UNIVERSITAT POLITÈCNICA DE VALÈNCIA

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

School of Industrial Engineering

Exploring Bayesian Optimization Methods on a Ball Pushing Robotic Application on a Golf-like Terrain

Master's Thesis

Master's Degree in Industrial Engineering

AUTHOR: Deruytter, Hannes

Tutor: Armesto Ángel, Leopoldo

ACADEMIC YEAR: 2023/2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

MASTER'S DEGREE IN INDUSTRIAL ENGINEERING

Exploring Bayesian Optimization Methods through Ball Navigation on the Trajectory

AUTHOR: HANNES DERUYTTER

TUTOR: LEOPOLDO ARMESTO ÁNGEL

Academic year: 2023-2024

UNIVERSITAT POLITÈCNICA DE VALÈNCIA

ESCUELA TÉCNICA SUPERIOR INGENIERÍA INDUSTRIAL VALENCIA

TRABAJO FIN DE MASTER EN INGENIERÍA INDUSTRIAL

Aplicación de métodos de optimización Bayesiana en un robot que empuja una pelota en un terreno tipo golf

AUTOR: HANNES DERUYTTER

TUTOR: LEOPOLDO ARMESTO ÁNGEL

Curso Académico: 2023-2024

Acknowledgment

First, I want to thank my parents, my family, and my friends for the support, love and encouragement that has been a real cornerstone of my journey.

I am deeply thankful to KU Leuven, my home university at Belgium, for providing me with qualitative lessons and study progress along the years. This qualitative education and experience shaped my academic and personal development.

I also want to thank my supervisor Leopoldo Armesto Ángel for his guidance and mentorship.

Abstract

Exploring Bayesian Optimization Methods through Ball Navigation on the Trajectory

H. Deruytter

This thesis explores the use cases of Bayesian optimization, in this example for optimizing robot policy parameters of a robot that pushes a ball on a golf-like terrain. On a simulated environment, the trajectory of the ball will be traced and policy parameters like the velocity and the pushing angle will be optimized to achieve the optimal performance.

In order to achieve this, we plan to use Bayesian optimization methods and thus the performance index to be optimized will be unknown but assumed that can be evaluated through experiments and modeled as a Gaussian process.

As part of the tasks of this Master Thesis will include to determine which is an optimal setup so that BO (Bayesian Optimization) methods can be applied to this problem, including the golf-terrain design and the performance index to optimize.

Resumen

Aplicación de métodos de optimización Bayesiana en un robot que empuja una pelota en un terreno tipo golf

H. Deruytter

Esta tesina de máster explora el uso de técnicas de optimización bayesiana para la optimización de parámetros de una política de un robot que empuja una pelota en un terreno similar al golf. En un entorno simulado, se trazará la trayectoria de la pelota y se optimizarán parámetros tales como la velocidad y el ángulo de empuje para lograr el rendimiento óptimo.

Para lograr esto, planeamos utilizar métodos de optimización bayesianos y, por lo tanto, se desconocerá el índice de rendimiento a optimizar, pero se asumirá que puede evaluarse mediante experimentos y modelarse como un proceso gaussiano.

Como parte de las tareas de esta tesina se incluirá determinar cuál es la configuración óptima para que los métodos BO puedan aplicarse a este problema, incluyendo el diseño del terreno de golf y el índice de rendimiento a optimizar, que a priori son desconocidos.

Table of contents

List of figures

List of tables

1 Introduction

1.1 The challenge of optimization

Optimizing the performance of a model is the cornerstone of successful machine learning. It involves finding the ideal configuration of parameters that minimize the chosen error metric and maximizes a desired outcome. Consider a self-driving car as a recent example of optimization. Its image recognition model needs optimized parameters to differentiate the different road users, like bikers or pedestrians, from obstacles in real time. Even minor errors in the optimizations from the optimal configuration could have catastrophic consequences.

1.2 Motivation for Bayesian optimization

Traditional optimization techniques like grid search or random search, which are also discussed in this thesis, are conceptually straightforward, but face significant limitations. While traditional methods often struggle to balance exploration (finding promising new regions) and exploitation focuses on areas likely to contain the optimum). Bayesian optimization addresses this by strategically selecting the most informative data points to evaluate, what leads to a more efficient search process. As we take the computational time to perform the tests as an expensive factor, bayesian optimization can help to minimize the number of function evaluations and thus the cost.

1.3 Objectives

As the objective for my thesis, 2 objectives are discussed here. As a first objective, the design for the golf-like terrain is made to get an optimal parkour for the bayesian optimization application. This section explores how the design of the golf-like terrain impacts the effectiveness of Bayesian Optimization (BO). Here the aim is to identify a terrain configuration that suits the BO.

As a second objective, different BO configurations will be compared to discuss what the best parameters are to perform a bayesian optimization. By evaluating the impact of kernel functions, acquisition functions and hyperparameter settings on the optimization, we can establish guidelines for selecting optimal BO configurations for similar robot policy optimization tasks.

2 Literature study

2.1 Machine learning

In the last few years, the field of machine learning has had a big growth due to the technological growth of speech recognition, image recognition and recommendation systems, and attracted many researchers and practitioners. With the exponential growth of data amount and the increase of model complexity, optimization methods in machine learning face increased challenges. (Sun et al., 2020)

2.1.1 Overview of ML

ML or machine learning is a subfield of artificial intelligence that focuses on enabling computers to learn from data and improve their performance over time. Just like the bayesian optimization that will improve their next guesses every time it tries a new angle and velocity, see next chapters. Unlike traditional rule-based systems, ML algorithms learn patterns from examples allowing them to predict without explicit programming. Learning in this context is understood as inductive inference, where one observes examples that represent incomplete information about some statical phenomenon (Rätsch, 2004).

In unsupervised learning, the primary objective is to unveil hidden patterns, such as clusters, or to identify anomalies within the data, like unusual machine behavior or network intrusion. On the other hand, supervised learning has a label associated with each example, representing the solution to a specific inquiry about that example. If the label is categorical, the task is termed a classification problem; otherwise, for continuous labels, it is referred to as a regression problem.

When using these examples along with their labels, the focus lies on predicting answers for new instances before they are observed. Therefore, learning here means not just memorization but also the ability to generalize to unseen scenarios. In this case of the thesis, the bayesian optimization tries to predict unseen scenarios with the use of a gaussian process.

2.2 Supervised classification

A part of machine learning is the classification, also referred to as pattern recognition, where one attempts to build algorithms capable off automatically constructing methods for distinguishing between different examples, based on their differentiating patterns. (Rätsch, 2004)

(Watanabe, 1985) describes a pattern as "the opposite of chaos, it is an entity, vaguely defined, that could be given a name." These patterns could be human faces, text documents, EEG signals, or DNA sequences associated with specific diseases. Imagine this example, you have a bunch of data (X), and you want to figure out what category (Y) each piece of data belongs to. For example, the data could be pictures and you want to know if each picture is a cat or a dog (Y).

To do this, you show some examples (training data) to a special program (classification algorithm). The program learns from these examples and builds a rule (functional relationship) that helps it guess the category (Y) of new, unseen data. The goal is for the program to get good at guessing the right category for these unseen pictures (data). Thus, the input to a pattern recognition task can be viewed as a 2-dimensional matrix, whose axes are the examples and the features. Pattern classification tasks are often divided into several sub-tasks: Data collection and representation, Feature selection and/or feature reduction and Classification.

2.3 Overview of traditional optimization methods

Traditional optimization methods have a wide range of algorithms designed to find the minimum or maximum value of a function within a search space. These methods play a role in scientific and

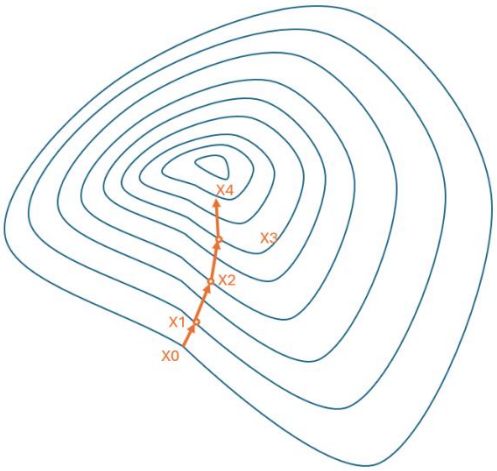
engineering disciplines, including robotics, machine learning, etc... In chapter 2.3, we will shortly overview some traditional optimization methods commonly used in research.

2.3.1 Gradient-based methods

Gradient-based methods is an optimization algorithm used to find a minimum or maximum of a function. They achieve this by iteratively moving in the opposite direction of the function's steepest ascent, a value called the gradient. This concept is like rolling down a hill - you naturally move in the direction of steepest descent. (*Optimization*, 2023)

Gradient descent, the most common gradient-based method, starts with an initial guess and then keeps adjusting that guess based on the gradient. Imagine yourself in a bumpy landscape; the gradient tells you which way is most downhill from your current position. By taking small steps in that direction, you eventually reach the lowest point, which is the minimum of the function. Examples of gradient-based methods include Gradient Descent and Stochastic Gradient Descent. ("Gradient Descent," 2024)

To explain the gradient descent in more detail, the reader can see FIGURE below, this image can for example describe a valley with the circle in the middle of the lowest point. Imagine a pedestrian walking to the deepest point of the valley.



To explain how the gradient based method for optimization finds the minimum point of the valley, we must understand formula below:

$$b = a - \gamma \cdot \Delta f(a) \qquad (1)$$

Legend:

- b: next position of our optimization algorithm (vector)
- a: current position (vector)
- γ: size of the step taken
- $\Delta f(a)$: direction of the steepest descent

This formula represents how the next position (e.g., b=X1) is computed by subtracting the current position (e.g., a=Xo) with a scaled vector ($\gamma \cdot \Delta f(a)$) in the direction of the deepest descent. In simple words, when optimizing through the gradient-based methods, it adjusts the current position by taking steps proportional to the gradient of the function at that position, thereby moving towards mostly a local minimum or absolute minimum if the function is convex and possesses a single minimum. This iterative process continues until a satisfactory convergence criterion is met, effectively guiding the optimization algorithm towards an optimal solution.

2.3.2 Direct search methods

Pattern search, also known as direct search, is a type of optimization algorithm well-suited for problems where calculating a function's slope is difficult or impossible. Unlike gradient-based methods, they rely solely on evaluating the function itself at different points and do not make use of derivatives.

Direct search methods belong to the class of derivative free optimizations (DFO). Thus, direct search methods are ideal for functions where we can only evaluate their outputs (like a black box) but do not have much other information. This is because they do not rely on complex calculations or properties of the function itself. (Kungurtsev et al., 2023)

In this thesis, this method is used to search for an objective function that could be interesting to perform a bayesian optimization on.

2.3.3 Linear programming

This optimization, linear programming, is a mathematical modeling technique in which a linear function is maximized or minimized when subjected to various constraints. This technique has been helpful for guiding quantitative decisions in business planning, in industrial engineering, and a bit in the social and physical sciences.

Linear programming finds optimal solutions to problems by minimizing or maximizing a linear objective function, subject to linear constraints. It was first used in the 1930's and gained popularity with George Dantzig's simplex method. As problems grew more complex, Leonid Khachiyan and Narendra Karmarkar developed polynomial-time algorithms to handle them. (*Mathematical Programming | Optimization, Algorithms & Models | Britannica*, 2024)

2.4 Introduction to Bayesian Optimization

To understand how and why we use the Bayesian optimization, we can take a deeper look at the Bayesian optimization, and how it can optimize our solution for the speed and the angle in a more efficient way. This includes a discussion on its theoretical foundations and practical considerations in this thesis example.

2.5 Understanding the Optimization Problem

To understand why the Bayesian optimization is used, we look at different problems we see in daily life or for example in the realm of software development. In this first example, the optimization of parameters in products like IBM ILOG CPLEX has become a complex challenge due to the many parameters that have to be known to use the program. This solver has 76 free parameters, which this designer must tune manually, that is of course too much to handle. In this example with IBM ILOG CPLEX, it could be a promising idea to optimize all the 76 CPLEX parameters to improve healthcare delivery. As a second example, in massive online games involving these 3 parameters, content providers, users and the analytics company that sits between them. In this example, it could be a wise idea to optimize the user experience and maximize the content provider's revenue.

In the example of this research, Bayesian optimization is tried out for a golf ball on a terrain to get a deeper understanding of how this optimization works. (Shahriari et al., 2016)

2.6 Gaussian Process Regression

In the next chapters of this thesis, we will use the MATLAB function fitrgp. To understand this function, we first need to understand the Gaussian process regression (GPR). These models are nonparametric, kernel-based probabilistic models. It provides a framework for modelling the relationship between input variables and x and output variables y, in this case the angle and velocity are the input variables, and the solution J is the output variable y.

If we consider the training set {(xi, yi); i=1, 2, ..., n}, where xi∈ℝd and yi∈ℝ are the input and output variables. A GPR model addresses the question of predicting the value of a response variable y_new, given the new input vector x_new and the training data (known start data). So, a Gaussian Process is just a set of random variables, such that any finite number of them have a joint Gaussian distribution. (*Gaussian Process Regression Models - MATLAB & Simulink - MathWorks Benelux*, n.d.)

To shortly have a quick understanding of what the GP regression does, FIGURE Shows how a GP regression with 0 and 4 trained points looks like. You can see that the 4 trained points already make a good guess of how the function will look alike. The red line shows the predicted mean value, while the shaded gray region shows the uncertainty of the predictions (2 standard deviations from the mean).

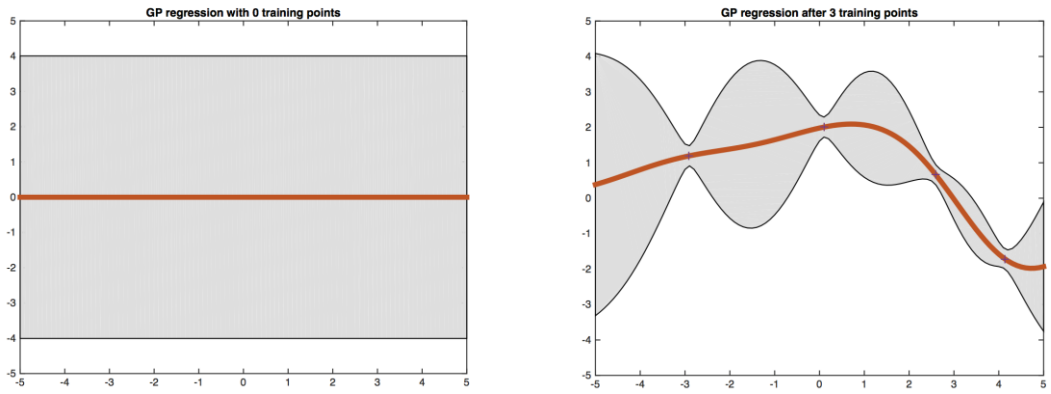


Figure 1 GP regression after 0 and 3 training points (1 dimensional). (Gaussian Process, *n.d.)*

In the next chapters, we will discuss how Bayesian optimization helps to choose which points to test out first to find a quick maximum or minimum value.

2.7 Basic Concepts of Bayesian Optimization

2.7.1 Objective Function

In Bayesian optimization, the objective function $f(x)$ represents the performance metric aimed to be optimized. In this thesis example, $f(x)$ could function as the distance traveled by a golf ball, for

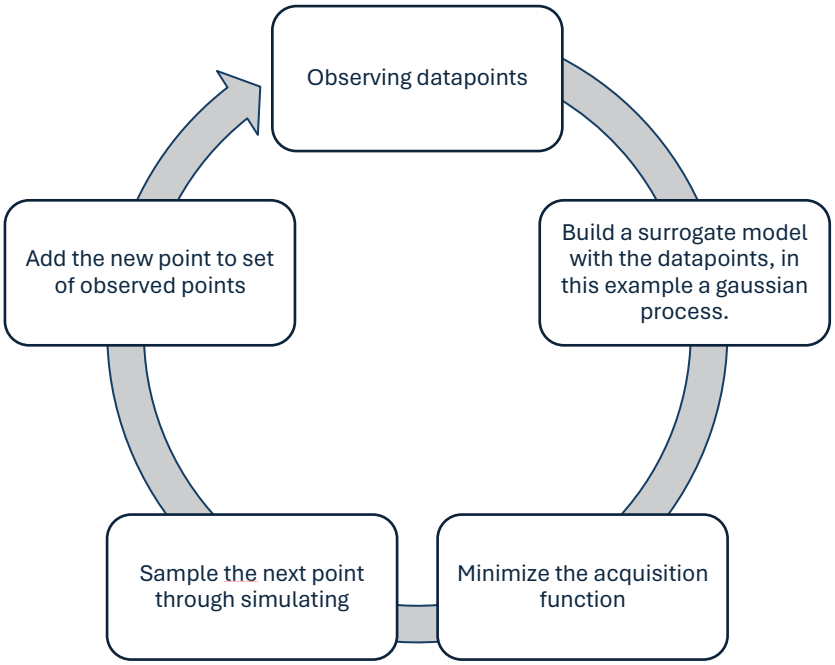


Figure 2 Steps of the Bayesian Optimization

the distance to the end goal and the route it takes. The design space X includes all configurations of the parameters like the angle or velocity, which influences the behavior of the ball on the golf terrain. In Figure 2, you can see the main steps that will be discussed for the bayesian optimization starting with observing datapoints.

2.7.2 Global maximization/minimization

The goal is to find the best configuration of the robot policy parameters, such as velocity (v) and the angle (α) to achieve the optimal results in terms of ball trajectory, distance covered, …

The first important part is a probabilistic model that is like a representation of the objective function we are trying to find the best value for. This model is a starting guess about how the objective function behaves, mostly made by a gaussian process. In this example we will start with some starting guesses or even 0 points of input with the help of MATLAB and coppeliaSim.

The second part is to measure how good our sequence of guesses is, called the "loss function". This will tell us how far off we are from finding the best solution, in our case we want to minimize this loss function to make the best possible guesses. After we make each guess and see the outcome, we update our initial guess about the objective function. This update will give a better idea of what the objective function might really look like. Moving from the initial guess to a more informed guess helps us make better guesses in the future. You can see an example of this process in Figure 3 (Shahriari et al., 2016) / (Galuzzi et al., 2020), where algorithm 1 is being processed.

You can see in the next code "algorithm 1", that we search for the maximum in the acquisition curve. In the next chapters we will discuss how Bayesian optimization works for more variables, including Velocity and the Angle of which the ball is pushed from. This will create a 3d surface, where the solution J is to be minimized can be predicted based on the effects of both velocity (V) and angle (α) using a Gaussian Process Regression or GPR model. Here the GPR will take an input vector containing both velocity and angle for predicting the output value J.

```
Algorithm 1: Bayesian Optimization for maximization
1: for n = 1, 2, ... do
2:     select new x_{n+1} by optimizing acquisition function α
         x_{n+1} = argmax_{x} α(x; D_n)
3:     query objective function to obtain y_{n+1}
4:     augment data D_{n+1} = {D_n, (x_{n+1}, y_{n+1})}
5:     update statistical model
6: end for
```

- n represents the iteration number.
- x_{n+1} is the new query point selected for iteration n+1.
- α is the acquisition function used for selecting new query points.
- D_n represents the set of data collected up to iteration n.
- y_{n+1} is the output of the objective function at the new query point x_{n+1}.
- D_{n+1} is the updated set of data after including the new observation.
- "argmax" denotes the argument that maximizes the acquisition function.

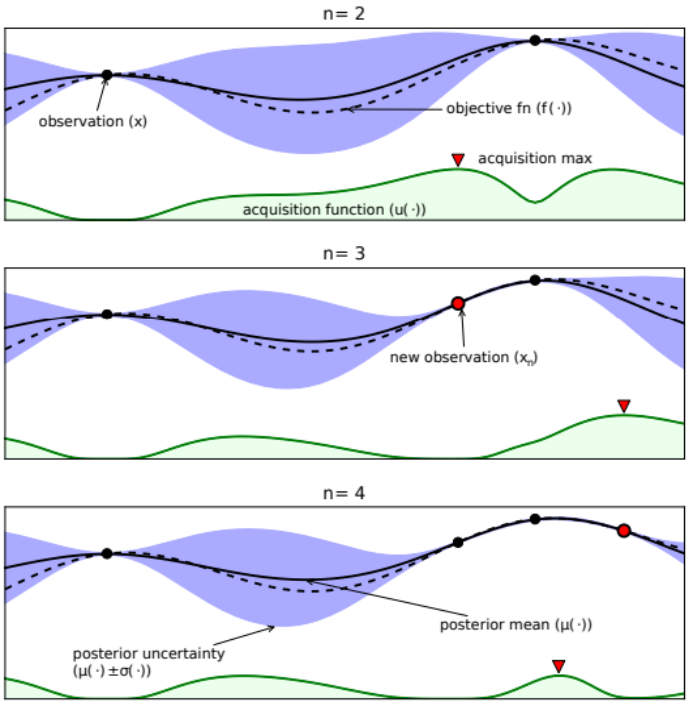


Figure 3 Illustration of the Bayesian optimization procedure over three iterations.

2.8 Configuration of the Bayesian Optimization

2.8.1 basic kernels

This chapter will discuss the basic kernels used in gaussian processes. Also see chapter 2.6. I would like to acknowledge the work of David Kristjanson Duvenaud at the University of Cambridge for his insightful information on the use of kernels. (Duvenaud, n.d.)

2.8.1.1 Squared exponential kernel

This kernel, SE, also known as the radial basis function kernel, has the form:

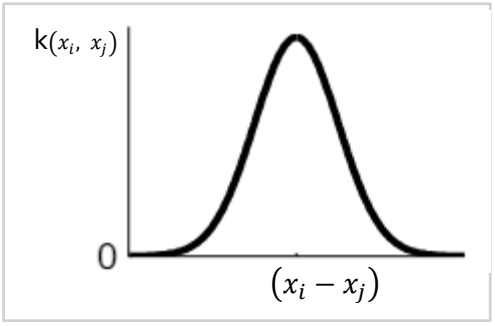


Figure 4 Squared exponential kernel

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \exp\left(\frac{-(x_i - x_j)^T (x_i - x_j)}{2\,{\sigma_l}^2}\right) \quad (2)$$

Legend:

- $\mathrm{k}(x_i, x_j)$: Kernel or covariance function
- σ_f^2: Output variance
- σ_l: length scale parameter

The SE kernel is the default kernel of the fitrgp() function because it has suitable properties. It is universal and is good for most functions. It has 2 parameters, The length scale parameter σl determines the smoothness of the function modeled by the Gaussian process. A smaller σl: leads to more rapid fluctuations, while a larger σl: results in smoother variations. The output variance parameter σ_f^2 controls the overall variability of the function sampled from the Gaussian process. It acts as a scale factor, adjusting the amplitude or variability of the function.

2.8.1.2 exponential kernel

This kernel can be specified in the bayesopt() function by adding the 'KernelFunction', 'exponential' name-value pair argument. This function is defined by:

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{r}{\sigma_l}\right)$$
$$\text{with } \mathrm{r} = \sqrt{(x_i - x_j)^T (x_i - x_j)} \quad (3)$$

Legend:

- r: Euclidean distance between xi and xj

2.8.1.3 Rational quadratic kernel

The RQ kernel or the rational quadratic kernel is another commonly used kernel in GPR. It has the form:

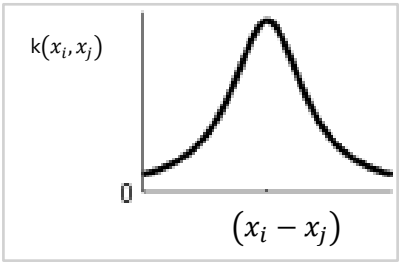


Figure 5 Rational quadratic kernel

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \left(1 + \frac{r^2}{2\,\alpha\,{\sigma_l}^2}\right)^{-\alpha} \quad \text{with } \mathrm{r} = \sqrt{(x_i - x_j)^T (x_i - x_j)} \tag{4}$$

Legend:

- α: positive-valued scale-mixture parameter that determines the relative weighting (when α→∞, RQ = SE)

This kernel is equivalent to adding together many SE kernels with different length scales. It gives additional flexibility introduced by the alpha parameter. The length scale parameter σ_l controls the smoothness of the function, like the SE kernel. However, the additional scale mixture parameter α allows the RQ kernel to adjust the balance between short-range and long-range correlations in the data. Smaller values of α lead to smoother functions, while larger values result in more complex and potentially oscillatory functions. In the thesis's case, we have a less smooth curve, and thus a more complex function.

2.8.1.4 Periodic kernel

This kernel is used to capture periodic patterns in data. It has the form:

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \exp\left(-\frac{2\,sin\left(\frac{\mathrm{pi}\,|x_i - x_j|}{p}\right)^2}{{\sigma_l}^2}\right) \tag{5}$$

Legend:

- p: period parameter

This periodic kernel allows to model functions which repeat themselves exactly. The parameter p determines the distance between repetitions of that function.

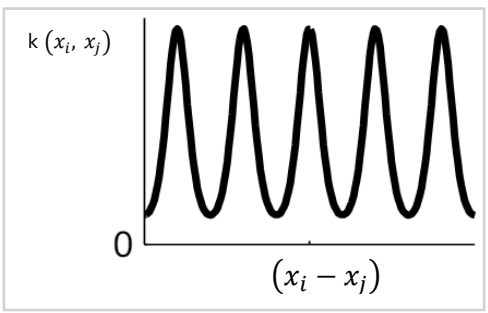


Figure 6 Periodic kernel

2.8.1.5 Matern52

The `'matern52'` kernel is a kernel of the Matérn families like matern52 matern32 ardmatern52, etc... This covariance function is defined by:

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \left(1 + \frac{\sqrt{5}\, r}{\sigma_l} + \frac{5}{3\sigma_l}\, r^2\right) \exp\left(-\frac{\sqrt{5}\, r}{\sigma_l}\right) \qquad (6)$$

$$\text{with } \mathrm{r} = \sqrt{(x_i - x_j)^T (x_i - x_j)}$$

2.8.1.6 Ardmatern52

The `'ardmatern52'` kernel is a specific type of kernel used here in the gaussian process regression. It belongs to the family of Matérn kernels. Which are characterized by their smoothness and differentiability properties. The `'ard'` prefix stands for automatic relevance determination, what indicates that the kernel has separate length scale parameters for each input dimension, what allows it to adjust the importance of different input dimensions. The `'52'` stands for the smoothness parameter of 5/2. This parameter controls how quickly the covariance between points decreases as the distance between them increases. The kernels Matern32 and ARDMatern32 are the same kernels but with a lower smoothness parameter.

$$\mathrm{k}(x_i, x_j) = \sigma_f^2 \left(1 + \sqrt{5}\, r + \frac{5}{3}\, r^2\right) \exp\left(-\sqrt{5}\, r\right) \qquad (7)$$

$$\text{with } \mathrm{r} = \sqrt{\sum_{m=1}^{d} \frac{(x_{im} - x_{jm})^2}{\sigma_m^2}}$$

Legend:

- σ_m^2: Separate length scale for each predictor m =1,2...d

2.8.2 Minimize the acquisition functions

In this chapter, other types of acquisition functions are discussed, that could also have been a good option to explore space effectively. For instance, Probability of Improvement (PI), Upper Confidence Bound (UCB) are commonly used acquisition functions used in Bayesian optimization.

2.8.2.1 Lower or upper Confidence bound

As this thesis wants to minimize the objective J, the lower confidence bound is a possible good acquisition function. This acquisition can be described as the function:

$$LCB(x;\, \lambda) = \mu(x) + \lambda\sigma(x) \qquad (8)$$

Legend:

$\mu(x) = the\ mean\ function\ of\ the\ gaussian\ process$

$\lambda = parameter\ to\ tune\ exploitation/exploration\ (negative\ with\ LCB)$ (default -2)

$\sigma(x) = uncertainty, captured\ by\ the\ standard\ deviation\ of\ the\ gaussian\ process$

This acquisition function is quite simple. As $\mu(x)$ and $\sigma(x)$ are both captured by the gaussian process, the only thing that defines the LCB is the parameter lambda. To explain how this function reacts, Figure 7 can be seen. In this example, a lambda value of 1.2 is chosen and you can see that the acquisition function or the UCB is slightly above the upper confidence interval. This is because

it takes 1.2 times the uncertainty. In the example of the thesis, so not like Figure 7, lambda is negative, so the minimum can be found with the LCB.

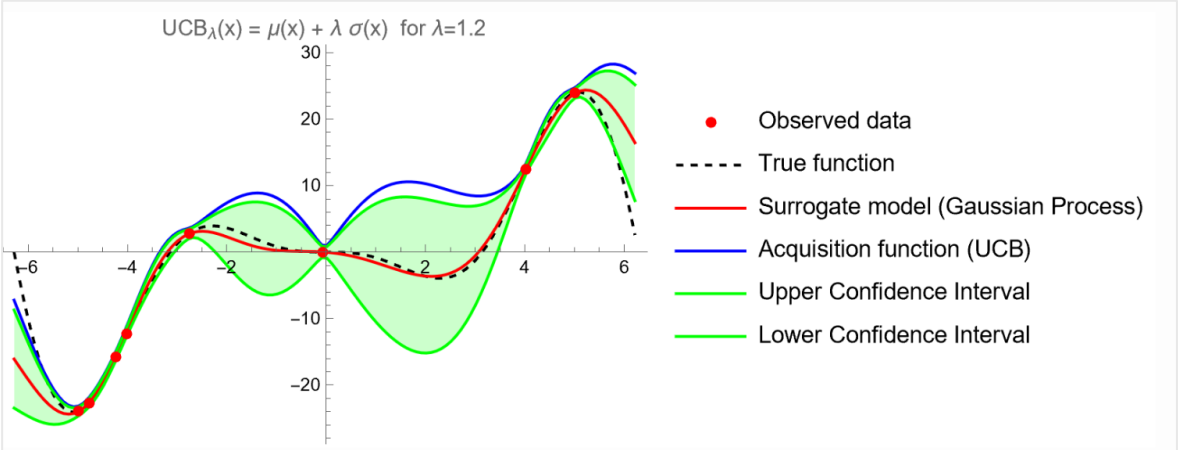


Figure 7 UCB for λ=1.2 (Kamperis, 2021)

2.8.2.2 Probability of improvement (PI)

The acquisition function PI takes the probability of improvement as the function to minimize. This function calculates for every x the probability of improvement, compared to the best point x* already found.

$$PI(x) = \Phi\left(v_Q(x)\right)$$

$$\text{Where } v_Q(x) = \frac{\mu_Q(x_{best}) - m - \mu_Q(x)}{\sigma_Q(x)} \tag{9}$$

Legend:

- Φ: CDF of the standard normal distribution
- $\mu_Q(x)$: Mean of the GP
- $\mu_Q(x_{best})$: Mean of the GP at position x_{best}
- $\sigma_Q(x)$: standard deviation at x
- m: estimated noise standard deviation

2.8.2.3 Expected improvement.

The expected improvement acquisition function can be defined as formula (10), where the first summation term is the exploitation term, and the second summation term is the exploration term.

$$EI(x,\xi) = \begin{cases} d \times \Phi(Z) + \sigma(x)\varphi(Z) & if\ \sigma(x) > 0 \\ 0 & if\ \sigma(x) = 0 \end{cases}$$

$$\text{Where } Z = \begin{cases} d/\sigma(x) & if\ \sigma(x) > 0 \\ 0 & if\ \sigma(x) = 0 \end{cases} \tag{10}$$

Legend:

- d: $f(x^*) - \mu(x) - \xi$ (in case of minimization), with f(x*) the best solution so far
- σ(x): standard deviation at x
- Φ: CDF of the standard normal distribution
- φ: PDF of the standard normal distribution
- ξ: exploration factor

2.8.2.4 Expected improvement per second

In scenarios where the evaluation time of a function is different across different regions, the bayesopt() function can here enhance efficiency by incorporating time weighted acquisition functions. This per second function aims to maximize improvement relative to the time spent. This is achieved by maintaining a bayesian model predicting evaluation time for each point in the search space. The formula is as follows:

$$EI_{pS}(x) = EI(x)/\mu_s(x) \tag{11}$$

Legend:

- $EI_{pS}(x)$: Expected improvement per second
- $EI(x)$: Expected improvement
- $\mu_s(x)$: Posterior mean of the timing gaussian process model

2.8.2.5 Expected improvement plus

The expected improvement plus is an extra function of MATLAB so that it can estimate whether there are overexploiting regions or not. This function facilitates a more balanced exploration and exploitation.

2.9 Advantages of Bayesian Optimization

The main advantage of Bayesian optimization is for optimizing expensive black box functions because it can efficiently locate global maximum or minimum without requiring gradient information or large number of evaluations. Also, the Bayesian optimization can be customized for specific tasks by adjusting parameters like the acquisition functions and training samples.

3 Openmanipulator X

The robot we are simulating in the CoppeliaSim program is the Openmanipulator-X, this robot has Open-Source software and hardware, which means that everyone can access it. The robot is compatible with OpenCR, this is an embedded controller developed for ROS or robot operating system. To simulate the real-life golf like terrain, we will use this robot to push the ball in the chosen direction and speed, although in this thesis, we will only discuss the virtual simulation of it. In Figure 8, you can see both the CAD files and the robot as a picture.

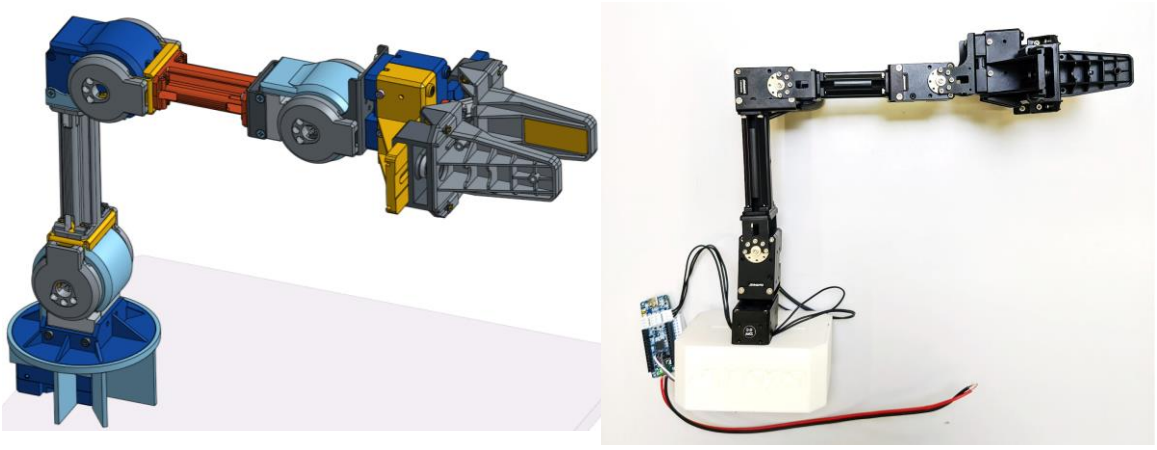


Figure 8 Openmanipulator X

The Openmanipulator X has 5 degrees of freedom, of which 1 is for controlling the gripper. This also means that the robot is unable to adjust the angle around the ball, but instead we need to move the ball, so it can be shot from the right angle. In Figure 9, you can see the visualization of the robot with the given rotating possibilities. As you can see it can only rotate around its main z-axis to change the angle of the ball, with θ0.

To control all the motors of the Openmanipulator X, the Dynamixel X430 is used, discussed in chapter 3.1.

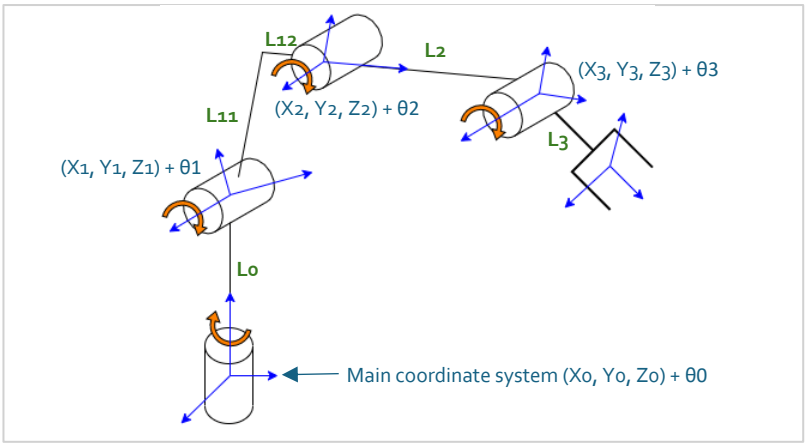


Figure 9 4-DOF robot arm

Joint number	Length (mm)
L0	77
L11	128
L12	24
L2	124
L3	126

Table 1 Openmanipulator dimensions

3.1 Dynamixel motors

The openmanipulator X uses the Dynamixel X series to control the robot, specifically the Dynamixel XM403-W350. These servo motors have a resulotion of 4096 pulses/revolution and has 6 operation modes including Current Control mode, velocity Control mode and position control mode, for controlling the current, velocity or position and supports PWM (pulse width modulation). The motors can handle a radial load of 40 N (at 10 mm (about 0.39 in) away from the horn) and a 20 N axial load and needs a recommended input voltage of 12 V. The communication between the motors and the OpenCR board is done by a TTL Level Multidrop BUS.

4 Methodology

4.1 Experimental Setup

To find a good parkour to explore the effectiveness of the Bayesian, multiple approaches in finding a satisfactory solution were tried that display the results or the objective in function of the angle and the velocity in a 3d graph. To get an understanding of what the solution means, a formula that balances 2 objectives is used:

$$\text{result } J = \alpha \times d\_last^2 + \beta \times \sum (d\_i)^2 \quad (12)$$

Legend:

- d_last: Distance from endpoint to last recorded point
- d_i: Distance from endpoint to any recorded point (i iterates over all points)
- α: Weighting factor for end goal proximity
- β: Weighting factor for trajectory efficiency

By using this formula, a solution can be found that evaluates both the importance of the endpoint and the importance of the trajectory efficiency with alpha prioritizing the end solution and beta prioritizing the trajectory.

4.2 CoppeliaSim (V-rep)

4.2.1 Introduction to CoppeliaSim

CoppeliaSim is a program that is used to simulate a golf ball that is pushed in each angle and speed. This software tool is used for both industrial and academic purposes. In this thesis, CoppeliaSim Edu, version 4.1.0. (rev. 1) was used.

4.2.2 Simulation set-up

In Figure 10, the parkour can be seen, made in CoppeliaSim. The light gray part can be seen as the terrain in which the ball will be pushed on. The flat figure below the starting point of the ball and the wall near the parkour is created with a cuboid and is attached to the terrain to ensure a good start. Because when using pure shapes like the cuboid, the simulator responds more naturally than mesh shapes like the terrain. In real life the terrain can be 3D printed and the walls + start position plank can be made with a laser cutter; in this example a wooden plate of 4mm (about 0.16 in) is chosen, which is suitable for laser cutting in the future.

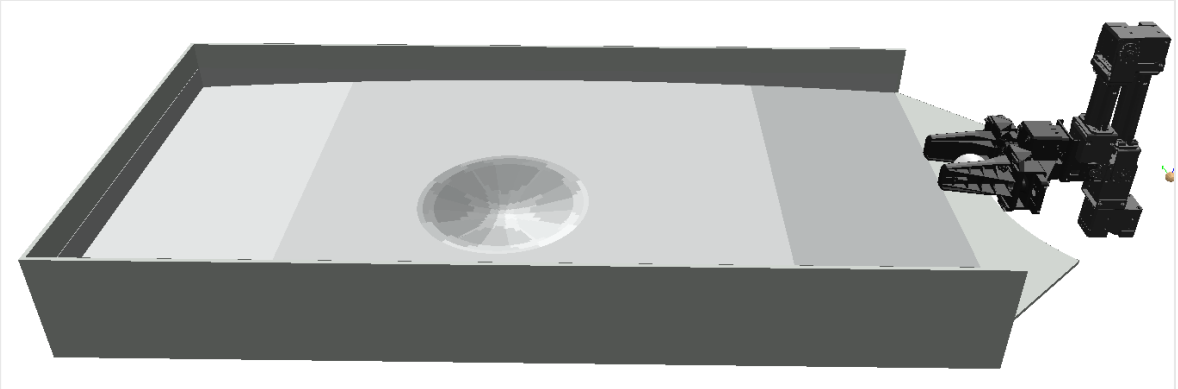


Figure 10 Simulation Set-Up

To simulate the robot gripper and the foosball ball (diameter 35mm), the gripper was replicated and the ball out of pure shapes (cuboids/sphere), so the program does not have difficulties calculating the interactions between the ball and the gripper (pure shapes). In Figure 11, all the elements are included in the simulation environment. Including the foosball Ball, the terrain, and the robot with 1 joint for horizontal movement.

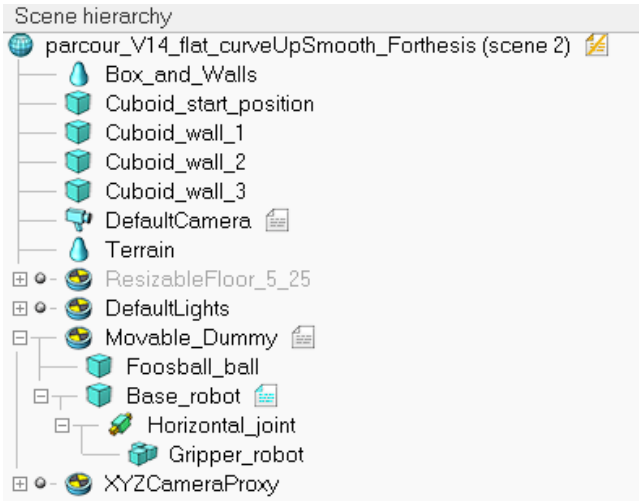


Figure 11 Scene hierarchy.

The joint (horizontal joint) and the position/rotation of the robot arm + ball (Movable Dummy) is controlled by remote API functions in MATLAB, see following chapters. There can also be seen that the gripper of the robot is attached to the joint, which is also connected to the base, in this case the base is displayed as a simple cuboid. In Figure 12, you can see the comparison between the robot in real life and the simulation.

It is not needed to simulate all servo motors and their joints, as we only must perform a horizontal movement at a certain speed, this can be controlled as a single prismatic joint. In future work, this can be extended in real life.

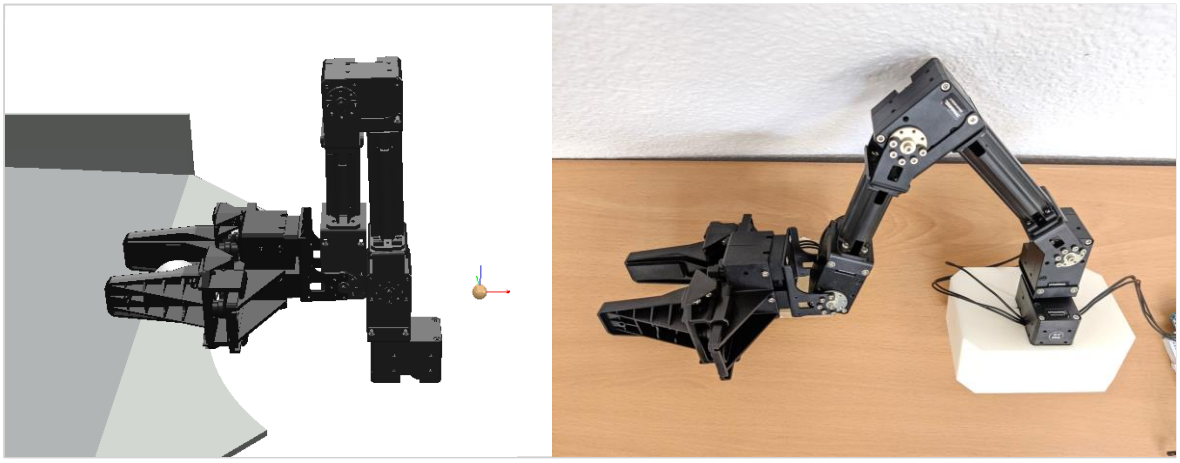


Figure 12 comparison between robot and CoppeliaSim.

4.3 Grid search for optimization

To take over the control of the joints and the rotation of the robot through MATLAB, we need to use remote API functions, here we will cover the commands that were used to remotely control the CoppeliaSim objects/joints and the API functions to retrieve the data of the ball. In the appendix, chapter 10.1 'MATLAB code for grid search', you can find the code to make a grid of all the collected points. In the code, we collect all the data for 30x30 points from -25 till 25 degrees

and from 0.1 till 1.5 m/s. In the code given in appendix chapter 10.1, we collect all the data points in a cell called Results that contains all the tests including all the data points per test, the distance to the endpoint and the solution J per test. This code is used in chapter 4.4

4.3.1 MATLAB connection to CoppeliaSim

Below is a snippet of MATLAB code from appendix chapter 10.1 that shows how we make the connection to MATLAB.

```
% Connect to CoppeliaSim
disp('Program started');
sim = remApi('remoteApi'); % Using the prototype file (remoteApiProto.m)
sim.simxFinish(-1); % Close all opened connections
clientID = sim.simxStart('127.0.0.1', 19997, true, true, 5000, 5);

% Check if connection is successful
if (clientID > -1)
    disp('Connected to remote API server');
    % ...
else
    disp('Failed to connect to CoppeliaSim');
end

% End communication
sim.simxFinish(clientID);
% Clean up
sim.delete();
```

This code initiates a connection between MATLAB and CoppeliaSim. Firstly, it closes all the opened connections and then with simxStart() it tries to connect to the IP address 127.0.0.1 and port number 19997, these are the standard of CoppeliaSim. If this connection is successful, it will display that it is connected, otherwise it displays failed to connect.

4.3.2 Object Handles

Below you can see the snippet code of appendix chapter 10.1 that takes over the handle of 3 objects, the ball, the dummy, and the joint. See chapter 4.2.2 for more information about these objects.

```
% Get handle to the object whose position you want to track
[~, objectHandleBall] = sim.simxGetObjectHandle(clientID, 'Foosball_ball',
sim.simx_opmode_blocking);
[~, objectHandleDummy] = sim.simxGetObjectHandle(clientID, 'Movable_Dummy',
sim.simx_opmode_blocking);
[~, prismaticJointHandle] = sim.simxGetObjectHandle(clientID, 'Horizontal_joint',
sim.simx_opmode_blocking);
```

These handles are needed to interact and control these objects programmatically. For example, the objectHandleBall handle enables us to track the x and y location of the ball, where x and y are the specific coordinates. See next code:

```
% Get object position
[~,    position]    =    sim.simxGetObjectPosition(clientID,    objectHandleBall,    -1,
sim.simx_opmode_blocking);


% Extract XY coordinates
x = position(1);
y = position(2);
```

Because we want to collect data synchronously while simulating, we also use the following code. simxSynchronous starts the synchronous mode and triggers it every time in the while loop with

simxSynchronousTrigger. This last function is in charge of informing CoppeliaSim to execute the next simulation step. See detailed code in appendix chapter 10.1.

```
sim.simxSynchronous(clientID,true); % Enables synchronous mode
% code in between
sim.simxSynchronousTrigger(clientID); % Trigger simulation step
```

4.4 Simulation environments

4.4.1 Use of SolidWorks

SolidWorks 2022 was used for the CAD files (computer-aided design) of the parkour elements used in the simulations. This software made it possible to create 3D models to create a golf-like terrain. These models were then exported in STL file format (Stereo Lithography), a widely used standard. We can import those compatible files into CoppeliaSim.

4.4.2 Random generated parkour

For the first experiment, a random parkour with randomly generated hills was tested as an initial parkour. The parkour was made using solid works, using the extrusion 'freeform'. To get an end point, a low point or pit was created where the ball can land in with coordinates (x: 83 mm, y: 336 mm). These coordinates determine the ball's distance to the end point through Pythagoras.

Figure 13 Random generated parkour in Solid Works and CoppeliaSim

In Figure 13, the random parkour is displayed, and the end coordinates are clearly visible as the lowest point in the parkour (see arrow). As we collect all the points (XY-coordinates) seen in Figure 14 at a speed of 2m/s and an angle of -60° → 60°, the routes the ball makes can be seen. To

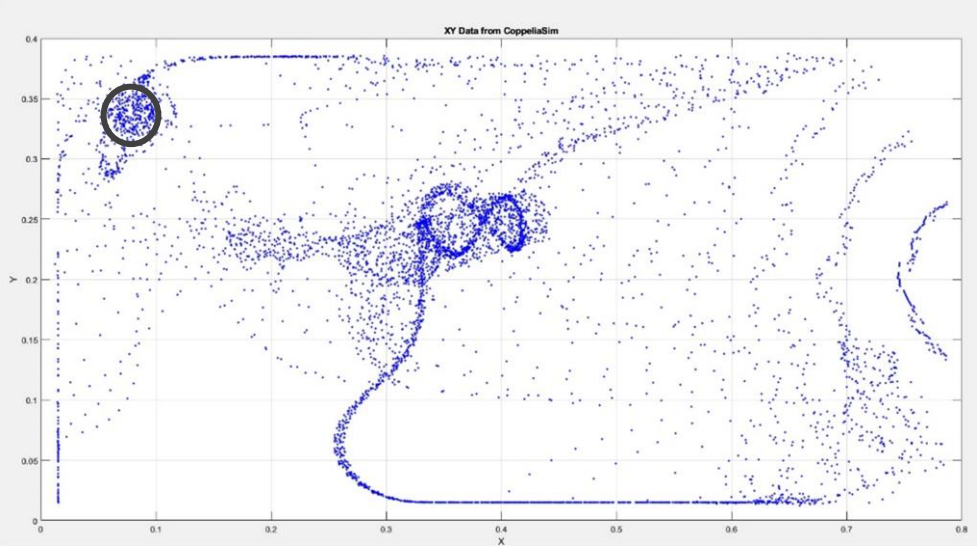


Figure 14 Simulation coppeliaSim

calculate the solution "J", all the distances are calculated from each point to the end coordinate, displayed in the circle of Figure 14. These distances are used for formula (12) to calculate solutions dependent on the end point and route it takes. Figure 14 only makes changes in the angle but not in the speed. For the next experiment, the speed was changed from 0.1m/s → 2m/s and the angle between -30° → 30°, this in both 10 x 10 experiments. The solutions are seen in Figure 15, in this example, the optimal solution is the one where the speed is 2 m/s (max speed) and the angle is -0.29 rad or approximately 16.66 degrees, aligning in the direction of the end point. However, this is not useful, because a Bayesian optimization would aim to confirm this already-known optimal outcome of achieving the maximum speed of 2 m/s but would only focus on optimizing the solution for the angle.

In this example 200 datapoints of each dt=10ms per test were collected, where the last collected point is the final distance. Here the importance of the route alpha, is 100 and the importance of the endpoint beta, is set on 1. This is used in formula (12) to calculate the solutions J per sample.

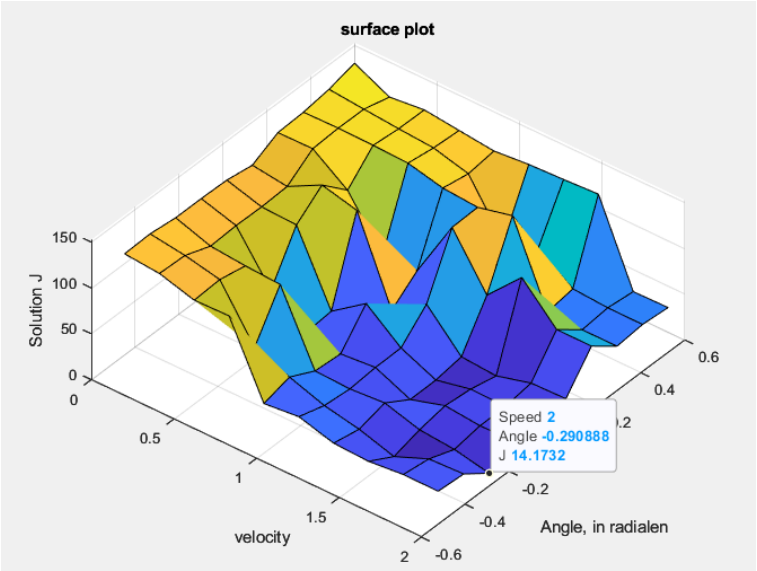


Figure 15 Solutions in 3d graph. (random parkour)

4.4.3 Flat parkour

In a second attempt, an easier flat surface is made. This should lead to a plot where the optimal solution is less predictable and forms a surface. The flat surface can be seen in "Figure 16 Flat surface". As with this end goal, if the ball rolls too fast it will roll over the endpoint and bounce back.

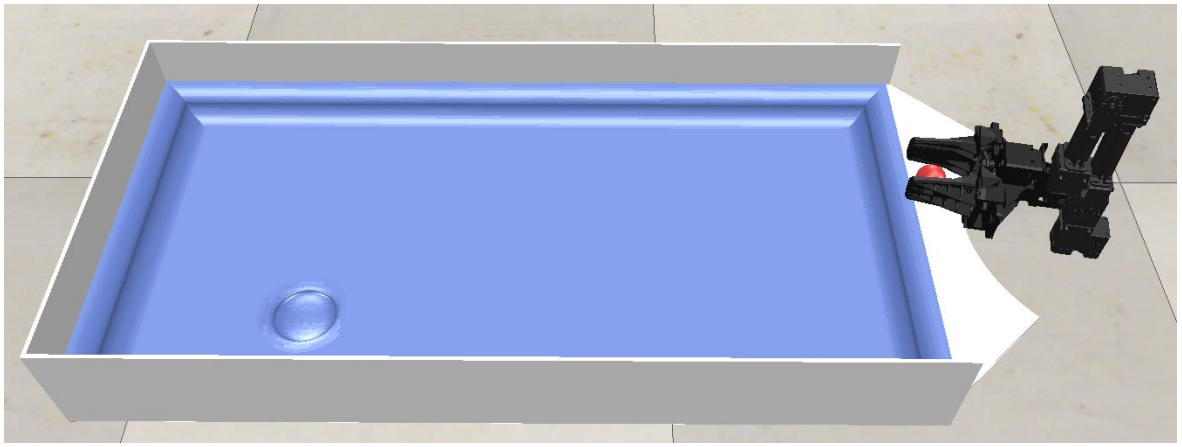


Figure 16 Flat surface.

In Figure 16, the velocity is changed from 0.1 → 2 m/s again and from -30° → +30° with a sample time of 10ms and the number of samples at 200, the end goal in this setup is x=0.443m and Y=0.117m. If we split the speed and the velocity into 25 parts, we can become a 3d graph with the solutions that has a 25x25, so 625 samples. In FIGURE 5, there can again been seen that this is not what is wanted. The optimal solution lays again at a certain angle and the maximum speed of 2m/s.

Per information, alpha is here 200 and beta 1, so for 200 points with a sample time of 10 ms, the route and the endpoint are of the same importance.

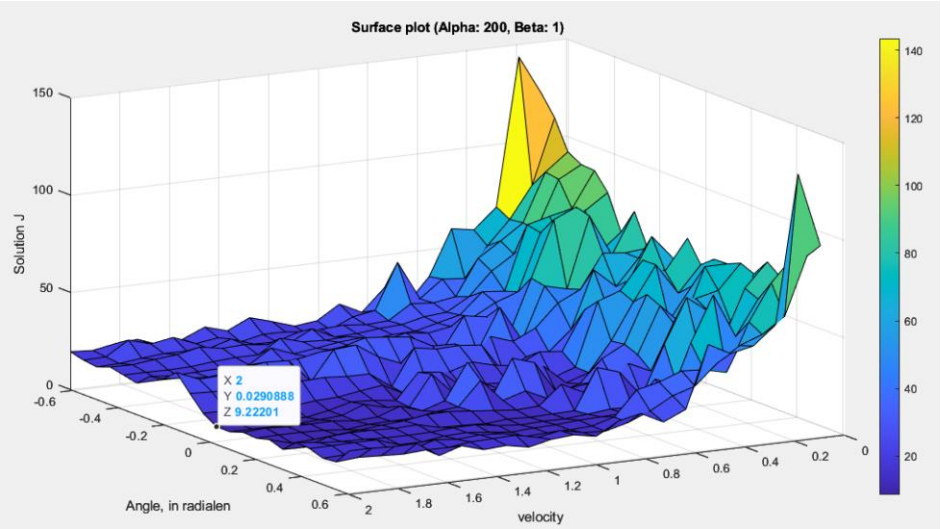


Figure 17 Solutions in 3d graph (flat parkour).

4.4.4 Flat parkour with different end coordinates

Given the limitations of the previous flat parkour configuration in generating a usable 3D graph, the end coordinates were modified to x = 154 mm and y = 374 mm. This modification aims to reposition the endpoint closer to the center of the parkour. Our objective is to prevent the ball from reaching the end goal with maximum speed, a scenario observed with the previous configuration.

In a good scenario, the ball should traverse over the end goal at a high speed and will not be able to get to the end goal at a low speed. This revised endpoint placement represents a potentially

helpful solution, so we can use this to optimize the speed and the angle with a Bayesian optimization. In Figure 18, the arrangement of my parkour is displayed.

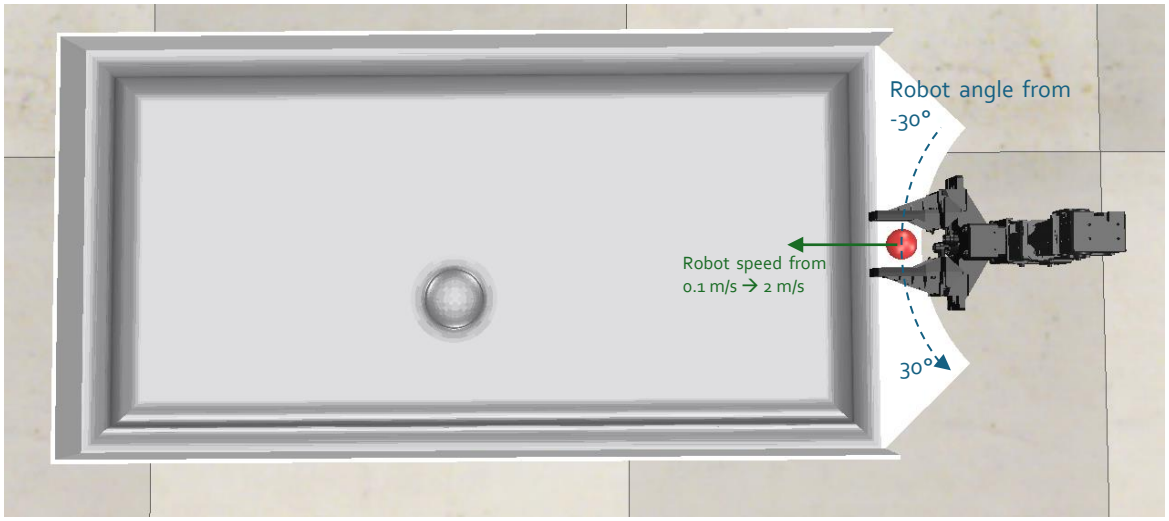


Figure 18 robot arrangement.

If we now want to perform the simulation, that runs the ball from -30° till 30° and the velocity from 0.1 m/s till 2m/s, each linear divided into 30 parts. That gives us a 3d graph of the solutions with 30 x 30 or 900 collected points. In Figure 19, you can clearly see some unusual behavior and even when having some slight changes in velocity, the result is different than the result of the test with a slightly bigger or smaller velocity. To explain this problem, the reader can refer to chapter 4.5.

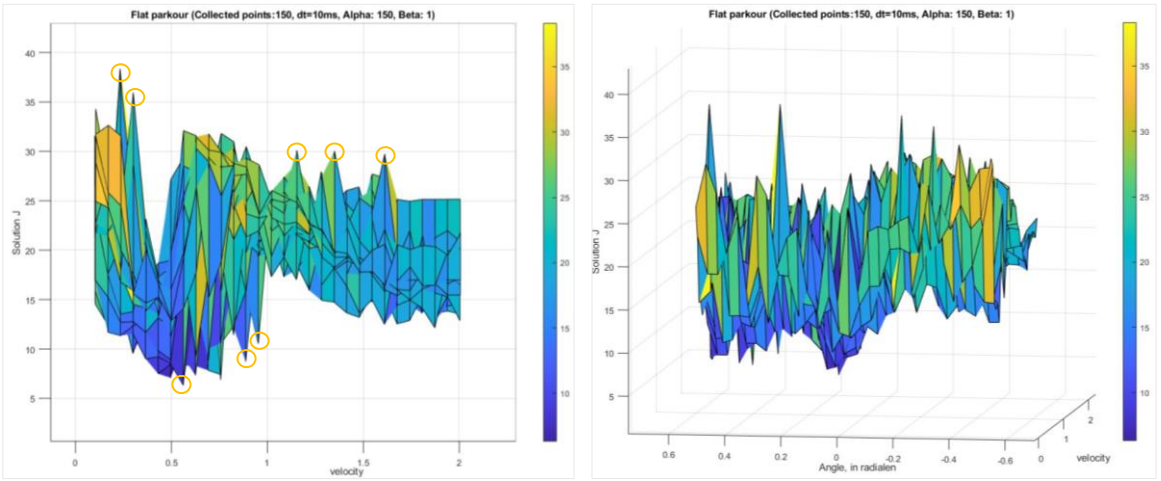


Figure 19 Flat parkour.

4.5 Limitations of CoppeliaSim

As a result of the peaks seen in 4.4.4, where it is clearly seen that a slight change in the angle can result in a substantial change of the result J. Additional tests were done to better understand how CoppeliaSim reacts. To see the accuracy of CoppeliaSim, we precisely controlled the robot's gripper velocity at 1 m/s while systematically changing its angle from -11.1° to -11.0° in 11 equal steps (0.01° increments). To get a better understanding of how the robot gripper is positioned, see Figure 20.

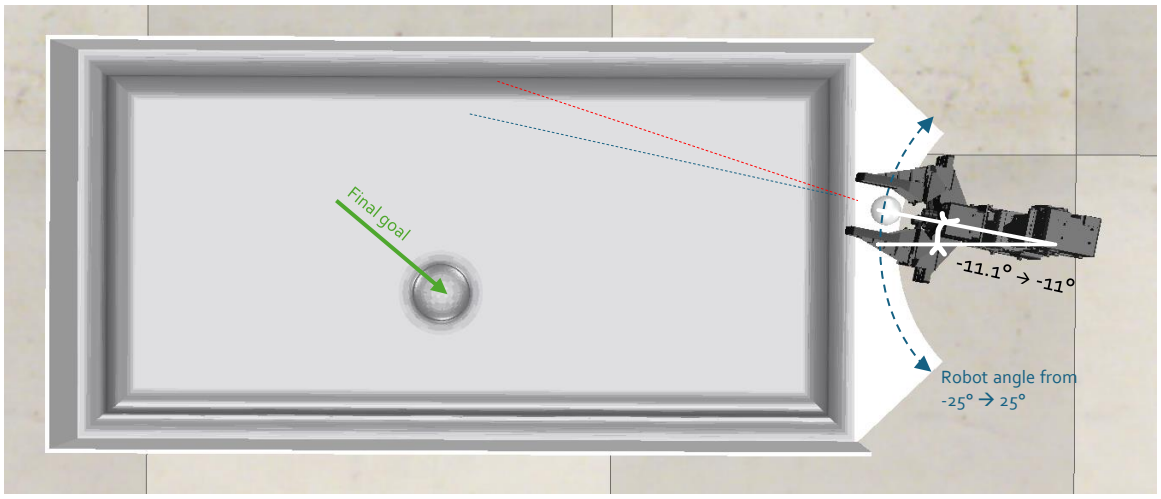


Figure 20 testing limitations of CoppeliaSim.

After doing eleven tests and plotting the resulting XY data on a graph, it becomes clear that CoppeliaSim struggles to accurately turn the robot and ball in increments of 0.01°. In Figure 21, the starting position of the ball deviated from the expected trajectory in three out of eleven trials (see zoomed in area of Figure 21). These deviations exceeded a displacement of more than 5 mm (about 0.2 in) despite the minimal intended angular change (0.01°) of the robot + ball. This issue persisted regardless of whether you controlled the robot and ball via the MATLAB remote API functions or manually within CoppeliaSim, so we can also conclude that it is not the MATLAB remote API functions that is responsible for the issue.

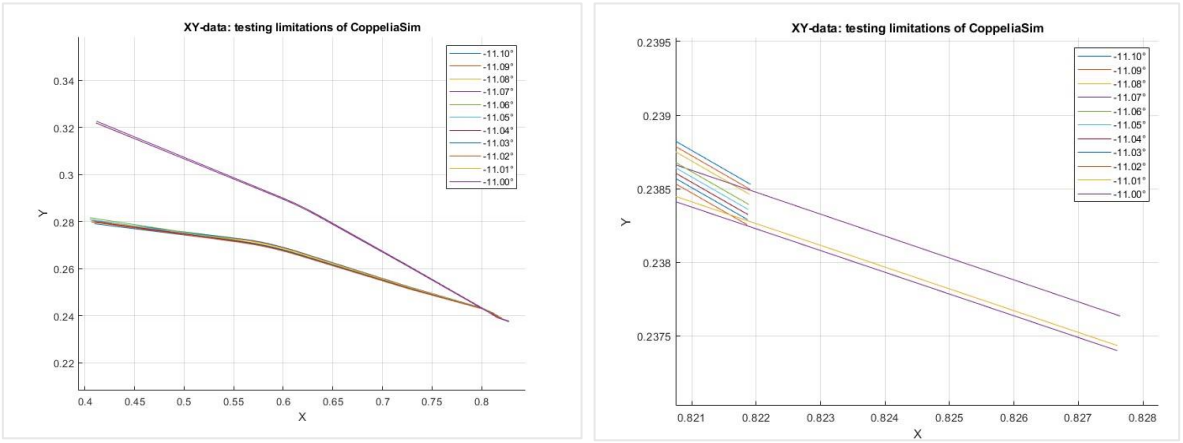


Figure 21 XY-data: testing limitations of CoppeliaSim at a speed of 1 m/s.

4.6 Choosing a Suitable 3D Curve for Bayesian Optimization

To minimize the inconsistent ball trajectories of CoppeliaSim discussed in chapter 4.5, the parkour geometry near the robot was modified. The beginning of the parkour (the side of the parkour near the robot) is changed from a fillet to a flat surface, and the gripper of the robot has changed to a gripper more aligned with the open manipulator X, so it is directed better with respect to the angle of the robot. See Figure 22, in this arrangement, we minimize the problems discussed in chapter 4.5.

Figure 22 parkour with minimized CoppeliaSim problems

In Figure 22, simulation was performed where the robot angle goes from -25° → 25° and the velocity from 0.1 m/s → 2 m/s each divided in 40 tests, which creates a 3d graph (angle, velocity, and solution J). Figure 23 shows promising results, with the optimal solution around 0.636 m/s and 0.0783 radians (approximately 4.5 degrees). We collected 50 data points at intervals of 50 milliseconds, resulting in a total simulation time of 2.5 seconds.

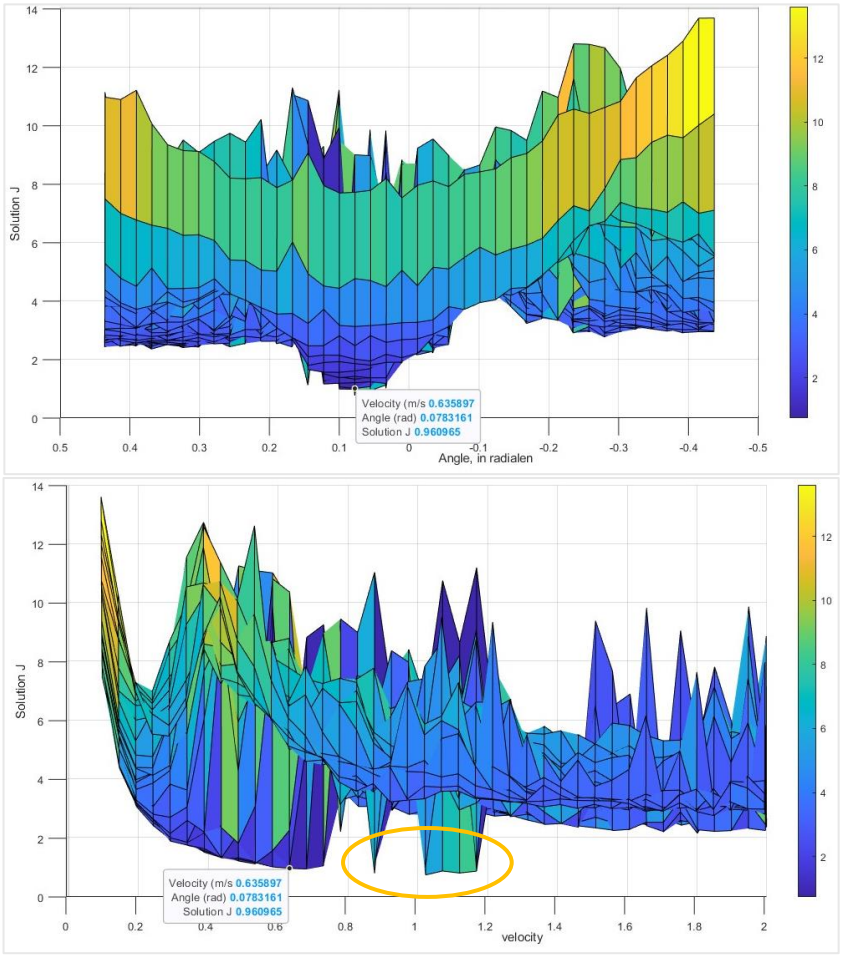


Figure 23 Suitable solution plot. (J, Angle) / (J, Velocity)

Figure 23 shows some unexpected behavior seen in the yellow circle; this is due to the fast speed of the ball. If we watch the made trajectory of the ball, we notice that the ball overshoots over the end goal and forceful rebound in the end goal too fast. This is also due to the mesh of the parkour.

To get a deeper understanding for the reader of why we get to this solution, 3 plots are plotted with the correct guessed angle of 0,0783 rad and 3 different speeds of 0.3 m/s, 0.636 m/s and 0.8 m/s, this can be seen in Figure 24. We notice that for a lower speed, the ball reaches the goal slowly in the given time of 2.5 seconds, we also notice that for a speed of 0.8 m/s, the ball rolls too fast and escapes from the end goal.

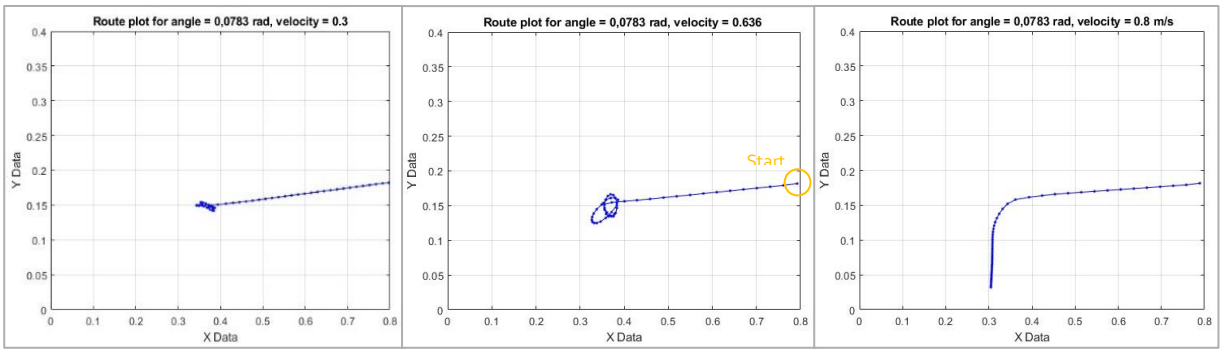


Figure 24 Recorded paths (Angle=0.0783 rad / Velocity = 0.3;0.636;0.8 m/s)

To get a more fluent outcome, several parkours were tested out, including the previous parkour, a curved down parkour, and a curved-up parkour. There can be concluded that the parkour like Figure 22, but with a slightly curved-up slope had the best outcome. See Figure 25 for the final selected parkour. There can be seen that the best solution found through grid search is at a speed of 1.01m/s and 0.105 rad (J = 0.86748). For that solution, the datapoints it took on the route are displayed in Figure 26.

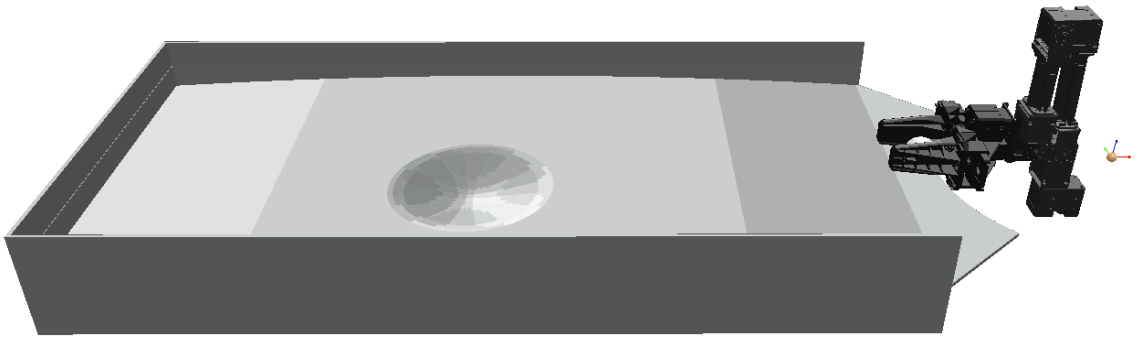


Figure 25 Final parkour

Each test took 50 ms/datapoint x 50 datapoints or 2.5 seconds in real life, although the simulation time per test was 16 seconds (depending on calculation power/computer). The total runtime for this graph below was thus 16 x 30 x 30 = 14400 seconds (about 4 hours). If we simulate this in real life, this will also take around 30 seconds per test (to place the ball in the right position each time). Utilizing Bayesian optimization can be a good strategy to minimize the number of tests and thus also the time, especially when time is the critical factor/expensive factor.

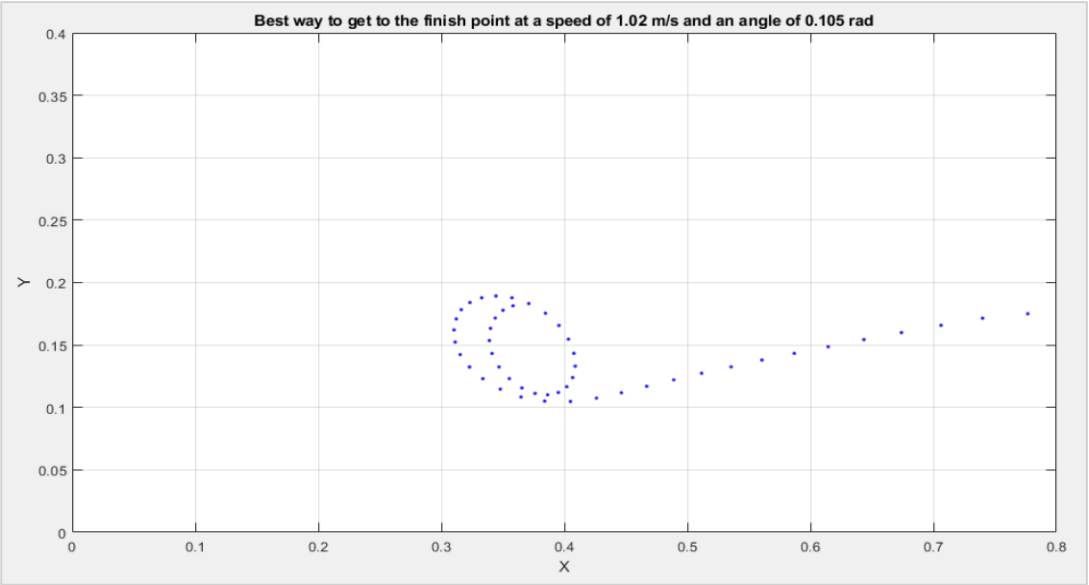


Figure 26 Best trajectory of the ball found by grid search

5 Perform Bayesian optimization

The Bayesian optimization algorithm attempts to minimize a scalar objective function f(x) for x in a bounded domain. The function can be deterministic or stochastic (Bayesian Optimization Algorithm - MATLAB & Simulink - MathWorks Benelux, n.d.), in the case of the simulation program CoppeliaSim, the simulator reacts in a deterministic way, this was tested out by doing the same angle and velocity for 10 times, and the exact route was taken repeatedly.

To start with, there is a z_matrix where each element z_matrix(i, j) represents the solution value for the i-th velocity and j-th angle. And there are separate arrays for the velocity and angle containing the corresponding values for each sample. So, this z_matrix contains 30x30 or 900 measured points, see chapter 4.3 for this grid search. The goal for this Bayesian optimization is to find the best solution (the lowest J value), with way less iterations.

So, the idea is to start with a few input data points (like 4 or 0 points for example) and let the Bayesian optimization guess what the best next point is to test. As the Bayesian optimization passes the next velocity and angle, CoppeliaSim will execute this and out of that data, the solution J can be calculated again.

5.1 Manual code review of Bayesian optimization (expected improvement)

In the appendix chapter 10.2, the full code for the Bayesian optimization can be seen. This code was adapted from the work of Karl Ezra Pilario at the university of the Philippines – Diliman (*AI221/Gaussian_Process+BayesOpt at Main · Kspilario/AI221*, n.d.), because it gives a good visualization to the Bayesian optimization. The code in chapter 10.2 displays 4 graphs, the first one displays the grid search we discussed in chapter 4.6. In the 2nd graph, all the observed points are displayed and, in the 3rd, and the 4th graph, the uncertainty and the expected improvement (EI) is displayed. See chapter 6.1 for these produced graphs.

In the next snapped code from the appendix chapter 10.2, x is the input data (Velocity and Angle) and y_true is the output data J. These are each stored in the columns of a matrix, in this case x holds the velocity in a column and the angle in a column, while 'y_true' holds the solutions J in a column. In the for loop where 'obs' observations are made, a model is generated with the function fitrgp or a gaussian process regression. This function returns a GPR model for predictors X and continuous response vector 'y_true'. The function predict(mdl,xyfine) predicts the output values 'y-pred' using the trained GPR model (mdl) for a new set of input datapoints ('xy_fine'). These predictions form a surface plot in the subplot in the upper right corner of the visualization grid. Each point on this surface represents a predicted output value 'y_pred' corresponding to a specific input combination 'xyfine'. Also consider that the standard deviation is kept by the variable 'sd'. This quantifies the uncertainty associated with each predicted output value and is also plotted in the corner at the left below side.

```
x = [reshape(velocity_matrix, [], 1), reshape(angle_matrix, [], 1)];
y_true = reshape(J_matrix, [], 1);
[Xfine,Yfine] = meshgrid(linspace(xrange(1,1),xrange(1,2)),...
                         linspace(xrange(2,1),xrange(2,2)));
for j = 0:obs
   mdl = fitrgp(x,y_true,'KernelFunction','ardmatern52');
   %ardmatern52 kernel was recommended in https://arxiv.org/pdf/1206.2944.pdf
   xyfine = [Xfine(:), Yfine(:)]; %grid of input points (Velocity and angle)
   [y_pred,sd] = predict(mdl,xyfine);
   zlim = [0,20];

   subplot(222);
   surf(Xfine,Yfine,reshape(y_pred,size(Xfine)));  % GPR prediction
   shading interp; hold on; view(angle);
   scatter3(x(:,1),x(:,2),y_true,10,'g','filled',...
       'MarkerEdgeColor','k');                    % Plot seen data
   title(sprintf('No. of observed points: %d',length(x)));
   axis([xrange([1 3 2 4]),zlim]); box on; hold off;
```

The following code under this text, sets xi (or ξ) at 0.1, this is the exploration-exploitation parameter, a higher value of xi will encourages more exploration, a lower value means more exploitation of known promising regions. The improvement for minimization 'd' is calculated with the formula d=min(y_true)−y_pred−ξ, it is calculated as the difference between the minimum observed function value and the predicted function value, adjusted by the parameter ξ. By subtracting ξ, points that are predicted to be close to the minimum observed value but are not significantly better, are penalized. This encourages exploration by points that are not close to the current minimum. (*Bayesian Optimization - Martin Krasser's Blog*, n.d.)

The EI or expected improvement is used as the acquisition function. This acquisition function is seen and is discussed in chapter 2.8.2

In the next part of the code after calculating the EI, the coordinates are taken where the expected improvement is the highest, these are stored in x. Then the GetSolutionFromCoppeliaSim() will run the simulation with the right angle and velocity and will return the solution J and add this to y_true. See appendix chapter 10.3 for this code.

```
%this code is in the for loop.
    %% Expected Improvement
    % This EI is from http://krasserm.github.io/2018/03/21/bayesian-optimization/
    xi = 0.1;  % Exploration-exploitation parameter (greek letter, xi)
             % High xi = more exploration
             % Low xi = more exploitation (can be < 0)

    if tobemax, d = y_pred - max(y_true) - xi; % (y - f*) if maximization (will never
be reached. tobemax=0)
    else,       d = min(y_true) - y_pred - xi; % (f* - y) if minimiziation
    end

    EI = (sd ~= 0).*(d.*normcdf(d./sd) + sd.*normpdf(d./sd));

    [eimax,posEI] = max(EI); xEI = xyfine(posEI,:);
    x(end+1,:) = xEI;               % Save xEI as next
    y_true(end+1) = GetSolutionFromCoppeliaSim(xEI);
```

6 Results

As seen in chapter 5, different acquisitions and kernels are discussed. In this chapter, the results of the Bayesian optimization with different kernels and different acquisition functions are compared. As a goal for these results, J=0.9 is chosen. In this thesis we want to discuss which kernel, which acquisition function or which parameters get the fastest to that goal. As the grid search needed 30x30 or 900 points to get to the solution of 0.87<0.9. This took 4 hours to complete, so it is useful to know which parameters the best is to use.

6.1 Comparison of kernels

In this chapter, the difference between the kernels will be discussed, with each a Bayesian optimization with Expected Improvement ($\xi = 0.1$, see chapter 6.2 why we chose 0.1) as acquisition curve. In the subchapters we use different kernels for the fitrgp process including the Matérn family, exponential and the squared exponential. These are the most common kernels for the fitrgp() function. Also note that we keep the xi parameter at 0.1, this makes sure that the balance between exploration and exploitation is optimal and that all default parameters of the kernels are used.

6.1.1 ARD Matérn 5/2 kernel

With 150 collected datapoints, the ending result with the ARD matern52 kernel is 0.912186 (>0.9) with a speed of 0.750505 and an angle of 0.004444 rad. See Figure 27 for the recreated model through bayesian optimization, here the reader can find the uncertainty, the expected improvement and the recreated model found through the bayesian optimization after 150 observed points. Note that these bayesian optimizations do not happen with the original Bayesopt() function from MATLAB, check chapter 5.1 for more information.

This kernel is also the kernel that is standard used in the MATLAB bayesopt() function. We can see that the bayesian optimization did a decent job in recreating the grid search surface, especially in the minimum of the graph. For further experiments, every estimated objective minimum is plotted per kernel in chapter 6.1.2.

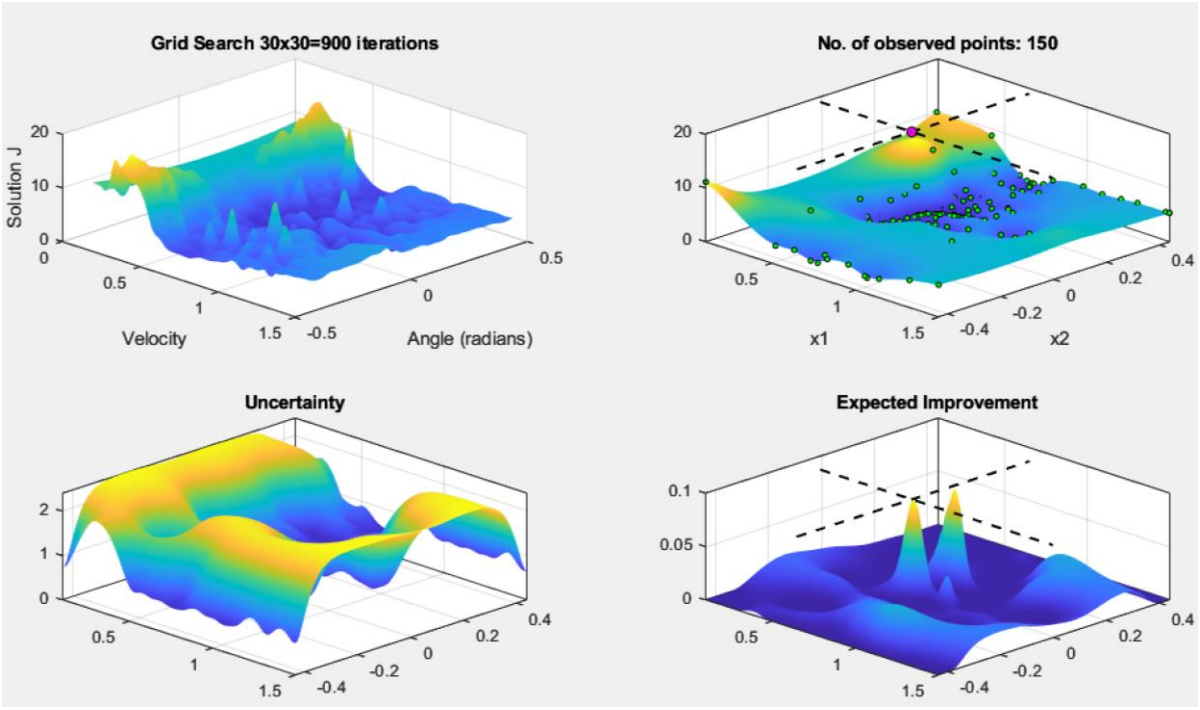


Figure 27 ARD Matérn 5/2 at 150 collected points

6.1.2 ARD Matérn 3/2 kernel

The ARD Matérn 32 reacts less fluently. It cannot handle the quick changes in the surface of the graph. The minimum result that this kernel got to after collecting 150 points is 0.9122, which is not even below 0.9, so did not really succeed in finding the minima. The same can also be said about the ARD Matérn 5/2 kernel.

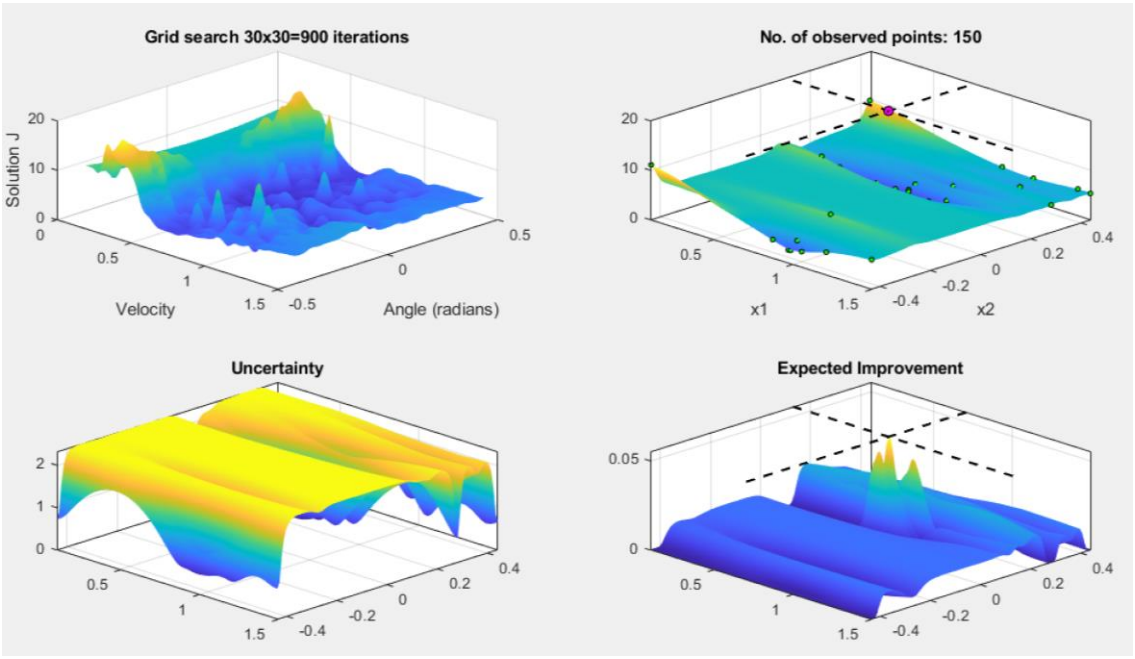


Figure 28 ARD Matérn 3/2 kernel at 150 collected points

6.1.3 Matérn 5/2 kernel

While the recreated surface does not look the same as the grid search, it kept searching around the right velocity and angle. After 150 collected points, it got to a solution of even 0.8344. After 88 tries it got under 0.9 already.

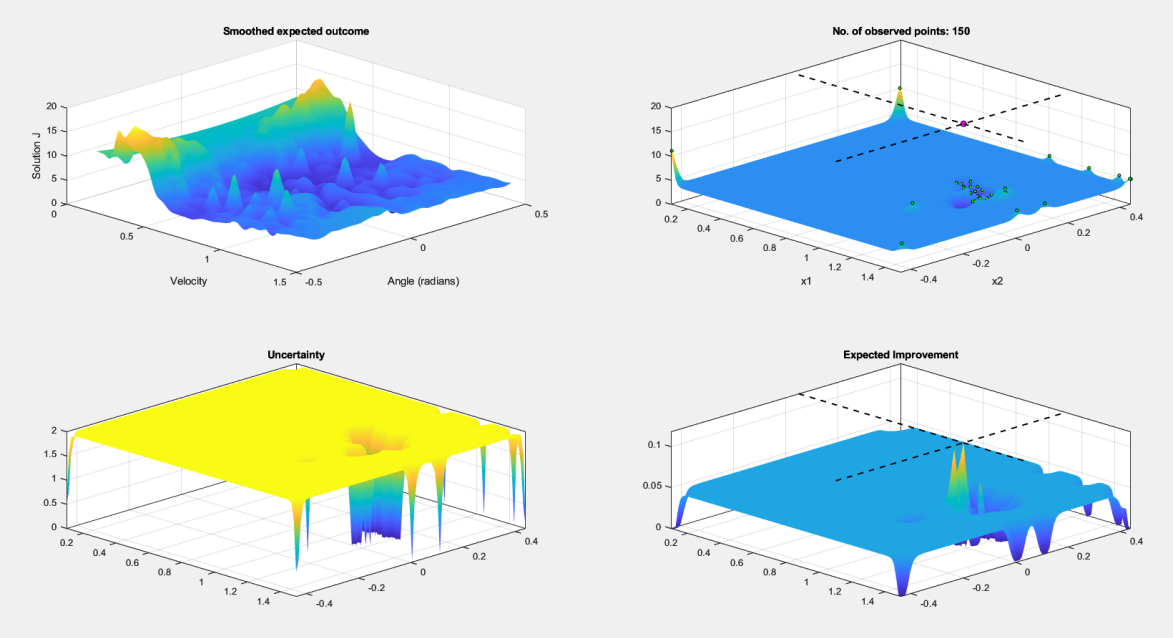


Figure 29 Matérn52 kernel at 150 collected points

6.1.4 Matérn 3/2 kernel

This kernel reacted the same as the Matérn 5/2 kernel, here it reached a solution of 0.8344 after 148 observations. This is the same minimal solution as the Matern 5/2 kernel. After 82 tries, it got under the value of 0.9.

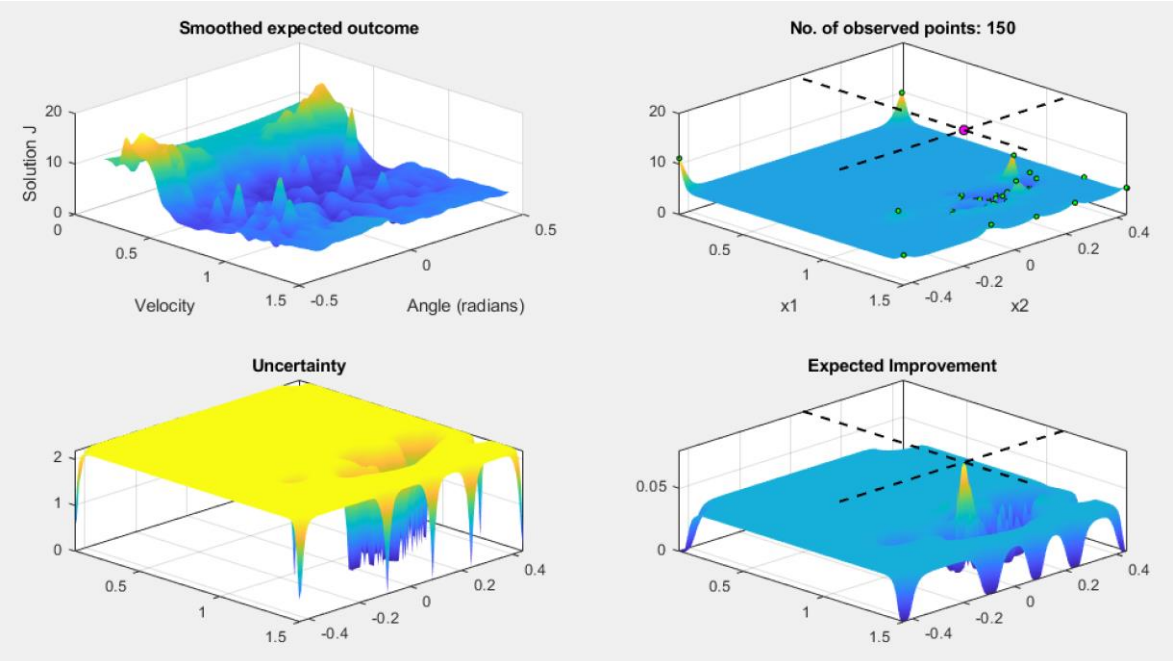


Figure 30 Matérn 3/2 kernel at 150 collected points

6.1.5 Exponential

While the exponential kernel got under the value of 0.9 in 60 observations. It did not reach the low value of the Matern52 and Matern32 kernels. The final minimal solution is here 0.8642, already found after 60 tries, but does not improve any further after that.

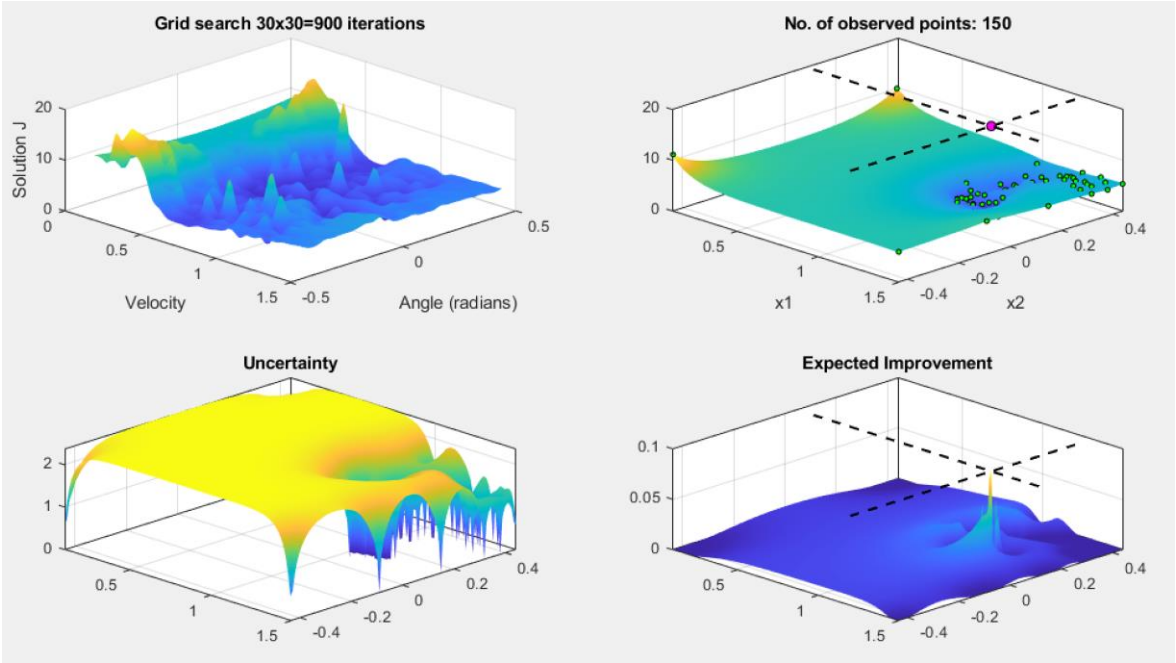


Figure 31 exponential kernel at 150 collected datapoints

6.1.1 Squared exponential

This kernel is the default kernel for the fitrgp() function, while the bayesopt() uses the ARD matérn 52 functions. This kernel performed the worst for the bayesian optimization, it never got under a solution of 0.9 and ended after 150 observations at a solution of 0.9829.

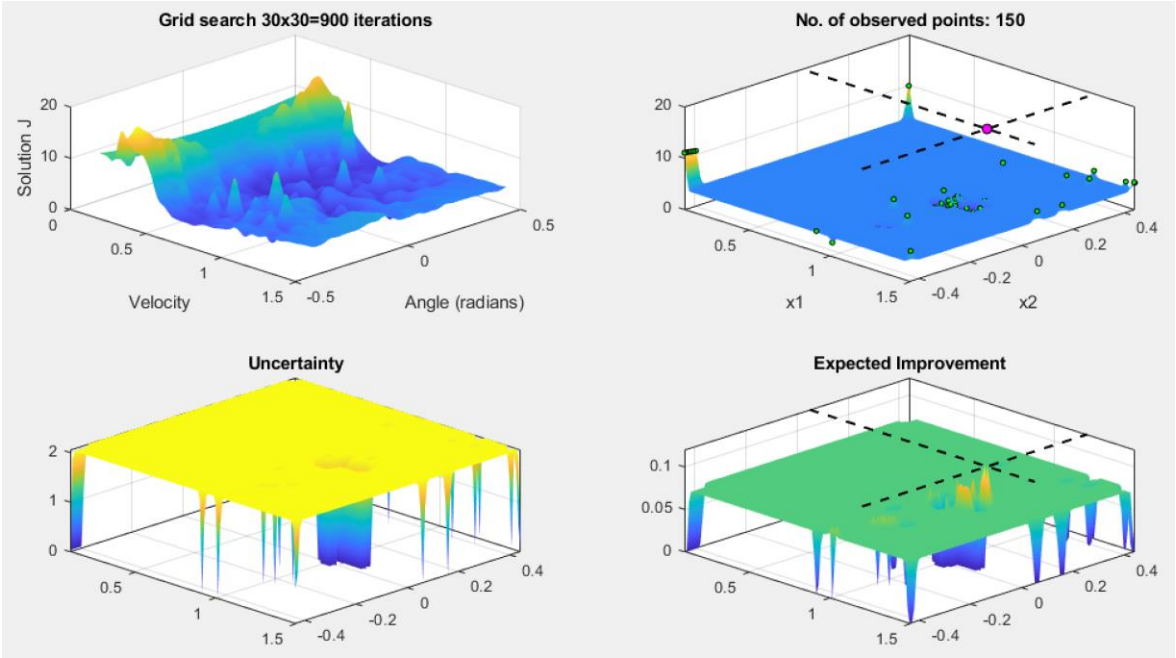


Figure 32 Squared exponential at 150 collected points

6.1.2 Estimated Objective Minimum on all kernels.

As a last chapter, the comparison between the minimum test results and the number of tests is displayed. The ARDMatern52 was the best for the recreation of the grid search, while the Matern52 was the best to get even after 30 observations a solution of 0.9372. And this kernel even found a solution of 0.8344. This solution was found at a speed of 1.1465 m/s and an angle of 0.0044 rad.

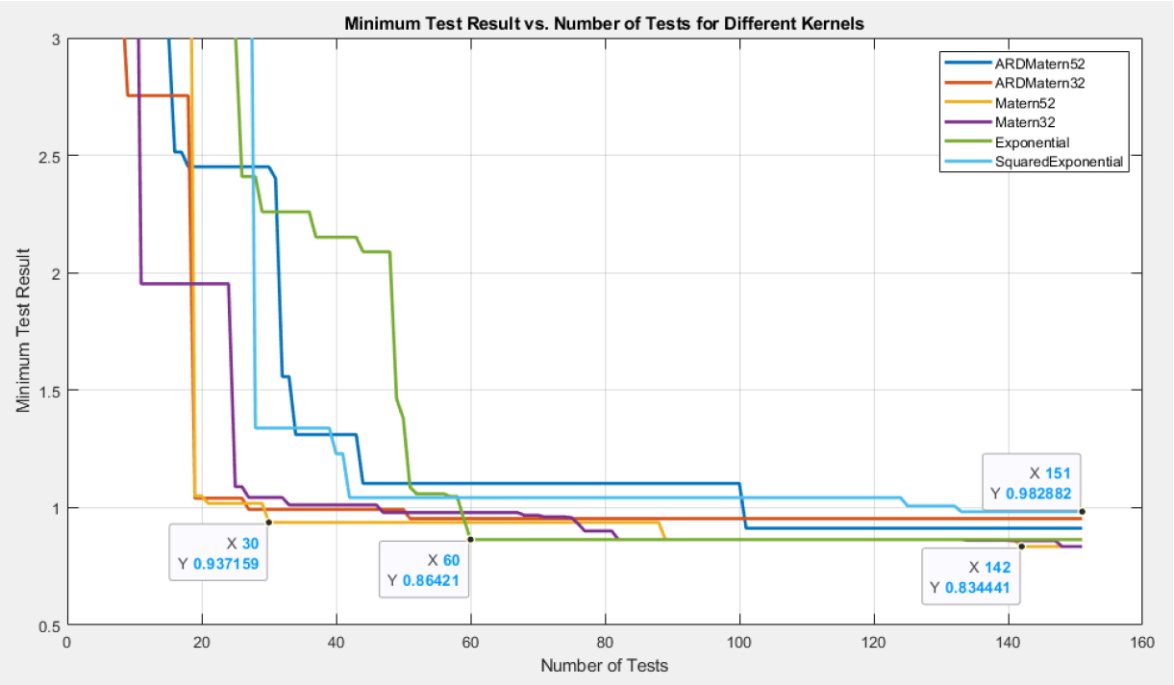


Figure 33 Estimated objective minimum compared

6.2 Effects of changing exploration ratio in EI.

To test out what the best Exploration Ratio is for the expected improvement formula, tests were made so the objective or solution in function of the number of observations can be plotted. For

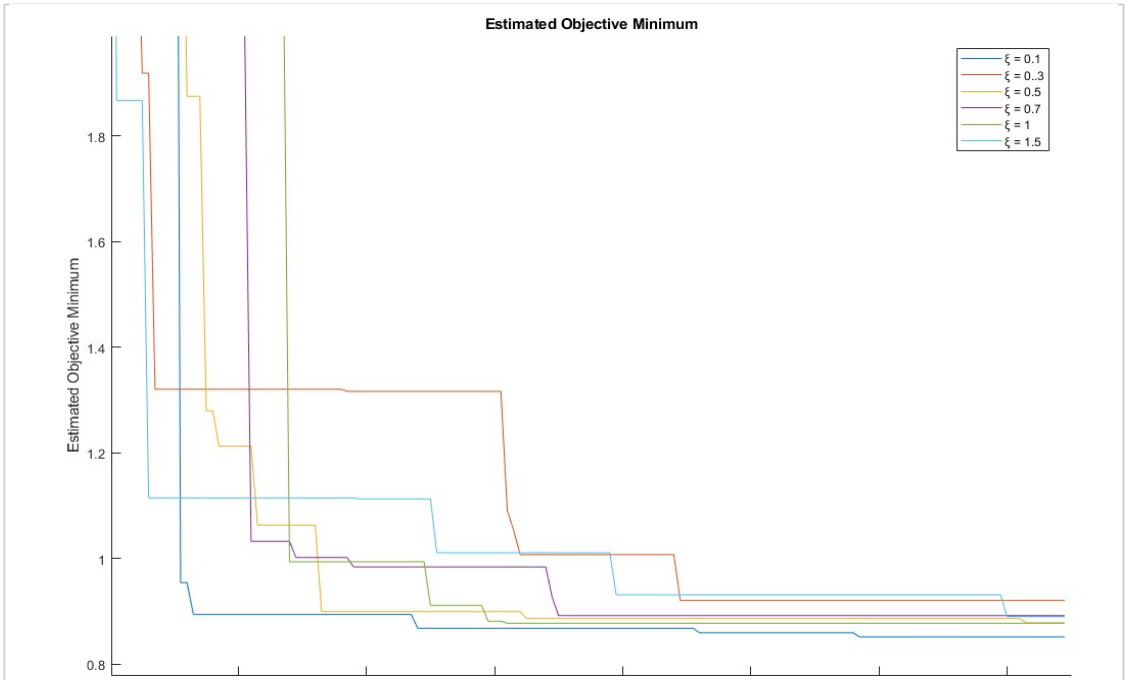


Figure 34 EI: ξ = 0.1 / 0.3 / 0.5 / 0.7 / 1 / 1.5

information, here the bayesopt() function from MATLAB is used, with an ARD Matérn 5/2 kernel, see the code in the appendix chapter 10.4. In *Figure 34*, different exploration ratios are tested out, this was done for ξ = 0.1 / 0.3 / 0.5 / 0.7 / 1 / 1.5. As can be seen, the lower the exploration ratio, the faster we get the solution. ξ = 0.1, is also the only test that reached a solution of only 0.8519, this is a lower solution as the grid search. But here only circa 120 iterations were needed, while in the grid search, we did 900 iterations. As for the other ratios, they ended up in a more local minimum. The exploration ratio of 0.3 failed, and started searching in a local minimum, the reason for this can also be because of chapter 4.5 Limitations of CoppeliaSim.

ξ	Number of tests needed to get J under 0.9	Best angle	Best velocity	Best Solution J
0.1	14	-0.01021	0.95801	0.8519
0.3	N/A: 150+	-0.00980	0.79911	0.9209
0.5	34	-0.00810	0.90881	0.8787
0.7	71	0.13770	1.09473	0.8923
1	60	0.11704	1.04118	0.8778
1.5	141	0.01435	1.0633	0.8903

Table 2 Effects of changing exploration ratio in EI

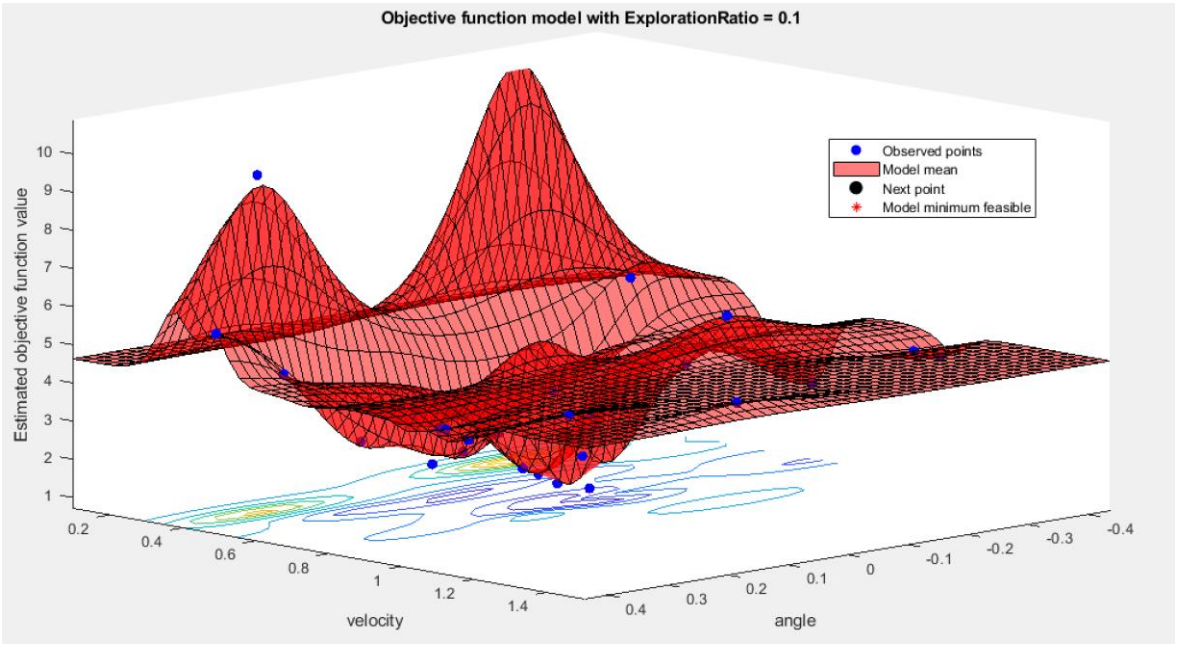


Figure 35 Representation of the objective through bayesopt(), ξ = 0.1, observations = 30

As we can see, with the expected improvement of 0.1, it already reached 0.9 after 14 tries. See Table 3 for all the guesses of the bayesopt(). There can also be noted from Figure 35 that the objective mean function is well recreated after 30 observations.

Obs nr.	Angle	Velocity	Reached minimum Objective
1	0.178863316499393	1.38906379979340	5.25912440932357
2	0.417258404346453	0.538948891968537	4.61589480564367
3	-0.364546021270933	1.47064152233831	4.61589480737713
4	-0.429756972560486	0.342269229161440	4.61589483415296

5	0.375395410213563	0.899831399455075	4.43340383395011
6	0.435794479407962	1.49918950459219	4.43340384137015
7	0.436215779857502	0.839819660182023	4.43340395304987
8	0.231397061028392	0.354697899462613	4.43340382534181
9	0.137721001375371	1.45400117406350	4.43340390475640
10	-0.433423251940695	1.02081822528617	4.34793602086259
11	0.434831609833118	1.07270276002257	3.91203652750060
12	0.125800915319664	1.01903381788736	0.954495628311187
13	0.00575985526234091	1.07212490265078	0.954495720539229
14	-0.00887814503403788	0.982333795088929	0.894487260006642

Table 3 first 14 tries of bayesopt(EI, ratio=0.1)

6.3 Comparing Expected Improvement types

The MATLAB bayesopt() function has 6 acquisition functions available to choose from. In these 6, there are 4 expected improvements functions. The default acquisition function is the expected-improvement-per-second-plus, but also the standard expected improvement, expected-improvement-plus and expected-improvement-per-second are available. These acquisition functions are discussed in the literature study. For this study, the Expected improvement is used with an exploration ratio of 0.1.

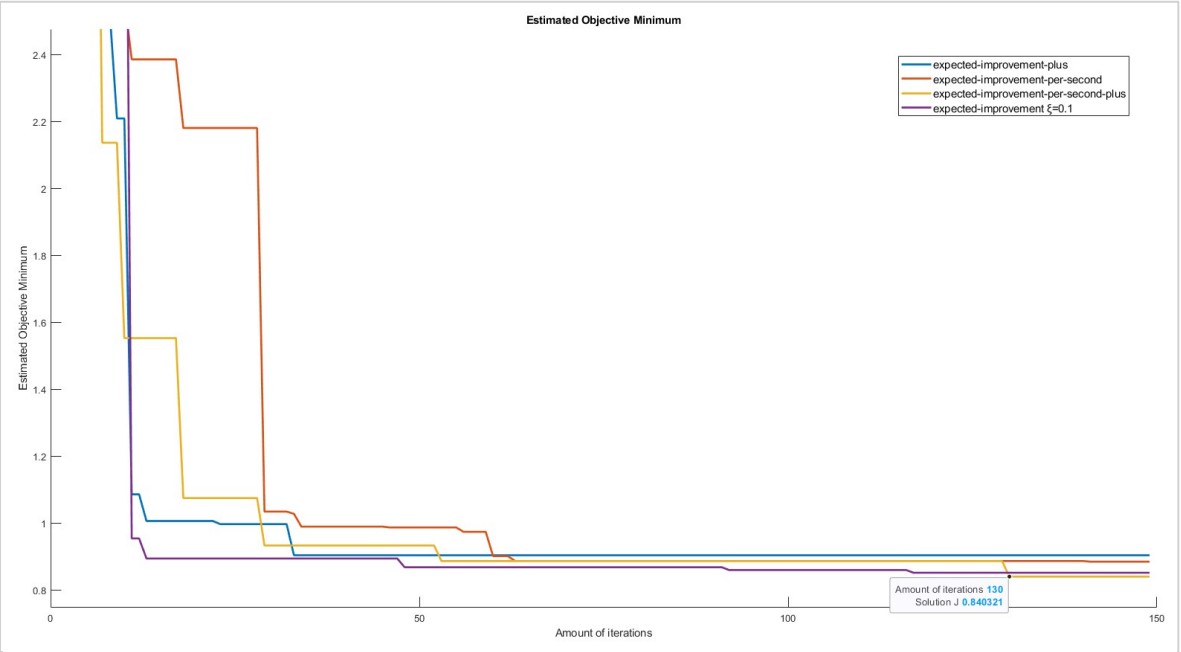


Figure 36 Comparing Expected Improvements

Note that the standard expected improvement with an exploration ratio of 0.1 is still an excellent choice. But the expected improvement per second plus stands out, as it has even found a better solution than the grid search in chapter 4.6, with a found solution at 0.8403. This only at 130 iterations while a grid search took 900 iterations.

Type of EI	Number of tests needed to get J under 0.9	Best angle	Best velocity	Best Solution J
plus	N/A 150+	-0.00214	0.97306	0.9043
per second	64	0.12239	0.92935	0.8849
Per second plus	54	0.11285	1.02832	0.8403
EI at ξ=0.1	14	-0.01021	0.95801	0.8519

Table 4 Comparing expected improvements

6.4 Lower confidence bound

Lower confidence bound as seen in the literature study, is tested out here. There can be seen that The Bayesian optimization gets stuck in a local minimum and doesn't really explore new places in the function model, see Figure 37. It gets stuck and keeps trying the points around angle 0, which is still not a bad solution.

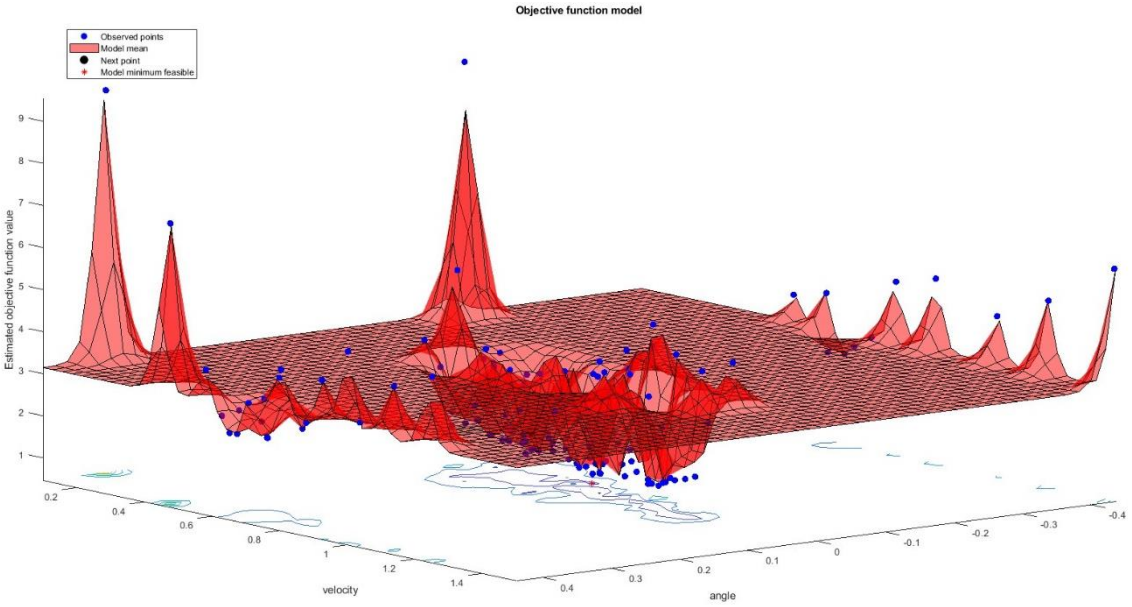


Figure 37 Objective function model Lower Confidence Bound

At 51 iterations it gets to a solution J=0.9143, but after that it does not improve any further and never gets under 0.9. If we compare this to all previous functions. It is the worst performing acquisition function in this case of the master thesis.

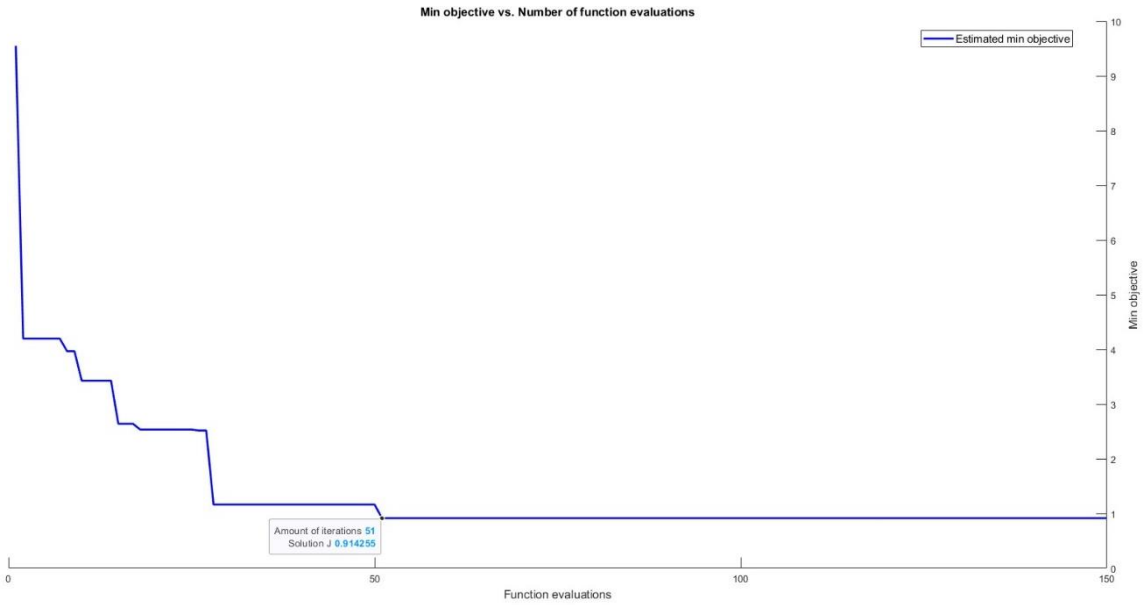


Figure 38 Min. Objective Lower Confidence Bound

6.5 Probability-of-improvement

This acquisition function is not performing well. It keeps trying the same angle and velocity. It could be due to the Limitations of CoppeliaSim that it has some wrong solutions, the gaussian fitrgp function could not make the gaussian process anymore because it has a poor model fitting. The most flexible kernel Matérn 5/2 is not flexible enough to capture the underlying model, because the Bayesian optimization keeps capturing solutions at the same speed and velocity. The lowest solution found by the acquisition function PI is 2.1868. Farly higher than other acquisitions functions. Here it would be useful to change parameters of the bayesopt() function, and specifically the parameters of the gaussian process. In the conclusions, the limitations of the bayesopt() will be discussed.

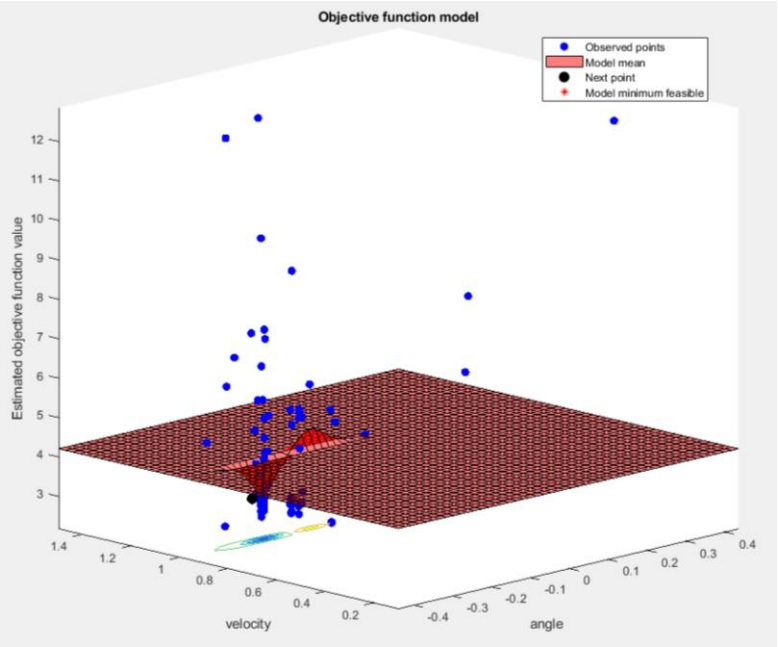


Figure 39 Objective function Probability of Improvement

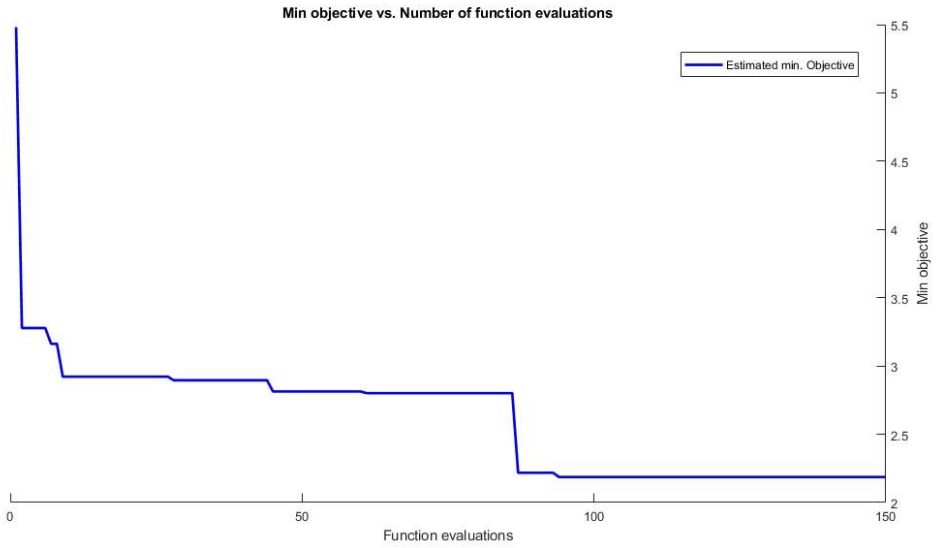


Figure 40 Min. Objective Lower Confidence Bound

6.6 Best route

As seen in this thesis, the best solution found was with the matern52 kernel, and the exploration factor of the EI on 0.1. This set of parameters guessed the lowest outcome of J=0.83 after 142 tries, this with a speed of 1.1465 m/s and an angle of 0.0044 rad. To even find a closer solution at that minimum, a bayesopt was performed between 1.14→1.15m/s and 0.004 → 0.005 rad. For this, the

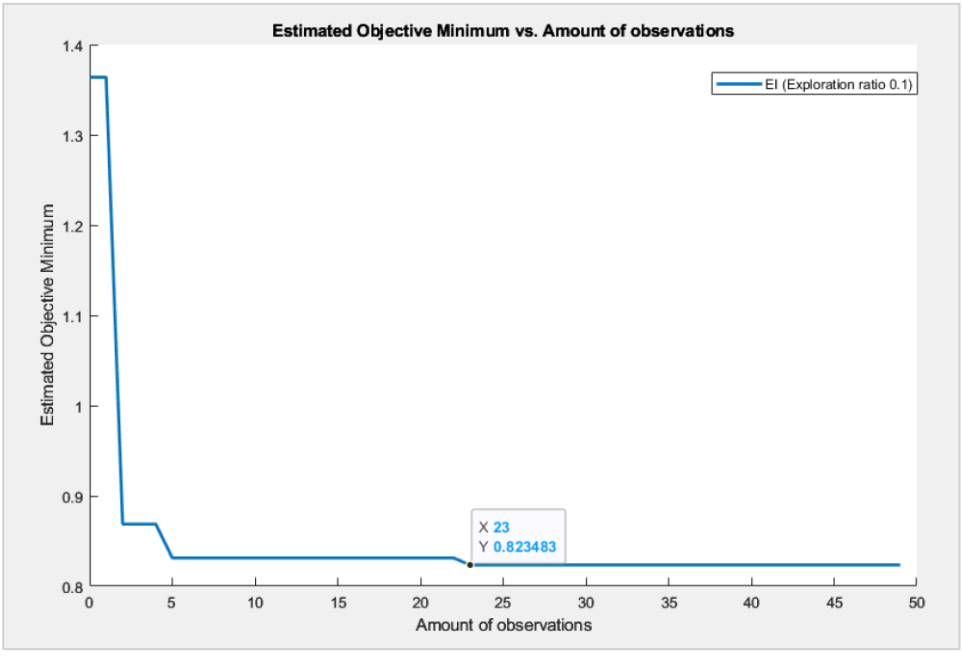


Figure 41 Bayesopt() for angle between 0.004 →0.005 and speed between 1.14 →1.15

standard parameters of bayesopt() and an exploration ratio of 0.1 were used. In Figure 41, the reader can see the objective minimum solution is found where J=0.823483, the lowest value found in the thesis. This value is found at an angle of 0.00423115994012618 rad and a speed of 1.14698072064436 m/s, which is a particularly good and specific solution.

To visualize what the ball does in this specific angle and speed at a solution of J=0.823483, the figure below shows the trajectory of the ball. As the reader can see, the best route is the route when the balls go as fast as possible while maintaining rolling in the valley made as the end goal.

If it is too fast, it jumps out of the valley. If the ball goes too slow, it collects more datapoints far away from the end goal.

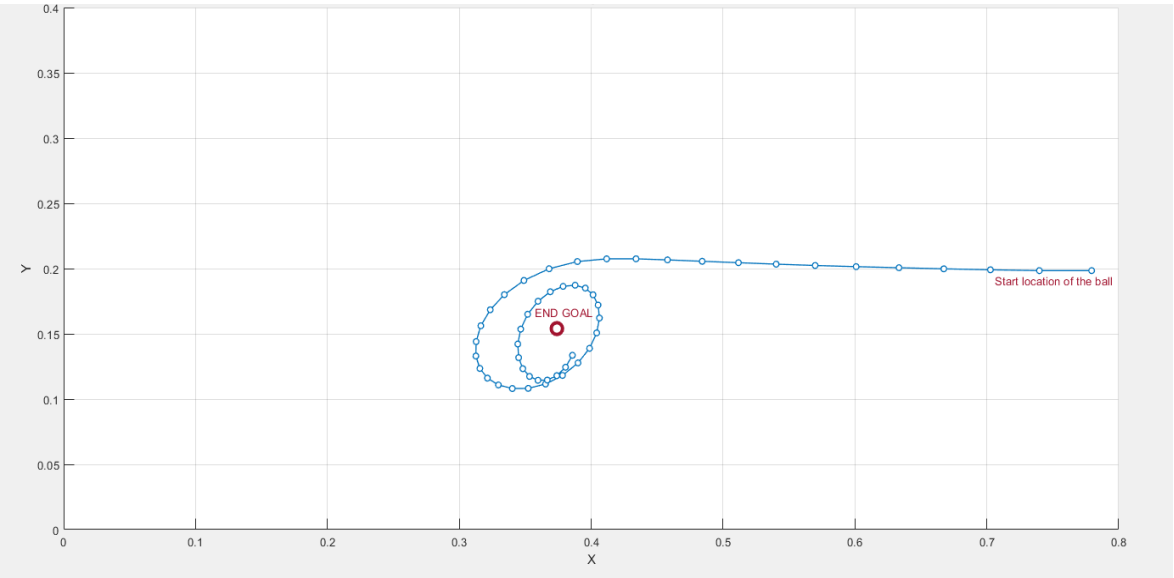


Figure 42 Final best trajectory

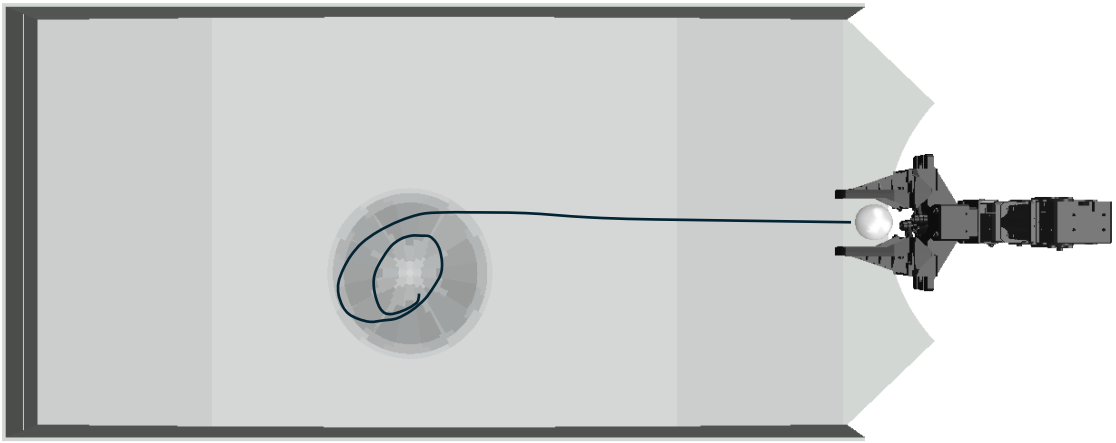


Figure 43 Final parkour + final trajectory

6.6.1 Comparing with expected best course

As we are expecting a best route directly to the end goal, with an angle of exactly 0.0711 rad or 4.07 degrees, there can be compared why the best solution found by the bayesian optimization discussed in chapter 6.6 is better than the solution straight to the end goal. To determine the maximum speed at which the ball will not roll past the end goal's valley, we conducted a grid search specifically focusing on changes in speed at a 4.07-degree angle. This gives us the highest speed of 0.876923 m/s. The solution calculated when having this highest speed and the direct angle to the endpoint resulted in a solution of 1.10149, which is significantly larger than the solution discussed in chapter 6.6. That is normal because as you can see in Figure 44, the route taken by the 'best result found by bayesian optimization' takes less points in the beginning far away from the end goal, and collects more points close to the end goal. That is because the speed of the blue trajectory is larger than the speed of the orange trajectory. If we use a slightly larger speed for the orange line, the ball will roll over the end goal and would finish at an x coordinate of 0.

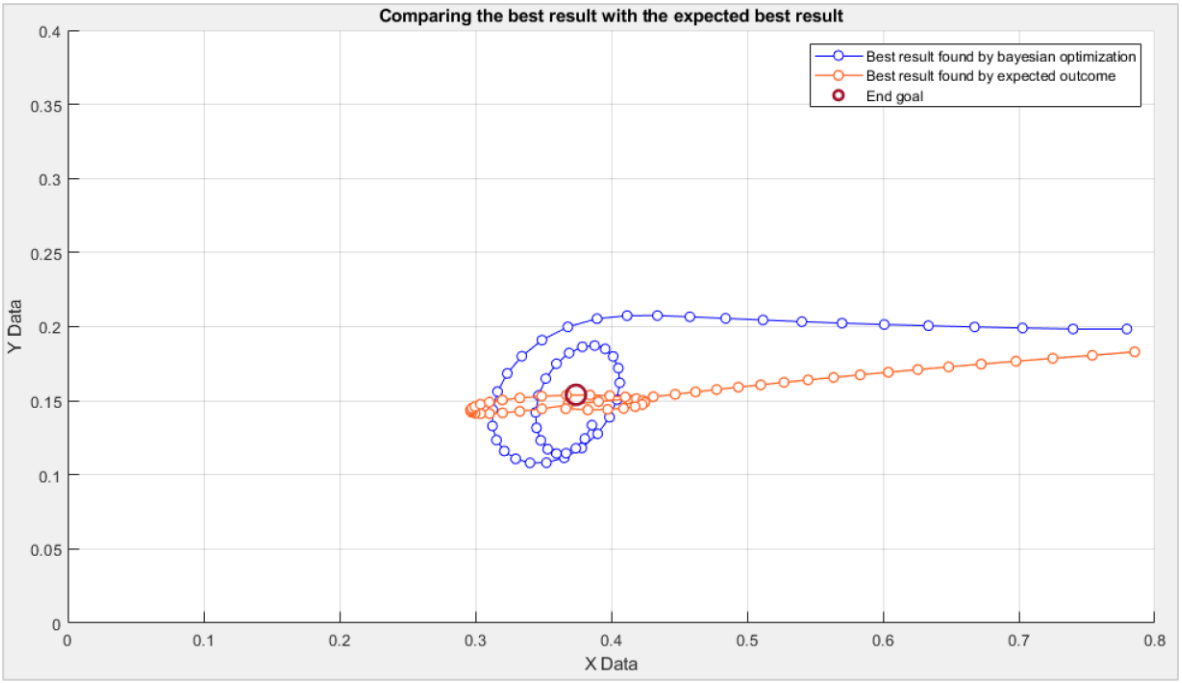


Figure 44 Understanding why the solution straight to the end goal is not the best

The comparison between the initial solution, J=1.10149, and the refined solution, J=0.82348 reveals an enhancement in performance. To quantify this improvement, we can compute the percentage decrease in the solution value.

$$\text{Percentage Improvement} = \left(\frac{1{,}10149 - 0{,}82348}{1{,}10149}\right) \times 100\% = 25.21\% \qquad (13)$$

Therefore, the solution's achieved a 25.21% improvement in performance if we compare the expected outcome versus the best-found outcome through bayesopt.

7 Conclusions

This master thesis's goal was to create a golf-like terrain, where the ball must be hit in different directions and velocities. In this terrain the solution J or the objective minimum was calculated for each angle and velocity. While trying all different parkours, like random generated parkours, flat parkours or parkours with a slight hill, some limitations of CoppeliaSim were noticed. While making slight changes in the angle or slight changes in the velocity, in terms of 0.01° in changes, the ball could make a whole other trajectory. Due to these errors, it was difficult to find a parkour with a smooth surface as the objective. When trying to minimize these errors by changing the parkour, there was concluded that a parkour with an end goal around the middle point and a slightly curved up parkour was the most suitable parkour to create an objective function J with a suitable minimum. While taking the solution J for 30 different angles and 30 different velocities between -25° and 25° and between 0.1m/s and 2 m/s, we got 900 samples plotted in a 3d surface. Here we found that the minimum solution J we found with the best trajectory was at 1.01 m/s and 0.105 radians with J=0.8675. This grid search had a simulation time of 14400 seconds (about 4 hours) as the expensive factor.

While 0.8675 is quite a workable solution, it still had to make quite a lot of calculations (4 hours) and 900 observations. To improve the expensive factor of the simulation time, the bayesian optimization was explored. In this thesis, the minimum objective we wanted to achieve is taken on 0.9, which is just slightly worse than the grid search. Defining a specific goal for our solution allows us to evaluate how different parameter or kernel selections can most rapidly reach that goal. While exploring all the different kernels that the gaussian process uses, like the Matérn and the exponential kernels, there could be concluded that the Matern52 kernel performed in average the best with a found solution J of 0.9372 after 30 observations. While continuing to improve it found a solution of 0.8344 at 142 tries. (note that while changing these kernels of the GP, the expected improvement with an EI ratio of 0.1 was used as an acquisition function).

When using the default kernel of bayesopt() (ARDMatern52) to compare how the change of the exploration ratio effects the results, there can be concluded that a low exploration ratio of 0.1 was better than using a larger exploration ratio. So, in this case of the thesis, it was better to use less exploration and more exploitation. The bayesian optimization already reached the goal of J=0.9 in 14 observations when using an exploration ratio of 0.1. In this case it took 16 seconds x 14 or 4 minutes to achieve a solution<0.9. while a grid search took 4 hours to find the solution of 0.8675.

Throughout trying the bayesopt() function of MATLAB, we also compared 4 different EI types while there could be seen that the standard expected improvement with the exploration ratio at 0.1 was still the best. Also, the Probability of Improvement and Lower confidence bound were explored but did not provide reliable results.

While wanting to explore the capabilities of the bayesian optimization through the function of bayesopt() in MATLAB, some limitations were encountered regarding the customization of the gaussian process. With manual coding the bayesian optimization, we could discuss different kernels, while the bayesopt() function lacks the ability to directly modify kernel parameters. This restricts the user's ability to change the gaussian process used in the BO. For future work, investigating alternative bayesian implementations in MATLAB or other toolkits could be of effective use.

At last, to even find a better solution for the trajectory, a new bayesopt was performed around the previous solution of 0.83, when performing an extra 23 points closer to the solution on top of the 142 of the previous bayesopt(), we got a final solution of 0.823483. When comparing the trajectory of this solution to the expected trajectory of going straight to the final goal, we encountered that the minimal solution found by going straight to the end goal at the maximum possible speed while not going over the end goal was 1.10149, this is because the speed is lower. It could not reach a

larger speed because it would travel over the end goal, instead of going in a spiral trajectory-like Figure 43.

7.1 Future work

As this thesis is only for the simulating part of the bayesian optimization. An overview can be given of how this work can be extended. As in the future work, the robot and the terrain can be made in real life. While pushing the ball with the robot on the made terrain, the following of the ball could be done by taking snapshots with a camera every 50ms. To extend the possibilities of the bayesian optimization while transitioning to a real-world scenario. The fundamental form of the objective function remains consistent, so the input data points of the bayesian optimization could be taken out of this thesis where the form of the objective function will be quite the same, allowing for better integration with the optimization process.

8 Budget

8.1 Introduction

In this chapter of the thesis, a detailed calculation of the budget used in this project will be provided. There are 2 main resources used, which are labor and material resources used, including software licenses.

For this work, an approximate time of 480 hours are spent, which is 30 hours per week (16 weeks).

At last, the costs of all the resources that are involved, including licenses, use of hardware, etc. will be analyzed.

8.2 Labor

All types of resources that were used:

- industrial engineer with a unit cost of 60 EUR/h

Concept	Unit cost (EUR/h)
Industrial engineer	60

Table 5 Labor cost table

8.3 Materials

The materials we are using in this thesis are shown below, which details the price and usage.

- Personal computer (Asus Zephyrus):

This computer was used to perform all the tests due to its powerful specifications and portability. This includes using it with CoppeliaSim, MATLAB, Microsoft office, ... This computer costs 1350 EUR, with an estimate's useful life of 9000 hours. It should also be noted that the devaluation of this computer has not been considered.

- MATLAB license:

The academic version of MATLAB costs 262 EUR/year with an estimated average used time of 2160 hours per year

- CoppeliaSim EDU:

This version is free of charge. This program was used to simulate the ball behavior on the surface.

- Microsoft Office 365:

This software was used to draft the thesis and the PowerPoint presentation. This software costs 69.99 EUR/year and is on average used for 1800 hours/year.

with this data, the unit cost for this thesis can be calculated with formula below:

$$Unit\ cost\ (\frac{EUR}{h})\ = Cost(EUR)/Used\ time(h) \quad (14)$$

Concept	Price (EUR)	Average used time (h)	Unit cost (EUR/h)
Personal computer	1350	10400	0,129807692
MATLAB	262	2160	0,121296296
CoppeliaSim EDU version	0	N/A	N/A
Microsoft 365	69,99	1800	0,038883333

Table 6 Materials cost table

8.4 Total cost of labor

Indirect costs of 2% were included to ensure that all costs are covered. These include administrative costs, overhead, and other organizational expenses.

Concept	Unit cost (EUR/h)	Quantity (h)	Total cost (EUR)
Industrial engineer	60	480	28800
Indirect costs		2%	576
Total costs			29376

Table 7 Total cost of labor

8.5 Total cost of materials

To calculate the costs for all materials, there should be noted that the MATLAB and CoppeliaSim EDU packages were used each for 350 hours. And the Microsoft 365 license was used for 130 hours.

Concept	Unit cost (EUR/h)	Quantity (h)	Total cost (EUR)
Personal computer	0,129807692	480	62,30769231
MATLAB	0,121296296	350	42,4537037
CoppeliaSim EDU version	N/A	350	0
Microsoft 365	0,038883333	130	5,054833333
Sub Total			109,8162293
Indirect costs	2%		2,196324587
Total costs			112,0125539

Table 8 Total cost of materials

8.6 Total budget

Partial budgets	Cost (EUR)
Item 1: Labor	29376
Item 2: Materials	112,0125539
Total budget	29488,01255

Table 9 Material execution budget

8.7 Total budget with added expenses and profit

Budgets	Cost (EUR)
Total budget	29488,01255
13% general expenses	3833,441632
6% industrial profit	1769,280753
Total budget with added expenses and profit	35090,73494

Table 10 Total budget with added expenses and profit

8.8 Bid price

Bid price	Cost (EUR)
Total budget with added expenses and profit	35090,73494
21% VAT	7369,054337
TOTAL	42459,78928

Table 11 Bid price

So, the total cost amount is 42459,78928 EUR.

9 Bibliography

AI221/Gaussian_Process+BayesOpt at main · kspilario/AI221. (n.d.). GitHub. Retrieved April 23, 2024, from https://github.com/kspilario/AI221/tree/main/Gaussian_Process%2BBayesOpt

Bayesian Optimization Algorithm—MATLAB & Simulink—MathWorks Benelux. (n.d.). Retrieved April 18, 2024, from https://nl.mathworks.com/help/stats/bayesian-optimization-algorithm.html

Bayesian optimization—Martin Krasser's Blog. (n.d.). Retrieved May 9, 2024, from http://krasserm.github.io/2018/03/21/bayesian-optimization/

Duvenaud, D. K. (n.d.). *Automatic Model Construction with Gaussian Processes*.

Galuzzi, B. G., Giordani, I., Candelieri, A., Perego, R., & Archetti, F. (2020). Hyperparameter optimization for recommender systems through Bayesian optimization. *Computational Management Science*, *17*(4), 495–515. https://doi.org/10.1007/s10287-020-00376-3

Gaussian Process. (n.d.). Retrieved April 21, 2024, from https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote15.html

Gaussian Process Regression Models—MATLAB & Simulink—MathWorks Benelux. (n.d.). Retrieved April 21, 2024, from https://nl.mathworks.com/help/stats/gaussian-process-regression-models.html

Gradient descent. (2024). In *Wikipedia*. https://en.wikipedia.org/w/index.php?title=Gradient_descent&oldid=1215767849

Kamperis, S. (2021, June 11). *Acquisition functions in Bayesian Optimization*. Let's Talk about Science! https://ekamperi.github.io/machine%20learning/2021/06/11/acquisition-functions.html

Kungurtsev, V., Rinaldi, F., & Zeffiro, D. (2023). Retraction-Based Direct Search Methods for Derivative Free Riemannian Optimization. *Journal of Optimization Theory and Applications*. https://doi.org/10.1007/s10957-023-02268-3

Mathematical programming | Optimization, Algorithms & Models | Britannica. (2024, April 12). https://www.britannica.com/science/linear-programming-mathematics

Optimization: Gradient-Based Algorithms | Baeldung on Computer Science. (2023, January 15). https://www.baeldung.com/cs/optimization-gradient-based-algorithms

Rätsch, G. (2004). *A Brief Introduction into Machine Learning*. https://www.semanticscholar.org/paper/A-Brief-Introduction-into-Machine-Learning-R%C3%A4tsch/fab926b5da1587077607679ebd56985735023d0

Shahriari, B., Swersky, K., Wang, Z., Adams, R. P., & De Freitas, N. (2016). Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE*, *104*(1), 148–175. https://doi.org/10.1109/JPROC.2015.2494218

Sun, S., Cao, Z., Zhu, H., & Zhao, J. (2020). A Survey of Optimization Methods From a Machine Learning Perspective. *IEEE Transactions on Cybernetics*, *50*(8), 3668–3681. https://doi.org/10.1109/TCYB.2019.2950779

Watanabe, S. (1985). *Pattern recognition: Human and mechanical*. John Wiley & Sons, Inc.

10 Appendix I

10.1 MATLAB code for grid search

```
% Connect to CoppeliaSim
disp('Program started');
sim=remApi('remoteApi'); % using the prototype file (remoteApiProto.m)
sim.simxFinish(-1);      % just in case, close all opened connections
clientID = sim.simxStart('127.0.0.1', 19997, true, true, 5000, 5);

if (clientID>-1)
    disp('Connected to remote API server');
    % Get handle to the object whose position you want to track
    [~, objectHandleBall] = sim.simxGetObjectHandle(clientID, 'Foosball_ball',
sim.simx_opmode_blocking);
    [~, objectHandleDummy] = sim.simxGetObjectHandle(clientID, 'Movable_Dummy',
sim.simx_opmode_blocking);
    [~, prismaticJointHandle] = sim.simxGetObjectHandle(clientID, 'Horizontal_joint',
sim.simx_opmode_blocking);

    %End point data
    X_FINAL = 0.374;
    Y_FINAL = 0.154;
    alpha = 50;  % Importance of the last collected point
    beta = 1;    % Importance for the followed path

    angRobot=linspace(-25*pi/180, 25*pi/180, 30); %lineair divided from -25° till 25° in
30 parts
    velocityRobot=linspace(0.1, 1.5, 30);         %same for speed

    neededPoints=50; %Collect 50 points every dt ms (dt=50ms)
    AmountOfTests=0; %holds on the amount of tests that has been performed

    %data used for the 3d surface:
    num_rows=length(angRobot);
    num_cols=length(velocityRobot);
    %creating the z matrix with all points of velocity and angle
    z_matrix = zeros(num_rows, num_cols);

    f_res=figure;

    Result=cell(length(angRobot), length(velocityRobot));

    for a = 1:length(angRobot) %degrees
        new_orientation = [0, 0, angRobot(a)];  %sets the euler angles for the next
orientation

        for v = 1:length(velocityRobot) %velocity
            AmountOfTests = AmountOfTests +1;
            disp(['testnr.: ', num2str(AmountOfTests)]); %displays the amount of tests

            new_velocity = velocityRobot(v);    %sets the velocity for the next speed in
the test

            sim.simxSynchronous(clientID,true);
            sim.simxStartSimulation(clientID,sim.simx_opmode_blocking); pause(0.1)
            returnCodeangle = sim.simxSetObjectOrientation(clientID, objectHandleDummy,
objectHandleBall, new_orientation, sim.simx_opmode_blocking); pause(0.1)
            returnCodevelocity = sim.simxSetJointTargetVelocity(clientID,
prismaticJointHandle, new_velocity, sim.simx_opmode_blocking);

            d_data = zeros(1, neededPoints); %empties the array again, setting index
back to 1
```

```
            x_data = zeros(1, neededPoints);
            y_data = zeros(1, neededPoints);
            z_data = zeros(1, neededPoints);
            index = 1;

            while true
                % Trigger simulation step
                sim.simxSynchronousTrigger(clientID);

                % Get object position
                [~, position] = sim.simxGetObjectPosition(clientID, objectHandleBall, -
1, sim.simx_opmode_blocking);
                [~, velocity] = sim.simxGetObjectVelocity(clientID, objectHandleBall,
sim.simx_opmode_blocking);

                % Extract XY coordinates
                x = position(1);
                y = position(2);
                z = position(3);
                d = sqrt((X_FINAL - x)^2 + (Y_FINAL - y)^2);
                Vx = velocity(1);
                Vy = velocity(2);
                V_mag = sqrt(Vx^2 + Vy^2); % Magnitude van de snelheid

                % Append coordinates to data arrays
                x_data(index) = x;
                y_data(index) = y;
                z_data(index) = z;
                d_data(index) = d;
                index = index + 1;

                % Update cell
                Result{a, v}.D_data=d_data;
                Result{a, v}.X_data=x_data;
                Result{a, v}.Y_data=y_data;
                Result{a, v}.Z_data=z_data;


                %checks if enough points are collected
                collectedpoints = length(Result{a, v}.D_data(Result{a, v}.D_data ~=0));
                if collectedpoints >= neededPoints
                    disp('Points collected. Exiting the loop.');
                    break;
                end
            end

            figure(1)
            hold on
            plot(Result{a, v}.X_data,Result{a, v}.Y_data)
            drawnow;
            xlabel('X');
            ylabel('Y');
            title(['Test with angle number ', num2str(a)]);
            xlim([0, 0.8]); % Set X axis limit from 0 to Xmax
            ylim([0, 0.4]); % Set Y axis limit from 0 to Ymax
            grid on;

            Result{a, v}.ang=angRobot(a); %calculates from i to degree
            Result{a, v}.velocity=velocityRobot(v);        %velocity is the same as
                                                           %the value j
             Result{a, v}.J= Function_Calculator(d_data,neededPoints,alpha,beta);

            % End simulation
            sim.simxStopSimulation(clientID, sim.simx_opmode_blocking);
        end
    end
```

```
    %changing results of mesh matrix z values:
    for a = 1:num_rows
        for v = 1:num_cols
            z_matrix(a, v) = Result{a, v}.J;
        end
    end
    %plot mesh
    % Plot the surface
    [X,Y] = meshgrid(velocityRobot,angRobot);

    figure;
    surf(X, Y, z_matrix);
    xlabel('velocity');
    ylabel('Angle, in radialen');
    zlabel('Solution J');
    title('surface plot');
else
    disp('Failed to connect to CoppeliaSim');
end

% End communication
sim.simxFinish(clientID);

% Clean up
sim.delete();
```

10.1.1 Function_Calculator ()

```
function result = Function_Calculator_V2(all_d,neededPoints, alpha, beta)
% Function_Calculator calculates a result based on the given expression.
% all_d: Array containing all distances
% d: Current distance
% alpha: Coefficient for the importance of the ending point
% beta: Coefficient for the importance of the route it took.
% datalength: amount of datapoints per test

% Calculate the result using the given expression
part1=alpha * all_d(neededPoints)^2;
part2=beta * sum(all_d(1:neededPoints-1).^2);
result = part1 + part2;
end
```

10.2 MATLAB code for Bayesian optimization

```
%This loaded data is used to display the original grid search function.
load('OFFICIEEL_P14_a5b1_30x30.mat', 'Result', 'velocityRobot', 'angRobot', 'z_matrix');
result1 = Result;
velocityRobot1 = velocityRobot;
angRobot1 = angRobot;
z_matrix1 = z_matrix;

load('OFFICIEEL_P14_a5b1_2x2.mat','Result','velocityRobot','angRobot','z_matrix'); %Here
4 input points are given as a start for the bayesian optimization.
Amount = length(velocityRobot); %or angrobot
velocity_matrix = zeros(Amount,Amount);
angle_matrix = zeros(Amount,Amount);

%gathering data out of Results cell
for i = 1:Amount
  for j = 1:Amount
    % Access data within each cell
    velocity = Result{i,j}.velocity;
    angle = Result{i,j}.ang;
    solution_J = Result{i,j}.J;
```

```
    % Assign values to J_matrix
    J_matrix(i, j) = solution_J;

    % Optionally, store velocity and angle in separate matrices:
    velocity_matrix(i, j) = velocity;
    angle_matrix(i, j) = angle;
  end
end

close all; clc; angle = [45 45];  %angle for the view
obs = 150;       % No. of more observations to perform

% Black-box
xrange = [0.1 1.5;         % Search range for x1
          -0.44 0.44];     % Search range for x2

%fprintf('Enter [1] if function is to be maximized\n');
%fprintf('      [0] if function is to be minimized\n');
tobemax = 0; %set on 0 for minimization


x = [reshape(velocity_matrix, [], 1), reshape(angle_matrix, [], 1)];
y_true = reshape(J_matrix, [], 1);
[Xfine,Yfine] = meshgrid(linspace(xrange(1,1),xrange(1,2)),...
                         linspace(xrange(2,1),xrange(2,2)));
for j = 0:obs
    mdl = fitrgp(x,y_true,'KernelFunction','ardmatern52');
    % ardmatern52 kernel was recommended in https://arxiv.org/pdf/1206.2944.pdf

    xyfine = [Xfine(:), Yfine(:)];
    [y_pred,sd] = predict(mdl,xyfine);
    zlim = [0,20];

    subplot(222);
    surf(Xfine,Yfine,reshape(y_pred,size(Xfine)));        % GPR prediction
    shading interp; hold on; view(angle);
    scatter3(x(:,1),x(:,2),y_true,10,'g','filled',...
        'MarkerEdgeColor','k');                     % Plot seen data
    title(sprintf('No. of observed points: %d',length(x)));
    axis([xrange([1 3 2 4]),zlim]); box on; hold off;

    % Define a finer grid for interpolation
    [Xq, Yq] = meshgrid(linspace(min(velocityRobot1), max(velocityRobot1), 100), ...
                    linspace(min(angRobot1), max(angRobot1), 100));
    % Interpolate the data onto the finer grid
    [X,Y] = meshgrid(velocityRobot1, angRobot1);
    z_matrix1_smooth = interp2(X, Y, z_matrix1, Xq, Yq, 'cubic');
    % Plot the smoothed surface
    subplot(221);
    surf(Xq, Yq, z_matrix1_smooth);
    xlabel('Velocity');
    ylabel('Angle (radians)');
    zlabel('Solution J');
    title('Smoothed expected outcome');
    shading interp;
    view(angle);
    hold on;
    zmax = 19;

    subplot(223);
    surf(Xfine,Yfine,reshape(sd,size(Xfine)));     % Uncertainty (std. dev.)
    shading interp; hold on; view(angle);
    axis([xrange([1 3 2 4]), 0, max(2,max(sd))]);
    title('Uncertainty'); box on; hold off;

    %% Expected Improvement
```

```
    % This EI is from http://krasserm.github.io/2018/03/21/bayesian-optimization/
    xi = 0.1;  % Exploration-exploitation parameter (greek letter, xi)
             % High xi = more exploration
             % Low xi = more exploitation (can be < 0)

    if tobemax, d = y_pred - max(y_true) - xi; % (y - f*) if maximization (will never be
reached. tobemax=0)
    else,       d = min(y_true) - y_pred - xi; % (f* - y) if minimiziation
    end

    EI = (sd ~= 0).*(d.*normcdf(d./sd) + sd.*normpdf(d./sd));

    [eimax,posEI] = max(EI); xEI = xyfine(posEI,:);
    x(end+1,:) = xEI;               % Save xEI as next
    y_true(end+1) = GetSolutionFromCoppeliaSim(xEI);

    subplot(224);
    surf(Xfine,Yfine,reshape(EI,size(Xfine)));
    shading interp; hold on;
    plot3(xEI([1 1]),xrange(2,:),eimax*[1 1],'--k','LineWidth',1.5);
    plot3(xrange(1,:),xEI([2 2]),eimax*[1 1],'--k','LineWidth',1.5);
    title('Expected Improvement'); grid on; hold off;
    view(angle); axis(xrange([1 3 2 4])); box on;

    subplot(222); hold on; xlabel('x1'); ylabel('x2');
    scatter3(x(end,1),x(end,2),zmax,'m','filled','MarkerEdgeColor','k');
    plot3(xEI([1 1]),xrange(2,:),zmax*[1 1],'--k','LineWidth',1.5);
    plot3(xrange(1,:),xEI([2 2]),zmax*[1 1],'--k','LineWidth',1.5);
    hold off; axis(xrange([1 3 2 4])); view(angle); pause(0.5);

end

if tobemax, [ae,be] = max(y_pred); % (will never be reached, because tobemax = 0)
            [ao,bo] = max(y_true); str = 'Maximum';
else,       [ae,be] = min(y_pred);
            [ao,bo] = min(y_true); str = 'Minimum';
end
fprintf('Bayesian Optimization\n');
fprintf('  %s (estimated):\n\ty(%.6f,%.6f) = %.6f\n',...
    str,xyfine(be,1),xyfine(be,2),ae);
fprintf('  %s (observed):\n\ty(%.6f,%.6f) = %.6f\n',...
    str,x(bo,1),x(bo,2),ao);
```

10.3 GetSolutionFromCoppeliasim()

```
function J =  GetSolutionFromCoppeliaSim(xEI)
    load('OFFICIEEL_P14_a5b1_30x30.mat','alpha','beta','neededPoints');

    X_FINAL=0.374;
    Y_FINAL=0.154;


    %start coppeliaSim
    disp('Program started');
    sim = remApi('remoteApi');
    sim.simxFinish(-1);
    clientID = sim.simxStart('127.0.0.1', 19997, true, true, 5000, 5);

    angRobot1=xEI(2);
    velocityrobot1=xEI(1);


    % Object handles
```

```
    [~, objHandleBall] = sim.simxGetObjectHandle(clientID, 'Foosball_ball',
sim.simx_opmode_blocking);
    [~, objHandleDummy] = sim.simxGetObjectHandle(clientID, 'Movable_Dummy',
sim.simx_opmode_blocking);
    [~, prismaticJointHandle] = sim.simxGetObjectHandle(clientID, 'Horizontal_joint',
sim.simx_opmode_blocking);

    sim.simxSynchronousTrigger(ClientID, true);
    sim.simxStartSimulation(clientID,sim.simx_opmode_blocking);

    % Set orientation and velocity
    new_orientation = [0, 0, angRobot1];
    sim.simxSetObjectOrientation(clientID, objHandleDummy, objHandleBall,
new_orientation, sim.simx_opmode_blocking);
    sim.simxSetJointTargetVelocity(clientID, prismaticJointHandle, velocityrobot1,
sim.simx_opmode_blocking);

    % Collect data
    d_data = zeros(1, neededPoints);
    x_data = zeros(1, neededPoints);
    y_data = zeros(1, neededPoints);
    z_data = zeros(1, neededPoints);
    index = 1;

    while true
                sim.simxSynchronousTrigger(clientID);
                [~, position] = sim.simxGetObjectPosition(clientID, objHandleBall, -1,
sim.simx_opmode_blocking);
                %[~, velocity] = sim.simxGetObjectVelocity(clientID, objHandleBall,
sim.simx_opmode_blocking);

                x = position(1);
                y = position(2);
                z = position(3);
                d = sqrt((X_FINAL - x)^2 + (Y_FINAL - y)^2);

                x_data(index) = x;
                y_data(index) = y;
                z_data(index) = z;
                d_data(index) = d;
                index = index + 1;

                % Check if enough points are collected
                if index > neededPoints
                    break;
                end
    end
    sim.simxStopSimulation(clientID, sim.simx_opmode_blocking);
    J=Function_Calculator_V2(d_data, neededPoints, alpha, beta);
end
```

10.4 UseBayesopt()

```
fun = @(x) GetSolutionFromCoppeliaSimForBayesopt(x);

% Define optimization variables and bounds
vars = [
    optimizableVariable('angle', [-0.436332313, 0.436332313]);
    optimizableVariable('velocity', [0.1, 1.5]);
];

close all;

% Call bayesopt
```

```
results0 = bayesopt(fun, vars, ...
    'MaxObjectiveEvaluations', 150, ...
    'IsObjectiveDeterministic', true, ...
    'AcquisitionFunctionName', 'lower-confidence-bound', ... % here Acquisition
function is defined, also other parameters can be defined here.
    'Verbose', 1);
```

10.5 GetSolutionFromCoppeliaSimForBayesopt(x)

```
function J =  GetSolutionFromCoppeliaSimForBayesopt(x)
    load('OFFICIEEL_P14_a5b1_30x30.mat','alpha','beta','neededPoints');
    X_FINAL=0.374;
    Y_FINAL=0.154;

    %start coppeliaSim
    disp('Program started');
    sim = remApi('remoteApi');
    sim.simxFinish(-1);
    clientID = sim.simxStart('127.0.0.1', 19997, true, true, 5000, 5);


    velocityrobot1 = x.velocity;
    angRobot1 = x.angle;

    % Object handles
    [~, objHandleBall] = sim.simxGetObjectHandle(clientID, 'Foosball_ball',
sim.simx_opmode_blocking);
    [~, objHandleDummy] = sim.simxGetObjectHandle(clientID, 'Movable_Dummy',
sim.simx_opmode_blocking);
    [~, prismaticJointHandle] = sim.simxGetObjectHandle(clientID,
'Horizontal_joint', sim.simx_opmode_blocking);

    sim.simxSynchronous(clientID,true);
    sim.simxStartSimulation(clientID,sim.simx_opmode_blocking);

    % Set orientation and velocity
    new_orientation = [0, 0, angRobot1];
    sim.simxSetObjectOrientation(clientID, objHandleDummy, objHandleBall,
new_orientation, sim.simx_opmode_blocking);
    sim.simxSetJointTargetVelocity(clientID, prismaticJointHandle,
velocityrobot1, sim.simx_opmode_blocking);

    % Collect data
    d_data = zeros(1, neededPoints);
    x_data = zeros(1, neededPoints);
    y_data = zeros(1, neededPoints);
    z_data = zeros(1, neededPoints);
    index = 1;

    while true
                sim.simxSynchronousTrigger(clientID);
                [~, position] = sim.simxGetObjectPosition(clientID,
objHandleBall, -1, sim.simx_opmode_blocking);
                %[~, velocity] = sim.simxGetObjectVelocity(clientID,
objHandleBall, sim.simx_opmode_blocking);

                x = position(1);
                y = position(2);
```

```
                z = position(3);
                d = sqrt((X_FINAL - x)^2 + (Y_FINAL - y)^2);

                x_data(index) = x;
                y_data(index) = y;
                z_data(index) = z;
                d_data(index) = d;
                index = index + 1;

                % Check if enough points are collected
                if index > neededPoints
                    break;
                end
    end
    sim.simxStopSimulation(clientID, sim.simx_opmode_blocking);
    J=Function_Calculator_V2(d_data, neededPoints, alpha, beta);
end
```