean_s, covar_x_s, u_s,
                                                covar_u_s, delta_u_s],
                                   [self.__cost_l(mean_s, mean_ref_s, covar_x_s, u_s,
                                    covar_u_s, delta_u_s, Q, R, S)])
            self.__lf_func = ca.Function('lf', [mean_s, covar_x_s, P_s],
                                       [self.__cost_lf(mean_s, mean_ref_s, covar_x_s, P_s)])
        elif costFunc is 'sat':
            self.__l_func = ca.Function('l', [mean_s, covar_x_s, u_s,
                                                covar_u_s, delta_u_s],
                                   [self.__cost_saturation_l(mean_s, mean_ref_s,
                                        covar_x_s, u_s, covar_u_s, delta_u_s, Q, R, S)])
            self.__lf_func = ca.Function('lf', [mean_s, covar_x_s, P_s],
                                       [self.__cost_saturation_lf(mean_s,
                                            mean_ref_s, covar_x_s,  P_s)])
        else:
             raise NameError('No cost function called: ' + costFunc)


    def __cost_lf(self, x, x_ref, covar_x, P, s=1):
        """ Terminal cost function: Expected Value of Quadratic Cost
        """
        P_s = ca.SX.sym('Q', ca.MX.size(P))
        x_s = ca.SX.sym('x', ca.MX.size(x))
        covar_x_s = ca.SX.sym('covar_x', ca.MX.size(covar_x))

        sqnorm_x = ca.Function('sqnorm_x', [x_s, P_s],
                               [ca.mtimes(x_s.T, ca.mtimes(P_s, x_s))])
        trace_x = ca.Function('trace_x', [P_s, covar_x_s],
                               [s * ca.trace(ca.mtimes(P_s, covar_x_s))])
        return sqnorm_x(x - x_ref, P) + trace_x(P, covar_x)


    def __cost_saturation_lf(self, x, x_ref, covar_x, P):
        """ Terminal Cost function: Expected Value of Saturating Cost
        """
        Nx = ca.MX.size1(P)

        # Create symbols
        P_s = ca.SX.sym('P', Nx, Nx)
        x_s = ca.SX.sym('x', Nx)
        covar_x_s = ca.SX.sym('covar_z', Nx, Nx)

        Z_x = ca.SX.eye(Nx) + 2 * covar_x_s @ P_s
        cost_x = ca.Function('cost_x', [x_s, P_s, covar_x_s],
                             [1 - ca.exp(-(x_s.T @ ca.solve(Z_x.T, P_s.T).T @ x_s))
                                     / ca.sqrt(ca.det(Z_x))])
        return cost_x(x - x_ref, P, covar_x)


    def __cost_saturation_l(self, x, x_ref, covar_x, u, covar_u, delta_u, Q, R, S):
        """ Stage Cost function: Expected Value of Saturating Cost
        """
        Nx = ca.MX.size1(Q)
        Nu = ca.MX.size1(R)

        # Create symbols
        Q_s = ca.SX.sym('Q', Nx, Nx)
        R_s = ca.SX.sym('Q', Nu, Nu)
        x_s = ca.SX.sym('x', Nx)
        u_s = ca.SX.sym('x', Nu)
        covar_x_s = ca.SX.sym('covar_z', Nx, Nx)
        covar_u_s = ca.SX.sym('covar_u', ca.MX.size(R))

        Z_x = ca.SX.eye(Nx) + 2 * covar_x_s @ Q_s
        Z_u = ca.SX.eye(Nu) + 2 * covar_u_s @ R_s

        cost_x = ca.Function('cost_x', [x_s, Q_s, covar_x_s],
                             [1 - ca.exp(-(x_s.T @ ca.solve(Z_x.T, Q_s.T).T @ x_s))
                                     / ca.sqrt(ca.det(Z_x))])
        cost_u = ca.Function('cost_u', [u_s, R_s, covar_u_s],
                             [1 - ca.exp(-(u_s.T @ ca.solve(Z_u.T, R_s.T).T @ u_s))
                                     / ca.sqrt(ca.det(Z_u))])

        return cost_x(x - x_ref, Q, covar_x)  + cost_u(u, R, covar_u)


    def __cost_l(self, x, x_ref, covar_x, u, covar_u, delta_u, Q, R, S, s=1):
        """ Stage cost function: Expected Value of Quadratic Cost
        """
        Q_s = ca.SX.sym('Q', ca.MX.size(Q))
        R_s = ca.SX.sym('R', ca.MX.size(R))
        x_s = ca.SX.sym('x', ca.MX.size(x))
        u_s = ca.SX.sym('u', ca.MX.size(u))
        covar_x_s = ca.SX.sym('covar_x', ca.MX.size(covar_x))
        covar_u_s = ca.SX.sym('covar_u', ca.MX.size(R))
        sqnorm_x = ca.Function('sqnorm_x', [x_s, Q_s],
                               [ca.mtimes(x_s.T, ca.mtimes(Q_s, x_s))])
        sqnorm_u = ca.Function('sqnorm_u', [u_s, R_s],
                               [ca.mtimes(u_s.T, ca.mtimes(R_s, u_s))])
        trace_u  = ca.Function('trace_u', [R_s, covar_u_s],
                               [s * ca.trace(ca.mtimes(R_s, covar_u_s))])
        trace_x  = ca.Function('trace_x', [Q_s, covar_x_s],
                               [s * ca.trace(ca.mtimes(Q_s, covar_x_s))])

        return sqnorm_x(x - x_ref, Q) + sqnorm_u(u, R) + sqnorm_u(delta_u, S) \
                + trace_x(Q, covar_x)  + trace_u(R, covar_u)


    def __constraint(self, mean, covar, H, quantile, ub, lb, eps):
        """ Build up chance constraint vectors
        """

        r = ca.SX.sym('r')
        mean_s = ca.SX.sym('mean', ca.MX.size(mean))
        S_s = ca.SX.sym('S', ca.MX.size(covar))
        H_s = ca.SX.sym('H', 1, ca.MX.size2(H))
        S = covar
        con_func = ca.Function('con', [mean_s, S_s, H_s, r],
                                [H_s @ mean_s + r * H_s @ ca.diag(S_s)])

        con = []
        con_lb = []
        con_ub = []
        for i in range(ca.MX.size1(mean)):
            con.append(con_func(mean, S, H[i, :], quantile[i]) - eps[i])
            con_ub.append(ub[i])
            con_lb.append(-np.inf)
            con.append(con_func(mean, S, H[i, :], -quantile[i]) + eps[i])
            con_ub.append(np.inf)
            con_lb.append(lb[i])
        cons = dict(con=con, con_lb=con_lb, con_ub=con_ub)
        return cons


    def __debug(self, t):
        """ Print debug messages during each solve iteration
        """

        print('_______________  Debug  ________________')
        print('* Mean_%d:' %t)
        print(self.__mean[t])
        print('* u_%d:' % t)
        print(self.__u[t])
        print('* covar_%d:' % t)
        print(self.__covariance[t, :])
        print('----------------------------------------')


    def plot(self, title=None,
             xnames=None, unames=None, time_unit = 's', numcols=2):
        """ Plot MPC
        # Optional Arguments:
            title: Text displayed in figure window, defaults to MPC setting.
            xnames: List with labels for the states, defaults to 'State i'.
            unames: List with labels for the inputs, default to 'Control i'.
            time_unit: Label for the time axis, default to seconds.
            numcols: Number of columns in the figure.

        # Return:
            fig_x: Figure with states
            fig_u: Figure with control inputs
        """

        if self.__mean is None:
            print('Please solve the MPC before plotting')
            return

        x = self.__mean
        u = self.__u
        dt = self.__dt
        Nu = self.__Nu
        Nt_sim, Nx = x.shape

        # First prediction horizon
        x_pred = self.__mean_prediction
        var_pred = self.__var_prediction

        # One step prediction
        var = np.zeros((Nt_sim, Nx))
        mean = self.__mean_pred
        for t in range(Nt_sim):
            var[t] = np.diag(self.__covariance[t])


        x_sp = self.__x_sp * np.ones((Nt_sim, Nx))

        if x_pred is not None:
            Nt_horizon = np.size(x_pred, 0)
            t_horizon = np.linspace(0.0, Nt_horizon * dt -dt, Nt_horizon)
        if xnames is None:
            xnames = ['State %d' % (i + 1) for i in range(Nx)]
        if unames is None:
            unames = ['Control %d' % (i + 1) for i in range(Nu)]

        t = np.linspace(0.0, Nt_sim * dt -dt, Nt_sim)
        u = np.vstack((u, u[-1, :]))
        numcols = 2
        numrows = int(np.ceil(Nx / numcols))

        fig_u = plt.figure(figsize=(9.0, 6.0))
        for i in range(Nu):
            ax = fig_u.add_subplot(Nu, 1, i + 1)
            ax.step(t, u[:, i] , 'k', where='post')
            ax.set_ylabel(unames[i])
            ax.set_xlabel('Time [' + time_unit + ']')
        fig_u.canvas.set_window_title('Control inputs')
        plt.tight_layout()
```

```
        fig_x = plt.figure(figsize=(9, 6.0))
        for i in range(Nx):
            ax = fig_x.add_subplot(numrows, numcols, i + 1)
            ax.plot(t, x[:, i], 'k-', marker='.', linewidth=1.0, label='Simulation')
            ax.errorbar(t, mean[:, i], yerr=2 * np.sqrt(var[:, i]), marker='.',
                            linestyle='None', color='b', label='One step prediction')
            if x_sp is not None:
                ax.plot(t, x_sp[:, i], color='g', linestyle='--', label='Setpoint')
            if x_pred is not None:
                ax.errorbar(t_horizon, x_pred[:, i], yerr=2 * np.sqrt(var_pred[:, i]),
                            linestyle='None', marker='.', color='r',
                            label='1st prediction horizon')
            plt.legend(loc='best')
            ax.set_ylabel(xnames[i])
            ax.set_xlabel('Time [' + time_unit + ']')

        if title is not None:
            fig_x.canvas.set_window_title(title)
        else:
            fig_x.canvas.set_window_title(('MPC Horizon: {x}, Feedback: {y}, '
                                           'Discretization: {z}'
                                           ).format( x=self.__Nt,
                                                     y=self.__feedback,
                                                     z=self.__discrete_method
                                           ))
        plt.tight_layout()
        plt.show()
        return fig_x, fig_u


def lqr(A, B, Q, R):
    """Solve the infinite-horizon, discrete-time LQR controller
        x[k+1] = A x[k] + B u[k]
        u[k] = -K*x[k]
        cost = sum x[k].T*Q*x[k] + u[k].T*R*u[k]

    # Arguments:
        A, B: Linear system matrices
        Q, R: State and input penalty matrices, both positive definite

    # Returns:
        K: LQR gain matrix
        P: Solution to the Riccati equation
        E: Eigenvalues of the closed loop system
    """

    P = np.array(scipy.linalg.solve_discrete_are(A, B, Q, R))
    K = -np.array(scipy.linalg.solve(R + B.T @ P @ B, B.T @ P @ A))
    eigenvalues, eigenvec = scipy.linalg.eig(A + B @ K)

    return K, P, eigenvalues


def plot_eig(A, discrete=True):
    """ Plot eigenvelues

    # Arguments:
        A: System matrix (N x N).

    # Optional Arguments:
        discrete: If true the unit circle is added to the plot.

    # Returns:
        eigenvalues: Eigenvelues of the matrix A.
    """
    eigenvalues, eigenvec = scipy.linalg.eig(A)
    fig,ax = plt.subplots()
    ax.axhline(y=0, color='k', linestyle='--')
    ax.axvline(x=0, color='k', linestyle='--')
    ax.scatter(eigenvalues.real, eigenvalues.imag)
    if discrete:
        ax.add_artist(plt.Circle((0,0), 1, color='g', alpha=.1))
    plt.ylim([min(-1, min(eigenvalues.imag)), max(1, max(eigenvalues.imag))])
    plt.xlim([min(-1, min(eigenvalues.real)), max(1, max(eigenvalues.real))])
    plt.gca().set_aspect('equal', adjustable='box')

    fig.canvas.set_window_title('Eigenvalues of linearized system')
    plt.show()
    return eigenvalues
```