Human Movements Prediction using on-line Gaussian Processes

eingereichte
BACHELORARBEIT
von

Ziyu Wang

geb. am 21.06.1992
wohnhaft in:
Frankfurter Ring, 2b
80807 München
Tel.: 0160 91484492

Lehrstuhl für
STEUERUNGS- und REGELUNGSTECHNIK
Technische Universität München

Univ.-Prof. Dr.-Ing./Univ. Tokio Martin Buss

Betreuer:	M. Sc. Matteo Saveriano, Prof. Dongheui Lee
Beginn:	29.02.2016
Zwischenbericht:	22.03.2016
Abgabe:	28.06.2016

TECHNISCHE UNIVERSITÄT MÜNCHEN
LEHRSTUHL FÜR STEUERUNGS- UND REGELUNGSTECHNIK
UNIV.-PROF. DONGHEUI LEE

01 February 2016

B A C H E L O R   T H E S I S
for
Ziyu Wang
Student ID 03637901, Degree EI

Human Movements Prediction using On-line Gaussian Processes

Problem description:

A promising strategy for the rehabilitation of patients with impaired body balance consists in providing a "light touch" balance support, for example by lightly resting a hand on the back of a patient without taking patient's weight. If a robotic caregiver is asked to provide this light touch, Cartesian impedance control strategies [1] can be adopted to guarantee a soft interaction. Cartesian control approaches require high frequency ($\approx 1ms$) control loops and the current human tracking systems are not able to provide references at the required frequency.

In this Bachelor Thesis work the student has to implement an algorithm, based on on-line Gaussian process regression [2, 3], to predict human's body part movements in real time given their current state. These predictions will be used as references for a Cartesian impedance controller providing robotic light touch.

Tasks:

- Literature overview on on-line Gaussian processes
- Movements prediction algorithm implementation in C++
- Comparison with state-of-the-art approaches (on-line support vector machines)
- Experimental evaluation with a Kuka LWR IV+ robot (optional)

Bibliography:

[1] C. Ott. Cartesian Impedance Control of Redundant and Flexible-Joint Robots, in *Springer Tracts in Advanced Robotics (STAR)*, 2008.
[2] C. E. Rasmussen and C. K. I. Williams. Gaussian processes for machine learning *MIT Press*, 2006.
[3] L. Csató. Gaussian Processes - Iterative Sparse Approximations *Aston University - PhD dissertation*, 2002.

Supervisor:	M. Sc. Matteo Saveriano
Start:	29.02.2016
Intermediate Report:	22.03.2016
Delivery:	28.06.2016

(D. Lee)
Univ.-Professor

Abstract

With drastic growth of the computing power in the recent time, robots can be utilized in many diverse ways, e.g. as a caregiver for patient with impaired balance control. Due to the high frequency required by the Impedance Controller, which cannot be provided by the current tracking system, robot might behave in an unstable manner while interacting with patients. Therefore, we want to develop and implement a model that can predict human movements given the current state. Properties such as high accuracy and flexibility make us decide to use Gaussian Process Regression(GPR) model for this purpose. Different approaches are suggested in this thesis to lower the computing time of GP drastically while still remaining the flexibility and accuracy. First, we investigate the calculation of kernel inversion within the GP. Two kernel inversion update techniques are proposed and benchmarked against each other. Choosing the approach with better results, we integrate it into a better real-time-orientated model, Local Gaussian Process and benchmark it against standard GP as well as another state-of-the-art model for real-time robot control, Support Vector Regression(SVR).

Contents

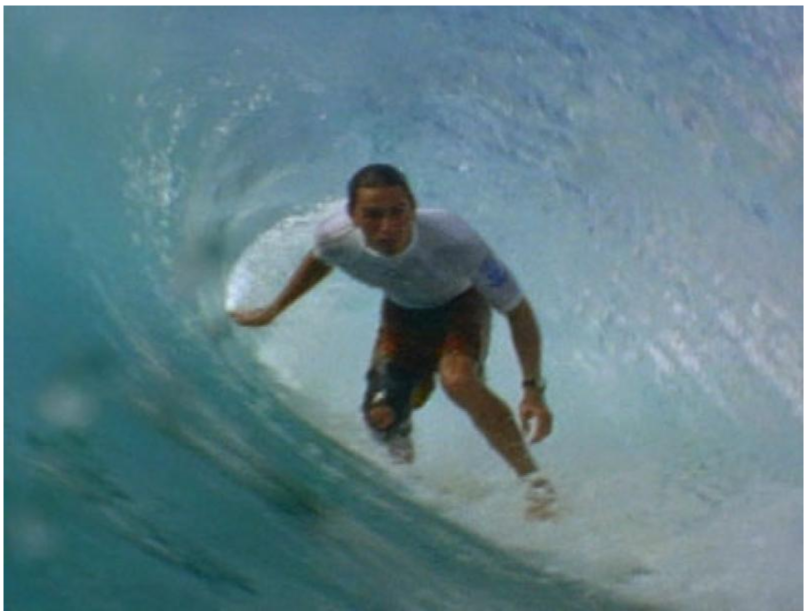
Chapter 1

Introduction

With the drastic increase of computer performance and academic research and development in the field of machine learning in the recent time, the potential area of application for robots is getting wider and more diverse than ever. Not only for industrial purpose as a machine with low failure rate and high efficiency, a robot can also facilitate as a caregiver in normal life.

1.1 Background

Figure 1.1: Surfers use light touch for stability



Source: J. Jeka. Light Touch Contact: Not Just for Surfers. The Neuromorphic Engineer. A Publication of INE- WEB.ORG, 2006.

A strategy "light touch" for rehabilitation of patients with impaired balance has been developed, which intends to facilitate a robot caregiver as an assistant to support the patient's balance control by resting a hand on the patient without taking the patient's weight. It aims to give the patient feedback signals that allow him /her to gain enhanced body balance control. The force provided by the caregiver is small comparing to the force that would be necessary for the actual lifting of patient's body, thus, the name "light touch". The strategy has been proven to be efficient in various studies, e.g. [10]. A Cartesian impedance controller is regarded to be the approach in order to provide the soft and comfortable interaction needed for the strategy.

1.2 Motivation and Challenges

The Cartesian Impedance Controller operates at a very high frequency, around 1 ms. This is essential for the soft and fluid interaction between the robot and the patient. However, the current human tracking systems are not yet able to provide the required frequency, which might lead to unstable behaviour.

The model Gaussian Process (GP) has a lot of characteristics such as the flexibility for being non-parametric and high accuracy, which are attractive qualities as a model for our problem. Thus, the thesis aims to develop and implement an efficient algorithm that predicts humans body movement in real time given their current state using on-line Gaussian Process regression, to "fill the gaps".

An intuitive introduction of GP can be found in the pioneer work in the field by Rasmussen:
"A *Gaussian Process* is a generalization of the Gaussian probability distribution. Whereas a probability distribution describes random variables which are scalars or vectors (for multivariate distributions), a stochastic process governs the properties of functions. Leaving mathematical sophistication aside, one can loosely think of a function as a very long vector, each entry in the vector specifying the function value $f(x)$ at a particular input x. It turns out, that although this idea is a little naive it is surprisingly close to what we need. Indeed, the question of how we deal computationally with these infinite dimensional objects has the most pleasant resolution imaginable: if you ask only for the properties of the function at a finite number of points, then inference in the Gaussian Process will give you the same answer if you ignore the infinitely many other points, as if you would have taken them all into account! And these answers are consistent with answers to any other finite queries you may have. One of the main attractions of the Gaussian Process framework is precisely that it unites a sophisticated and consistent view with computational tractability." [16]

1.3 Related Work

Non-parametric models, such as Gaussian Process do not make or make fewer assumptions on the size or the distribution of the data than parametric ones. While having more accurate and energy-efficient robot control, parametric models do not cope well with unmodeled non-linearities, such as complex friction or actuator dynamics. However, these factors have few impact on non-parametric models. Being less restrictive on the data also offers more flexibility to the model and for these reasons, non-parametric models are often selected as the standard real-time robot control approach.
Other such models are e.g. Locally Weighted Projection Regression(LWPR) and Support Vector Regression(SVR)[19][14].

LWPR is a model suggested by S. Vijayakumar, S. Schaal, and A. D'Souza in 2005. In its core, it is a non-parametric regression model with locally linear clusters. Due to this linear nature, the model has a fantastic scaling of $O(n)$ in data size and has been shown to perform well in domains of high-dimensions[25][26]. The fast learning speed combined with other properties such as low requirement for training data memorization, weighting kernels adjustment based on local information and the ability do to deal with potentially redundant information make this model the standard robot control method when it comes to real-time learning[17]. An introductory explanation of its basic principles are explained in section 2.2.1.

Support Vector Machine(SVM) on the other hand, can be traced back to the *Generalized Portrait Algorithm*, suggested by Vapnik and Lerner in the 1960s which is essentially the linear version of SVM. Since then, SVM has been researched actively for the last several decades. In recent time, SVM has established itself as a standard machine learning method. Although commonly, it is applied on classification problems, the basic idea remains the same for regression: to minimize the error by individualizing a hyperplane that maximizes the margin ϵ, while trying to stay low-cost with the help of skilful selection of the parameters. This is also more elaborated on in section 2.3.

1.4 Contribution

While being a highly performant model for machine learning, one of the major drawbacks of GP lies in its poor scaling regarding the size of the dataset and the computation time. Therefore, while keeping the accuracy and flexibility intact, this thesis aims to lower the computation time by optimizing the model, so that its computation time can also stand toe-to-toe with other state-of-the-arts. More specifically, the In-/Decremental Cholesky and In-/Decremental Inversion techniques for the kernel inversion of GP are investigated, implemented and compared to each other in terms of efficiency and accuracy. The suitable approach will be then integrated into the

Local Gaussian Process implementation which should further lower the computing time. The approach will be benchmarked against standard GP and another state of the art approach, Support Vector Regression. The optimized model is planned to be contributed as reference for the Cartesian impedance controller later on.

1.5 Outline

Main part of the thesis is divided into two chapters:

Chapter two introduces various regression models. This chapter contains the all the theories behind the models and their on-line update techniques, as well as their implementations in C++:

First section of this chapter explains the theoretical principles of our GP optimization approaches, namely, the Incremental Inversion using Block form and the Decremental Inversion using Sherman-Morrison, Incremental Cholesky. In the second section, the theory behind LGP approach is introduced. The third section describes the C++ implementations of the previous sections.

Chapter three can be viewed in three sections:

Comparison results between the implementations of the inversion techniques are presented in the first section.

The different regression models are compared to each other in terms of time and accuracy with 1-D data as input in the second section.

In section three, we test these models with a set of positional data sampled from KUKA robot as input. First the setup and the proceedings are explained, a conclusion is drawn at the end of the chapter.

The last chapter summarizes the work. The limitations of the implemented model are pointed out and possibilities for future work are suggested.

Chapter 2

Approaches for on-line Robot Control

2.1 Gaussian Process

2.1.1 Gaussian Process Regression

A GP can be used as means for regression as following:
Assuming y being the observation made at the coordinate $\mathbf{x}$ and the process is observed n times. Also assuming the prior mean is set to zero-constant as well as the covariance matrix is $\mathbf{K}\left(\Theta, \mathbf{X}, \mathbf{X}'\right)$, Θ being the vector of hyperparameters of the chosen kernel, then we can denote the GP as following:

$$f(\mathbf{X}) \sim \mathcal{GP}\left(0, \mathbf{K}\left(\Theta, \mathbf{X}, \mathbf{X}'\right)\right) \tag{2.1}$$

And the log marginal likelihood is:

$$\log p(\mathbf{f}|\mathbf{X}, \Theta) = -\frac{1}{2}\mathbf{f}^T\mathbf{K}^{-1}\mathbf{f} - \frac{1}{2}\log\det(\mathbf{K}) - \frac{n}{2}\log(2\pi) \tag{2.2}$$

For simplicity and readability, a few terms are shortened as following: $\mathbf{K} = \mathbf{K}(\Theta, \mathbf{X}, \mathbf{X})$, $\mathbf{K}_* = \mathbf{K}(\mathbf{X}, \mathbf{X}_*)$ and $\mathbf{k}_* = k(\mathbf{x}_*)$.
Maximizing the marginal likelihood function with respect to Θ will deliver us the full specification of the GP. The first two parts on the right hand side of the equation can each be thought as the penalty terms for the fitting accuracy and for the model's complexity, respectively. With Θ given, making predictions about testing values $\mathbf{x}^*$ should work as following, given training dataset $\mathbf{X}$, the predictive distribution $p(y^*|\mathbf{x}^*, f(\mathbf{x}), \mathbf{X}) = \mathcal{N}(y^*|\mathbf{m}, \mathbf{var})$, with $\mathbf{m}$ being the posterior mean estimate and $\mathbf{var}$ being the posterior variance estimate, the predictive results m and var for a single test point $\mathbf{x}^*$ can be calculated as following:

$$\mathrm{m}(\mathbf{x}^*) = \mathbf{k}_*^T\mathbf{K}^{-1}\mathbf{y} \tag{2.3}$$

$$\mathrm{var}(\mathbf{x}^*) = k(\mathbf{x}^*, \mathbf{x}^*) - \mathbf{k}_*^T \mathbf{K}^{-1} \mathbf{k}_* \tag{2.4}$$

For GP Regression, the most time consuming operation is the inversion of the kernel matrix $\mathbf{K}^{-1}$, which can take up to $O(n^3)$ flops with Gauss-Jordan. For the time-crucial operations in strategy "light touch", calculating the matrix from scratch every time a new training pair is added/removed, i.e. adding/removing a new row and column in the kernel matrix $\mathbf{K}$, this approach will not suffice. In the following sections, two approaches are introduced, which should lower the computing time significantly.

2.1.2 On-line Inversion

The main idea behind on-line matrix-inversion is to make use of the old matrix ($\mathbf{K}_{t+1} = f(\mathbf{K}_t)$) instead of disposing it entirely and calculating the new matrix from the scratch.

Incremental Inversion An efficient way of calculating the new matrix $\mathbf{K}_{t+1}$ is to use the block form introduced in [6]. If the new entries added to the kernel matrix are $\mathbf{b}$ and c, the incremented kernel matrix would have the following structure:

$$\mathbf{K}_{t+1} = \begin{bmatrix} \mathbf{K}_t & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix} \tag{2.5}$$

The inverse $\mathbf{K}_{t+1}^{-1}$ can be calculated in the following fashion

$$\mathbf{K}_{t+1}^{-1} = \begin{bmatrix} \mathbf{K}_t^{-1} + \frac{1}{k}\mathbf{K}_t^{-1}\mathbf{b}\mathbf{b}^T\mathbf{K}_t^{-1} & -\frac{1}{k}\mathbf{K}_t^{-1}\mathbf{b} \\ -\frac{1}{k}\mathbf{b}^T\mathbf{K}_t^{-1} & \frac{1}{k} \end{bmatrix} \tag{2.6}$$

where $k = c - \mathbf{b}^T\mathbf{K}_t^{-1}\mathbf{b}$. Since the previous inverse matrix $\mathbf{K}_t^{-1}$ is given, the cost for the new inverse consists of matrix-vector multiplications, each of which costs up to $O(2n^2)$ flops.

Decremental Inversion In the decremental case, Sherman-Morrison is the formula that allows us the make use of the old inverse. Let $\mathbf{K}_t$ and $\mathbf{K}_{t+1}$ be matrices before and after the removal of the last entry:

$$\mathbf{K}_t = \begin{bmatrix} \mathbf{K}_{t+1} & \mathbf{b} \\ \mathbf{b}^T & c \end{bmatrix} \tag{2.7}$$

and $\mathbf{K}_t^{-1}$ with following structure:

$$\mathbf{K}_t^{-1} = \begin{bmatrix} \mathbf{X} & [n \times 1] \\ [1 \times n] & [1 \times 1] \end{bmatrix}$$

using Sherman-Morrison formula combined with block form [7] to calculate the new inverse $\mathbf{K}_{t+1}^{-1}$ can be obtained by:

$$\mathbf{K}_{t+1}^{-1} = \mathbf{X} - \frac{\mathbf{Xb}(\mathbf{Xb})^T}{c + \mathbf{b}^T\mathbf{Xb}} \tag{2.8}$$

Like in the incremental case, the operations consuming the most time are still matrix-vector multiplications with $O(2n^2)$ flops of cost.
However, the entry that we intend to remove is usually not the last one. Thus, in order to use this technique, we can simply swap the desired entries that should be removed, with the last ones. This swapping is illustrated as following:

$$\begin{bmatrix} 1 & \alpha & 2 & 3 & \beta \\ \alpha & \alpha & \alpha & \alpha & \alpha \\ 4 & \alpha & 5 & 6 & \beta \\ 7 & \alpha & 8 & 9 & \beta \\ \beta & \alpha & \beta & \beta & \beta \end{bmatrix} \rightarrow \begin{bmatrix} 1 & \beta & 2 & 3 & \alpha \\ \beta & \beta & \beta & \beta & \alpha \\ 4 & \beta & 5 & 6 & \alpha \\ 7 & \beta & 8 & 9 & \alpha \\ \alpha & \alpha & \alpha & \alpha & \alpha \end{bmatrix} \tag{2.9}$$

2.1.3 Cholesky Decomposition

The state-of-the-art technique, Cholesky Decomposition is an efficient way of solving the problem $\mathbf{b} = \mathbf{Ax}$ given $\mathbf{A}$ and $\mathbf{b}$. Instead of calcualting $\mathbf{A}^{-1}$, the idea is to compute a lower triangular matrix $\mathbf{L}$ s. t.

$$\mathbf{A} = \mathbf{LL}^T \tag{2.10}$$

Equation $\mathbf{b} = \mathbf{Ax}$ can then be solved, by solving $\mathbf{Ly} = \mathbf{b}$ using forward substitution and $\mathbf{L}^T\mathbf{x} = \mathbf{y}$ using back substitution.

Incremental Cholesky Similar to the case with incremental inversion in subsection 2.1.2, there is also a way to make use of the old $\mathbf{L}$ matrix of the Cholesky decomposition, instead of calculating the entire decomposition from the scratch. This technique, introduced in [15], is built-in in the library we use.
Let $\mathbf{K}$ be our kernel matrix and $\mathbf{L}$ the lower-triangular matrix of its Cholesky Decomposition. $\mathbf{k}_{new}$ and k_{new} are the new entries added in the new kernel matrix $\mathbf{K}_{new}$:

$$\mathbf{K}_{new} = \begin{bmatrix} \mathbf{K} & \mathbf{k}_{new} \\ \mathbf{k}_{new}^T & k_{new} \end{bmatrix}, \qquad \mathbf{L}_{new} = \begin{bmatrix} \mathbf{L} & 0 \\ \mathbf{l}_{new}^T & l_{new} \end{bmatrix} \tag{2.11}$$

then $\mathbf{l}_{new}$ and l_{new} can be computed by solving:

$$\mathbf{Ll}_{new} = \mathbf{k}_{new}, \qquad l_{new} = \sqrt{k_{new} - ||\mathbf{l}_{new}||^2} \tag{2.12}$$

The term $\mathbf{Ll}_{new} = \mathbf{k}_{new}$ can be effectively solved by using back substitution, which has quadratic cost.

Decremental Cholesky Decremental Cholesky, however, is only viable for the removal of the last entry of the kernel matrix $\mathbf{K}$. In that case, simply removing the last row of $\mathbf{L}$ will suffice. If we wish to remove e.g. the first row of $\mathbf{K}$, the resulting $\mathbf{L}$ cannot be done as on-line decremental inversion case by simply swapping the entry positions, since the rows, specially the first row, influences the rest of the entries. Although one approach is introduced in [20], it is stated that the method is not to be recommended from the author's experience. Therefore, when calculating the inversion of the kernel matrix after removal of a training pair, the Cholesky decomposition is computed again from the scratch.

2.2 Local Gaussian Process

While reducing the complexity from $O(n^3)$ to $O(n^2)$, the scaling is still not quite as satisfying when facing a very large dataset. With Local Gaussian Process(LGP), we aim to limit the complexity to a certain dimension, regardless of the size of the dataset. Combining this with the on-line inversion techniques, the model should achieve a performance that is able to stand toe-to-toe with state-of-the-art, such as Support Vector Regression, which we will explain in the next section.

2.2.1 Locally Weighted Projection Regression

Locally Weighted Projection Regression(LWPR)[25] is often applied on simple robot control tasks. Due the linear nature of this model, it provides one of the best computation-scaling, but often suffers from suboptimal accuracy. The model is summarized as following:
As the name may suggest, LWPR predicts value by evaluating several local linear model with individual weights and approximating the combination of the regression results of these local models.
Let the *weighted mean prediction* be $\hat{y} = \mathbb{E}\{\bar{y}_k|\mathbf{x}\} = \sum_{k=1}^{M} \bar{y}_k p(k|\mathbf{x})$ and $\bar{y}_k$ be the regression result at k-th local linear model. M describes the relevant local models for the regression. Probability of input point $\mathbf{x}$ being in model k is according to *Bayes' rule* as following:

$$p(k|\mathbf{x}) = \frac{p(k,\mathbf{x})}{p(\mathbf{x})} = \frac{p(k,\mathbf{x})}{\sum_{k=1}^{M} p(k,\mathbf{x})} = \frac{w_k}{\sum_{k=1}^{M} w_k} \tag{2.13}$$

The regression of the entire model can then be calculated as

$$\hat{y}(\mathbf{x}) = \frac{\sum_{k=1}^{M} w_k \bar{y}_k(\mathbf{x})}{\sum_{k=1}^{M} w_k} \tag{2.14}$$

A *similarity measure* function should be introduced for the weight w_k, which describes the relevance of the local model k to the input point $\mathbf{x}$.

$$w_k = f(\mathbf{x}, \mathbf{c}_k, \Theta_k) \tag{2.15}$$

with c_k and Θ_k being the *centroid* and *parameter-set* of local model k, respectively. The parameter-set Θ_k should be tuned in the way that error between prediction and observation minimal.

While having fantastic computation scaling of $O(n)$ due to the linear nature of the local models, there are also the following drawbacks to be considered:

The manual parameter-tuning for w_k as well as for $\bar{y}_k$ within each linear model is difficult since they are highly data-dependent. Since the local models are linear, it will also require a fairly large amount of them to achieve results with reasonable accuracy on non-linear dataset.

2.2.2 Local Gaussian Process

Since LWPR and GPR seem to complement each other, we consider the model LGP[13], which is an approach that combines elements from both, while retaining a balanced trade-off between accuracy, complexity and flexibility. The regression process can be divided into three parts: data-clustering, model-learning and prediction.

For data-clustering, we simply divide our training data into K subsets. Note that if $K = 1$, this model is no different than the standard GPR. Each subset can be limited to a certain certain size N_{max}. In our case, we simply divide the dataset in a chronological order, but principally, the clustering function can be any arbitrary function. The parameters needed are for the LGP are therefore: amount of subsets K, maximum size of the subset N_{max} and the clustering function. Since the poor scaling of ($O(n^3)$, n is the size of the entire trainingset), which is the major drawback of GP is, like previously mentioned, due to the inversion of the kernel matrix, splitting the data into smaller subsets leads to inversion of a much smaller kernel matrix and thus, reducing the computing time drastically, both for adding data and calculating regression. By limiting the subsets to a fixed size, the total complexity is $O(MN^3)$, where M is number of relevant models for the prediction and N is the size of the subsets. In most cases, $MN^3 \ll n^3$.
The model-learning, i.e. adding data to the model and adjust the kernel inverse, can be done with methods introduced in previous sections 2.1.2 and 2.1.3, which reduce the complexity dimension of each local model from $O(n^3)$ to $O(n^2)$. A commonly used function for weight calculation is the *Radial Basis Function*:

$$w_k(\mathbf{x}) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{c}_k\|^2}{2l^2}\right) \tag{2.16}$$

l defines the *characteristic lengthscale* and $\mathbf{c}_k$ the *centroid* of k-th local model.

Assuming points belonging to the same region are more informative for the prediction, this function is our function of choice to determine the relevance w_k of a certain cluster is to a query point. When adding a new point $\mathbf{x}$, the cluster i with the highest relevance w_i will be chosen. If the cluster size exceeds a certain limit, an old point from the dataset will be removed in order to keep the size of the subset. A weight threshold w_{th} is introduced, so that if none of the cluster has a sufficiently high weight, a new cluster will be created with with $\mathbf{x}$ as its centroid. These two steps are can be summarized in Algorithm 1.

For prediction, we can apply the formulas (2.3) from GPR for local regression:

$$\bar{y}_k(\mathbf{x}^*) = \mathbf{k}_*^T \mathbf{K}^{-1} \mathbf{y} \tag{2.17}$$

and eq.(2.14) from LWPR for final regression:

$$\hat{y}(\mathbf{x}) = \frac{\sum_{k=1}^{M} w_k \bar{y}_k(\mathbf{x})}{\sum_{k=1}^{M} w_k} \tag{2.18}$$

where M is number of relevant models for the prediction. This is summarized in Algorithm 2.

Algorithm 1 Partitioning the data and updating the kernel

```
Require: New observation data $\{\mathbf{x}_{new}, y_{new}\}$
  for $k = 1$ to $M$ = amount of all clusters do
    Calculate the weights of these clusters
    $w_k = f(\mathbf{x}_{new}, \mathbf{c}_k, \Theta_k)$
  end for
  Choose the cluster with the highest weight
  $w_{max} = \max(w_1, ..., w_M)$
  if $w_{max} > w_{th}$ then
    Add the data pair to the cluster with the highest weight
    $\mathbf{X}_{new} = [\mathbf{X}, \mathbf{x}_{new}],\ \mathbf{y}_{new} = [\mathbf{y}, y_{new}]$
    Update the centroid of the cluster
    $\mathbf{c}_{new} = \mathrm{mean}(\mathbf{X}_{new})$
    Update kernel inverse with techniques introduced in section 2.1.2 and section
    2.1.3
  else
    Build new cluster with $\mathbf{x}_{new}$ as centroid
    $\mathbf{c}_{M+1} = \mathbf{x}_{new},\ \mathbf{X}_{M+1} = [\mathbf{x}_{new}],\ \mathbf{y}_{M+1} = [y_{new}]$
  end if
```

Algorithm 2 Prediction using LGP

```
Require: test point $\mathbf{x}^*$, amount of clusters $M$
  for $k = 1$ to $M$ do
    Calculate the weights of the clusters
    $w_k = f(\mathbf{x}^*, \mathbf{c}_k, \Theta_k)$
    Calculate the mean $\bar{y}_k$ using parameters from the local Gaussian Process with
    eq.(2.3) and eq.(2.17)
    $\bar{y}_k = m_k(\mathbf{x}^*) = \mathbf{k}_*^T \mathbf{K}^{-1} \mathbf{y}$
  end for
  Calculate the weighted mean prediction $\hat{y}$ from the local clusters with eq.(2.18)
  $\hat{y}(\mathbf{x}^*) = \frac{\sum_{k=1}^{M} w_k \bar{y}_k(\mathbf{x}^*)}{\sum_{k=1}^{M} w_k}$
```

2.3 Support Vector Regression

Support Vector Machine(SVM) is one of the state-of-the-arts when it comes real-time robot control. Being mainly used for classification purposes, it is also applicable on regression problems.
Similar as in the classification approach, the regression is done by defining a *loss function* that ignores errors within a certain margin. This type of function is also called *ϵ-insensitive loss function.* The points within the ϵ-band are called *Support Vectors.*

$$L(y, f(\mathbf{x}, w)) = \begin{cases} 0, & \text{if } |y - f(\mathbf{x}, w)| < \epsilon \\ |y - f(\mathbf{x}, w)| - \epsilon, & \text{otherwise} \end{cases} \tag{2.19}$$

where $f(\mathbf{x}, w)$ is a linear model function

$$f(\mathbf{x}, w) = \sum_{j=1}^{m} w_j g_j(\mathbf{x}) + b \tag{2.20}$$

with $g_j(\mathbf{x})$ as a set of transformation functions that are aimed to map input $\mathbf{x}$ to a m-dimensional feature space. b is the bias term, which can be ignored when the data is preprocessed to be zero-mean.

The function for empirical risk is then:

$$R = \frac{1}{N} \sum_{i=1}^{N} L(y_i, f(\mathbf{x}_i, w)) \tag{2.21}$$

The linear regression and complexity reduction is done by minimizing the weight term. The Quadratic Minimization Problem can be formulated as:

$$\min \frac{1}{2} \|w\|^2$$

In order to make the model also feasible for non-linear data, so-called (non-negative) slack variables ξ_i and ξ_i^* have to be introduced as error tolerances outside the ϵ-band in the positive and negative, respectively. The updated minimization problem functional can be formulated as a Lagrangian with constraints as following:

$$\text{minimize} \qquad \frac{1}{2}\|w\|^2 + C \sum_{i=1}^{N} (\xi_i + \xi_i^*) \qquad \text{s.t.} \begin{cases} y_i - f(\mathbf{x}, w) \leq \epsilon + \xi_i^*, \\ f(\mathbf{x}, w) - y_i \leq \epsilon + \xi_i, \\ \xi_i, \xi_i^* \geq 0 \end{cases} \tag{2.22}$$

The solution to it can be formulated as

$$f(\mathbf{x}) = \sum_{i=1}^{n_{SV}} (a_i - a_i^*) K(\mathbf{x}_i, \mathbf{x}) \quad \text{s.t. } 0 \leq a_i^* \leq C \text{ and } 0 \leq a_i \leq C \tag{2.23}$$

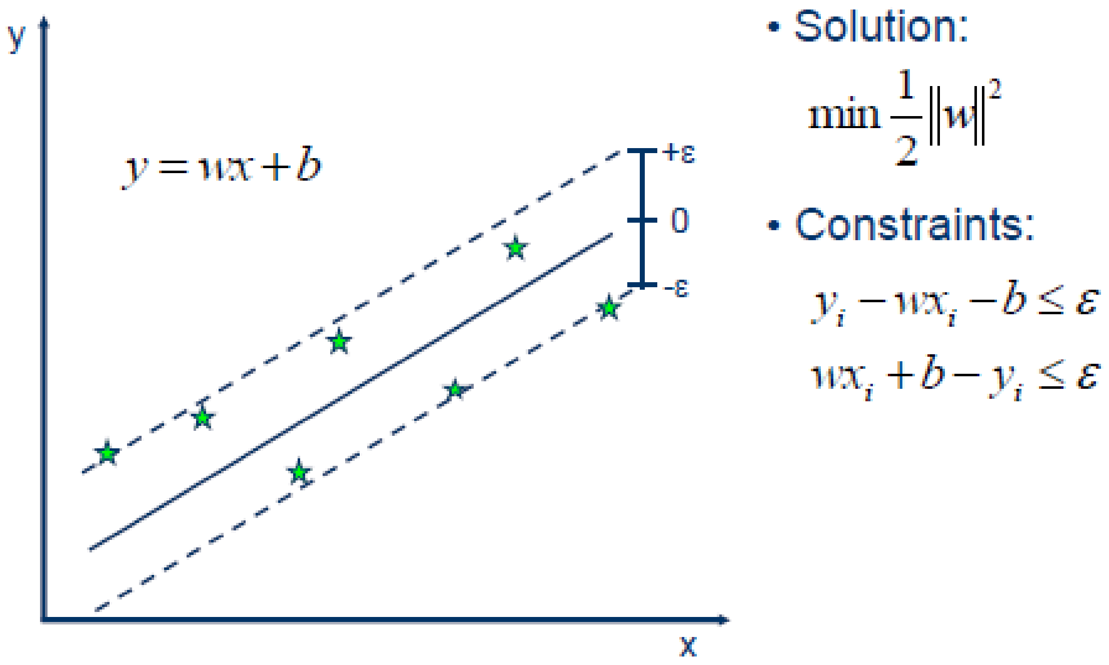
n_{SV} is the number of support vectors and $K(x_i, x)$ is the *kernel* for the SVM, which transforms the data into higher dimensional space:

$$K(\mathbf{x}_i, \mathbf{x}) = \sum_{i=1}^{m} g_j(\mathbf{x}_i) g_j(\mathbf{x})$$

One of the disadvantages of SVR is that the right parameters ϵ, C and *kernel parameters* are crucial to achieve competitive results while remaining low-cost. Unlike the hyperparameters in GP, these need to be tuned manually.

All of these are illustrated in Figure 2.1 and Figure 2.2.

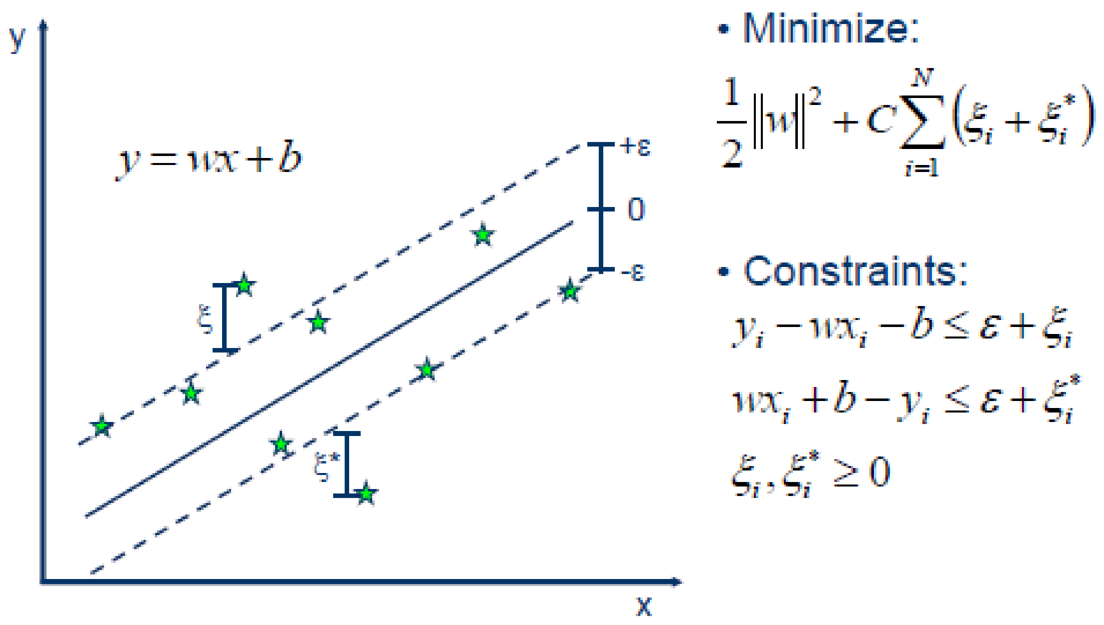
Figure 2.1: Illustration of the basic principles of SVR


y
x
$y = wx + b$
$+\varepsilon$
0
$-\varepsilon$
• Solution:
$\min \frac{1}{2}\|w\|^2$
• Constraints:
$y_i - wx_i - b \le \varepsilon$
$wx_i + b - y_i \le \varepsilon$

Source: `http://www.saedsayad.com/images/SVR_1.png`

The regression and the complexity reduction is done by minimizing the term $\frac{1}{2}\|w^2\|$.

Figure 2.2: Illustration of SVR with slack variables


y
x
$y = wx + b$
$+\varepsilon$
0
$-\varepsilon$
ξ
ξ^*
• Minimize:
$\frac{1}{2}\|w\|^2 + C\sum_{i=1}^{N}(\xi_i + \xi_i^*)$
• Constraints:
$y_i - wx_i - b \le \varepsilon + \xi_i$
$wx_i + b - y_i \le \varepsilon + \xi_i^*$
$\xi_i, \xi_i^* \ge 0$

Source: `http://www.saedsayad.com/images/SVR_2.png`

Deviations outside the ϵ-band in the positive and negative can be denoted as ξ and ξ^*, which are so-called slack variables. The updated minimization problem functional for non-linear regresssion then becomes eq.(2.22)

2.4 Implementations in C++

2.4.1 GP and LGP Implementation

The C++ library `libgp`[1] by Manuel Blum is used in the implementations of the thesis. It provides the basic functionalities for the purpose of the our subject, such as the basic structure of GP, two hyperparameter optimization routines, a set of covariance functions and a built-in function for on-line usage.

`libgp` Introduction `libgp`[1] is a light-weight library for basic GP regression usage. Despite not providing the range of features that e.g. MatLab-toolbox "gpml" by Rasmussen[1] does, it still offers the functionalities that this thesis needs, while also possessing some on-line properties. Its dependencies are `cmake`[2] , an open-source and cross-platform build system and `Eigen3`[3], a state-of-the-art template library for linear algebra, which not only supports dynamic-sized matrix operations, but also speeds them up especially for large matrices. In this library, the prior mean is fixed to zero-constant.
The library has the following structure:
A class with prefix "`cov_`" in the name indicates that it is an implementation of one of the various covariance functions (with "`cov_factory`" being the generator of the functions and "`cov`" being the base class).
Class `sampleset` stores the training dataset. It can add, remove new pairs to the set and read data at a specific index.
Class `gp` is the class that we utilize in `main` to perform GP regression.
Classes `gp_utils` and `input_dim_filter` provide various utilities for the `gp`-class.
Classes `cg` and `rprop` are implementations of two algorithms for the hyperparamter-optimzation, conjugated gradient and resilient back-propagation[2]. In our thesis, the hyperparameters are optimized using the latter.

Built-in On-line Function `libgp` uses the efficient Cholesky decomposition instead of regular Gauss-Jordan matrix-inversion to solve the kernel inverse. The on-line incremental update of this Cholesky decomposition is implemented in the function `add_pattern()`: while adding a new pair of training data into `sampleset`, it also performs incremental Cholesky to update the kernel matrix inverse.

[1]`http://www.gaussianprocess.org/gpml/code/matlab/doc/index.html`
[2]`https://cmake.org/`
[3]`http://eigen.tuxfamily.org/`

My Modifications

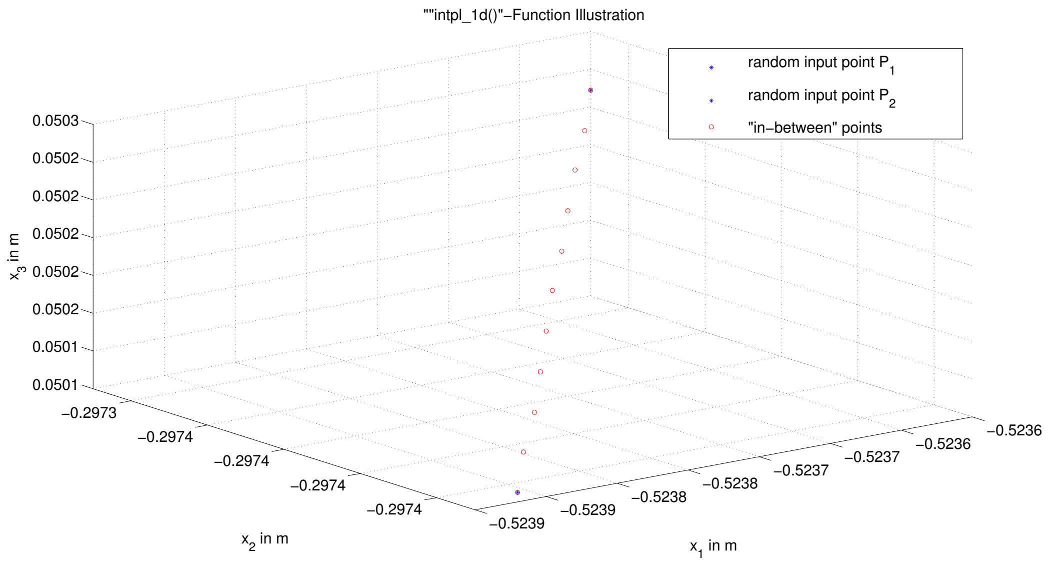
Function for Training Data Removal Matching the functions `add_pattern()` in `gp` and `add()` in `sampleset` for adding new training data, the functions `remove_pattern()` and `remove()` delete an entry with given index from the training data while also updating the kernel matrix.

Functions using In-/Decremental Inversion Functions with prefix "`my_`" (together with the attribute `Kinv` and the function `computeKinv()`) are a set of implementations that aims to use on-line inversion techniques to update the kernel matrix.
Most of the "`my_`"-functions are the system-equivalents to their counterparts, functions without "`my_`" prefix. (the attribute `Kinv` of `gp`-class has similar function as `L` in the built-in Cholesky calculation routine). `computeKinv()` uses Eigen(which uses LU-decomposition) to compute the inverse of the kernel matrix. However, some functions are created from the scratch to perform the algorithms in-/decremental inversion:
`my_IncrInv()` returns the incremental inversion matrix $\mathbf{K}_{t+1}^{-1}$, given the input $\mathbf{K}_t^{-1}, \mathbf{b}, c$ from eq.(2.5) and size of the kernel entry n.
Similarily, `my_DecrInv()` returns the decremented kernel matrix inverse $\mathbf{K}_{t+1}^{-1}$, given the input $\mathbf{K}_t^{-1}, \mathbf{b}, c$ from eq.(2.7) and size of the kernel entry n.
`my_swap()` is a function that swaps the entry to be removed with the last one, as illustrated in eq.(2.9).

Functions for LGP-Framework A special class "`lgp`" is created for the implementation of the Local Gaussian Process. This class functions as a framework that creates multiple GPs from a file containing training dataset. User can define parameters such as the amount of subsets for initial partitioning, the maximum size of a subset and the weight threshold. The functions `add_data()` and `regression()` from class "`lgp`" as well as the functions `updateCenter()`, `getCenter()` and `getWeight()` from class `gp` are the implementations of the theories introduced in section 2.2.2.

Functions for Movement Prediction based on Velocity Given direction of the velocity, coordinates of a starting point in [m] and the time interval in [ms], the function `predict_1d()` of class `gp` uses GP regression to predict the position in the given direction and returns this value.
Unlike the functions in the previous paragraphs, this function is created specifically for the sake of velocity calculation and point prediction, i.e. when using this function, it is assumed that the given input-output training pairs represent coordinates of the point and their current velocity in an arbitrary direction.
`intpl_1d()` is a standalone-function in `main`-file. It takes one coordinate from the start- and end-point of the same direction, the time intervall T and sampling time step t and interpolate between these two coordinates. The output is a vector listing $n = \frac{T}{t} + 2$ coordinates between the two input coordinates, with first entry being the starting and last one being the end coordinate of the input points. Its function is illustrated in the Figure 2.3.

Figure 2.3: `intpl_1d()`-function illustration.

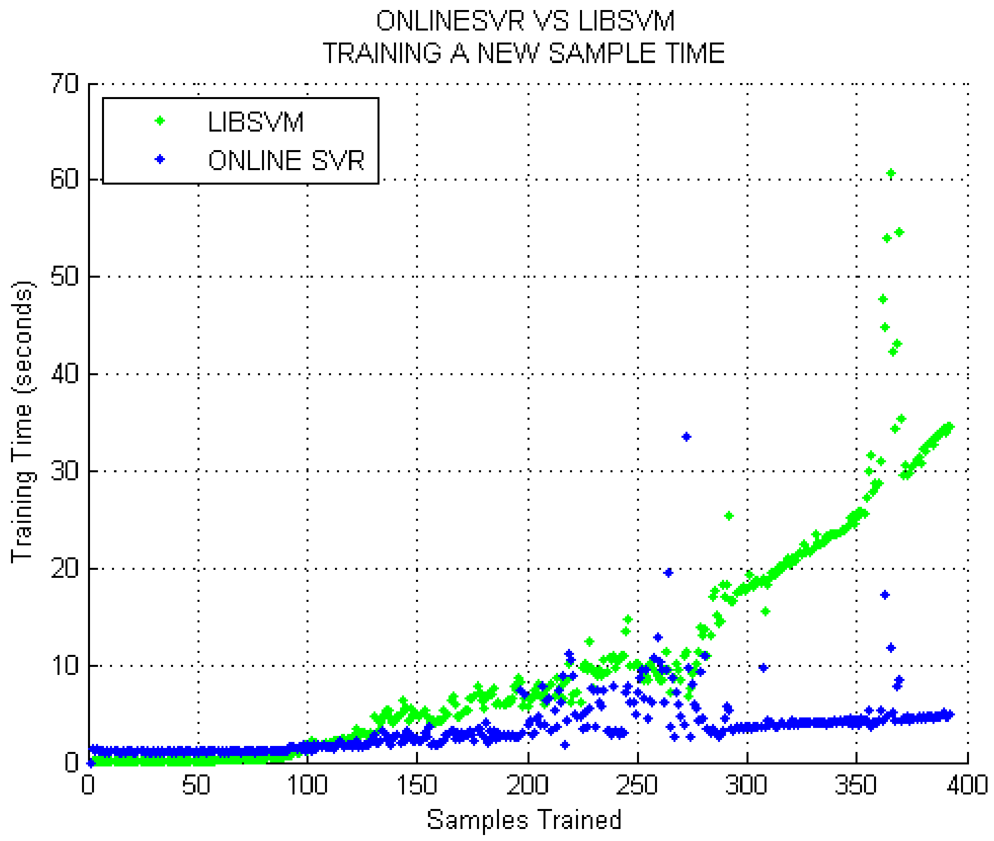


For this figure, the `intpl_1d()`-function is called three times for positions in every direction. Inputs are the two blue stars, which are randomly created points. The red circles illustrate the output of the function, the "in-between"-points. Parameters for the function are $T = 50$ ms and $t = 5$ ms.

2.4.2 SVR Implementation

`onlineSVR`[4] is the implementation of choice to benchmark our code against, because it is, similar to `libgp`, light-weighted, yet still provides the necessary regression functionality we need. On top of efficiently implementing the SVR as explained in section 2.3, it also researches the technique to add/remove new samples without having to train the SVM entirely from the scratch, which is principally similar to what we did with GP in the section 2.1.2 and section 2.1.3. Comparing to more popular and extensive C++ SVM-libraries, such as `libsvm`[5], `onlinSVR` is able to achieve lower computing time when executing on-line tasks.[14]. This is shown in Figure 2.4.

Figure 2.4: OnlineSVR and LibSVM compared in an online training.[14]



[4]`http://onlinesvr.altervista.org/`

[5]`https://www.csie.ntu.edu.tw/~cjlin/libsvm/`

Chapter 3

Experimental Results

In this chapter, the computing time and accuracy between the two kernel inversion techniques are benchmarked against each other. We then integrate the less time-consuming approach into the LGP-framework and test it with a set of 3-D robot positional data and benchmark the standard online GP, LGP and state-of-the-art, SVR using implementation introduced in section 2.4.2 to each other.

RBF-kernel from eq.(2.16) is selected as the covariance function for GP, LGP and the kernel for SVR.

$$k(\mathbf{x}, \mathbf{x}^*) = \exp\left(-\frac{\|\mathbf{x} - \mathbf{x}^*\|^2}{2l^2}\right)$$

The prior mean of GP and LGP is fixed to be zero-constant.

All of following tests in this chapter are done on a PC running 64bit-Ubuntu 12.04LTS (Precise Pangolin) and equipped with an Intel Core i5-2500 CPU @3.30GHz×4.

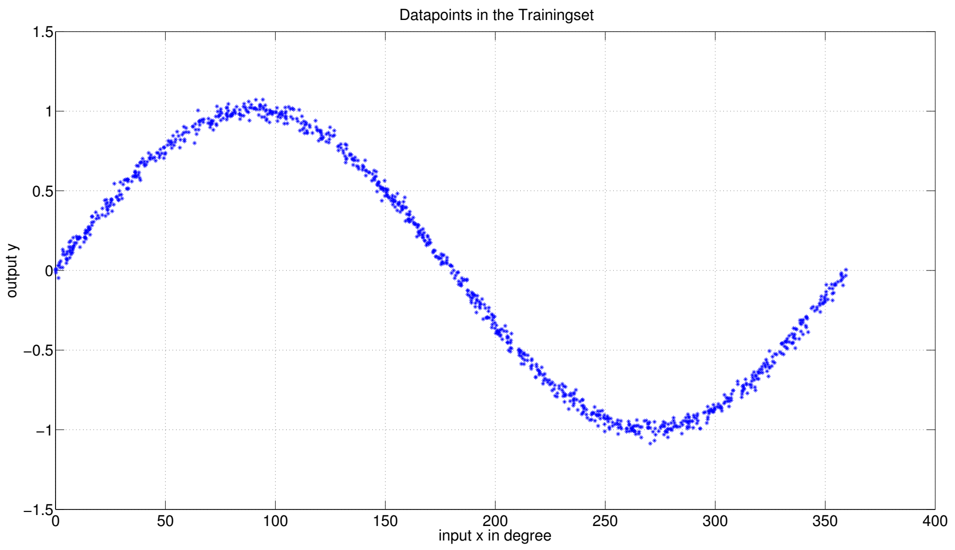
3.1 Benchmark between different Inversion Techniques

For visibility, the training input x in this chapter are 1-D, pseudo-random values from $0 \leq x \leq 360$ generated using `drand48()*360` and a random seed function. The output is a sine-function with additive Gaussian noise. Figure 3.1 shows the distribution of the training datapoints.

3.1.1 Benchmark of Training Data Manipulation Functions

The computing time between functions `add_pattern()`, which uses incremental Cholesky and `my_add_pattern()`, which uses block form in eq.(2.6) to update the kernel matrix inversion, is logged. Several scenarios are tested: having a sample set of $n = 100$ and $n = 1000$ training pairs, the time for addition/removal of an extra pair is logged. This would measure the time that both inversion techniques need for a matrix inversion of dimension 100×100 and 1000×1000, respectively.

Figure 3.1: Distribution of the Sample Data



$n = 1000$ datapoints are sampled from the basic $y = \sin(x) + \sigma$ function, with σ being a 0.03 factorized additive Gaussian noise.

Training Data Addition Functions benchmarked With a pre-existing kernel matrix, the computing time for adding one training pair are to be seen at Table 3.1.

Table 3.1: Computing time for one training pair addition

	`add_pattern()`	`my_add_pattern()`
$100 \rightarrow 101$	0.055ms	0.143ms
$1000 \rightarrow 1001$	6.873ms	38.399ms

Time needed for adding $n = 100$ and $n = 1000$ pairs to the training set is to be seein at Table 3.2

Table 3.2: Total time needed for adding $n = 100$ and $n = 1000$ training pairs

	with `add_pattern()`	with `my_add_pattern()`
$n = 100$	0.089ms	4.721ms
$n = 1000$	146.444ms	7204.78ms

The significant longer computing time needed for `my_add_pattern()` causes us to wonder, so we dig further into the function and only time the essential block responsible for the mathematics introduced in eq.(2.6) and in paragraph 2.1.3, to leave out the minor incompatibility possiblity that our code might have with `libgp` structure. The result is given in Table 3.3.

Table 3.3: Computing time for calculating inversion

	Incremental Cholesky	Incremental Inversion
$100 \rightarrow 101$	0.076ms	0.13ms
$1000 \rightarrow 1001$	0.09ms	2.64ms

Training Data Removal Functions benchmarked With a pre-existing 1000×1000 kernel matrix, the computing time for removing one training pair are to be seen at Table 3.4.

Table 3.4: Computing time for one training pair removal

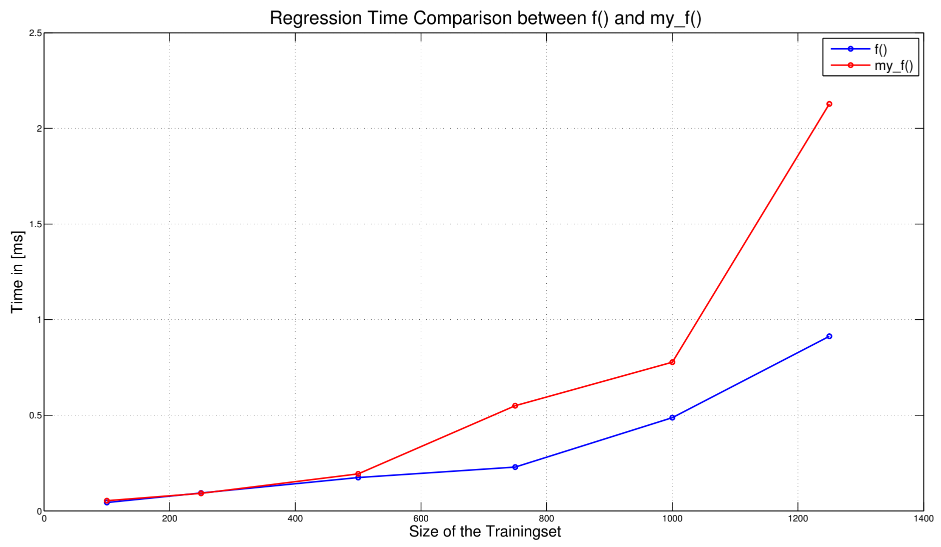
	`remove_pattern()`	`my_remove_pattern()`
remove last pair	0.059ms	24.403ms
remove first pair	108.757ms	44.292ms
remove random pair (at index=250)	112.414ms	38.899ms

The decremental inversion function takes less time in every category except when the training pair to be removed is the last pair. This is due to the fact, like explained in paragraph 2.1.3, that in this case, the new Cholesky matrix can simply be generated by removing the last row.

3.1.2 Comparison between Regression Functions

The functions to be benchmarked in this section are regression functions for prediction purpose. So instead of focusing purely on computing time, the Root-Mean-Square Error(RMSE) is also logged.
As mentioned in the "problem statement" section 1.2, the controller needs to operate at around 1 ms. We observe the time needed for one prediction made for matrix of different sizes, to determine what is a "safe" size for the matrix so that the tiem for prediction can stay under 1 ms. The regression time for `f()` and `f_()` can be seen in Figure 3.2 and the RMSE in Table 3.5.

Figure 3.2: Regression Time Comparison `f()` and `my_f()`



For trainingset of size $n = [100, 250, 500, 750, 1000, 1250]$, the time needed for predicting a new query point is logged in [ms].

Table 3.5: RMSE for `f()` and `my_f()`

	`f()`	`my_f()`
$n = 100$	1.53×10^{-3}	1.53×10^{-3}
$n = 250$	3.94×10^{-4}	3.94×10^{-4}
$n = 500$	2.32×10^{-4}	2.32×10^{-4}
$n = 750$	7.38×10^{-5}	7.38×10^{-5}
$n = 1000$	1.00×10^{-4}	1.00×10^{-4}
$n = 1250$	1.22×10^{-4}	1.22×10^{-4}

One would soon recognize that the RMSE for `f()` and `my_f()`, which are tested and logged separately from each other, are exactly the same, proving the correctness of the results obtained by the implemented code. In terms of computing time, built-in routine still beats our modification slightly, being able to handle $n = 1250$ training pairs while still staying under 1 ms.

Conclusion With incremental Cholesky outperforming incremental inversion considerably, we chose the Cholesky routine to build the LGP framework on and benchmark it with the standard GP as well as SVR [14].

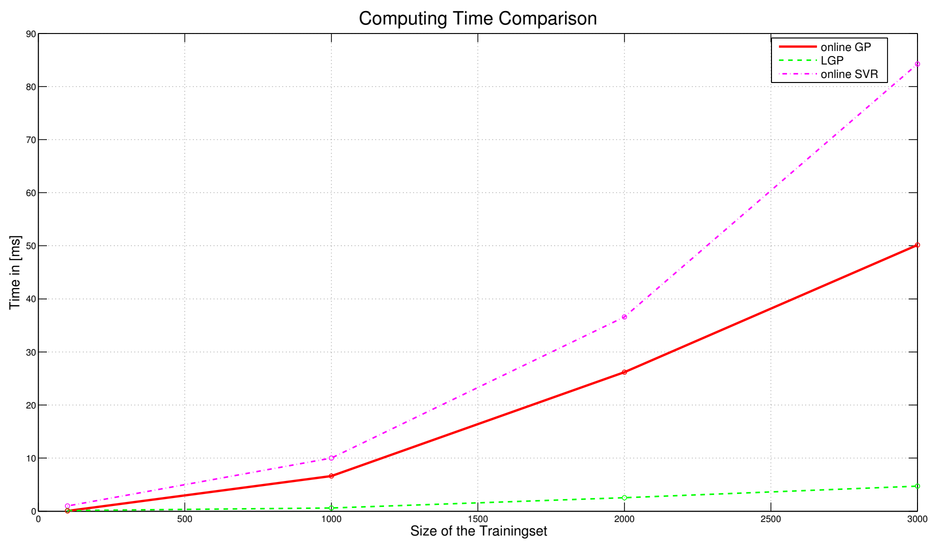
3.2 Benchmark between different Models

This section benchmarks the LGP approach introduced in section 2.2 to standard on-line GP in section 2.1 and incremental SVR, section 2.3.
For LGP, the trainingset is partitioned into 3 subsets. RBF function eq.(2.16) is selected as the weighting function for the local clusters. The weight threshold w_{th} is set to 1×10^{-6}.
For SVR, we also use the RBF function as the kernel. Parameters C, ϵ and the *kernel parameter* l are tuned after experimenting so that the RMSE is held relatively low.

3.2.1 Computing Time Comparison for Data Addition

For computing time comparison, we still used the 1-D sine function from Figure 3.1, and vary the number of sample points between 100, 1000, 2000, 3000. The computing time in [ms] is logged and plotted, the result is shown in Figure 3.3. It shows that especially when dealing with very large amount of data, LGP outperforms GP and SVR noticeably.

Figure 3.3: Computing Time Comparison between GP, LGP and SVR



For trainingset of size $n = [100, 1000, 2000, 3000]$, the time needed for learning a new sample point is logged.

3.2.2 Accuracy Comparison on Trajectory Data

After choosing the routine we think is less time-consuming, we want to test the accuracy of these models. In this chapter, one of the 3D-trajectory data collected from a KUKA robot is used as training input.

The Setup and Proceedings

In the following test, the 3D positional data of an arbitrarily chosen trajectory T99 is used as input. It is a $T = 20$ s long motion sampled with $t_s = 5$ ms , of which we take the first 15 s from, due to noisy and off-target data in the last 5 s. The values of the 3D-velocity in x_1-, x_2- and x_3-direction are used as corresponding outputs for three GPs/LGPs/SVMs. To gain these values, we simply do

$$v = \frac{\Delta x}{0.001 t_s}$$

with Δx being the difference between two consecutive points and t_s the sampling time. The physical unit of the positional data is given in [m] and sampling time is given in [ms], the resulting velocity in [m/s].
After the three models are fully trained and parameters / hyperparameters optimized [1], we predict the velocities in all three coordinates at all points. We decide to compare the results for two sampling time step variations: $t_s = 15$ ms and $t_s = 100$ ms.

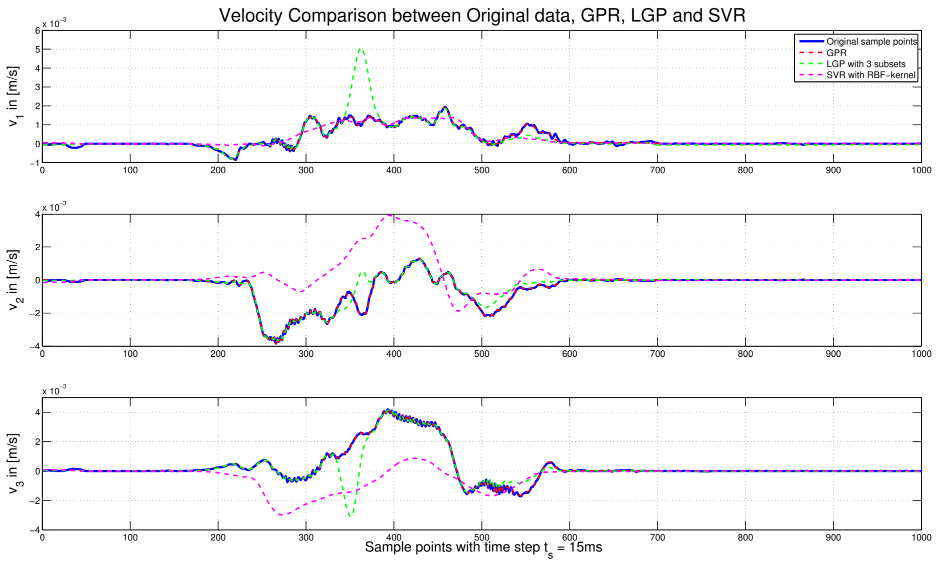
Velocity and Trajectory Comparisons

For sampling time $t_s = 15$ ms and $n = 1000$ datapoints Figure 3.4a, 3.4b and 3.6 show the velocity, 1-D and 3-D trajectory comparison between original data and reconstructed data from different models. Table 3.6 shows the RMSE of each model in x_1-, x_2- and x_3-direction and is plotted in Figure 3.5.

[1] 100 `rprop` iterations for each GP are performed. An iterations does not provide any significant change to the hyperparameters anymore. For SVM, the parameters are tuned manually.

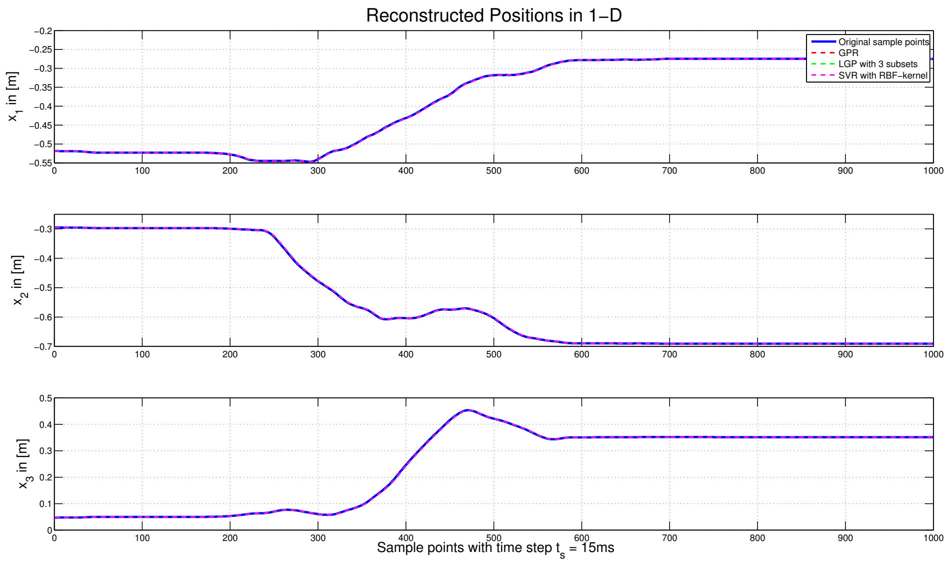
Figure 3.4: 1-D Camparison for $t_s = 15$ ms

(a) Velocity Comparison



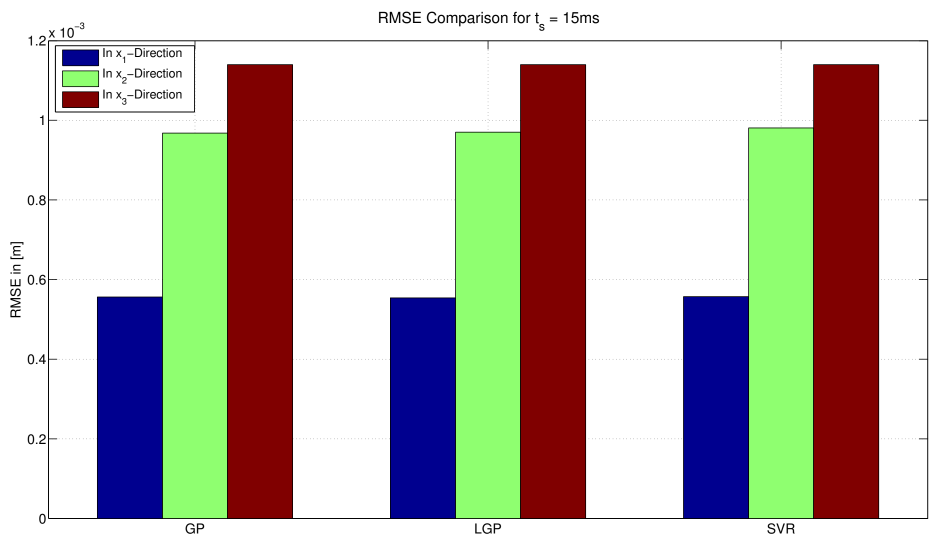
The spikes in green dashed line(LGP) around sample point #350 is due to fact that the centroids of all three local models are too far away to be informative about value of the output. The off-target of the magenta line(SVR) values in v_2 and v_3 are due to the fact that the weights for the SVM exceeds machine epsilon, therefore the training data have to be scaled up to perform proper training and regression, which leads to instability of in the bias term.

(b) Position Comparison



The data are collected with a time step $t_s = 15$ ms and the off-target values from the velocity Comparison are only off by max. 4×10^{-3}[m/s], meaning the effect on the reconstructed the points will be barely noticeable here.

Figure 3.5: RMSE plot for $t_s = 15$ ms

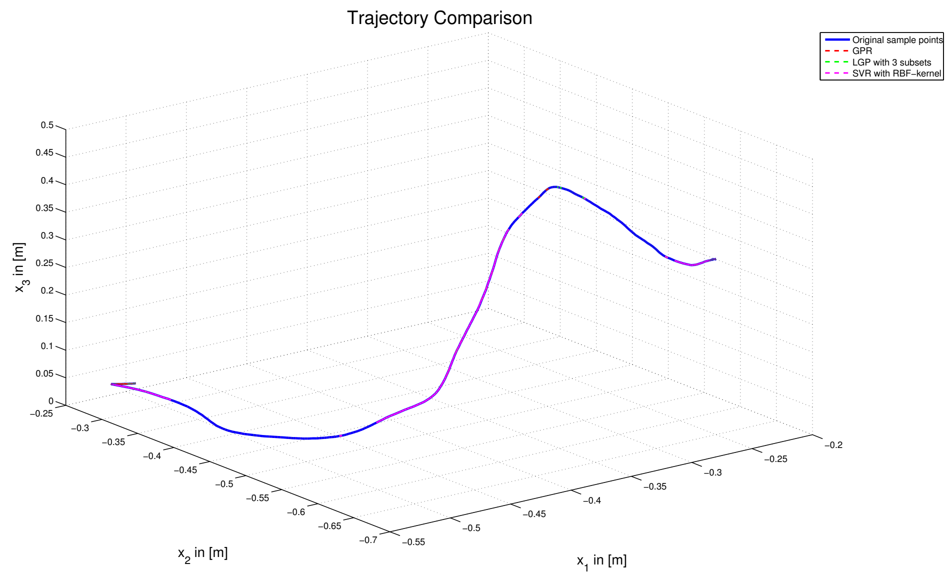


With a large training data size of 1000 points, these three models' RMSE barely differ from each other. In order to make this difference more clear, we further downsample the data in the next paragraph.

Table 3.6: RMSE Comparison between GP, LGP and SVR for $t_s = 15$ ms

	GP	LGP	SVR
In x_1-direction	5.56×10^{-4}	5.54×10^{-4}	5.57×10^{-4}
In x_2-direction	9.68×10^{-4}	9.70×10^{-4}	9.81×10^{-4}
In x_3-direction	1.14×10^{-3}	1.14×10^{-3}	1.14×10^{-3}

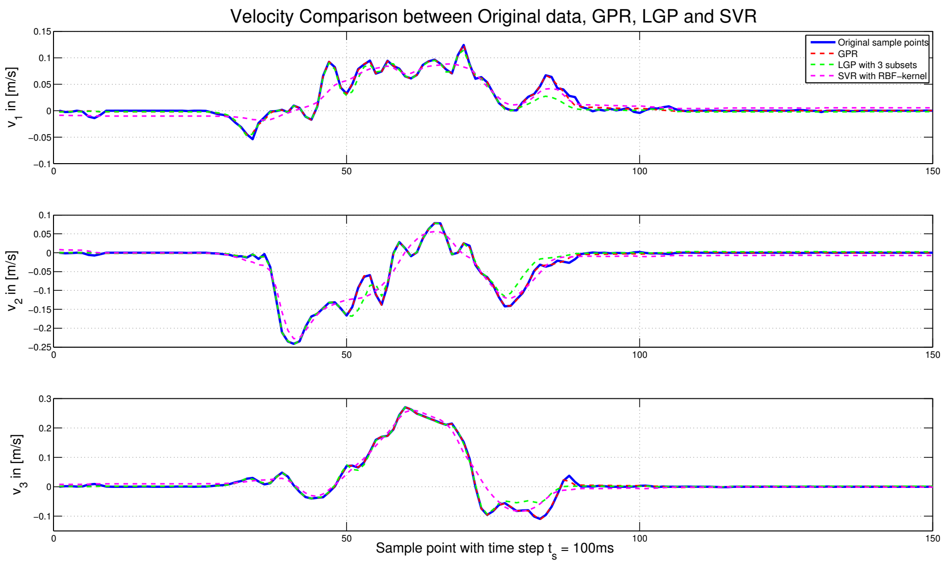
Figure 3.6: 3-D Trajectory Comparison for time step $t_s = 15$ ms.



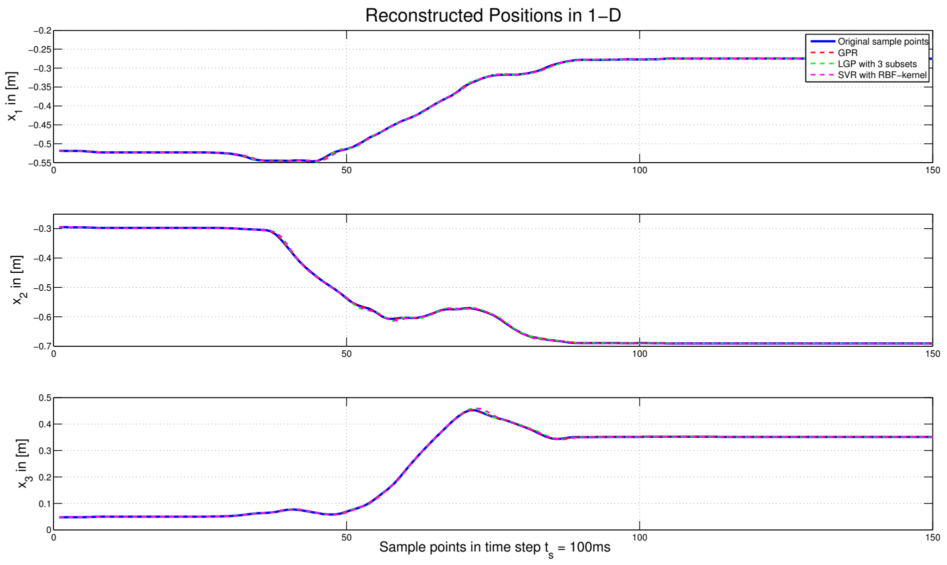
For sampling time $t_s = 100$ ms and $n = 150$ datapoints To test the accuracy of the models when receiving only a low amount of training data, we increased the sampling time step to $t_s = 100$ ms, which results in a trainingset of 150 points. Also, the training output does not need to be scaled for `onlineSVR` to obtain proper weights, so the following figures should make the difference between these four models more noticeable. Table 3.7 shows the RMSE of each model in x_1-, x_2- and x_3-direction. The RMSE is plotted in Figure 3.8. Figure 3.7a, 3.7b and 3.9 show the velocity, 1-D and 3-D trajectory comparison between original data and reconstructed data from different models.

Figure 3.7: 1-D Comparison for $t_s = 100$ ms

(a) Velocity Comparison

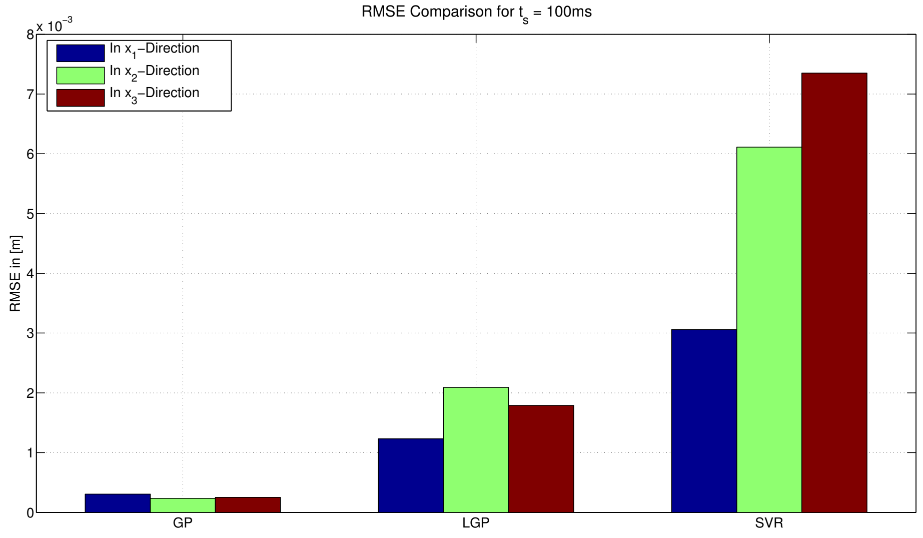


(b) Position Comparison



x-axis shows the sampled points with time step $t_s = 100$ ms and y-axis the velocity in [m/s]. Blue line shows the flow of the original values. Red dashed line shows the data obtained from GPR. Green dashed line shows the flow of values obtained from LGP. The magenta dashed line the values from SVR. The RMSE is listed in Table 3.7.

Figure 3.8: RMSE plot for $t_s = 100$ ms

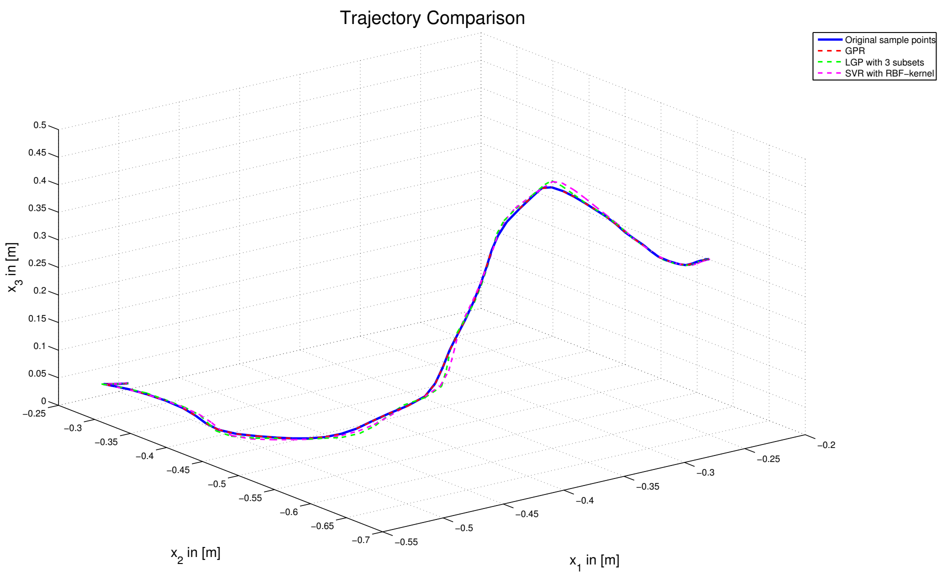


For $t_s = 100$ ms, the difference between these three models becomes more clear: while outperforming GP and SVR in computing time, LGP is still able to achieve reasonable results. Note that in this plot, the y-axis goes from 0 to 8×10^{-3}, whereas the scale in Figure 3.5 goes from 0 to 1.2×10^{-3}, which might give the false impression of achieving better result given less data.

Table 3.7: RMSE Comparison between GP, LGP and SVR for $t_s = 100$ ms

	GP	LGP	SVR
In x_1-direction	3.06×10^{-4}	1.23×10^{-3}	3.06×10^{-3}
In x_2-direction	2.36×10^{-4}	2.09×10^{-3}	6.11×10^{-3}
In x_3-direction	2.50×10^{-4}	1.79×10^{-3}	7.35×10^{-3}

Figure 3.9: 3-D Trajectory Comparison for time step $t_s = 100$ ms.



3.2.3 Discussion

As illustrated last two sections, the predicted velocities are very close to the original ones and with that also a precise trajectory reconstruction, even with only $n = 150$ training pairs. This proves that the LGP model is, in terms of accuracy, indeed viable.

While remaining flexible and highly accurate, our model still outperforms both GP and the state-of-the-art, SVR in computing time. Defining the parameter N for the maximum size of the clusters and M for the number of relevant clusters in LGP, user can rest assured that the computation time will not exceed $O(MN^2)$, which gives the operator control over the computation time instead of being almost entirely data-reliant. This is a major advantage comparing to other models.

Chapter 4 Conclusion

In this thesis, we analyze Gaussian Process Regression and aim to develop an online version of it. First, we investigate calculations within the GP and propose two techniques to optimize the computing time[7][6]. After implementing them both, the two techniques are benchmarked against each other. Furthermore, we chose the technique with better results and integrate it into a more real-time orientated GP model, LGP[13]. At last, we benchmark the standard GP, LGP and SVR[14] against each other and draw the conclusion that, the implemented LGP model is indeed viable.

However, the LGP model can still be improved in the following aspects:

- the training output for the GP/LGP is still 1-D, meaning in order to predict 3-D positions, three of the models have to be created. A model that takes arbitrary m-dimensional data as training output can be considered for future work;

- while the improvement might be small due to the small size of the subsets, a real-time pattern removal function can still be implemented in case the subset reaches its maximum size;

- predictions still occasionally deviate from the target value when e.g. none of the cluster-centroids are relevant enough to give informations, as shown in Figure 3.4a. This effect should become less relevant the larger the sample-set get. In order to prove/refute that, test with much larger sample-set should be performed;

- more trajectories can be used for testing to have a comprehensive benchmark results on the model. Evaluation on a real robot can also considered as part of the future work.

List of Abbreviations

GP	Gaussian Process
GPR	Gaussian Process Regression
LGP	Local Gaussian Process
LWPR	Locally Weighted Projection Regression
RBF	Radial Basis Function
RMSE	Root-Mean-Square Error
RPROP	Resilient Back-Propagation
SVM	Support Vector Machine
SVR	Support Vector Regression

List of Figures

Bibliography

[1] M. Blum. Gaussian Process Library for Machine Learning. `https://github.com/mblum/libgp`, 2012.

[2] M. Blum and M. Riedmiller. Optimization of Gaussian Process Hyperparameters using Rprop. In *European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning*, 2013.

[3] J. Q. Candela and C. E. Rasmussen. A unifying view of sparse approximate gaussian process regression. *Journal of Machine Learning Research*, pages 1939–1959, 2005.

[4] L. Csató. *Gaussian Prcocesses: Iterative Sparse Approximations.* PhD. dissertation, Aston University, 2002.

[5] L. Csató and M. Opper. Sparse online gaussian processes. Technical report, Neural Computing Research Group, 2002.

[6] N. Drakos. *Computer Based Learning Unit*, chapter Network Intrusion Detection: Evasion, Traffic normalization and End-to-End protocol semantics (1997). University of Leeds, 1997.

[7] Lars Eldén, Misha E. Kilmer, and Dianne P. O'Leary. *G.W. Stewart: Selected Works with Commentaries*, chapter Updating and Downdating Matrix Decompositions, pages 45–58. Birkhäuser Boston, 2010.

[8] A. Gijsberts and G. Metta. Real-time model learning using Incremental Sparse Spectrum Gaussian Process Regression. *Neural Networks*, August 2012.

[9] D. H. Grollman and O. C. Jenkins. Sparse incremental learning for interactive robot control policy estimation. In *Robotics and Automation, 2008. ICRA 2008. IEEE International Conference on*, pages 3315–3320, 2008.

[10] J. Jeka. Light Touch Contact: Not Just for Surfers. *The Neuromorphic Engineer. A Publication of INE-WEB.ORG*, 2006.

[11] D. Nguyen-Tuong, M. Seeger, and J. Peters. Local gaussian process regression for real-time model-based control. In *International Conference on Robots and Systems (IROS 2008)*, 2008.

[12] D. Nguyen-Tuong, M. Seeger, and J. Peters. Local gaussian process regression for real time online model learning and control. In *Advances in Neural Processing Systems (NIPS 2008)*, 2008.

[13] D. Nguyen-Tuong, M. Seeger, and J. Peters. Model Learning with Local Gaussian Process Regression. *Advanced Robotics*, 2009.

[14] F. Parella. Online support vector regression. Master's thesis, University of Genoa, 2007.

[15] L. Polok, V. Ila, M. Solony, P. Smrz, and P. Zemcik. Incremental Block Cholesky Factorization for Nonlinear Least Squares in Robotics. In *Proceedings of Robotics: Science and Systems*, 2013.

[16] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press, 2006.

[17] S. Schaal, C. G. Atkeson, and Vijayakumar. Real-time robot learning with locally weighted statistical learning. In *International Conference on Robotics and Automation(ICRA)*, 2000.

[18] S. Schaal, C. G. Atkeson, and S. Vijayakumar. Scalable techniques from nonparametric statistics for real time robot learning. *Applied Intelligence - Special Issue on Scalable Robotic Applications of Neural Networks Vol. 16 No.1*, 2002.

[19] B. Schölkopf and A. Smola. *Learning with Kernels: A Tutorial Introduction*, chapter Support Vector Regression. the MIT Press, 2002.

[20] M. Seeger. Low rank updates for the cholesky decomposition. Technical report, 2004.

[21] A. Simpkins. *Robotic Systems - Applications, Control and Programming (2012)*, chapter Real-Time Control in Robotic Systems, pages 209–234. InTech, 2012.

[22] E. Snelson and Z. Ghahramani. Local and global sparse gaussian process approximations. In *Proceedings of the Eleventh International Conference on Artificial Intelligence and Statistics (AISTATS 2007)*, pages 524–531, 2007.

[23] J. Sun de la Cruz, Kulić. D., and Owen W. *Autonomous and Intelligent Systems*, chapter Online Incremental Learning of Inverse Dynamics Incorporating Prior Knowledge, pages 167–176. Springer-Verlag Berlin Heidelberg, 2011.

[24] J. Ting, M. Kalakrishnan, S. Vijayakumar, and S. Schaal. Bayesian kernel shaping for learning control. In D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, editors, *Advances in Neural Information Processing Systems 21*, pages 1673–1680. Curran Associates, Inc., 2009.

[25] S. Vijayakumar and S. Schaal. Locally weighted projection regression: An o(n) algorithm for incremental real time learning in high dimensional space. In *in Proceedings of the Seventeenth International Conference on Machine Learning (ICML 2000)*, pages 1079–1086.

[26] S. Vijayakumar, S. Schaal, and A. D'Souza. Incremental online learning in high dimensions. In *Neural Computation 17 (2005)*, pages 2602–2634, 2005.

License