**TU Berlin**
**Fakultät IV**
**Department of Software Engineering and Theoretical Computer Science**
**Methoden der Künstlichen Intelligenz, Projekt**

# Report

# Projects in Machine Learning and Artificial Intelligence winter term 2012/2013

# Project: Balancing a Pole

**by**
**Andreas Ley (314674)**
**and**
**Stephan Meyer (314497)**

April 14, 2013

## Abstract

The following document describes the work and results of the "Balancing a Pole" project, which aimed to reproduce the results of [Deisenroth2010]. We will show, that the algorithm indeed works, but that the computational burden is extremely large, and that without further structural changes, the algorithm scales very poorly. We will outline some low level optimizations that we employed to cut down the computational time and we will also discuss the importance of the full estimation of the states' covariances on the algorithm's ability to design a policy. We will close with a short qualitative evaluation of the algorithm and the produced policys.

# Contents


**1 Introduction** **5**
- 1.1 Scenario definition . . . 5
- 1.2 Overview over the algorithm . . . 6
  - 1.2.1 Optimizing the policy . . . 6
  - 1.2.2 Updating the world model (modeling phase) . . . 9

**2 Theoretical background** **9**
- 2.1 Introduction to Gaussian Processes . . . 9
- 2.2 Gaussian Processes with uncertain input . . . 10
- 2.3 Introduction to Radial Basis Function - Networks . . . 11
- 2.4 RBF-Network as a determinisic GP . . . 12

**3 Optimizing the cost function** **14**
- 3.1 Analytical gradient computation . . . 14
- 3.2 Powel's method of conjugated directions . . . 16

**4 Implementation** **17**
- 4.1 Software architecture . . . 17
- 4.2 Optimizations . . . 17
- 4.3 Adapted parameter optimization . . . 18

**5 Results** **19**
- 5.1 The importance of computing covariances . . . 19
- 5.2 Evalutation of the produced policy for the inverted pendulum . . . 21

**6 Conclusion** **22**
- 6.1 Prospects . . . 22


# List of Figures


1 Inverted pendulum setup . . . 5
2 The two phases of the algorithm . . . 6
3 Estimation of the policy's trajectory . . . 7
4 A Gaussian Process with 3 observations . . . 10
5 Gaussian Process moment matching . . . 11
6 Influence reagions of controll points in a RBF-Network . . . 12
7 This figure shows the resulting function of a RBF-Network . . . 12
8 A deterministic GP . . . 13
9 Deterministic GP expressed as RBF-Network . . . 13
10 Toy example to assess the importance of covariances . . . 20
11 Trajectory without covariances . . . 24
12 Trajectory with covariances . . . 25

13 Final trajectory of the optimized policy . . . . . . . . . . . . . . . . . . . . 26
14 Progression of the expected errors . . . . . . . . . . . . . . . . . . . . . . . . 27

# 1 Introduction

In this document, we will describe our work and our results for the project “Balancing a Pole”. The goal of the project was to reproduce the results of [Deisenroth2010]. On the following pages, we will give a brief explanation of the algorithm and it’s mathematical background and then present our implementation and our results.

The algorithm presented in [Deisenroth2010] solves the problem of designing a control policy for an arbitrary control engineering problem with little to no expert knowledge and only a minimal amount of interaction with the real system. The approach described in the original paper uses machine learning methods to learn the real system’s dynamics through a small amount of observations and then employs this learned numerical model, which emulates the system’s dynamics, to design and refine the control policy in an offline process. The alorithm is of immense interest, because it allows the design of control policies without intimate knowledge of the system at hand, and can also be used to create non linear control policies, a field of control theory, which is not yet as well explored as that of linear control theory.

Similarly to the original paper, we applied the algorithm to the control engineering problem of the inverted pendulum.

## 1.1 Scenario definition

The specific control engineering problem, to which we applied the algorithm and on which we tried to gauge the algorithm’s performance, is the problem of the inverted pendulum.

It is a commonly used toy problem of control theory, such that it’s properties and dynamics are well explored and understood. It also facilitates the comparison of the algorithm’s performance with the performance of common control theory approaches. In this project we will however refrain from the latter.

The inverted pendulum consists of a cart, that can be accelerated forward or backward by an actuator. Mounted to the cart via a rotational bearing is a pole such that the pole can rotate freely in the vertical 2D plane around the cart (see figure 1). This makes the pole a pendulum, that can not be explicitly controlled by the cart’s actuator.

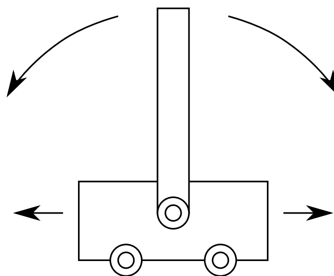


Figure 1: Experiment setup. A pole is mounted to the middle of a cart. The cart balances this pole by driving to the left or right.

The goal of the inverted pendulum problem is to accelerate the cart via it's actuator in a way, such that the pole swings into an upright state and is stabilized there. Since the goal state is instable, this requires constant action from the actuator.

## 1.2 Overview over the algorithm

In order to design a control policy that can accomplish the task defined in section 1.1, the algorithm described by [Deisenroth2010] operates by alternating between two phases, a modeling phase and an optimization phase. During the modeling phase, a set of Gaussian Processes (GPs, see section 2.1 for more details) learns the dynamics of the real world system, such that during the optimization phase, the policy (which is also a variation of a GP) can work offline with the model. Both phases can be seen in figure 2.

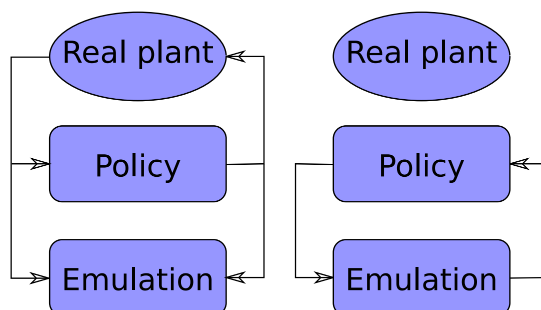



Figure 2: Two main operation modes of the algorithm. The left scheme shows the modeling phase, where observations are collected from the real world to improve the model of the world. The right scheme shows the optimization phase, where the policy is trained by estimating it's performance using the emulation of the world.

### 1.2.1 Optimizing the policy

For the purpose of this section, we will assume that we have a model of the world's dynamics, called the emulation, that is "good enough". Using this emulation, a policy's performance can be estimated and thus optimized in an offline process. This section shall give an overview over this phase, section 1.2.2 will describe, how the emulation learns the real world's dynamics in the first place.

The policy's performance is gauged by estimating the state trajectory, that the policy would produce on the real system, and then applying an error metric to this state trajectory which returns a single scalar value. The error metric is designed in a way, that it produces large values if the trajectory contains long spans of bad states (such as the pendulum not standing upright at the target position).

To estimate the state trajectory of a policy, the policy is put in a closed loop with the emulation such that starting from a constant initial state, the state's progression can be emulated for a limited time horizon (see figure 3). The current state estimate is fed into

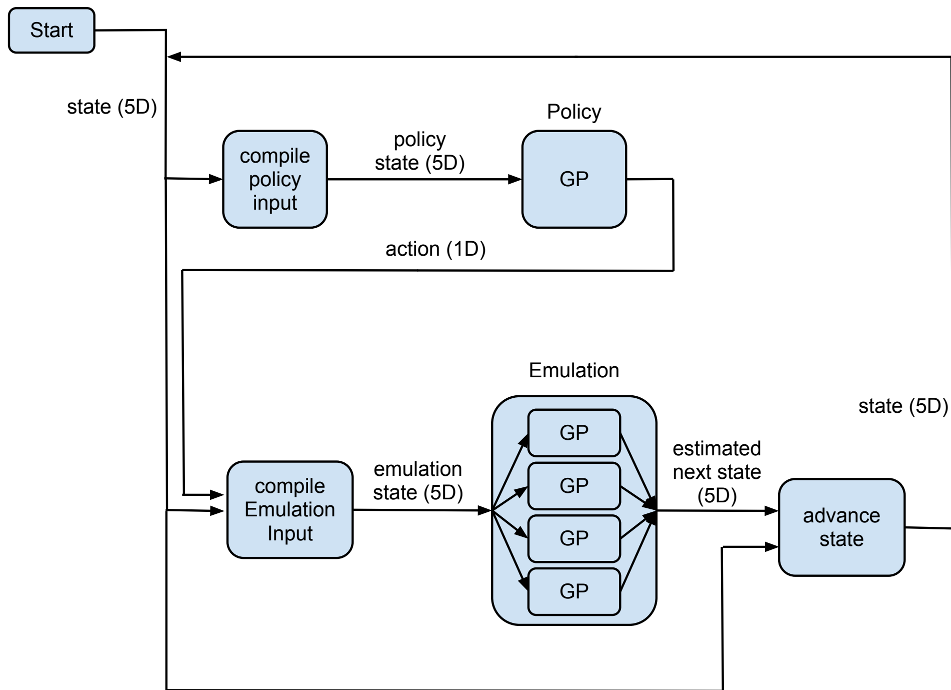



Figure 3: This figure shows the trajectory estimation for a given policy in the optimization phase of the algorithm. Starting from the initial state, the current state is fed into the policy, which creates a force to control the cart. The force and the current state are then put into the emulation which creates the next state. Performing this loop for a couple of iterations produces an estimate of the state trajectory.

the policy which then returns the force that should be applied by the actuator. This force together with the current state estimate is in turn fed into the emulation which returns the state estimate of the next time step.

In the inverted pendulum scenario, we use a five dimensional vector to encode the state of the system. The five components are

- Position of the cart
- Velocity of the cart
- Cosine of the angle of the pole
- Sine of the angle of the pole
- Angular velocity of the pole

Since the pole can rotate all the way around the cart, a continous representation of the angle is needed. We employed the method proposed by [Deisenroth2010] of splitting the angle up into the sine and cosine of the angle.

These states are however not actually fed directly to the policy and the emulation. Instead, the input vector of the policy consists of:

- Distance of the cart to the target position
- Velocity of the cart
- Cosine of the angle of the pole
- Sine of the angle of the pole
- Angular velocity of the pole

The policy outputs a single value, the force that the actuator should apply. Since the actuator's force is limited, for example by the maximum torque of the motor involved, the policy's output must be clamped to a confined interval. In this project we used the sine function as such a squashing function, as proposed in the original paper [Deisenroth2010].

The input of the emulation contains:

- (Squashed) force
- Velocity of the cart
- Cosine of the angle of the pole
- Sine of the angle of the pole
- Angular velocity of the pole

The output of the emulation is:

- Movement of the cart ($\Delta$position)
- New velocity of the cart
- New cosine of the angle of the pole
- New sine of the angle of the pole
- New angular velocity of the pole

To map between the different state representations and subsets, as well as to perform the squashing of the policy's output into a confined interval, special steps must be performed in addition to the policy's and emulation's computations. These are labeled "compile policy input", "compile emulation input" and "advance state" in figure 3. Note, that our emulation outputs are, apart from the position, absolute values and not differences as proposed by the original paper.

Using the estimation of the policy's performance, as outlined in this section, the optimization of the policy is actually equivalent to the well studied problem of maximizing/minimizing a scalar valued function by modification of it's input arguments. The input arguments are in our case the parameters of the policy, while the scalar value, that is to be minimized, is the cost returned by the error metric. The minimization process is described in more detail in section 3.

The optimization of the policy depends highly on the accuracy of the emulation. The idea of alternating between the optimization and modeling phases (see figure 2) ensures, that the policy gets better with each iteration due to the improved emulation, while the emulation also gets better around the interesting trajectories.

Also keep in mind, that the states and input/output vectors are not deterministic, but Gaussian distributed to model the emulation's uncertainty in it's predictions. That is, every state is actually encoded by a vector of it's means and a covariance matrix.

### 1.2.2 Updating the world model (modeling phase)

During the modeling phase, the current control policy interacts with the "real world". The policy's output as well as the trajectory of the real world's states (as reported by the sensors) is collected and used to refine the emulation's model of the real world.

Given triples from the recorded trajectory, which contain a state, the policy's response to the state and the followup state, pairs of input/output vectors for the emulation can be built. These represent the desired output, that the emulation should produce for a given input. The input/output pairs can be fed directly to the emulation to improve the predicitions, as we will show in section 2.1.

Due to lack of the actual hardware, we used a simulation in place of an actual real world inverted pendulum. That is, the real behaviour of the pole and the cart was implemented by numerically solving the differential equations that physically describe the system (see [Sultan2003] for a derivation of the ODEs that we used).

# 2 Theoretical background

This section gives a detailed description of the theoretical background for Gaussian Processes and RBF-Networks.

## 2.1 Introduction to Gaussian Processes

A Gaussian Process (in the following "GP") is a stochastic process that maps an input vector to a Gaussian distributed scalar output value. That is, the output of a Gaussian Process has a mean $\mu$ and a variance $\sigma$. It can be thought of as a function approximator, which performs an elaborate interpolation scheme on a set of control points. However, contrary to simple function approximators, a GP is also able to model it's uncertainty when sampled far away from any control point, or when the control points are noisy.

Figure 4 shows a Gaussian Process with its mean (the red line) and its confidence interval (the blue area).

Initially the GP has a prior of the function which is $\mu = 0$ and $\sigma = 2$ in our project for every input vector. When control points, or in our case observations of the world, are added, the approximation around the given control points change. That is, the mean near the control points bends over to the value of the control points and the variance around that control points decreases. Intuitively speaking, a GP is more confident in it's approximation of a function around the area of an observation and therefore becomes more accurate and less uncertain in that area.

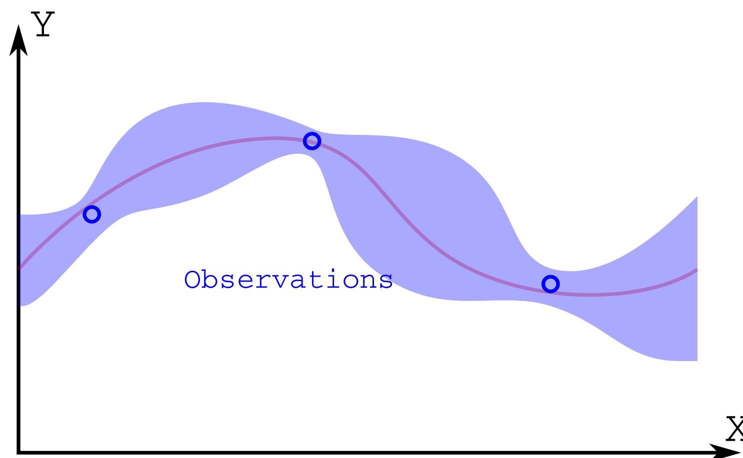



Figure 4: This figure shows a Gaussian Process with 3 observations. The red line is the mean and the blue area is the confidence interval. This way, a Gaussian distributed random scalar $y$ is assign to each input $x$. The blue circles are observations of the function that is to be approximated.

The more observations are given, the more accurate the approximation of the underlying function becomes. [1]

## 2.2 Gaussian Processes with uncertain input

One of the major challenges of the approach of [Deisenroth2010] is that the input state vector for each GP is not a deterministic vector but a Gaussian distributed random vector. When a GP calculates the output for a Gaussian distributed input, the output is not necessarily Gaussian distributed. Figure 5 shows one possible outcome of such a calculation.

Further calculations with a non Gaussian distributed input vector can result in an even more complex distribution. To keep the complexity fixed, [Deisenroth2010] proposes the approximation of the output distribution with a Gaussian distribution by only computing the first two moments of the output distribution and setting all other moments to zero.

[1] A more detailed description of the Gaussian Process can be found in [Deisenroth2010] chapter 2.2.

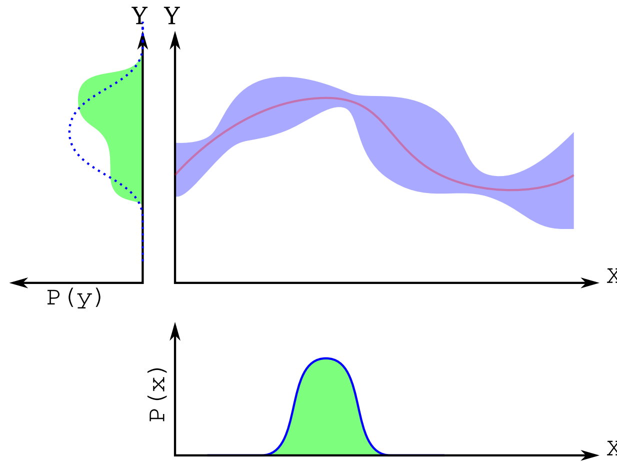



Figure 5: This figure shows the output (upper left corner) of a Gaussian Process (upper right corner) for a Gaussian distributed input (bottom). The outcome distribution is not necessarily Gaussian distributed (green area on the left side).

In figure 5 the blue dotted line in the upper left plot shows the approximated Gaussian distribution of a Gaussian distributed input vector. [1]

## 2.3 Introduction to Radial Basis Function - Networks

A RBF-Network is historically considered a special case of an artificial neuronal network. It maps multidimensional input vectors to a scalar output using a special form of interpolation between a set of control points. That is, for a given input it computes the distances of the input vector to all control points, and then applies a so called radial basis function to each of the distances, which returns a weight for each control point. Typical radial basis functions are scaled Gaussian curves.

Figure 6 shows a RBF-Network for two control points. When two or more control points lie close to each other, the radial basis functions and thus the areas of influence of the control points may overlap.

The output of the RBF-Network is the weighted sum of all the control point's labels, using the weights computed by the radial basis function for each distance. The result is pictured as the blue line in figure 7.

It should be pointed out that a RBF-Network is actually not that dissimilar from a GP, a fact that [Deisenroth2010] leveraged to simplify the implementation. We will elaborate on this in the next section.

[1] The equation to compute the multivariate prediction for a Gaussian distributed input, is explained in more detail in section 2.3 [Deisenroth2010].

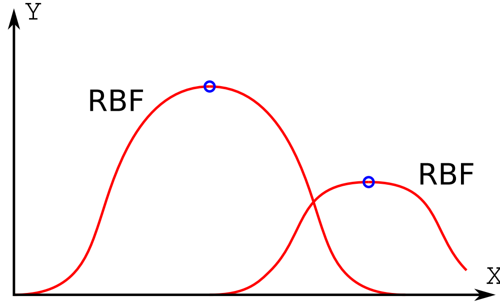



Figure 6: This figure shows a RBF-Network for two input vectors. The blue circles denote the control points, the horizontal positions denoting the control points' position in the input space and the vertical positions denoting the control points' labels. The red curves show the radial basis functions multiplied by the control points' labels. They represent the area of influence of the control points and can overlap, as in this case.

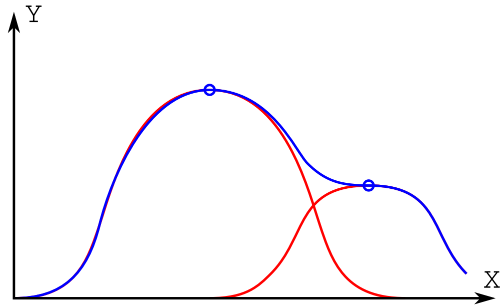



Figure 7: This figure shows the resulting function of a RBF-Network

### 2.4 RBF-Network as a determinisic GP

As indicated in the last section a RBF-Network can also be interpreted as a deterministic GP. This can only be done without loss of generality if the approximating GP has a prior mean of $m = 0$. Figure 8 shows a GP without confidence interval (deterministic). Therefore "deterministic" means, that there is no uncertainty in the approximation.

The exact same curve can be generated if one uses the sum of two gaussian functions with the correct parameters. For all intents and purposes this is a RBF-Network.

In detail we can define a RBF-Network as follows

$$\tilde{\pi}(\mathbf{x}_*) = \sum_{s=1}^{N} \beta_s k_\pi(\mathbf{x_s}, \mathbf{x}_*) = \beta_\pi^{\mathbf{T}} k_\pi(\mathbf{X}_\pi, \mathbf{x}_*) \tag{1}$$

where $\beta$ is the weight vector of the neurons and can be choosen freely, $k_\pi(\mathbf{x_s}, \mathbf{x}_*)$ is the neuron function. The function $k_\pi$ is defined as the kernel function described in [Deisenroth2010] section 2.2.1.

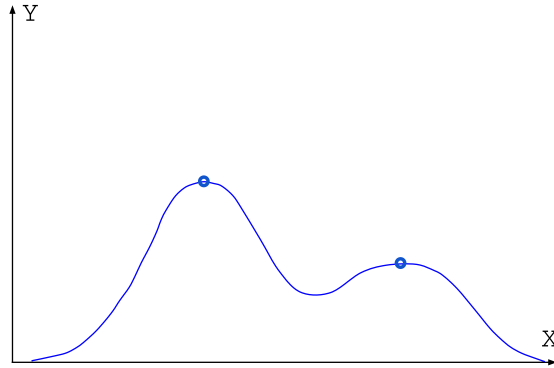



Figure 8: A deterministic GP. This figure shows the same GP as figure 4 without variance

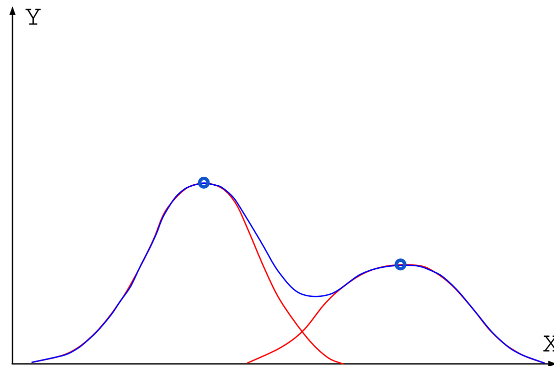



Figure 9: This figure shows the same curve as figure 8. Here the curve is formed by a RBF-Network according to the schema described in section 2.3

If $\beta$ is defined as

$$\beta := (k_h(\mathbf{X}, \mathbf{X}) + \sigma_\epsilon^2 \mathbf{I})^{-1} \mathbf{y} \tag{2}$$

then equation 1 is identical to the formular of a deterministic GP (GP without confidence interval). By exploiting this, the RBF-Network can be replaced or reinterpreted as a deterministic GP. Figure 9 shows how the same function can be constructed by using a RBF-Network. We employed this property, as suggested by [Deisenroth2010] to reduce implementation effort. That is, we did not implement a RBF-Network in addition to our GP implementation, but rather performed a slight modification of the GP implementation, such that it can be operated in “RBF mode” as well as in “GP mode”.

# 3 Optimizing the cost function

As was hinted in section 1.2.1, in order to optimize the policy, an error metric or cost function is required. We used a saturating cost function as suggested by [Deisenroth2010], which is defined as

$$c(\mathbf{x}) = 1 - exp(-\frac{1}{2a^2}d(\mathbf{x}, \mathbf{x}_{target})^2) \tag{3}$$

where $\mathbf{x}_{target}$ is the target state and $\mathbf{x}$ is the current state of the system. This cost function only depends on the distance from the current state to the target state. Since the distance at one point in time gives only limited information of how good the policy is working it is necessary to measure the behavior of the policy over time. The longer the policy is able to hold the pole near the target state, the better is the policy. Thus, the optimization is performed with respect to the long term cost

$$V^{\pi}(\mathbf{x}_0) = \mathbb{E}_{\tau}[\sum_{t=0}^{T} c(\mathbf{x}_t)] = \sum_{t=0}^{T} \mathbb{E}_{x_t}[c(\mathbf{x}_t)] \tag{4}$$

where $\tau = (\mathbf{x}_0, ..., \mathbf{x}_T)$ is the (estimated) state trajectory that the policy would produce. So the long term cost is the sum of all expected values of the cost function $c(\mathbf{x})$ over a fixed time horizon of $T$ time steps. In the following sections two ways of optimizing the policy with respect to the long term cost will be explained.

## 3.1 Analytical gradient computation

The original paper [Deisenroth2010] pointed out, that the gradients of the error function can be computed analytically, because all components (GPs, RBF-Network and squashing function) are expressed analytically in closed form. This is of tremendous interest, because it allows for very efficient optimizations with off the shelf algorithms such as conjugated gradient descent or similar methods.

The gradient of the long term cost is defined as

$$\frac{dV^{\pi_\Psi}(\mathbf{X}_0)}{d\Psi} = \sum_{t=0}^{T} \frac{d}{d\Psi}\mathbb{E}_{\mathbf{x}_t}[c(\mathbf{x}_t)|\pi_\Psi] \tag{5}$$

where $\psi$ is the whole parameter set of the policy $\pi$.

Instructions, on how to derive the equations for the gradients, are given in detail in [Deisenroth2010] chapters 3.6 and 3.7. Unfortunately the equations themselves were not expicitly given in the paper, nor in any other paper describing the same approach. Therefore the basic equations of the gradient had to be derived. In this project we only derived the gradient with respect to the labels of the policy's control points ($\mathbf{y}_\pi$). This is due to an optimization, that we will describe in section 4.3.

The partial solutions to the gradient computation will be presented here. At first we define $\beta$ and $q_i$ as suggested by [Deisenroth2010]

$$\beta_\pi = (K_\pi + \sigma_\pi I)^{-1}\mathbf{y} \tag{6}$$

where $K_\pi$ is the kernel matrix of the GP of the policy having $m$ training targets/control points, $\sigma$ is the training target noise and $\mathbf{y}$ are the training labels.

$$q_i = \alpha^2 |\Sigma\Lambda^{-1} + I|^{-\frac{1}{2}} exp(-\frac{1}{2}(x_i - \mu)^T (\Sigma + \Lambda)^{-1}(x_i - \mu)) \tag{7}$$

with $i = [1, ..., m]$ and $q_i \in \mathbb{R}$. $q_i$ is the expected covariance between the training input $x_i$ and a test input $x_*$. $\mu$ and $\Sigma$ are the mean and the covariance matrix of a test input $x_* \sim \mathcal{N}(\mu, \Sigma)$. $\Lambda$ are the characteristic length scales of the kernel function (see [Deisenroth2010] chapter 2.2). $\alpha^2$ is the signal variance.

The following formular calculates the gradient of the mean with respect to $\mathbf{y}$.

$$\begin{aligned} \frac{\partial \mu_t}{\partial \mathbf{y}} &= \beta_\pi^T \cdot \frac{\partial q}{\partial \mathbb{E}[\mu]} \\ &\cdot \left[ \frac{\partial \mathbb{E}[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial \mathbb{E}[\tilde{\pi}(\cdot)]} \cdot \frac{\partial \mathbb{E}[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} + \frac{\partial \mathbb{E}[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial cov[\tilde{\pi}(\cdot)]} \cdot \frac{\partial cov[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} \right] \\ &\cdot \beta_\pi^T \cdot \frac{\partial q}{\partial cov[\mu]} \\ &\cdot \left[ \frac{\partial cov[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial \mathbb{E}[\tilde{\pi}(\cdot)]} \cdot \frac{\partial \mathbb{E}[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} + \frac{\partial cov[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial cov[\tilde{\pi}(\cdot)]} \cdot \frac{\partial cov[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} \right] \end{aligned} \tag{8}$$

Fully expanded, equation 8 becomes very large. Hence the parts of the formular are presented separately.

$$\begin{aligned} \frac{\partial \mathbb{E}[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial \mathbb{E}[\tilde{\pi}(\cdot)]} &= u_{max} \cdot exp\left(-\frac{\sigma^2}{2} \cdot cos(\mathbb{E}[\tilde{\pi}(\cdot)])\right) \\ &= u_{max} \cdot exp\left(-\frac{\sigma^2}{2} \cdot (\beta_\pi^T \mathbf{q})\right) \end{aligned} \tag{9}$$

$$\begin{aligned} \frac{\partial \mathbb{E}[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial cov[\tilde{\pi}(\cdot)]} &= u_{max} sin(\mathbb{E}[\tilde{\pi}(\cdot)]) \cdot \frac{\partial exp\left(-\frac{\sigma^2}{2}\right)}{\partial \sigma^2} \\ &= u_{max} sin(\beta_\pi^T \mathbf{q}) \cdot -\frac{1}{2} exp\left(\frac{-\sigma^2}{2}\right) \end{aligned} \tag{10}$$

$$\begin{aligned} \frac{\partial cov[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} &= \frac{\partial \beta_\pi^T Q \beta_\pi^T \mathbf{q}}{\partial \mathbf{y}} - \frac{\beta_\pi^T \mathbf{q})^2}{\partial \mathbf{y}} \\ &= (K_\pi + \sigma_\pi I)^{-1^T} \cdot (Q + Q^T) - 2(\beta_\pi^T \mathbf{q}) \cdot (K_\pi + \sigma_\pi)^{-1} \mathbf{q} \end{aligned} \tag{11}$$

$$
\begin{aligned}
\frac{\partial cov[u_{max} sin(\tilde{\pi}(\cdot))]}{\partial \mathbb{E}[\tilde{\pi}(\cdot)]} =& \frac{\partial}{\partial \mathbb{E}[\tilde{\pi}(\cdot)]} \cdot \\
& u_{max}^2 \cdot \\
& \left[ sin(\beta_p i^T k_\pi(\mathbf{x}_\pi, \mathbf{x}_*)) - exp\left(-\frac{\sigma^2}{2}\right) sin(\beta_\pi^T \mathbf{q}) \right] \cdot \\
& \left[ sin(\beta_\pi^T k_\pi(\mathbf{x}_\pi, \mathbf{x}_*)) - exp\left(-\frac{\sigma^2}{2}\right) sin(\beta_\pi^T \mathbf{q}) \right]^T
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
\frac{\partial \mathbb{E}[\tilde{\pi}(\cdot)]^T}{\partial \mathbf{y}} &= \frac{\partial \beta_\pi^T}{\partial \mathbf{y}} \mathbf{q} \\
&= (K_\pi + \sigma_\pi I)^{-1} \mathbf{q}
\end{aligned} \tag{13}
$$

These derivatives describe the gradient of the mean of the system states $\mathbf{x}$ with respect to $\mathbf{y}$. However, the explicit calculation of $\frac{\partial \Sigma}{\partial \mathbf{y}}$ would go beyond the scope of this project. Hence the optimization process was done without the analytical gradients, using a method called Powel's method of conjugated directions.

## 3.2 Powel's method of conjugated directions

In addition to the analytical computation of the error function's gradients, we also explored the feasibility of gradient free minimization. Luckily, optimization problems are a well studied field, and many algorithms have been proposed for it, some of which can minimize/maximize a function without explicit knowledge of it's gradients using only the function values. The algorithm we eventually settled on is Powel's method of conjugate directions (see for example [NumericalRecipes2007]).

Powel's method of conjugate directions finds the (local) minimum of a function with $n$ parameters by iteratively doing line searches for the minimum along each direction (in turn) of a set of $n$ directions. The set of directions is modified in a way such that after about $n$ iterations the directions form a basis of mutually conjugate vectors with respect to the local curvature of the error surface (see algorithm 1). This allows the algorithm to descent into the minimum without the oscillating behavior that simple algorithms like steepest descent display.

A detailed discussion of Powel's method of conjugate directions, or any other optimization technique, is outside the scope of this document. However we will note, that this approach is significantly slower than conjugated gradients using analytical derivatives, since the number of line searches required for Powel's method is roughly quadratic in the number of parameters for a quadratic function, whereas it is only linear for conjugated gradients (see [NumericalRecipes2007]).

**Algorithm 1** Powel's method of conjugate directions

```
                                                        ▷ Initialization
P ← initialParameterSet
for all 0 ≤ i < n do
    D_i ← e_i                       ▷ Use axes as initial set of directions
end for
                                                        ▷ Optimization
while not converged do
    P_old ← P
    for all 0 ≤ i < n do
        P ← LineSearchForMinimum(P, D_i)      ▷ Optimize along each direction
    end for
    find the D_k which made the biggest contribution in this iteration
    D_k ← P − P_old   ▷ Replace one direction by the joint direction of this iteration
end while
    return P
```

# 4 Implementation

For the actual implementation, we experimented with Matlab and C++ and soon realized, that Matlab scripts tend to become either clumsy or slow with growing complexity, while C++ offers a good balance between fidelity and speed. In addition, the superior reusability of C++ code also speaks in favor of a C++ implementation. This led to the decision, to use C++ for the project.

The following sections briefly describe the software architecture of our C++ implementation and discusses some of the quirks and problems that needed to be overcome.

## 4.1 Software architecture

One central architecture decision was to use meta programming to a large degree and leverage the flexibility it offers. This proved to be extremely helpfull during the development phase, since it enabled us to quickly build small test scenarios for debugging purposes.

We were able to push this to a point, where all the details of the inverted pendulum scenario are abstracted into a single class specification, while the rest of the implementation remains completely problem agnostic and can be used on a different scenario by simply instanciating it with a different scenario specification. For details, see our source code, which comes with extensive documentation.

## 4.2 Optimizations

One problem we ran into early on was the immense amount of computation time that the algorithm requires to optimize a policy. The optimization process is, as described earlier, the process of minimizing an error function by adjusting a set of parameters. The

parameters are, in our case, the control points of the policy. Since the policy has no sense of the dynamics' structure and is essentially just a fancy interpolation between control points, the number of control points grows (in theory) exponential in the number of input dimensions. That means, that the number of parameters that need to be optimized, also grows exponential in the number of policy input dimensions. With Powel's method requiring a number of line searches that is (again in theory) proportional to the square of the number of parameters, and given that each line search requires on the order of 10 to 15 evaluations of the error function, this quickly results in an immense amount of trajectories, which need to be estimated. If we now consider, that each trajectory evaluation requires on the order of 60 Gaussian Process evaluations, each of which has a computational complexity that is again quadratic in the number of observations/control points, it becomes obvious that the computations can run for a very long time.

The first and obvious solution is to design the policy for only a single start configuration and thus only for a single trajectory. This allows to arrange the policy's control points on the surface of a tube around the trajectory, which significantly limits the amount of required control points. Contrary to the original paper, we also kept the policy's control points fixed in the input space and only changed the output "labels" (see section 4.3 for details), which saved some costly matrix inversions.

With these modifications, the algorithm can converge within a human's lifespan but the required time is still somewhat inconvenient. The logical next step would be to employ further structural optimizations, some of which are hinted in the original paper. However, to keep the general structure close to that of the original paper, we refrained from such high level optimizations and only employed some low level optimizations to speed up various hotspots. While initially the algorithm took days to optimize a policy, we managed to reduce this time using low level optimizations to "mere" 10 to 15 hours.

One noteworthy optimization is the combination of multiple Gaussian Processes which generate a multidimensional output vector into a fused "multi Gaussian Process" which can internally share a large amount of computations across the output dimensions. The other is a custom implementation of the exponential function based on the approach taken in the cephes library ([cephes]). Our implementation is vectorized, using Intel's new SSE4.1 instructions, and outperforms the standard implementation in the runtime by a factor of two. Using Powel's method, we also experimented with a multicore implementation, where multiple line searches are performed in parallel. While this does break the convergence of the algorithm (at least in theory), we observed that it still performs "good enough" to be employed in debugging and test runs.

## 4.3 Adapted parameter optimization

In this section we discuss the motives behind and impact of our restriction of the optimization parameters to the policy's labels (denoted $\mathbf{y}_\pi$ in all formulas).

Ordinarily, the policy's behavior is controlled by the following parameters:

- The locations of the control points in the input space $\mathbf{x}_\pi$
- The "labels" or desired outputs of the control points $\mathbf{y}_\pi$

- The hyperparameters of the Gaussian kernel function (for example the length scales $\Lambda_\pi$)

In theory, all these parameters can be adapted to optimize the policy, and in fact, this is proposed by [Deisenroth2010]. This however leads to a big problem when a large number of control points is used. The policy operates according to equation 14

$$\pi(\mathbf{x}_*) = u_{max} sin(\tilde{\pi}(\mathbf{x}_*)), \quad \tilde{\pi}(\mathbf{x}_*) = \beta_\pi^T k_\pi(\mathbf{X}_\pi, \mathbf{x}_*), \quad \mathbf{x}_* \in \mathbb{R}^D \tag{14}$$

in which $\beta_\pi$ is the product of an inverted matrix and the vector of the control point's labels (see [Deisenroth2010]).

$$\beta_\pi = (\mathbf{K}_\pi + \sigma_\pi^2 \mathbf{I})^{-1} \mathbf{y} \tag{15}$$

The computational complexity of a matrix vector multiplication is $\mathcal{O}(n^2)$, while a matrix inversion is $\mathcal{O}(n^3)$ for all practical matrix sizes and becomes extremely costly for a large amount of control points. Since all parameters except $\mathbf{y}_\pi$ influence the matrix that needs inversion, we only optimize $\mathbf{y}_\pi$ and set the other parameters to fixed but reasonable values.

We saw a substantial speedup from this change, however, the modification comes with a price. The reduction of the parameter space leads to a suboptimal policy, because it can't reposition the control points. The results show, that the policy needs a higher amount of control points, but is still able to solve the problem (see section 5).

# 5 Results

We applied the algorithm to two different scenarios: The inverted pendulum scenario and a simplified scenario in which the pendulum is removed from the cart and the policy's goal is just to reposition the cart. While the inverted pendulum scenario is the more difficult problem, due to it's instable goal state, the simplified version is marginally stable and proved rather usefull in testing and understanding the algorithm. In this section, we will discuss our findings for both scenarios by pointing out a pitfall that we fell into, as well as evaluating the quality of the produced control policies.

## 5.1 The importance of computing covariances

Given the computational complexity of the algorithm, it can be tempting to assume the states to be independent during the emulation, so that only the main diagonal of the covariance matrices must be computed. This means, that the covariance between the outputs of the emulation GPs is omitted, as well as the covariance between the inputs and the output of the policy.

We initially followed this track of thought only to find out, that the covariances are actually of vital importance for the algorithm's ability to design a policy, and that without them the policy optimization for instable problems fails (in some cases it can still work for stable and to some degree for marginally stable problems).

To understand the importance of the covariances, consider the following toy example, where a random state is passed to a policy, and the policy's output in turn is passed together with the state into the emulation. We only consider a single iteration (that is we "emulate" only a single timestep) and in addition, our state as well as the policy's output is scalar (see figure 10).

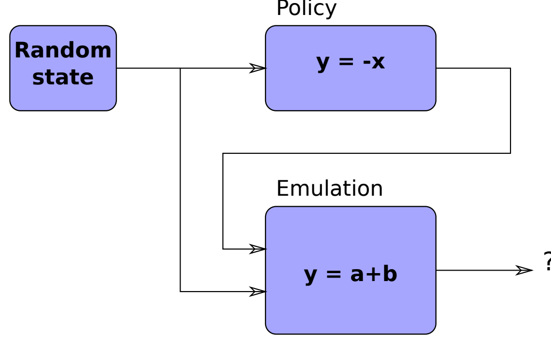



Figure 10: Toy example to assess the importance of covariances for the optimization of the policy.

Let us now assume, that the "dynamics" of the real world system are perfectly modeled by the emulation and that they satisfy the simple formula $S_{N+1} = S_N + P_N$, that is the emulation just computes the sum of the last "state" and the policy's output. Let us further assume, that our goal is to build a policy which brings the system into the state $S = 0$, that is which drives the emulation's output to zero. It is obvious, that the best policy, which perfectly achieves this goal, is the policy which just returns the negative state as output. That way, no matter what the distribution of the inital random state was, the emulation's ouput will always be zero.

Let us now consider the same scenario, if all variables are considered to be independent. Let us also assume, w.l.o.g., that the inital state is mean free and Gaussian distributed with:

$$S_N \sim N(\mu_n = 0, \sigma_n^2) \tag{16}$$

The (perfect) policy's output is then:

$$P_N = -S_N \sim N(-\mu_n = 0, \sigma_n^2) \tag{17}$$

The resulting new state (the emulation's output) is

$$S_{N+1} = S_N + P_N \tag{18}$$

with the variance (note, that the new state is also mean free):

$$E\left[S_{N+1}^2\right] = E\left[(S_N + P_N)^2\right] \tag{19}$$

$$= E\left[S_N^2\right] + 2 \cdot E\left[S_N \cdot P_N\right] + E\left[P_N^2\right] \tag{20}$$

If we now omit the covariance (because we make the assumption that all variables are independent), then the output variance observed by the algorithm is:

$$E\left[S_{N+1}^2\right] = E\left[S_N^2\right] + E\left[P_N^2\right] \quad (21)$$

$$= 2 \cdot E\left[S_N^2\right] \quad (22)$$

So the resulting new state, as seen by the algorithm, is not zero, but rather even less likely to be zero than the initial state, due to the increased variance. In fact, in the eyes of the algorithm, the best policy in this case is the policy which does nothing.

This becomes especially problematic for instable scenarios, like the inverted pendulum. Once the state approches the goal state, any remaining variance should be counteracted by the policy. However, since all the algorithm sees from the policy is uncorrelated noise (because the covariances were assumed zero), the best policy is to do nothing when the pendulum approches it's goal state, a behavior that is obviously not stabilizing the pendulum.

The problem also has an impact on stable and marginally stable scenarios, because uncertainties decrease significantly less without covariances. Figure 11 shows the trajectory of an optimized policy for the simple scenario of just moving the cart from position zero to position one by accelerating and breaking. Note, that the initial state was chosen to have a slight variance. During the course of the trajectory, this variance never decreases, although one would assume, that given an optimized policy which drives the cart into a predefined state, the variance should decrease. Figure 12 shows the same scenario, but this time using full covariance matrices in the optimization phases. As can be seen, the estimated variance is decreasing in this case because the policy tries to bring the cart into a defined state.

## 5.2 Evalutation of the produced policy for the inverted pendulum

Using the proposed algorithm together with the described optimizations, we were able to produce a policy which swings the inverted pendulum up and stabilizes it in a predefined target position (see figure 13).

As can be seen, the policy solves the problem quite well and with the minimum amount of “swing bys”. The policy requires 80 control points, instead of the 20 control points in the original paper, because in our implementation the policy's control points are not repositioned and thus a higher inital density of control points is needed. We did, however, place the control points around the states that the policy was likely to traverse. Our inital observations also differ from those in the original paper. While the original paper uses random forces on a hanging pendulum to produce the initial observations, we seed the algorithm with various observations of a standing pendulum falling over. This gives us a better emulation around the otherwise hard to reach and thus badly approximated goal state.

Figure 14 shows the expected errors over the 30 steps long time horizon for different numbers of modeling and optimization iterations. Compared to the original work, we require significantly more iterations, and thus more observations of the real world, to

arrive at a good policy. We assume, that this is due to our deviation from the original paper in the encoding of the emulation's output, where the original paper proposed to output state differences instead of absolute states.

A major downside of the produced policy is it's restriction to the single trajectory, that the algorithm "programmed" into it during optimization. As soon as the pendulum's state leaves the narrow tube of this choreographed trajectory, be it through disturbances or drift, the policy no longer produces usefull control ouputs. We consider this a fundamental problem of the structure of the policy. The policy is modeled by a number of control points, which must be kept small to keep the algorithm's runtime within reasonable limits. This means, that the (five dimensional) input space of the policy can not be filled completely with control points in a usefull density. Rather the limited number of control points must be placed (or get placed by the algorithm) around the trained trajectory which limits the policy to this one trajectory.

Another downside, not of the policy but of the algorithm that produces it, is the amount of computational time that is required to find a good policy. While the algorithm only requires a few seconds to minutes of interaction with the real system, it requires half a day of computational time for a small problem like the inverted pendulum. This makes it's usage extremely inconvenient.

# 6 Conclusion

In this project, we reproduced the results of [Deisenroth2010]. We gave a brief explanation of the algorithm and the involved mathematical background and presented our implementation and our results. We applied the algorithm to the problem of the inverted pendulum and demonstrated a successfull optimization.

We showed, how the algorithm can be optimised to keep the runtime within reasonable bounds and discussed the importance of the full estimation of the state's covariance.

We closed with a short qualitative evaluation of the algorithm and the policy's produced by it.

## 6.1 Prospects

Of course it would be interesting, to apply the algorithm to a wider range of problems, to better judge it's performance, something which can easily be accomplished with the framework created of this project. A nice addition would however be the implementation of an optimization scheme based on the analytical gradient computation, because we expect to see a significant speedup from this.

We also see a lot of room for improvement (or specialisation) in a change of the policy's underlying structure. [Deisenroth2010] only experimented with linear and RBF-Network structures, of which we only implemented the RBF-Network. It would be interesting to explore the benefits and drawbacks of other structures, possibly specialised to specific scenarios.

Along this theme, it should also be noted that in the sense of control theory the policys proposed in [Deisenroth2010] are PD-controllers with the differential parts being

encoded as “velocities” within the state vectors. It would be interesting to experiment with full PID-controllers by also incorporating low pass filtered “integral” parts in the state vectors.

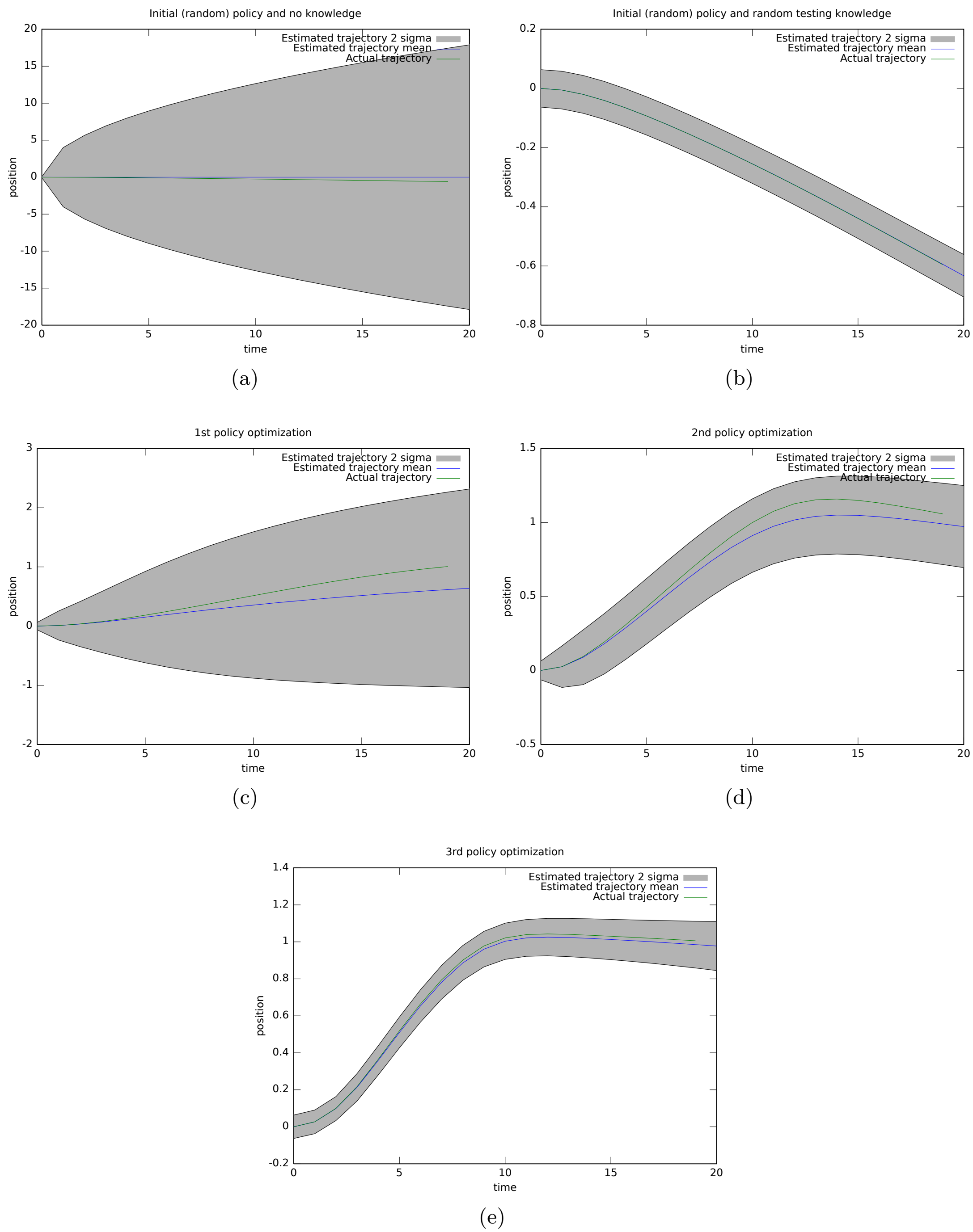



Figure 11: Trajectory of an optimized policy in the simplified scenario, where all covariances were assumed to be zero. As can be seen, the policy can not decrease the uncertainty.

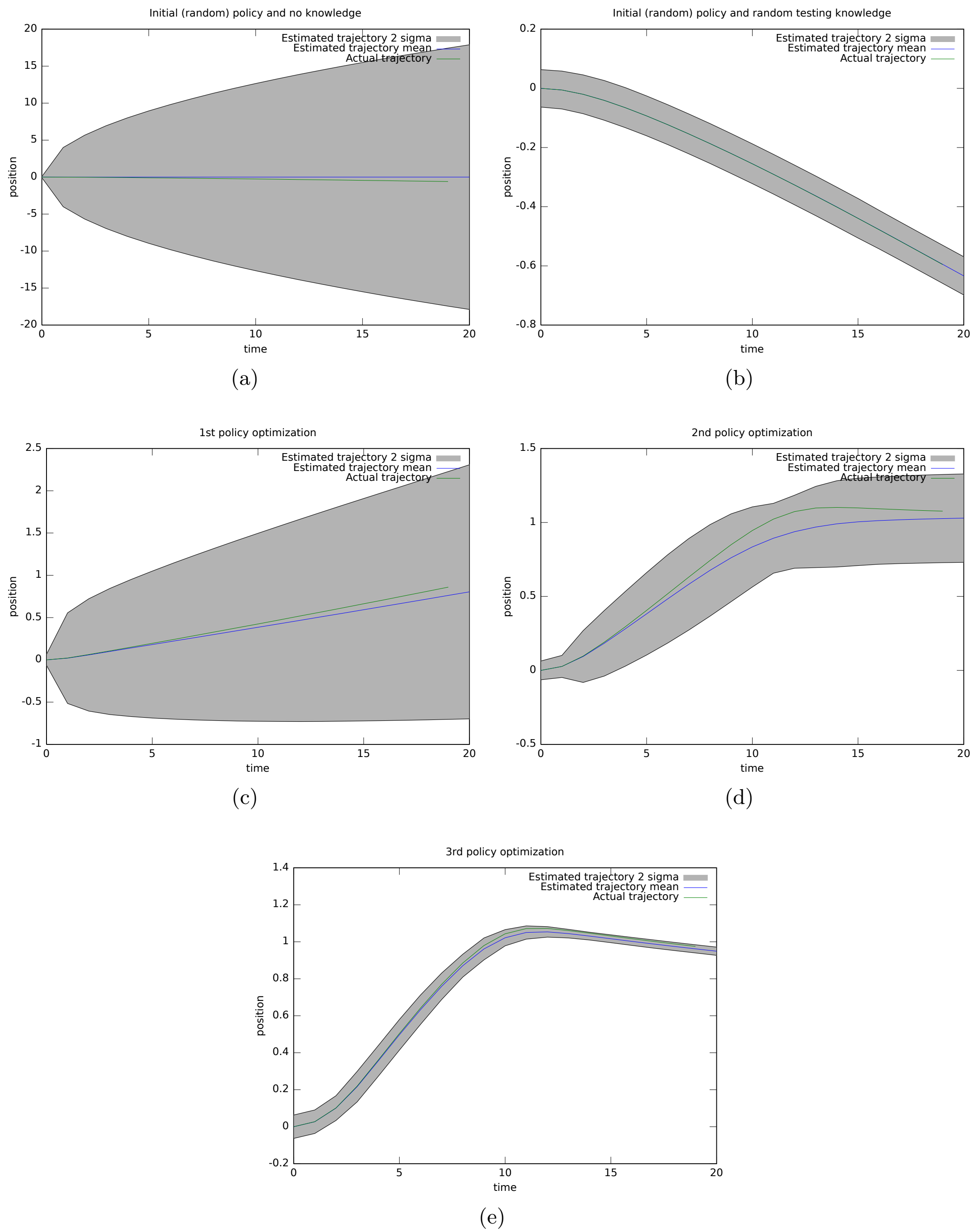
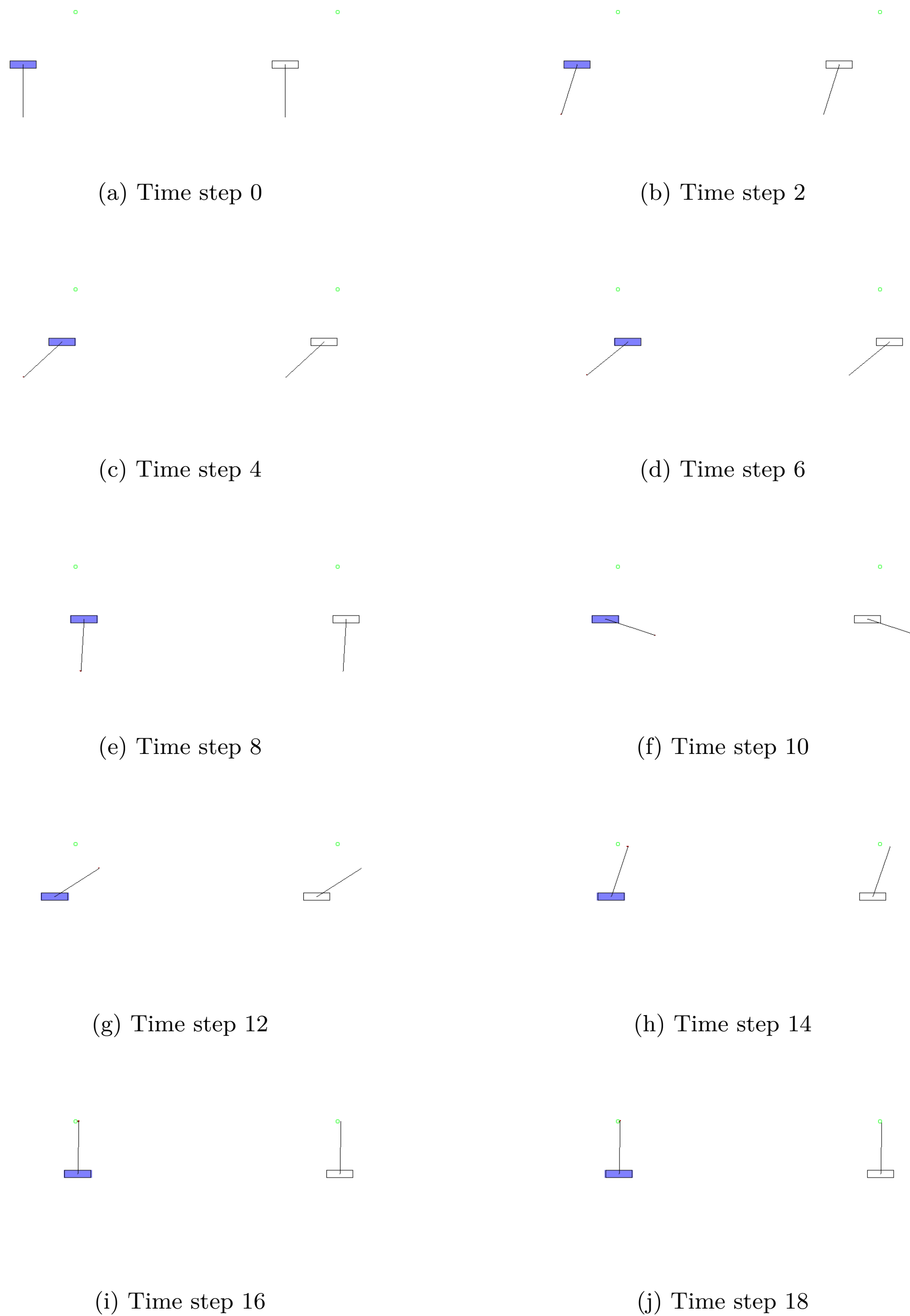



Figure 12: Trajectory of an optimized policy in the simplified scenario, where the full covariance matrices were computed during the optimization. As can be seen, the policy manages to significantly decrease the position's uncertainty when reaching the goal state.



(a) Time step 0
(b) Time step 2
(c) Time step 4
(d) Time step 6
(e) Time step 8
(f) Time step 10
(g) Time step 12
(h) Time step 14
(i) Time step 16
(j) Time step 18

Figure 13: Final trajectory of the optimized policy. The left part of each image shows the predicted state, the right part shows the actual state. The green circle is the target for the pendulum. For the full animation see the video clip that comes with this report.

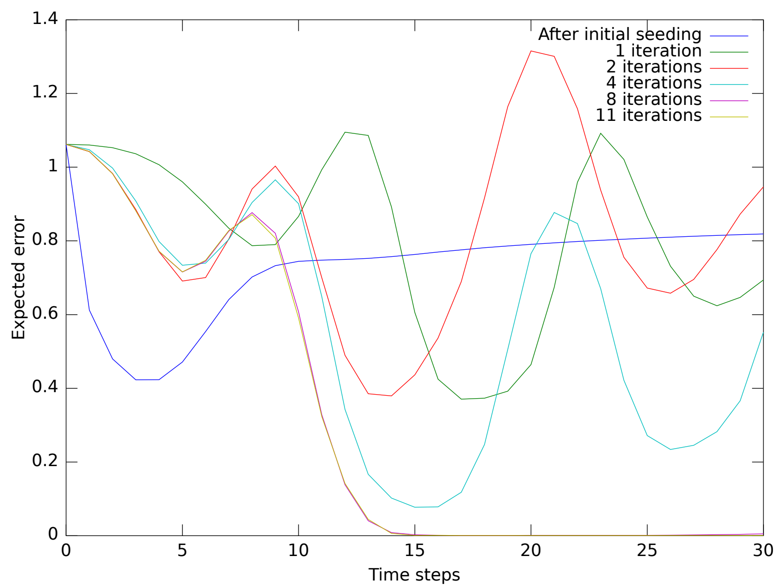



Figure 14: The expected errors in each time step for different numbers of iterations.

## References

[Deisenroth2010] Marc Peter Deisenroth, Dissertation “Efficient Reinforcement Learning using Gaussian Processes”

[Sultan2003] Khalil Sultan, “Inverted Pendulum - Analysis, Design and Implementation”

[NumericalRecipes2007] W.H.Press S.A.Teukolsky W.T.Vetterling B.P.Flannery, “Numerical Recipes - The Art of Scientific Computing - Third Edition”

[cephes] “Cephes Mathematical Library” http://www.netlib.org/cephes/