Bayesian optimization for a skeletal muscle using the simulation framework OpenDiHu

Bachelor thesis

Author:
Lukas Bauer

Examiner:
Prof. Dr. rer. nat. Dominik Göddeke

Co-Examiner:
Prof. Dr. rer. nat. Miriam Schulte

Supervisors:
Carme Homs-Pons M.Sc.
Robin Lautenschlager M.Sc.

Universität Stuttgart
Institute of Applied Analysis and Numerical Simulation
Institute for Parallel and Distributed Systems
November 2024

Preamble

STATUTORY DECLARATION

I, Lukas Bauer, student number 3535006, hereby declare that I have written this thesis independently and have not used any other than the declared sources and resources. I have also explicitly marked all material that I have quoted either literally or by content from the used sources. Parts of the code in [**Bauer**] have been created by ChatGPT 3.5 (www.chatgpt.com). I confirm that the printed version is identical to the digitally submitted version.

....

Date Signature

Note of thanks

First of all I want to thank Prof. Dominik Göddeke and Prof. Miriam Schulte for proofreading this thesis and giving me the opportunity to do this bachelor thesis with this muscle simulation project. I also want to thank Carme Homs-Pons and Robin Lautenschlager for helping me whenever I needed help with this project. Finally, I want to thank my parents for supporting me and proofreading this work.

Contents

Abstract

English version

In this bachelor thesis we want to understand how muscles contract if they have been stretched before the contraction. To do this, we simulate muscles using the simulation framework OpenDiHu. We pull at the muscle with a certain force and observe the length of contraction afterwards. The goal is to find the force that maximizes the length of contraction, which we want to do using Bayesian optimization. Since this optimization process is very diverse, we want to use a Bayesian optimization model, that can find our maximum as fast and as accurately as possible. The best model for our case we found uses a Matérn kernel with $\nu = 0.5$, the constant mean function, the entropy search acquisition function and a stopping-xy criterion. With this model we find the stretching force for the greatest length of contraction to be the maximal force before the muscle tears. The model also performs well with different test functions that have different properties of functions we might optimize in the future.

Summary in German/Zusammenfassung

In dieser Bachelorarbeit wollen wir herausfinden, wie sich Muskeln zusammenziehen, wenn sie davor gestreckt wurden. Das machen wir mit dem Simulationspaket OpenDiHu, mit dem wir die Muskeln simulieren. Dabei strecken wir die Muskeln zuerst mit einer bestimmten Kraft und beobachten dann, wie weit sich der Muskel in einer bestimmten Zeit danach zusammenziehen kann. Das Ziel ist es, die Streckkraft zu finden, mit der sich der Muskel am besten zusammenzieht. Das wollen wir mit Bayes'scher Optimierung tun. Da diese Optimierungsmethode sehr individualisierbar ist, wollen wir ein Modell der Bayes'schen Optimierung finden, das das Maximum in unserem Fall möglichst schnell und genau findet. Das optimale Modell, das wir finden, verwendet die Matérn Kovarianzfunktion mit $\nu = 0.5$, die konstante Erwartungsfunktion, die entropy search Akquisitionsfunktion und ein stopping-xy Kriterium. Das Maximum, das wir damit finden, liegt bei der maximalen Kraft, bei der der Muskel noch nicht reißt. Wir testen das

Optimierungsmodell auch erfolgreich mit verschiedenen Testfunktionen, die verschiedene Eigenschaften von Funktionen haben, die wir in Zukunft eventuell optimieren wollen.

Introduction

Following [**Homs-Pons et al.**], the human body is full of perfectly optimized systems that are completely in sync with each other. For example, the nervous system and the skeletal muscles are working so well together that you know exactly where your arms are even with your eyes closed using the agonist-antagonist system. A chemical amputation cuts this connection between agonist and antagonist and confuses the nervous system, which leads to people not only having less control in their residual muscles, but also phantom pain. A medical procedure to overcome these problems is the agonist-antagonist myoneural interface (AMI), where the residual muscle does not get connected to a tendon that is connected to a bone, but to a tendon that is connected to the muscles antagonist. With this, the agonist-antagonist relationship is being restored and lessens the pain and improves control in the muscles. A future idea is to even control prostheses with this. Here we want to take a closer look at the AMI.
When we inspect a normal muscle in the human body, we find that even in its relaxed state it is stretched a little, since it is connected to tendons and bones. The force a muscle is being stretched with is related to its range of motion. Knowing this, we want to prestretch our residual muscles before connecting them with a tendon, so that we get a maximum range of motion. We can define a function $f : \mathbb{R}_{\geq 0} \to \mathbb{R}_{\geq 0}$ that maps a certain prestretch force to the corresponding range of motion. If we want to carry out AMI, we should use the the maximizer of f as the prestretch force for the residual muscles. Finding this optimal prestretch is not easy, since we cannot test with an actual muscle how much it can contract after a certain prestretch. Thus, we need to simulate a muscle. For that we will use the simulation framework OpenDiHu [**Maier**]. Unfortunately, simulating a muscle takes quite some time, so we are very restricted in how we can find $\text{argmax}(f)$. Since we do not have any information about the kind of function we are dealing with, can only evaluate f at single points, and do not have any derivatives, we can call f a blackbox function. Optimizing blackbox functions is challenging, especially if every evaluation takes a lot of time. But there is a method we can use: Bayesian optimization.

To eventually help doctors improve amputees' quality of life, the overall goal would be to find the best prestretches for a pair of agonist-antagonist residual muscles that are connected with

a tendon. This is, however, very complicated regarding the implementation using OpenDiHu, so we want to start by looking at a single muscle and its maximum range of motion. This can later be used to implement the general case. As another simplification we will be using a simple muscle in form of a cuboid as a model, because this is an easy example inside of OpenDiHu. Once everything works with this model, one can exchange the example muscle for a more realistic one.

To implement the optimization loop for the cuboid muscle, we first have to take a look at the mathematics behind Bayesian optimization and understand how it works. We will do this in chapter 1.

Then in chapter 2, we will look into the simulation framework OpenDiHu and understand how we can simulate muscles with it.

Since Bayesian optimization is very diverse, we have to apply the model to our case and look at the different options we have. We will then implement these options in Python using the package BoTorch [**Balandat et al.**] in chapter 3.

In chapter 4, we will analyse the results of the optimization and compare the different options we have for our model to find the ones that are suited best for our case.

Finally, in chapter 5, we will give some conclusions and outlooks.

Chapter 1

Bayesian optimization

As the name suggests, Bayesian optimization is a tool for solving the problem $\max_{x \in A} f(x)$ with $A \subset \mathbb{R}^d$ and $f : A \to \mathbb{R}$ continuous. The difference to most other optimization methods is that we can globally optimize noisy, gradient-free blackbox functions. That means: Firstly, $f(x)$ does not have to be deterministic, but can be random to a certain degree. Secondly, we do not need any derivatives of our function (in contrast to e.g. gradient descent); thirdly we do not know if our function has any structure that would make it easier to optimize (like convexity), and lastly, we can find its global optimum. Bayesian optimization can also quantify the uncertainty of $f(x)$ at every input point x, which helps us a lot with finding unknown maxima. Bayesian optimization works best if we have a lower dimensional problem ($d \leq 20$) and if our set A has a special form like a hyper rectangle $A = \{x \in \mathbb{R}^d : a_i \leq x_i \leq b_i,\ a_i, b_i \in \mathbb{R}, 1 \leq i \leq d\}$, but that is not required [**Frazier**]. A common use for Bayesian optimization are expensive to evaluate functions that might take minutes to hours to evaluate and might have some statistical noise. Often, these are simulations of which we want to optimize some parameters, just like in our case.
The idea is to create a statistical model of which we believe that it represents our function, and update this model with data points $y_i = f(x_i)$. To select the next query point of our updated statistical model, we use an acquisition function that gives us the point that maximizes the chance to give us $\max f(x)$ (cf. algorithm 1). We will look into each step in more detail later.
Bayesian optimization is very diverse, but it can broadly be categorised in parametric optimization, which we will look at in section 1.1, and non-parametric optimization, which will follow in section 1.2. These sections mainly rely on [**Shahriari et al.**], [**Frazier**] and [**Rasmussen et al.**, Chapter 2].

1.1. Bayesian optimization with parametric models

For our optimization problem we will not use parametric models. This is still worthy of looking into, because it is more descriptive than section 1.2, which is easier to understand after this

Algorithm 1 General algorithm for Bayesian optimization

1: **Input:** Function $f : \mathbb{R}^n \supset A \to \mathbb{R}$, $x_1 \in A$ and an acquisition function $\alpha : \mathbb{R}^n \times \mathcal{D} \to \mathbb{R}^n$ with $\mathcal{D}$ being the dataset of evaluations of f
2: **Output:** Dataset D of tuples $(x_i, f(x_i))$
3: **for** $n = 1, 2, \dots$ **do**
4: Select new x_{n+1} by optimizing acquisition function α: $x_{n+1} = \arg\max_x \alpha(x; D_n)$
5: Evaluate $f(x_{n+1}) = y_{n+1}$
6: Update dataset $D_{n+1} = \{D_n, (x_{n+1}, y_{n+1})\}$
7: Update statistical model
8: **if** Stopping criterion is met **then**
9: Break
10: **end if**
11: **end for**

section. The statistical methods in this section rely on [**Czado et al.**] and the Bayesian optimization method relies on [**Shahriari et al.**].

In this section we assume that the function we want to optimize is the density function of a random variable X distributed with a parameter $\vartheta^* \in \Theta \subset \mathbb{R}^n$ that we do not know. We assume that we can evaluate f_{ϑ^*} at any point, but we do not know its exact form. If we evaluate our function $f_{\vartheta^*}(x_i)$ at a point x_i, we gain some insight of what our parameter ϑ^* might be. Our goal is to approximate ϑ^* as accurately as possible, because once we know ϑ^*, we can simply calculate $\max_{x \in A} f_{\vartheta^*}(x)$, because now we have a closed form and can maximize this function using other optimization methods. To find the parameter, we assume that also $\vartheta \in \Theta$ has a given distribution with density function $\pi(\vartheta)$, the so called prior distribution. In step n we have given the prior distribution $\pi_n(\vartheta)$. If we evaluate $f_{\vartheta^*}(x_i)$, we get new information about ϑ^* and can update $\pi_n(\vartheta)$ to the so called posterior distribution $\pi_{n+1}(\vartheta)$ that describes ϑ^* better than $\pi_n(\vartheta)$. As we can see in algorithm 2, we can bring this procedure in the form of the algorithm in algorithm 1.

Remaining unclear in this algorithm is only how to update the prior to the posterior distribution. Generally, this posterior distribution does not have to be of the same form as the prior distribution, which makes working with it hard. Ideally, we would get a distribution of the same kind as the prior for our posterior. A distribution that does exactly that is called conjugated to the distribution of X. Luckily, we can usually find such a conjugated distribution (see appendix A.1) and update the statistical model in algorithm 2 step 7 using Lemma A.1.3.

By updating the distribution of ϑ, we gain more and more insight in ϑ^* and $\pi_n(\vartheta)$ gives increasingly more emphasis on ϑ^* and less on other parameters. With this we are closing in on ϑ^* until we are close enough so that we can choose a good approximation of the maximum of f_{ϑ^*} in step 4 of the algorithm, and therefore we are done.

Algorithm 2 Bayesian optimization for parametric models

Input: Function $f_{\vartheta^*} : \mathbb{R}^n \supset A \to \mathbb{R}$, distribution $\pi_1(\vartheta)$ of $\vartheta \in \Theta \subset \mathbb{R}^n$, $x_1 \in A$ and $f_{\vartheta^*}(x_1) \in \mathbb{R}$
Output: Dataset D of tuples $(x_i, f(x_i))$
for $n = 1, 2, \dots$ **do**
    Draw a sample ϑ_n of $\pi_n(\vartheta)$ and select new x_{n+1} by optimizing f_{ϑ_n}: $x_{n+1} = \arg\max_x f_{\vartheta_n}(x)$
    Evaluate $f_{\vartheta^*}(x_{n+1}) = y_{n+1}$
    Update dataset $D_{n+1} = \{D_n, (x_{n+1}, y_{n+1})\}$
    Update prior distribution $\pi_n(\vartheta)$ to the posterior $\pi_{n+1}(\vartheta)$
    if Evaluations y_i have not significantly improved over the last few steps **then**
        Break
    end if
end for

1.2. Bayesian optimization with non-parametric models

For our case we will use Bayesian optimization with non-parametric models. Contrary to the previous section, we now want to optimize arbitrary continuous functions. To do that, we need to find another statistical model that we can use in the general Bayesian optimization algorithm 1. The almost exclusively used model is the Gaussian process. The mathematical description of it mainly relies on [**Rasmussen et al.**] and [**Frazier**].

1.2.1. Gaussian processes.

Definition 1.2.1. ([**Rasmussen et al.**], Definition 2.1)
A Gaussian process is a collection of random variables, any finite collection of which have a joint Gaussian distribution.

In our case the collection of random variables has the form $\{X_a \mid a \in A\}$ with
$X = (X_1, \dots, X_n) \sim \mathcal{N}(\mu_0(X), \Sigma_0(X, X))$ for any finite set $\{X_1, \dots, X_n\}$. $\mu_0(X)$ is the mean of this distribution and is an n-dimensional vector

$$\mu_0(X) = \begin{bmatrix} \mu_0^1 \\ \vdots \\ \mu_0^n \end{bmatrix}$$

and $\Sigma_0(X, X)$ is the covariance matrix

$$\Sigma_0(X, X) = \begin{bmatrix} \Sigma_{11} & \dots & \Sigma_{1n} \\ \vdots & & \vdots \\ \Sigma_{n1} & \dots & \Sigma_{nn} \end{bmatrix}.$$

We will look into their exact form later, but now we assume them to be given.
We assume that our function f follows this Gaussian distribution for any collection of points: $f(x_1, \dots, x_n) \sim \mathcal{N}(\mu_0(X), \Sigma_0(X, X))$. Especially, we assume that also $f(x) \sim \mathcal{N}(\mu_0(x), \Sigma_0(x, x))$ for a single point $x \in A$. If we have evaluated $f(x_i)$, if follows that $f(x_i) \sim \mathcal{N}(f(x_i), 0)$, at least if we assume a noise free evaluation. We will later add noisy evaluations.
If we evaluate our function at a point, we want to infer from this evaluation what f looks like at other points. That means we are looking for $f(x)|f(X)$ with $X = (x_1, \dots, x_n)$ if we have evaluated f in the points $x_1, \dots, x_n$. We know that the vectors (x) and X are distributed normally:

$$(x, X) \sim \mathcal{N}\left(\begin{bmatrix}\mu(x)\\ \mu(X)\end{bmatrix}, \begin{bmatrix}\Sigma_0(x,x) & \Sigma_0(x,X)\\ \Sigma_0(X,x) & \Sigma_0(X,X)\end{bmatrix}\right)$$

Using appendix A.2, we find $f(x)|f(X) \sim \mathcal{N}(\mu_n(x), \sigma_n(x)^2)$ with

$$\begin{aligned}\mu_n(x) &= \Sigma_0(x,X)\Sigma_0(X,X)^{-1}(f(X) - \mu_0(X)) + \mu_0(x)\\ \sigma_n^2(x) &= \Sigma_0(x,x) - \Sigma_0(x,X)\Sigma_0(X,X)^{-1}\Sigma_0(X,x).\end{aligned} \tag{1.1}$$

This is how we update our statistical model, and the resulting distribution is the posterior distribution of the Gaussian process, similar to section 1.1. To check if this updated model makes sense, we can plug in a point x_i at which we have evaluated f already, to see that $\mu_n(x_i) = f(x_i)$ and $\sigma_n^2(x_i) = 0$. Because this is what we would expect, this updated model does make sense.
Since a lot of applications of Bayesian optimization are models where we would expect some noise in form of uncertainties in the given data or the like, we have to add that in to our statistical model. For that we assume that every evaluation has the form $y = f(x) + \varepsilon_y$ with $\varepsilon_y \sim \mathcal{N}(0, \sigma_m^2)$. We also assume that these ε_y are independent of each other, so that $\mathrm{Cov}(\varepsilon_y, \varepsilon_z) = \sigma_m^2 \delta_{yz}$ and therefore $\mathrm{Cov}(y_p, y_q) = \Sigma_{pq}(x_p, x_q) + \sigma_m^2 \delta_{pq}$. In total we then get $\mathrm{Cov}(X, X) = \Sigma_0(X, X) + \sigma_m^2 I$ and can use this to update the model from above:

$$(x, X) \sim \mathcal{N}\left(\begin{bmatrix}\mu(x)\\ \mu(X)\end{bmatrix}, \begin{bmatrix}\Sigma_0(x,x) & \Sigma_0(x,X)\\ \Sigma_0(X,x) & \Sigma_0(X,X) + \sigma_m^2 I\end{bmatrix}\right)$$

and

$$\begin{aligned}\mu_n(x) &= \Sigma_0(x,X)(\Sigma_0(X,X) + \sigma_m^2 I)^{-1}(f(X) - \mu_0(X)) + \mu_0(x)\\ \sigma_n^2(x) &= \Sigma_0(x,x) - \Sigma_0(x,X)(\Sigma_0(X,X) + \sigma_m^2 I)^{-1}\Sigma_0(X,x).\end{aligned}$$

$\sigma_m^2 I$ is only added to $\Sigma_0(X, X)$, because we only add that to already evaluated points and leave non-evaluated points free until we evaluate them. For σ_m we can use a fixed value if we assume a constant randomness, but we can also use it as a parameter to be adjusted during the optimization process if we do not know the exact factor of randomness.
We now want to look at the mean μ_0 and covariance matrix Σ_0. For now we assume that all parameters and coefficients are given and we will look into choosing them later.
The mean function is a continuous function, where we assign the mean of the normal distribution to a set of elements in A. This function can look like $\mu_0(X) = \mu + \sum_{i=1}^{p} \beta_i \Psi_i(X)$ with (given) parameters μ, β_i and parametric functions Ψ_i that are usually low order polynomials in X. But

this is hard to work with, so the usually used mean function is the so called constant mean $\mu_0(X) = \mu$ or even the zero mean $\mu_0(X) = 0$, where we assume a very simple mean everywhere and add the real mean in later when we update our Gaussian process model using equation (1.1):

$$\mu_n(x) = \Sigma_0(x, X)\Sigma_0(X, X)^{-1} f(X)$$

The covariance function creates a covariance matrix using a kernel function $k : \mathbb{R}^d \times \mathbb{R}^d \to \mathbb{R}$. The kernel function is a way to quantify the influence of two points on each other, where points that are close together typically have more influence on each other than points that are far away from each other. From this we get the covariance matrix

$$\Sigma_0(X, X) = \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \vdots & & \vdots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix}$$

A kernel is called stationary if the output only depends on the relative distance of the two inputs, and it is called non-stationary if it also depends on the absolute location. Every kernel depends on at least one of the following parameters:

- $l \in [0, \infty)$: The length-scale describes the range of influence a point has. The greater l is, the greater the range of influence.
- $\sigma \in [0, 1]$: The variance is a scaling factor that determines the magnitude of influence two points have on each other.
- $\sigma_b \in [0, 1]$: The variance for the linear kernel is a constant offset that describes the uncertainty around a given point c.
- $p \in [0, \pi]$: The periodicity determines the period of the periodic kernel. Points that have distance p to each other are regarded as the same point.
- $\nu \in (0, \infty)$: This is the so called smoothness parameter used for the Matérn kernel.

Here are some examples of the most used kernels [**Görtler et al.**] with some visualizations in figure 1:

- Periodic kernel:

  $$k(x_1, x_2) = \sigma^2 \cdot \exp\left(-\frac{2\sin^2(\frac{\pi||x_1 - x_2||}{p})}{l^2}\right)$$

  This is a stationary kernel that is best used if we expect periodic behaviour. But since we do not expect this, we are not going to use this one.

- Linear kernel:

  $$k(x_1, x_2) = \sigma_b^2 + \sigma^2(x_1 - c)(x_2 - c)$$

  This is a non-stationary kernel that is best used if we expect linear behaviour. But since we do not expect this, we are not going to use this one either.

- Radial Basis Function (RBF) kernel:

$$k(x_1, x_2) = \sigma^2 \cdot \exp\left(-\frac{||x_1 - x_2||^2}{2l^2}\right)$$

This is a stationary kernel that is pretty simple and used a lot, so this will be a candidate for our case.

- Matérn kernel:

$$k(x_1, x_2) = \sigma^2 \cdot \frac{2^{1-\nu}}{\Gamma(\nu)} \cdot \left(\frac{\sqrt{2\nu}||x_1 - x_2||}{l}\right)^{\nu} \cdot K_\nu \left(\frac{\sqrt{2\nu}||x_1 - x_2||}{l}\right)$$

$\Gamma(\nu)$ is the gamma function and K_ν is the modified Bessel function.
This is a stationary kernel. It is the most commonly used kernel since it is very customisable. We can adjust the smoothness parameter ν to represent our belief of the smoothness of our function. For some values of ν we get special cases:

  - $\nu = \frac{1}{2}: \; k(x_1, x_2) = \sigma^2 \cdot \exp\left(-\frac{||x_1 - x_2||}{l}\right)$
  - $\nu = \frac{3}{2}: \; k(x_1, x_2) = \sigma^2 \left(1 + \frac{\sqrt{3}||x_1 - x_2||}{l}\right) \cdot \exp\left(-\frac{\sqrt{3}||x_1 - x_2||}{l}\right)$
  - $\nu = \frac{5}{2}: \; k(x_1, x_2) = \sigma^2 \left(1 + \frac{\sqrt{5}||x_1 - x_2||}{l} + \frac{5||x_1 - x_2||^2}{3l^2}\right) \cdot \exp\left(-\frac{\sqrt{5}||x_1 - x_2||}{l}\right)$
  - $\nu \to \infty: \; k(x_1, x_2) = \sigma^2 \cdot \exp\left(-\frac{||x_1 - x_2||^2}{2l^2}\right) \to$ the RBF kernel

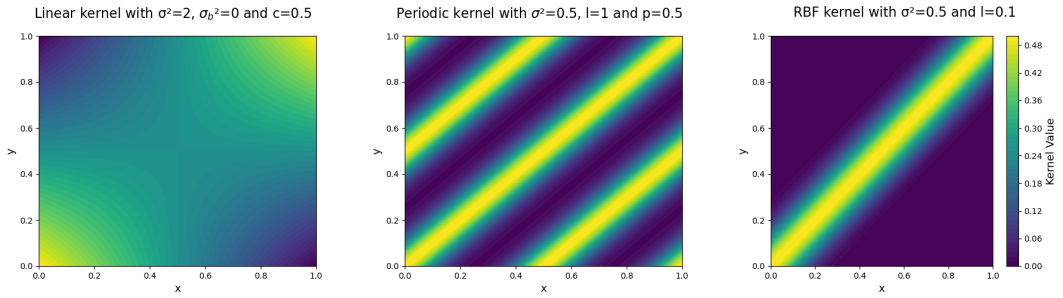

Figure 1. Here we can see three of the kernels visualized over the interval $[0,1]^2$ and their corresponding kernel value k(x,y) being plotted in color. Yellow corresponds to a high and dark blue to a low value.

Until now we have used all the parameters as if they are given. We now want to look into choosing them [**Frazier**]:
These parameters define the Gaussian process that is supposed to describe our function. Because we have some information about our function in form of evaluations $y_i = f(x_i)$, we want the Gaussian process to describe also these evaluations as precisely as possible using these parameters. The most intuitive approach to find the best parameters for the Gaussian process is the maximum

likelihood (ML) of our prior distribution:
The likelihood-function is defined as

$$L : \Theta \times \mathbb{R}^n \to [0,1],\ (\theta, x) \mapsto L(\theta, x) := p_\theta(x)$$

with $p_\theta(x)$ being the density function of our normal distribution with parameters θ. For our parameter space we choose the direct product of the spaces of all missing parameters, so that a single element in $\Theta = \mathbb{R} \times [0,1] \times [0,\infty)$ has the form $\theta = (\mu, \sigma, l)^T$ if we are using the constant mean and the RBF kernel. Our input space is $\mathbb{R}^n$, because we have n evaluations. The maximum likelihood is then defined as $\hat{\theta}_{\mathrm{ML}} := \mathrm{argmax}_{\theta\in\Theta}(L(\theta, Y))$ with $Y = (y_1, \ldots, y_n)$ being the vector of all the evaluations so far.
A different and a little more sophisticated way to choose the parameters would be the maximum a posteriori (MAP) estimate ([**Bassett et al.**]), where we assume a prior distribution $\pi(\theta)$ of the parameters to filter out unlikely parameters. The maximum a posteriori is then defined as $\hat{\theta}_{\mathrm{MAP}} := \mathrm{argmax}_{\theta\in\Theta}(\pi(\theta|Y)) = \mathrm{argmax}_{\theta\in\Theta}(p_\theta(Y) \cdot \pi(\theta))$ using Bayes' rule. One can see that the maximum likelihood is a special case of the maximum a posteriori using a uniform prior over the entire parameter space. That might result in unrealistic parameters, but if we do not know what the parameters might look like, that is the best option we have.
When we want to update our Gaussian process, we first choose our parameters using one of these methods, and then get our posterior distribution using equation (1.1). But there is also another way that does both at the same time, the fully Bayesian approach:
We want to find the best density function $p_{\tilde{\theta}}$ that describes $f(x)|Y$. For that we sample a few values $f(x) = y|Y$ and find with the law of total probability

$$p_{\tilde{\theta}}(f(x) = y|Y) = \int_\Theta p_\theta(f(x) = y|Y) \cdot \pi(\theta|Y) d\theta$$

for a prior density π of the possible parameters. Calculating this integral is usually not possible, too hard or too annoying, so that we approximate it:

$$p_{\tilde{\theta}}(f(x) = y|Y) \approx \frac{1}{J} \sum_{j=1}^{J} p_{\hat{\theta}_j}(f(x) = y|Y)$$

with $\hat{\theta}_j$ being samples from $\pi(\theta|Y)$. Using equation (1.1) we then can evaluate $p_{\hat{\theta}_j}(f(x) = y|Y)$. If we do this for multiple values of y, we can use the maximum likelihood to find the best parameters $\tilde{\theta}$ at the point x. This is computationally very expensive, but compared to the cost of evaluating f this might be vanishingly low.
Now we have a statistical model to describe our function and different strategies to update it. What we want to do now is to look at step 4 in algorithm 1 and find fitting acquisition functions.

1.2.2. Acquisition functions.

An acquisition function is a function $\alpha : \mathbb{R}^d \times \mathcal{D} \to \mathbb{R}$ with $\mathcal{D}$ being the set of pairs $(x_i, f(x_i))$ with the query points that we have evaluated our function at already and the corresponding f-value. From such a function we take the $\mathrm{argmax}(\alpha(x; D)) = x_{n+1}$ to be the next point for evaluation. Ideally, this α would give us sensible query points and is easy to maximize, so we

will look at some examples of commonly used acquisition functions. This section mainly relies on [**Shahriari et al.**] and [**Frazier**].

Probability of improvement.
One idea to find a good next query point is to find the point x_{n+1} that maximizes the probability that $f(x_{n+1}) > \tau$ for a given τ. Since $f(x) \sim \mathcal{N}(\mu_n(x), \sigma_n(x))$ using equation (1.1), the probability of that can be written as

$$\mathbb{P}[f(x) > \tau] = \Phi\left(\frac{\mu_n(x) - \tau}{\sigma_n(x)}\right)$$

with $\Phi(x) = \int_{-\infty}^{x} p_{0,1}(y)dy$ and $p_{0,1}(y)$ being the density function of the standard normal distribution. We can then define the probability of improvement (PI) acquisition function as

$$\alpha_{\mathrm{PI}}(x; D) := \Phi\left(\frac{\mu_n(x) - \tau}{\sigma_n(x)}\right).$$

The only thing left to choose is τ. A reasonable value would be the maximum of the evaluations so far: $\tau := \max_{i=0,\dots,n}(f(x_i))$.

Expected improvement.
A similar idea is to look at the expected improvement (EI). In order to find the point x where we are expecting the greatest improvement we not only factor in the probability of improvement, but also how much we are improving by. We again take the improvement threshold $\tau := \max_{i=0,\dots,n}(f(x_i))$ and look at the utility function $I(x, y) := (y - \tau) \cdot \mathbb{1}_{\{y>\tau\}}$. The EI is then

$$\alpha_{\mathrm{EI}}(x; D) := \mathbb{E}[I(x, y)] = (\mu_n(x) - \tau) \cdot \Phi\left(\frac{\mu_n(x) - \tau}{\sigma_n(x)}\right) + \sigma_n(x) \cdot p_{0,1}\left(\frac{\mu_n(x) - \tau}{\sigma_n(x)}\right).$$

In figure 2 we can see that EI increases with both $\mu_n(x)$ and $\sigma_n(x)$, so that also unexplored areas with high variance have a good chance of being selected. With this we get a tradeoff between low risk - low reward and high risk - high reward.

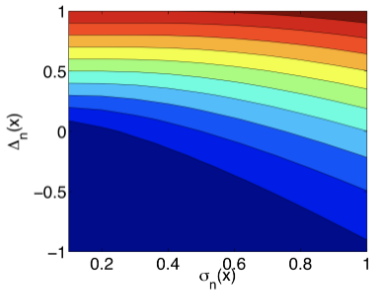

Figure 2. Here we see the value of $\alpha_{\mathrm{EI}}(x; D)$ increasing with both $\Delta_n(x) = \mu_n(x) - \tau$ on the y-axis and $\sigma_n(x)$ on the x-axis, with red being a high value and blue being low. [**Frazier**]

Generally, we cannot guarantee that we converge to the global maximum of our function using any acquisition function (see figure 3). But for EI with a special kind of functions and the Matérn kernel we can find convergence rates [**Bull**].

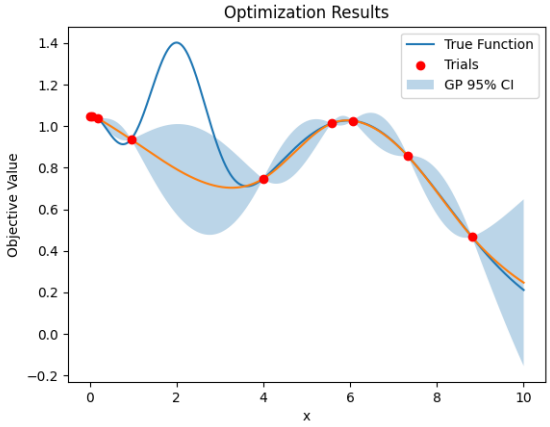

Figure 3. Example that Bayesian optimization with expected improvement does not necessarily converge to the global maximum. The blue graph is the true function we want to maximise. The red dots are the evaluations of f. Two of them have been chosen at random, the rest has been chosen by the EI acquisition function. The optimization loop terminated in the local maximum at 0 instead of the global maximum at 2. The orange function is the last approximation by the Gaussian process.

Knowledge Gradient.
The knowledge gradient (KG) is a way to try to improve EI. We define

$$\alpha_{\mathrm{KG}}(x; D) := \mathbb{E}[\mu^*_{n+1} | x_{n+1} = x]$$

with $\mu^*_{n+1} := \max_x(\mu_{n+1}(x))$ being the maximum of the posterior mean function after evaluating at another point. For $\alpha_{\mathrm{KG}}(x; D)$ we assume that we evaluate $f(x)$ and maximize the mean function of the posterior distribution to get μ^*_{n+1}. We do not want to actually evaluate $f(x)$, because that would defeat the purpose of an acquisition function. Instead we sample values $y_j \sim \mathcal{N}(\mu_n(x), \sigma_n(x))$ and approximate the expected value over the average of its $\mu^*_{n+1}(y_j)$:

$$\alpha_{\mathrm{KG}}(x; D) \approx \frac{1}{J} \sum_{j=1}^{J} \mu^*_{n+1}(y_j)$$

With this method we try to skip one evaluation of f and evaluate directly at the best point after this imagined evaluation. This has the potential to minimize the number of evaluations of f we need to find our maximum, but is has the drawback of being more risky, since we are approximating $f(x)$ using our posterior distribution, not knowing if our function does actually behave this way.

Entropy search.
The entropy search (ES, [**Hennig et al.**]) acquisition function is bringing an entirely new concept to the table. The idea is to reduce the entropy of the global maximizer, meaning that we try to decrease its uncertainty as much as possible. The entropy of a random variable X with distribution $\pi(x)$ is defined as $H(X) = -\int \pi(x) \cdot ln(\pi(x))\, dx$. We now want to find a random variable X^* that describes the distribution of the global maximizer x^* from the input space of f, of which we then can compute the entropy. For that, we sample functions f_j from the current posterior distribution of the Gaussian process and look for the global maximizer x_j^*. If we do this with enough samples, we get an approximate distribution X^* of the global maximizer, of which we can approximately calculate the entropy $H(X^*)$. What we want to do now is to reduce this entropy as much as possible using one evaluation of f. For that we look at $\mathbb{E}[H(X^*|f(x))]$, which is the expected entropy of the global maximizer after we have evaluated $f(x)$. To calculate that, we sample values $y_j \sim \mathcal{N}(\mu_n(x), \sigma_n(x))$ and calculate the corresponding $H(X^*|f(x) = y_j)$ by sampling new functions $\tilde{f}_k$ from the induced posterior distribution of the Gaussian process and calculating its maximizer. With this we can approximate

$$\mathbb{E}[H(X^*|f(x))] \approx \frac{1}{J} \sum_{j=1}^{J} H(X^*|f(x) = y_j).$$

Putting all that together, we find

$$\alpha_{\mathrm{ES}}(x; D) := H(X^*) - \mathbb{E}[H(X^*|f(x))].$$

Here we can also find smart ways to calculate everything, trying to bring the computational cost down, but none of the ways are actually cheap.

There are many more acquisition functions, all of which bring something new and potentially better to the table, but these are some of the most commonly used ones. Unfortunately, there is not one single acquisition function that gives an overall better performance over all possible problems than any other function [**Shahriari et al.**], so we have to try and look what the best acquisition function in our case is.

1.2.3. Stopping criteria.

The last step that we have not addressed yet in algorithm 1 is step 8, the stopping criterion.
The algorithm creates a sequence $(z_n)_{n\in\mathbb{N}}$ with $z_n = (x_n, y_n)$ and $y_n = f(x_n)$. A stopping criterion would use this sequence to evaluate if we are close enough in a metric d to the global maximum. If $z^* = (x^*, y^*)$ is the global maximum and ε a given constant, a point z_n is "close enough" to the maximum, if $d(z_n, z^*) < \varepsilon$. Examples of this metric d are:

- $d_1(z_n, z^*) = ||z_n - z^*|| := ||(x_n - x^*, y_n - y^*)||$
- $d_2(z_n, z^*) = ||x_n - x^*||$
- $d_3(z_n, z^*) = ||y_n - y^*||$

d_3 is only a half-metric, because $d_3((x_1, y_1), (x_2, y_2)) = 0$ can happen if $y_1 = y_2$ and $x_1 \neq x_2$, but since we can have multiple maxima, we can identify different points with the same f-value with each other without creating contradictions. d_2, however, is a full metric if we assume exact evaluations of f, otherwise it is also a half-metric, which also does not lead to any problems. For the norm we can use any norm of the $\mathbb{R}^{d+1}$.

Unfortunately, we do not know z^* or if $(z_n)_{n\in\mathbb{N}}$ converges. Using the Bolzano-Weierstraß theorem, we know that $(z_n)_{n\in\mathbb{N}}$ contains a convergent subsequence, because both $(x_n)_{n\in\mathbb{N}}$ and $(y_n)_{n\in\mathbb{N}}$ are bounded. We can even say that the subsequence of $(y_n)_{n\in\mathbb{N}}$ is monotonically increasing, since the probability that an acquisition function finds a value greater than the greatest value so far is non-zero. So we only have to consider these subsequences for our stopping criterion.

Using d_3 as metric, we only have to look at $(y_n)_{n\in\mathbb{N}}$ and can create this monotonically increasing subsequence by only adding points y_n that are greater than every other sequence member in this subsequence so far. We can define an improvement threshold $\varepsilon > 0$ and a number $m \in \mathbb{N}$ and use this to define a stopping criterion: If the current best value y_n is not being improved by more than ε in the next m iterations, we assume that y_n is close enough to the maximum and cannot be improved by more than ε, so we stop the loop.

With d_2 we can do something similar with again a number $\varepsilon > 0$ and $m \in \mathbb{N}$: If there are at least m members of $(x_n)_{n\in\mathbb{N}}$ are in the ε-neighbourhood of a certain x_i, we assume that this creates a subsequence that converges to a point in this ε-neighbourhood and stop the loop. However, this accumulation point does not have to be the maximizer of f, so this might not be a good time to stop. But we can combine this criterion with the criterion for d_3 to find a criterion for the metric d_1: In addition to the requirement that there have to be m members of $(x_n)_{n\in\mathbb{N}}$ in the ε-neighbourhood of x_i, we also demand that one of the points x_n in this neighbourhood corresponds to the maximum of the sequence $(y_n)_{n\in\mathbb{N}}$ so far. We can then stop the loop.

These are some practical stopping criteria.

With this we have every component of algorithm 1 to implement it and can try different combinations to see what suits our case best.

Chapter 2

Simulating a muscle in OpenDiHu

Before we start implementing Bayesian optimization, we should look into the case we want to optimize. As mentioned in the introduction, we have a given muscle that we want to stretch with a prestretch force and observe how much it can contract in a certain amount of time. For that we will use the simulation framework OpenDiHu [**Maier**]. In this chapter, we will look at the muscle models OpenDiHu uses in section 2.1 and then at the implementation of stretching and contraction with OpenDiHu in section 2.2.

2.1. Skeletal muscle model

We use the same model as [**Homs-Pons et al.**] and [**Bradley et al.**].
A skeletal muscle consists of fibers that themselves consist of sarcomeres, which are the smallest contracting unit. To activate a muscle, there are motor neurons that send out electrical current to the middle of the fibers. This electrical current runs along the fibers through the sarcomeres that consequently make the muscle contract.

OpenDiHu mimics this hierarchy and represents the muscle by a 3D and a $n\times$1D mesh for a fixed $n \in \mathbb{N}$, and 0D points that can be interpreted as a 0D mesh. The 3D mesh represents the entirety of the muscle and its geometry, like the position of the nodes of the lower dimensional meshes. The $n\times$1D meshes are the n fibers that lie within the 3D mesh. Inside them the propagation of the electrical current is being computed. These fibers consist of the 0D points which are the sarcomeres that generate the internal forces of the muscle which leads to it contracting (see figure 1). The spatial sizes of the three different dimensional meshes differ a lot, so we have to apply multi-scaling to the model. To capture the muscle's phenomena as accurately as possible, we also need to implement time sub-cycling and strang splitting, which means that we decouple the time steps of the simulation of these meshes. The sarcomeres need a much higher time step resolution than the fibers and the muscle do, so we synchronise the sarcomeres and the fibers after each time step of the fibers and several of the sarcomeres and transfer information. We do the same with the fibers and the muscle geometry, because also the fibers require a higher time step resolution

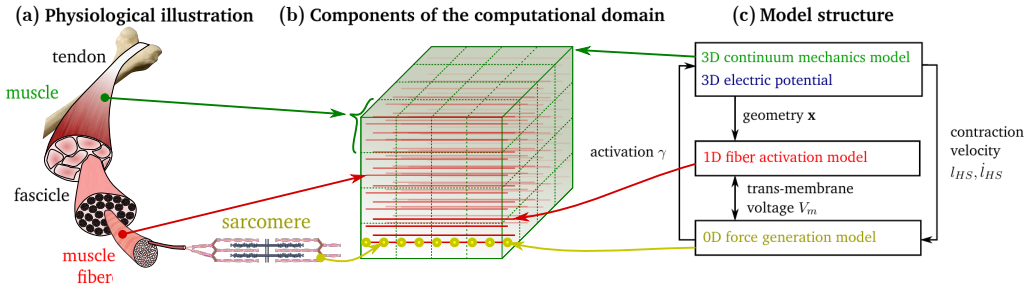

Figure 1. Visualisation of the muscle structure in OpenDiHu, [**Maier**], page 13. Apart from the 3D electric potential, this is the model we use.

than the muscle itself. OpenDiHu has implemented the solvers for the different meshes and the synchronisation between them.

The macroscopic deformations of the muscle are being represented by a continuum mechanics model with the conservation of momentum and mass as governing equations:

$$\rho \ddot{x} = \nabla \cdot P \tag{2.1}$$

$$\text{div } \dot{x} = 0 \tag{2.2}$$

These equations hold for every time step. x is the muscle's displacement, $\dot{x}$ its velocity, $\ddot{x}$ its acceleration, ρ its density and P is the first Piola-Kirchhoff stress tensor. P can be divided into a passive and an active part. The deformation of the muscle creates passive forces inside it. The resulting stress can be described with the passive part of the Piola-Kirchhoff stress tensor $P_{\text{passive}}(F)$. F is the deformation gradient tensor with $\det(F) = 1$ because we assume incompressibility. The active forces created by activating the muscle create the active stress $P_{\text{active}}(F, \gamma)$, where γ is the activation parameter that summarises the active force states of the sarcomeres. With the pressure p, we can combine these parts to the Piola-Kirchhoff stress tensor:

$$P = P_{\text{passive}}(F) + P_{\text{active}}(F, \gamma) - pF^{-T} \tag{2.3}$$

The summand $-pF^{-T}$ has to be added to keep the incompressibility constraint.

In the 1D meshes we compute the propagation of the electrical current. Its governing equation is a differential equation about the fiber's transmembrane potential V_m:

$$\frac{\partial V_m}{\partial t} = \frac{1}{C_m A_m} \left(\sigma_{\text{eff}} \frac{\partial^2 V_m}{\partial x^2} - A_m I_{\text{ion}}(V_m, y) + S(V_m) \right) \tag{2.4}$$

A_m is the fiber's surface to volume ratio, C_m is the membrane's electrical capacitance, σ_{eff} is the effective conductivity, and $S(V_m)$ is the source term of the fiber's activation by the motor neurons. $I_{\text{ion}}(V_m, y)$ is the ionic current across the membrane with the summary of state variables y.

For the 0D subcellular muscle model we compute the state variables through a set of coupled ordinary differential equations

$$\begin{aligned} I_{\text{ion}} &= I_{\text{ion}}(y, V_m) \\ \frac{\partial y}{\partial t} &= G(y, V_m) \end{aligned} \tag{2.5}$$

where $G(y, V_m)$ is given by a Hodgkin-Huxley model of the sarcomeres. With this we can also find the activation parameter γ that we need for the continuum mechanics through

$$\gamma(y, l_{hs}) = f_{f-l}(l_{hs}) \frac{A_2 - A_2^{\min}}{A_2^{\max} - A_2^{\min}} \tag{2.6}$$

with the force-length ratio f_{f-l} depending on the length of a half-sarcomere l_{hs}. $A_2 \in y$ is the concentration of post power-stroke cross-bridges and $A_2^{\max}$ and $A_2^{\min}$ the corresponding maximum and minimum.

Together these equations describe the contraction of a muscle which we now want to simulate.

2.2. Implementation in OpenDiHu

We apply this model structure to a toy muscle that we want to stretch and contract. For simplicity reasons we are using a very easy geometry for the muscle: a cuboid. It has the dimensions $3 \times 3 \times 12$ cm and contains 100 fibers with 100 sarcomeres each. The fibers are directed in the z-axis, which is the axis in which the muscle is the longest, see figure 2. The 3D mesh is divided into cubes with the side length 0.5 cm that we use as cells for the 3D solvers.

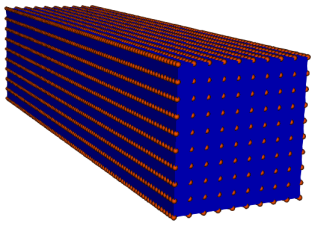

Figure 2. A cuboid muscle of dimensions $3 \times 3 \times 12$ cm containing 100 fibers with 100 sarcomeres each. The red points are the sarcomeres, which can be connected to create 100 1D meshes along the z-axis. The blue cuboid is the muscle geometry.

To stretch the muscle, we will use the tensile test example inside OpenDiHu. This example pulls with a given force along the z-axis. The force is applied gradually so that the muscle does not tear. We pull at the muscle until it cannot stretch any further with our given force. The muscle's volume remains constant throughout this stretching process, because we assume an incompressible material, more precisely an incompressible Mooney-Rivlin material with parameters $c_1 = 3.176 \cdot 10^{-10}$ and $c_2 = 1.813$.

After the muscle has been stretched, we want to contract it. As mentioned above, the muscle is being activated in the middle by neurons, which leads to electrical current flowing through the

fibers and activating the sarcomeres as described in equations (2.4), (2.5) and (2.6). These create a force which they transfer to the fibers that then contract. The fibers transfer this contraction information to the muscle geometry. The muscle geometry is what we use to visualise the results of the contraction. It also ministers to macroscopic forces that deform the muscle as described in equations (2.1) and (2.2), and transfers this to the fibers. The sarcomeres follow a very deterministic behaviour, because they get the input of the activation and generate the corresponding contraction forces. The fibers are more complicated, because they need to minister the deformation of the muscle, the contraction forces, and the propagation of the electical current. The muscle geometry has similar challenges. OpenDiHu has implemented solvers that approximate the solutions to the equations and allow us to simulate the muscle's behaviour.

The solvers we need for stretching and contracting a muscle in OpenDiHu are the FastMonodomainSolver, HyperelasticitySolver, and DynamicHyperelasticitySolver.
The FastMonodomainSolver solves equations (2.4) and (2.5) in the fibers by parallelizing the different fibers. It describes the propagation of the electrical current if the motor neurons activate the muscle. Otherwise it only updates the position of the fibers induced by macroscopic deformations of the muscle.
The HyperelasticitySolver solves equations (2.1) and (2.2) by illustrating the deformation of the macroscopic deformations in the muscle's geometry. It is a solver for static problems that uses both Finite elements and Taylor-Hood elements methods. By adding Dirichlet and Neumann boundary conditions, we can simulate a prestretch force. Without any boundary conditions the only forces in the muscle are internal contraction forces.
The DynamicHyperelasticitySolver has the same function as the HyperelasticitySolver, but only works with dynamic problems.
Algorithm 3 shows the combination of the different solvers that gives us the simulation of stretching and contraction.

Algorithm 3 Prestretch and Contraction in OpenDiHu

```
1: Coupling: Consecutive execution
2:     Prestretch:
3:         Coupling: Execute separately and transfer information in the end
4:             Fibers: FastMonodomainSolver without activation
5:             Solid Mechanics: HyperelasticitySolver with boundary conditions
6:     Contraction:
7:         Coupling: Execute separately and transfer information every 50 fiber-timesteps
8:             Fibers: FastMonodomainSolver with activation
9:             Solid Mechanics: DynamicHyperelasticitySolver without boundary conditions
```

Step 4 of algorithm 3 is necessary, even though the solver has no input since we are not activating the fibers. If we did not couple fibers and solid mechanics, the fibers would not be updated in the prestretch section of algorithm 3 and would therefore not be stretched. We can see this wrong

prestretch in figure 3. This leads to the fibers and solid mechanics not being synchronised and the muscle not being contracted correctly.

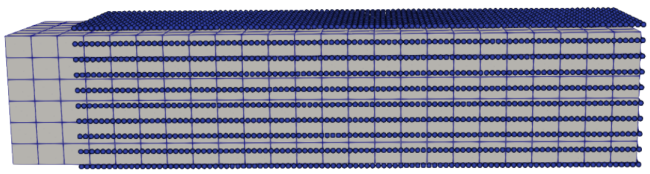

Figure 3. This is a visualisation of the muscle after it has been stretched without synchronising the fibers with the solid mechanics. The gray cuboid is the muscle geometry that has been stretched, the blue dots together form the fibers that have not been stretched. If we activated the muscle in this state, the fibers would only contract in the right part of the muscle geometry and only deform it there and leave the left part, where we are missing the fibers, untouched. Since this is not how a muscle contracts, we have to synchronise the fibers with the solid mechanics and with that stretch them as well.

Simulation time.

The simulation describes the behaviour of the muscle in a certain amount of time that we have to specify in the beginning. To find a good way to quantify the range of motion of a muscle, we have to choose this simulation time sensibly.

The activation of the muscle happens in phases of short pulses of electrical current induced by the neurons in the middle of the muscle. After the first pulse has spread far enough in the muscle, the next one is sent out. A way to define a muscle's range of motion would be to limit the activation phases to just one that we will let spread until it has reached both of the muscle's ends. Once the muscle is not reacting much to the activation anymore, we end the simulation and calculate the contraction in this time. For our muscle that is about 40 milliseconds. Another way would be to let the simulation run until the maximum contraction of the muscle has been reached, no matter how many phases of activation it takes. This has multiple drawbacks. For one, this takes a very long time which we do not have. For another, we might get the result that all prestretches can contract to the same minimal length, just in different times. A muscle that contracts very slowly because of its large prestretch would be rated better than a muscle that contracts extremely fast, but has not been prestretched by much. Since amputees do not want large contraction with extremely low contraction speed, but want to use their muscles as normally as possible, speed is also a very important factor. This is why we will only be looking at one activation phase.

Chapter 3

Implementation

We now know how Bayesian optimization and our muscle work. As we have seen, Bayesian optimization is very customisable and has a lot of options for different Bayesian optimization models. Since simulating the stretching and contraction of a muscle take much time, we want to find a Bayesian optimization model that finds an accurate maximum in as few iterations as possible. To find this optimal model, we want to have as little options as possible to compare in the end, so that the finding process is easier and clearer. For that we will summarise all the options from chapter 1 and discard some of them in section 3.1 to shrink the set of possible models. In section 3.2 we then want to take a closer look at implementing these options. We will use the results from this chapter to find the optimal Bayesian optimization model for our case in chapter 4.

3.1. Options for our model

For the Bayesian optimization model we have to choose an option from each of the following areas:

- The type of kernel function
- The type of mean function
- How to update the Gaussian process
- The kind of noise
- The acquisition function
- The stopping criterion

The type of kernel function.
In section 1.2.1 we have already summarised the most commonly used kernels: The periodic, linear, RBF and Matérn kernel. As mentioned there, we will not be using the periodic or linear kernel, since they only should be used if we expect f to be periodic or linear. For our case we

do not expect this behaviour of f, but even if it does turn out to behave linearly or periodically, we will not use these kernels, since we will want to use this optimization model for other cases as well. That leaves us with the RBF and Matérn kernel. They are very popular, because they can be used for a broad variety of functions and especially the Matérn kernel is very customisable by adjusting the smoothness parameter ν. Therefore we will use these kernels and try different parameters for ν.

The type of mean function.
The three options for the mean function are the constant mean of the form $\mu_0(X) = \mu$, the zero mean of the form $\mu_0(X) = 0$, and the polynomial mean of the form $\mu_0(X) = \mu + \sum_{i=1}^{p} \beta_i \Psi_i(X)$. The polynomial mean is usually being used if f is expected to have a special form that can be approximated by a polynomial like μ_0 to a certain degree. Since we do not know the form of f and want to keep it as general as possible, this is not going to be a good fit for our case. Additionally, this mean function contains a lot of parameters that need to be chosen when updating the Gaussian process, which makes this process more complicated. Therefore our approach will be based on the majority preference, using the constant and zero mean. These functions are very general and easy to work with and are thereby well suited for our case.

How to update the Gaussian process.
Our options are the maximum likelihood, the maximum a posteriori, and the fully Bayesian approach. Both the maximum a posteriori and the fully Bayesian approach require a prior distribution over the parameters we want to find, which represents our beliefs of these parameters. But we again do not know the form of the parameters or of f. If we did have an idea, what the parameters in the case of our cuboid muscle with the certain material parameters looked like, these methods might still not be optimal, because the form of our parameters could change if we use a different muscle or different material parameters. This is why we will not be using either of these methods and will solely rely on the maximum likelihood.

The kind of noise.
The noise of our evaluations of f is the randomness in our muscle simulation. For a muscle to contract it has to be activated by neurons. These neurons are located in the middle of the muscle and send out electrical current that travels through the muscle, which then contracts. In reality, these neurons are not located in exactly the middle of the muscle, but are randomly spread out by a certain amount. This leads to some significant randomness in the muscle contraction. The randomness can be adjusted inside OpenDiHu. If our randomness factor is 0, we get a deterministic contraction with the only noise being rounding errors. In this case we should use a small fixed noise. We will discuss the exact value in section 3.2. We could also use variable noise which is being chosen in the process of updating the Gaussian process, but since we already have some information about the noise, we should use it. If we have a realistic randomness factor of about 0.1, the contraction is not deterministic and therefore we do not have any information about the noise. In that case we should use variable noise.

The acquisition function.
We have looked at the acquisition functions expected improvement, probability of improvement, knowledge gradient and entropy search. There are a lot more acquisition functions we could have looked at, but we chose those four, because they are both popular and sound promising. Therefore, we will try all four of them.

The stopping criterion.
In section 1.2.3 we have found two stopping criteria. One only depends on the evaluations y_n of f which we will call the stopping-y criterion, the other one depends on both x_n and y_n which we will call the stopping-xy criterion. We do not know if any criterion is a better fit for our case than the other one, so we will try both of them. Both criteria require an $\varepsilon > 0$ and $m \in \mathbb{N}$. Through some experimentation we have found that $\varepsilon = 10^{-4}$ and $m = 3$ work fine for the stopping-y criterion, and $\varepsilon = 0.05$ and $m = 3$ for the stopping-xy criterion. We might need to adjust these parameters during the process of finding the best optimization model, but they are of this scale.

3.2. Implementation using BoTorch

BoTorch [**Balandat et al.**] is a python package for Bayesian optimization that is heavily based on PyTorch. It contains many functions for Bayesian optimization that we can put together to create an individual optimization loop, which is exactly what we want. However, there are some restrictions we have to keep in mind. For example, the smallest noise BoTorch allows is 10^{-6}. This is the value we will use for the fixed noise if we simulate the muscle without randomness. Another limitation is with the smoothness parameter ν of the Matérn kernel: The only implemented values of ν are 0.5, 1.5 and 2.5 which decreases the size of our search space for this parameter by a lot.
With this we have everything for the optimization loop itself, but we are still missing some initial parameters.

Initial points.
The optimization loop can only choose new points to evaluate based on previous points. Therefore, it requires some initial points that we have to choose somehow. We can choose evenly distributed points over our input space or we can choose them randomly. Evenly distributed points have the advantage of getting information about f everywhere. Any point in the input space is in the range of a point that we have evaluated already, so that we can induce a decently good approximation of its f-value with our Gaussian process. In higher dimensions this has the disadvantage of scaling very poorly. If we choose the initial points randomly with Monte-Carlo sampling, we might leave some areas without any information about f, but we do cover most of the input space and scale much better in higher dimensions than evenly distributed points [**Asmussen et al.**]. BoTorch offers an option that combines the advantages of both these methods using Sobol sequences. With that method we choose random points, but we ensure that they are covering the input space much more evenly than the Monte-Carlo method. This ensures that

we do not leave any "holes" in the input space, but we scale just as well in higher dimensions as Monte-Carlo. Our case is one dimensional, so there is not much of a difference between Sobol sampling and evenly distributed points, but we are still going to use the Sobol method, because we might want to reuse the code for higher dimensional problems.
We will use two initial points for our optimization process. We want to keep that number as low as possible, because we want to minimise the number of evaluations of f that we do without any information. But only one initial point is sometimes not enough for the acquisition function to choose a sensible next query point. It can happen that the acquisition function chooses two points very close to the initial point in the next two iterations, because it does not have enough information to choose points further away, which triggers the stopping-xy criterion. This leads to us not finding the maximum. With two points this does not happen.

Bounds.
The optimization process only works on a bounded input space. Our function f is defined on $\mathbb{R}_{\geq 0}$, which gives us a lower bound of 0. f is not bounded above, but if we stretch a muscle too far, it tears and cannot contract anymore. Stretching the muscle until it tears and taking a force just under this stretching force as upper bound would work well. In our case that is about 30 N. This might not be the best upper bound we can find, because we do not expect the maximum to be at a force that nearly tears the muscle. With a smaller interval we would need less evaluations of f to find the maximum than with a bigger interval. If we had information about the muscle's behaviour, we could reduce the upper bound for a more efficient optimization process. An assumption would be that the muscle contracts less with a prestretch force that stretches the muscle to e.g. double its length than it could with a lower prestretch force. We can find this force and with that the upper bound using bisection. Unfortunately, this is still just an assumption, so we will not be using this method, but if it turns out to be true, we should definitively use it. For now we will stick with the 30 N that we have found through experimentation.

Now we have looked at all the pieces that we need to create the optimization loop and we can bring them together in the following pseudo code in algorithm 4. The real code can be found under [**Bauer**] in the tag "Bachelor-thesis" in *cuboid_muscle/BayesOpt.py*.
The code has another feature that we can toggle on that we did not mention in the pseudo code to keep it simpler and clearer. Once the optimization loop is done and has found a maximum, we can choose to add another query point at a location we want to inspect a bit more closely. A situation where this might be useful would be if we have found a maximum without evaluating f in a certain area, because the acquisition function did not expect this area to hold a maximum, just like in figure 3 in section 1.2. If we want to figure out if there is actually no maximum in that area or if the optimization process overlooked it, we can add a query point in this area after the optimization process is done. If it turns out that there is a maximum, we can save all the evaluations of f so far, and restart the optimization loop with this data as initial data. The optimization loop should then converge to the maximum we have found in the untouched area.

Algorithm 4 Bayesian optimization loop for our case

1: **Input:** A kernel function, a mean function, the kind of noise, an acquisition function, a stopping criterion, an upper bound
2: **Output:** A maximizer and its maximum
3: Find 2 initial points using the sobol method
4: Calculate the muscle's length of contraction with the corresponding prestretch forces given by the initial points by simulating it in OpenDiHu. This is an evaluation of f
5: Initialise a Gaussian process with the evaluations of f, the kernel function, mean function and noise from line 1
6: Update the Gaussian process by finding fitting parameters and calculating the posterior distribution
7: **for** $n = 1, 2, \ldots$ **do**
8: Maximise the acquisition function from line 1 to find next query point
9: Evaluate f at the query point
10: Add the new evaluation of f to the dataset of f-evaluations of the Gaussian process
11: Update the Gaussian process
12: **if** The stopping criterion from line 1 is met **then**
13: Break
14: **end if**
15: **end for**
16: Return the maximum of the evaluations of f and its maximizer

Chapter 4

Results and discussion

In section 4.1 we will look at some first results. After that in section 4.2 we want to find an optimal Bayesian optimization model and discuss what "optimal" means. Finally, in section 4.3 we will test this optimal Bayesian optimization model.

4.1. First results

As a result of this bachelor thesis we want to find the maximiser of the function $f : [0, 30] \to \mathbb{R}_{\geq 0}$ that maps a prestretch force to the muscle's contraction with this force, just as mentioned in the introduction. 30 N is our maximum prestretch force, because the muscle tears if it is being stretched further. First of all we evaluate f at seven equidistant points, so every 5N, by simulating the muscle's contraction to approximate f using a Gaussian process. Without using Bayesian optimization yet, we can get a feeling for what f looks like and where the maximum might be. We can see the result in figure 1. As shown there, the muscle contracts best if it is being stretched as far as possible and f is almost linear. This is not what we expected, since intuition would say that a muscle that is close to tearing would contract less than a muscle that has been stretched less. One explanation would be that the simulation model does not capture the muscle's behaviour correctly, but for now we will assume that this is not the case and our simulation does represent the reality. Defining a good muscle model would be out of the scope of this work. This makes finding an optimal Bayesian optimization model hard, because most models will find the maximum in this case very fast, but we also want to apply this model to other functions, where some models might perform poorly.

One example where we want to use this optimization model is the AMI simulation with two coupled muscles mentioned in the introduction. There we stretch both muscles with our prestretch force and get the function $\hat{f}$ that maps this force to the corresponding range of motion of this coupled system of muscles. The maximum of $\hat{f}$ cannot be at the maximum force, because then no muscle can contract without tearing its antagonist. Therefore, it is expected that $\hat{f}$ has its

maximum somewhere in the middle, but is overall going to look similar to f, since it is based on the same simulation.

To find a good optimization model, we could test the different models on $\hat{f}$, which has the only small problem that we do not have $\hat{f}$. To ensure that the optimization model also works for non linear functions, we create a few test functions $\hat{f}_i$ that $\hat{f}$ could look like, test our models on these functions and choose the model that performs best over all these functions as the best model for our cases. The functions we are going to use are (for visualisation see figure 2):

- $\hat{f}_1(x) = -3x(x - 1.3) + 0.3$
- $\hat{f}_2(x) = e^{-(5x-3)^2} + 0.2 \cdot e^{-(30x-22)^2}$
- $\hat{f}_3(x) = x + e^{-(5x-5)^2} \cdot \sin(5x - 1.5)$
- $\hat{f}_4(x) = e^{-(10x-2)^2} + e^{-\frac{(10x-6)^2}{10}} + \frac{1}{(10x)^2+1}$
- $\hat{f}_5(x) = 0.5 - 3x(x - 1) \cdot \sin(5x)$

These functions have different characteristics than the real $\hat{f}$ might have. These include both only single maxima, multiple maxima, but also differently steep maxima, which can be described as the Lipschitz constant in a small neighbourhood of the maximum. We also chose functions of different kinds, e.g. partially linear, quadratic, trigonometric or exponential. Together these test functions give us a broad spectrum of functions our optimal optimization process can optimize.

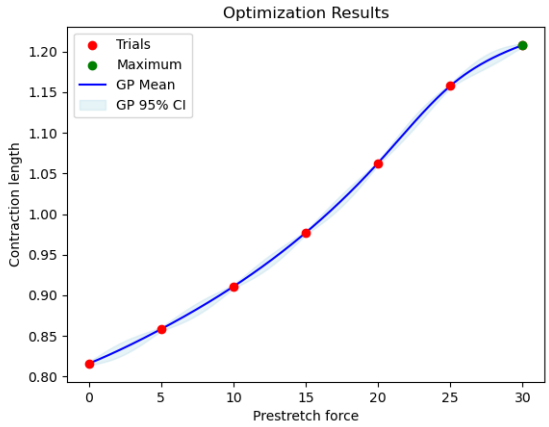

Figure 1. This is an approximation of f created by a Gaussian process initialised be the red equidistant query points. We find the maximum at 30 N. The blue graph is the mean function of the Gaussian process posterior, which turns out to be almost linear.

4.2. The optimal Bayesian optimization model

Our goal is to find a Bayesian optimization model that gives us a good approximation of the maximum in as few iterations as possible. One model is better than another one, if it needs less

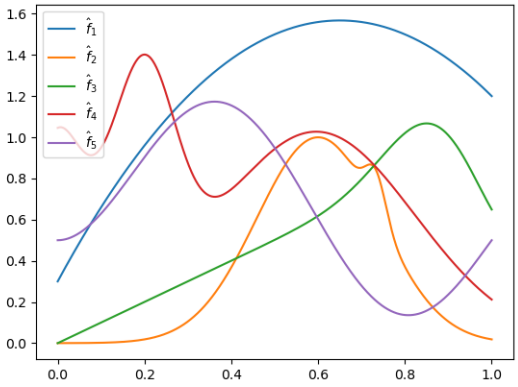

Figure 2. These are the test functions we are using to test the optimization models with. They include different characteristics the true function $\hat{f}$ might have.

evaluations of the objective function and has a higher accuracy. If one needs less evaluations but has a lower accuracy, there is no strictly better model. In this case, we have to weigh our options individually to value one model higher than the other one. Therefore, an optimal model might not be clearly defined, but as long as we can find a model that performs very well in comparison to most other models, we can define it as the optimal optimization model.

We will start by taking one model as basis for comparison, discard models that perform worse and take better models as new basis for comparison. This way we continue what we started in section 3.1 and shrink down the number of possible models even further until we have a small number left that we can compare directly.

The model we will start with, has a Matérn kernel with $\nu = 1.5$, a zero mean, uses expected improvement and the stopping-xy criterion. Using algorithm 4, we find that over all the test functions $\hat{f}_i$, this model takes on average 6.716 evaluations of $\hat{f}_i$, finds a local maximum in 79.4% of cases and the global maximum in 72.8% of cases. A maximum counts as being found if algorithm 4 returns a maximizer and a maximum which are within an ε-neighbourhood of the real maximizer and maximum. For ε we chose the value $3 \cdot 10^{-2}$ for the neighbourhood of the maximizer on the x-axis and $1 \cdot 10^{-2}$ for the neighbourhood of the maximum on the y-axis. The number of evaluations and the percentages are averages over 500 optimization processes, that is 100 per test function. Figure 3 shows the result of an optimization process of $\hat{f}_4$.

In appendix B we show the numbers of evaluations and percentages of maxima found again averaged over 100 optimization processes for each test function, so 500 optimization processes in total. This way we make up for statistical outliers. Since our test functions are deterministic, we are not using variable noise in the Bayesian optimization process, but a fixed noise of 10^{-6} as mentioned in section 3.2. We are also not using the knowledge gradient acquisition function, because this is a very expensive acquisition function that takes a lot of time to maximize. Doing

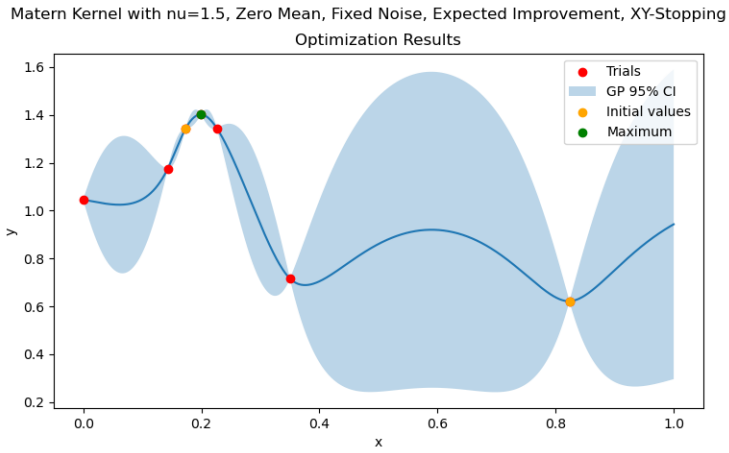

Figure 3. This graph shows the initial values in orange, the found maximum in green, the other evaluation points in red, the mean function of the Gaussian process posterior in dark blue, and the standard deviation of the Gaussian process posterior in light blue. The function that has been optimized is the test function $\hat{f}_4$. As we can see, the optimization process has converged to the maximum at about 0.2, which is also the global maximum.

500 optimization processes on a personal computer with this acquisition function is not possible due to limited resources. Letting it run on the cluster that we have been using for other long and expensive calculations is also not possible because of technical errors.

Probability of improvement.

We first want to inspect the probability of improvement (PI) acquisition function.

If we combine it with different kernel functions, different mean functions and the stopping-xy criterion, we find that algorithm 4 always takes exactly 4 iterations until it stops (see tables 2 and 3 in appendix B). This happens, because the PI acquisition function prefers low risk - low reward situations. The probability that we get a larger value than before is higher close to the current maximum, so every new query point will be close to the current maximum, which triggers the stopping criterion after the fourth evaluation.

If we combine it with different kernel functions, different mean functions and the stopping-y criterion, we find that it takes many more iterations than our basis of comparison model (see tables 4 and 5 in appendix B). That happens, because it again chooses only values close to the current maximum, but the stopping criterion is only triggered after it does not find any greater value. So the optimization process converges to a local maximum, but the steps are very small and we need many iterations (see figure 4).

Therefore, we can discard the PI acquisition function.

The stopping criterion.

We now want to directly compare the two stopping criteria and show that the stopping-xy criterion is strictly better than the stopping-y criterion. The advantage of the stopping-y criterion

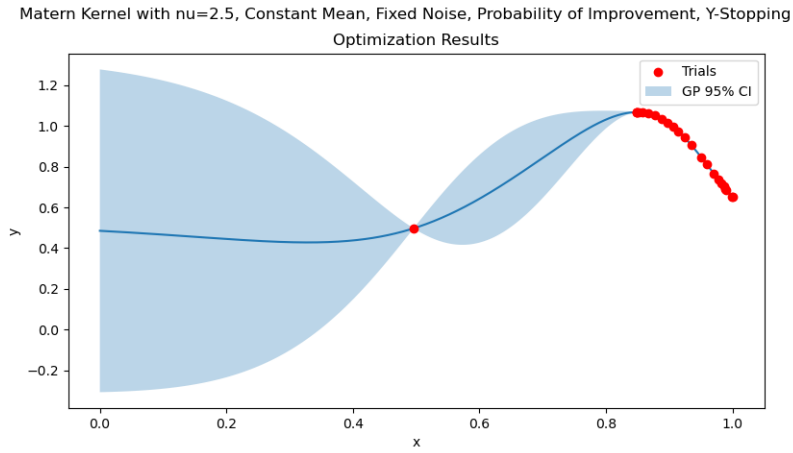

Figure 4. This is the optimization result of a Bayesian optimization with a Matérn kernel with $\nu = 2.5$, a constant mean, fixed noise, the probability of improvement acquisition function and the stopping-y criterion for our test function $\hat{f}_3$. The red dots are the query points chosen by the acquisition function, the blue line the mean function of the Gaussian process posterior and the blue area the standard deviation at the corresponding points. As we can see, this process takes a lot of iterations, 24 to be exact. It does find the global maximum at about 0.85, but there are two large areas we have not explored and that might hide other maxima.

is that it has a very high accuracy at finding local maxima, because it only stops after it has not found a better value in the last three iterations, no matter the acquisition function, which is not very likely when we have not found a local maximum yet. The disadvantage is that it takes a lot of iterations, because once it has found a local maximum, it takes three more iterations until the optimization process stops. When the stopping-xy criterion converges to a maximum, there have usually already been a few evaluations of $\hat{f}_i$ in the neighbourhood of the maximum, which is why it does not take three more iterations for the process to stop, but might stop instantly. The disadvantage of the stopping-xy criterion is that it is not as unlikely that the criterion is triggered when we have not found a maximum yet as with the stopping-y criterion. But in general it is expected that the smaller number of evaluations outweighs the possible lesser accuracy. The pairs of tables in appendix B that show the difference in the stopping criterion are 6 and 7, 8 and 9, and 10 and 11.

The mean functions.

Now we want to compare the two mean functions. Our expectation would be that the constant mean is better than the zero mean, because the Gaussian process has more information to work with with the constant mean. To compare the mean functions, we take a random model, e.g. the Matérn kernel with $\nu = 1.5$, entropy search and the stopping-xy criterion, and combine it with both the zero and the constant mean. Tables 12 and 10 in appendix B confirm our hypothesis, since algorithm 4 is both faster and more accurate with the same model paired with the constant mean instead of the zero mean. Unfortunately, this is not generally true as we can see in the pairs of tables 13 and 14, 1 and 8, and 15 and 16 in appendix B, where the zero mean always

needs a few more iterations, but is a bit more accurate. Therefore, no mean function is strictly better than the other one.

The kernel functions.
The four kernel functions we have given are forms of the Matérn kernel, with $\nu = 0.5$, $\nu = 1.5$, $\nu = 2.5$ and $\nu = \infty$, where $\nu = \infty$ is the same as the RBF kernel. The larger ν, the "smoother" we expect the function we want to optimize to be, that is why ν is called the smoothness parameter. Smoothness in this context has nothing to do with differentiability, but with the expected Lipschitz constant of our function. Of a smooth function we would expect its values not to change too much close to an already evaluated point. We can observe this in figure 5, where we evaluate test function $\hat{f}_1$ at three equidistant points and observe the expected Lipschitz constant close to the evaluated points by noticing how steep the standard deviation of the Gaussian process posterior is at these points. Based on this, it would make sense to discard kernel functions that contradict our belief of the smoothness of the functions we want to optimize, because the approximation of the Gaussian process would not be too accurate. But our goal is to find the maximum, not to approximate the function. Even though a good approximation would lead to the maximum, a good approximation is not necessary. What we do need is an acquisition function that gives us a sensible next query point based on the current approximation of $\hat{f}_i$. The combination of a non-smooth approximation with an acquisition function can lead to a faster optimization process than the same acquisition function with a smoother approximation, even though the true function might be as smooth as the smoother approximation. That can happen e.g., if the acquisition function does not like to take big risks, but it evaluates an area as less risky with a non-smooth approximation than with the smoother one.

Therefore, kernel functions and acquisition functions are not independent of each other and we cannot discard any of them.

We cannot continue to find the best model by discarding options. Luckily, the number of possible options is now very small and we can directly compare all the models we have left. We still have two mean functions, two acquisition functions and four kernel functions that make 16 Bayesian optimization models. To find the best one, we carry all of them out and select the best one. The table that averages the results over 100 optimization processes per test function, so 500 in total, can be found in the appendix B.2. The models in table 1 have an accuracy of finding local maxima of $\geq 85\%$. That table has been extracted from appendix B.2.

Even though the chance to find a local maximum is higher with the model of the last line in table 1, we can discard it because of the high number of evaluations it needs to find this maximum. The accuracy of the other models is not that much smaller that it is worth one extra evaluation of the objective function.
By comparing the first two lines of table 1, we can see that the second line is a bit more accurate than the first one, but it takes about 0.7 iterations more. Since these two models only differ in the mean function, this is the exact effect we saw when we compared the mean functions in this section. With the same argumentation as before, the model with the constant mean, Matérn

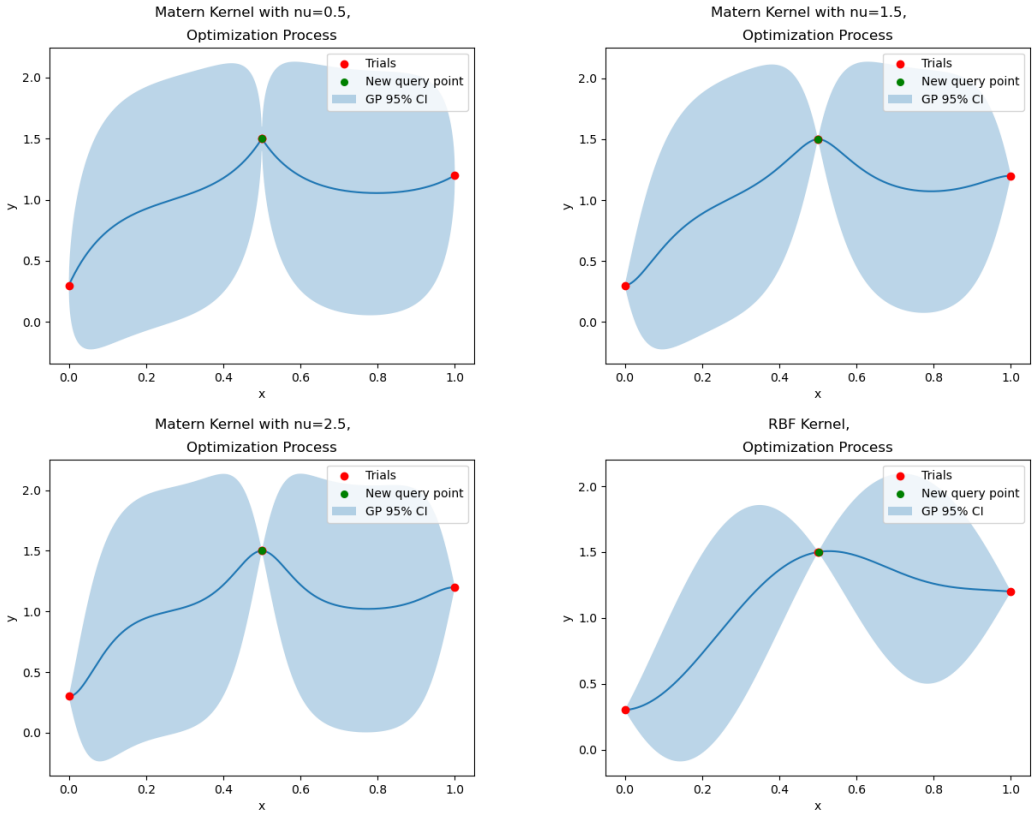

Figure 5. Here we see a Gaussian process approximate the test function $\hat{f}_1$ with the four different kernels and three equidistant query points. The blue line is the mean function and the blue area the corresponding standard deviation of the posterior of the Gaussian process.

Table 1. The best Bayesian optimization models

Mean	Kernel	Acquisition function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
constant	Matérn, $\nu = 0.5$	ES	7.352	85.2%	79.8%
zero	Matérn, $\nu = 0.5$	ES	8.012	85.8%	83.6%
zero	RBF	EI	9.068	88%	82.2%

kernel with $\nu = 0.5$ and the entropy search acquisition function is slightly more preferable than its counterpart with the zero mean function. The reduction in the number of evaluations of this amount outweighs the small increase in accuracy.

With this we have found our optimal Bayesian optimization model. It uses the constant mean function, the Matérn kernel function with $\nu = 0.5$, the entropy search acquisition function and the stopping-xy criterion.

4.3. Testing the optimal optimization model

We now want to test this model we have found. For that we will look at its detailed accuracy for the test functions $\hat{f}_1$-$\hat{f}_5$. The values in this table below are again averages over 100 optimization processes per test function and the definition of a maximum being found is the same as before.

Table 2. Matérn kernel with $\nu = 0.5$, constant mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	8.449	91%	91%
$\hat{f}_2$	7.020	76%	76%
$\hat{f}_3$	7.14	95%	95%
$\hat{f}_4$	6.87	76%	49%
$\hat{f}_5$	7.28	88%	88%
Average	7.352	85.2%	79.8%

These percentages might seem lower than we would like them to be, but some cases that missed the maximum still found a value close to it, just not close enough to be in the ε-neighbourhood of the maximum with the ε-neighbourhood being the same as before. As we can see, the model performs best if there is only one maximum meaning with test functions $\hat{f}_1$, $\hat{f}_3$ and $\hat{f}_5$. It performs worst with $\hat{f}_4$, which is the most complicated function and the function most models struggle with. It takes the most evaluations with $\hat{f}_1$, because the maximum has a very flat surrounding, which spreads the chance of a maximum over a larger area, which then leads to more evaluations being needed. Therefore, our model does not behave unexpectedly and performs very well with all test functions and particularly well with functions with only one maximum. To confirm this, we will test it with some other simple test functions (visualized in figure 6):

- $\hat{f}_6(x) = \sin(5x)^2$
- $\hat{f}_7(x) = x + 0.5x^2 \cdot \sin(18x)$
- $\hat{f}_8(x) = 1 - |x - 0.5|$
- $\hat{f}_9(x) = \sqrt{x} - e^{5(x-1)}$

As a result we get table 3, where we again average over 100 optimization processes for each of the new test functions. As we can see, the accuracy is very similar to the one with the first five test functions, which confirms that our model behaves consistently. It takes even less evaluations of $\hat{f}_i$ with these test functions. We can also see the effect of a flat maximum needing more evaluations to find it with test function $\hat{f}_9$ in comparison to the steeper maxima of the other functions.

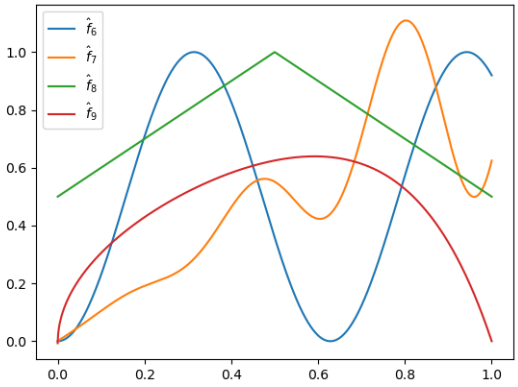

Figure 6. These are the new test functions we are using to test the optimal optimization models with.

Table 3. Matérn kernel with $\nu = 0.5$, constant mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_6$	5.83	80%	80%
$\hat{f}_7$	6.17	91%	80%
$\hat{f}_8$	6.79	81%	81%
$\hat{f}_9$	8.48	89%	89%
Average	6.818	85.25%	82.5%

Now that we know that our model works well with the additional test functions, we also want to know if it can find the maximum of our original problem with the optimal prestretch force for a single muscle. The approximated graph of the objective function can be seen in figure 1, and the result of the optimization process of this function with our optimal model in figure 7. As expected, our model finds the maximum of 30 N, and it does so with seven evaluations, which is consistent with the average number of evaluations of the test functions. This does not only confirm that our model works well, but it also confirms that we have chosen our test functions well, since they seem to represent the kind of function we are being confronted with with these simulations.

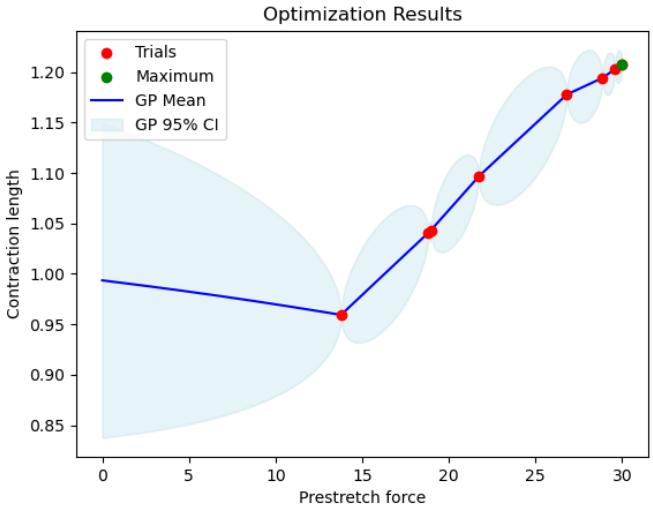

Figure 7. We see the optimization process of our optimal Bayesian optimization process for our original case about the optimal prestretch force for the maximum range of motion.

Chapter 5

Conclusion and outlook

We have now found an optimal Bayesian optimization model with which we can find maxima of blackbox functions as accurately and as fast as possible. It uses a Matérn kernel with $\nu = 0.5$, the constant mean function, the entropy search acquisition function, and the stopping-xy criterion. With this model we have found the maximum to the original problem of the best prestretch force for the maximum range of motion of a single muscle. Because the function that maps the prestretch force to its contraction length is almost linear, the maximum is located at the maximal prestretch force before the muscle tears. We have tested the optimal optimization model with non linear functions as well. Since the case of a single muscle is only a first step to a series of improvements until we can use this for medical procedures like the AMI. The main contribution of this thesis is the code for the Bayesian optimization process with the package BoTorch [**Balandat et al.**]. The code can be found on a GitHub repository [**Bauer**]. The final version of the code is under the tag "Bachelor-thesis". The optimization files are customisable for different kinds of optimization models and can be applied to multidimensional problems as well with only a few changes in the code. It is also compatible with OpenDiHu.

As a next step, this single muscle case can be applied to a more complicated example, e.g. a more realistic scenario like a biceps muscle. With this, we might observe different effects of the muscle's behaviour like non linearity, but we should be able to find the best prestretch again with our recommended optimization model.

After that, we might want to go from a single muscle to a system of two coupled muscles, just as mentioned in the introduction. Two coupled muscles as agonist and antagonist will most definitely show some interesting behaviour, which we will want to analyse and optimize, again with Bayesian optimization.

In the long run, our hope is to use this simulation and optimization in medical procedures that give patients the ability to use their muscles in a natural way after amputations, reduce their phantom pain and increase their quality of life.

Appendix A

Theorems

A.1. Conjugated distributions

Definition A.1.1. ([**Czado et al.**], Definition 2.14) A family of distributions $\{\mathbb{P}_\vartheta : \vartheta \in \Theta\}$ with $\Theta \subset \mathbb{R}^K$ is called a vector exponential family, if the density function can be written as

$$p(x,\vartheta) = \mathbb{1}_{x\in A} \cdot \exp\left(\sum_{i=1}^{K} c_i(\vartheta)T_i(x) \ + d(\vartheta) + S(x)\right)$$

for a set $A \subset \mathbb{R}^n$ and functions $c_i, d : \Theta \to \mathbb{R},\ T_i, S : \mathbb{R}^n \to \mathbb{R},\ i = 1,\dots,K$.

Example A.1.2. ([**Czado et al.**], Example 2.17) The 1-dimensional normal distribution is a 2-dimensional vector exponential family with $\vartheta = (\mu, \sigma^2)^T$, because we can write the density function as

$$p(x,\vartheta) = \exp\left(\frac{\mu}{\sigma^2}\cdot x - \frac{1}{2\sigma^2}\cdot x^2 - \frac{1}{2}\left(\frac{\mu^2}{\sigma^2} + \ln(2\pi\sigma^2)\right)\right)$$

Lemma A.1.3. *(*[**Czado et al.**]*, Lemma 2.16) Existence of conjugated distributions for exponential families:*
Let $x = (x_1,\dots,x_n)$ be an i.i.d. sample of a K-dimensional vector exponential family with density

$$p(x|\vartheta) = \mathbb{1}_{x\in A} \cdot exp\left(\sum_{i=1}^{K} c_i(\vartheta)\cdot\sum_{j=1}^{n} T_i(x_j) \ + \sum_{j=1}^{n} S(x_j) \ + nd(\vartheta)\right).$$

Then the (K+1)-dimensional vector exponential prior distribution

$$\pi(\vartheta; t_1,\dots,t_{k+1}) = a_1 \cdot exp\left(\sum_{i=1}^{K} c_i(\vartheta)t_i + t_{K+1}d(\vartheta)\right)$$

is a conjugated family to $p(x|\vartheta)$ with the posterior distribution

$$p(\vartheta|x) = a_2 \cdot \pi \left(\vartheta; t_1 + \sum_{j=1}^{n} T_1(x_j), \ldots, t_K + \sum_{j=1}^{n} T_K(x_j), t_{K+1} + n \right)$$

and a_1, a_2 being normalizing constants.

A.2. Conditioned normal distributions

Proposition A.2.1. *([**Eaton**], Proposition 3.13)*
If we have two vectors X_1 and X_2 and the vector (X_1, X_2) is distributed normally with

$$(X_1, X_2) \sim \mathcal{N}\left(\begin{bmatrix} \mu_1 \\ \mu_2 \end{bmatrix}, \begin{bmatrix} \Sigma_{11} & \Sigma_{12} \\ \Sigma_{12}^T & \Sigma_{22} \end{bmatrix} \right),$$

then $X_1|(X_2 = x_2)$ is also distributed normally with

$$X_1|(X_2 = x_2) \sim \mathcal{N}(\mu_1 + \Sigma_{12}\Sigma_{22}^{-1}(x_2 - \mu_2), \Sigma_{11} - \Sigma_{12}\Sigma_{22}^{-1}\Sigma_{12}^T).$$

Appendix B

Data created by Bayesian optimization

B.1. Detailed data

The following tables contain the detailed data of Bayesian optimization with algorithm 4 and the model from the respective captions. The data for each test function is averaged over 100 optimization processes.

Table 1. Matérn kernel with $\nu = 1.5$, zero mean, EI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	6.91	87%	87%
$\hat{f}_2$	5.9	56%	56%
$\hat{f}_3$	6.88	89%	89%
$\hat{f}_4$	6.82	81%	48%
$\hat{f}_5$	7.07	84%	84%
Average	6.716	79.4%	72.8%

Table 2. Matérn kernel with $\nu = 2.5$, constant mean, PI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	4	12%	12%
$\hat{f}_2$	4	9%	8%
$\hat{f}_3$	3.99	8%	8%
$\hat{f}_4$	4	22%	5%
$\hat{f}_5$	4	12%	12%
Average	3.998	12.6%	9%

Table 3. RBF kernel, zero mean, PI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	4	8%	8%
$\hat{f}_2$	4.08	15%	13%
$\hat{f}_3$	4	18%	18%
$\hat{f}_4$	3.97	28%	12%
$\hat{f}_5$	4.06	10%	10%
Average	4.022	15.8%	12.2%

Table 4. Matérn kernel with $\nu = 2.5$, constant mean, PI, stopping-y

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	15.55	100%	100%
$\hat{f}_2$	17.78	57%	57%
$\hat{f}_3$	14.13	100%	100%
$\hat{f}_4$	16.01	98%	52%
$\hat{f}_5$	18.06	99%	99%
Average	16.31	90.8%	81.6%

Table 5. RBF kernel, zero mean, PI, stopping-y

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	9.983	98%	98%
$\hat{f}_2$	16.57	63%	63%
$\hat{f}_3$	12.148	100%	100%
$\hat{f}_4$	14.22	100%	51%
$\hat{f}_5$	12.948	98%	98%
Average	13.178	91.8%	82%

Table 6. Matérn kernel with $\nu = 2.5$, zero mean, EI, stopping-y

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	9.35	98%	98%
$\hat{f}_2$	10.71	93%	93%
$\hat{f}_3$	10.08	99%	99%
$\hat{f}_4$	9.12	88%	42%
$\hat{f}_5$	9.7	96%	96%
Average	9.792	94.8%	85.6%

Table 7. Matérn kernel with $\nu = 2.5$, zero mean, EI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	6.33	75%	75%
$\hat{f}_2$	5.88	46%	46%
$\hat{f}_3$	7.41	96%	96%
$\hat{f}_4$	7.29	83%	45%
$\hat{f}_5$	6.81	76%	76%
Average	6.744	75.2%	67.6%

Table 8. Matérn kernel with $\nu = 1.5$, constant mean, EI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	6.84	86%	86%
$\hat{f}_2$	5.6	53%	53%
$\hat{f}_3$	6.22	86%	86%
$\hat{f}_4$	6.12	68%	29%
$\hat{f}_5$	6.6	82%	82%
Average	6.276	75%	67.2%

Table 9. Matérn kernel with $\nu = 1.5$, constant mean, EI, stopping-y

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	9.48	100%	100%
$\hat{f}_2$	11.23	99%	99%
$\hat{f}_3$	10.56	100%	100%
$\hat{f}_4$	9.95	98%	59%
$\hat{f}_5$	10.01	100%	100%
Average	10.246	99.4%	91.6%

Table 10. Matérn kernel with $\nu = 1.5$, constant mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	7.64	89%	89%
$\hat{f}_2$	7.09	69%	69%
$\hat{f}_3$	7.42	98%	98%
$\hat{f}_4$	8.12	72%	53%
$\hat{f}_5$	6.884	81%	81%
Average	7.421	81.8%	78%

Table 11. Matérn kernel with $\nu = 1.5$, constant mean, ES, stopping-y

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	9.19	96%	96%
$\hat{f}_2$	11.09	90%	90%
$\hat{f}_3$	9.8	93%	93%
$\hat{f}_4$	10.1	78%	54%
$\hat{f}_5$	9.79	94%	94%
Average	9.994	90.2%	85.4%

Table 12. Matérn kernel with $\nu = 1.5$, zero mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	8.23	94%	94%
$\hat{f}_2$	6.97	60%	57%
$\hat{f}_3$	7.74	98%	98%
$\hat{f}_4$	8.01	73%	52%
$\hat{f}_5$	7.54	86%	86%
Average	7.698	82.2%	77.4%

Table 13. Matérn kernel with $\nu = 0.5$, zero mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	9.21	92%	92%
$\hat{f}_2$	7.87	83%	83%
$\hat{f}_3$	7.62	90%	90%
$\hat{f}_4$	7.48	71%	60%
$\hat{f}_5$	7.88	93%	93%
Average	8.012	85.8%	83.6%

Table 14. Matérn kernel with $\nu = 0.5$, constant mean, ES, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	8.45	91%	91%
$\hat{f}_2$	7.02	76%	76%
$\hat{f}_3$	7.14	95%	95%
$\hat{f}_4$	6.87	76%	49%
$\hat{f}_5$	7.28	88%	88%
Average	7.352	85.2%	79.8%

Table 15. RBF, constant mean, EI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	6.4	91%	91%
$\hat{f}_2$	6.49	67%	66%
$\hat{f}_3$	7.4	99%	99%
$\hat{f}_4$	8.1	59%	27%
$\hat{f}_5$	6.38	88%	88%
Average	6.954	80.8%	74.2%

Table 16. RBF, zero mean, EI, stopping-xy

Test function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
$\hat{f}_1$	6.84	92%	92%
$\hat{f}_2$	7.58	72%	72%
$\hat{f}_3$	7.97	96%	96%
$\hat{f}_4$	15.78	85%	56%
$\hat{f}_5$	7.17	95%	95%
Average	9.068	88%	82.2%

B.2. Collection of data

In the following table we find the data to all the Bayesian optimization models, that we have not discarded yet. All of the models are using the stopping-xy criterion and assume non noisy evaluations. The data is averaged over 500 optimization processes, 100 per test function from chapter 4.

Table 17. Data of remaining Bayesian optimization models

Mean	Kernel	Acquisition function	Number of evaluations	Percentage of local maxima found	Percentage of global maxima found
constant	Matérn, $\nu = 0.5$	EI	5.674	47.2%	41.4%
constant	Matérn, $\nu = 0.5$	ES	7.352	85.2%	79.8%
constant	Matérn, $\nu = 1.5$	EI	6.276	75%	67.2%
constant	Matérn, $\nu = 1.5$	ES	7.431	81.8%	78%
constant	Matérn, $\nu = 2.5$	EI	6.476	78%	71.6%
constant	Matérn, $\nu = 2.5$	ES	7.429	82.6%	77.8%
constant	RBF	EI	6.954	80.8%	74.2%
constant	RBF	ES	7.846	81.8%	77.2%
zero	Matérn, $\nu = 0.5$	EI	6.218	58.2%	52.8%
zero	Matérn, $\nu = 0.5$	ES	8.012	85.8%	83.6%
zero	Matérn, $\nu = 1.5$	EI	6.716	79.4%	72.8%
zero	Matérn, $\nu = 1.5$	ES	7.698	82.2%	77.4%
zero	Matérn, $\nu = 2.5$	EI	6.744	75.2%	67.6%
zero	Matérn, $\nu = 2.5$	ES	7.469	76.6%	73.8%
zero	RBF	EI	9.068	88%	82.2%
zero	RBF	ES	8.140	81%	76%

Bibliography

[Asmussen et al.] Søren Asmussen, Peter W. Glynn, *Stochastic Simulation: Algorithms and Analysis*, Springer New York, NY, published 14.07.2007, retrieved 04.09.2024

[Balandat et al.] Maximilian Balandat, Brian Karrer, Daniel R. Jiang, Samuel Daulton, Benjamin Letham, Andrew Gordon Wilson, Eytan Bakshy, *BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization*, arXiv:1910.06403, published 08.12.2020, retrieved 17.08.2024

[Bassett et al.] Bassett, R., Deride, J., *Maximum a posteriori estimators as a limit of Bayes estimators*, Math. Program. 174, 129–144 (2019), https://doi.org/10.1007/s10107-018-1241-0, published 30.01.2018, retrieved 01.08.2024

[Bauer] Lukas Bauer, *BA*, https://github.com/Lukuas31415/BA, retrieved 08.09.2024

[Bradley et al.] Chris P. Bradley, Nehzat Emamy, Thomas Ertl, Dominik Göddeke, Andreas Hessenthaler, Thomas Klotz, Aaron Krämer, Michael Krone, Benjamin Maier, Miriam Mehl, Tobias Rau, Oliver Röhrle , *Enabling Detailed, Biophysics-Based Skeletal Muscle Models on HPC Systems*, Front. Physiol., published 12.07.2018 retrieved 19.09.2024

[Bull] Adam D. Bull, *Convergence rates of efficient global optimization algorithms*, arXiv:1101.3501, published 22.10.2011, retrieved 12.08.2024

[Czado et al.] Claudia Czado, Thorsten Schmidt, *Mathematische Statistik*, Springer Berlin, Heidelberg, published 27.05.2001, retrieved 14.06.2024

[Eaton] Morris L. Eaton, *Multivariate Statistics: A Vector Space Approach*, Project Euclid, published 28.11.2007, retrieved 30.05.2024

[Frazier] Peter Frazier, *A Tutorial on Bayesian Optimization*, arXiv:1807.02811, published 08.07.2018, retrieved 07.05.2024

[Görtler et al.] Jochen Görtler, Rebecca Kehlbeck, Oliver Deussen, *A Visual Exploration of Gaussian Processes*, DOI: 10.23915/distill.00017, published 02.04.2019, retrieved 30.06.2024

[Hennig et al.] Philipp Hennig, Christian J. Schuler, *Entropy Search for Information-Efficient Global Optimization*, Journal of Machine Learning Research 13 (2012) 1809-1837, published 01.06.2012, retrieved 15.08.2024

[Homs-Pons et al.] Carme Homs-Pons, Robin Lautenschlager, Laura Schmid, Jennifer Ernst, Dominik Göddeke, Oliver Röhrle, Miriam Schulte, *Coupled Simulations and Parameter Inversion for Neural System and Electrophysiological Muscle Models*, Wiley Online Library, published 31.03.2024, retrieved 04.04.2024

[Maier] Benjamin Maier, *Scalable Biophysical Simulations of the Neuromuscular System*, arXiv:2107.07104, published 13.07.2021, retrieved 21.08.2024

[Rasmussen et al.] Rasmussen, C. and Williams, C., *Gaussian Processes for Machine Learning*, MIT Press, Cambridge, MA, published 2006, retrieved 30.05.2024

[Shahriari et al.] Bobak Shahriari, Kevin Swersky, Ziyu Wang, Ryan P. Adams, Nando de Freitas, *Taking the Human Out of the Loop: A Review of Bayesian Optimization*, IEEE, published 10.12.2015, retrieved 22.04.2024