Gaussian Processes for Automatic Controller Gains Tuning in Robotics and Control

Schmitz, Maximilian , Lu, Yuwei , Gray, Justin , Oh, Jaeyo , Kanwar, Bharat

***Abstract*—Tuning controller parameters is one of the most fundamental problems when designing controllers for dynamic systems and is usually done manually. To automate this, methods from machine learning, such as Bayesian Optimization, have been used. The downside is that this may lead to safety-critical failures when evaluating different controller parameters on a real system. This problem is solved in this paper by using an optimization algorithm, SafeOpt, which, given an initial, low-performance controller, automatically optimizes the parameters of a control law to guarantee safety. The controller parameters are evaluated by a performance function modeled by a Gaussian process. Only parameters with a performance higher than a certain threshold with high probability are considered as safe.**

I. Introduction

Tuning a controller is a time consuming and challenging task. It requires significant domain knowledge or experience. In this paper we provide and evaluate a method for automatic controller tuning given by [2]. This method automatically tunes controller parameters without requiring a model of the underlying, dynamic system while explicitly avoiding the evaluation of unsafe controller parameters. The controller parameters are therefore evaluated by a performance function which is initially unknown where we use Gaussian processes (GPs) to approximate this performance function. In the following, first, we outline Gaussian processes and present a mathematical intuition. Then, we explain in more detail how to use GPs for automatic controller tuning and tune one- and two-dimensional examples. Finally we add sensor noise and disturbances and examine possible downsides of this approach.

The authors are members of the class ECE6254 - Statistical Machine Learning at the Georgia Institute of Technology, United States. Email: {ylu645,jgray73,oj, mschmitz7,bkanwar3}@gatech.edu

II. Gaussian Processes

Gaussian processes are a powerful tool in machine learning which allows for model prediction by combining a prior model and data with real-time input data [1]. GPs are random processes which make inference and learning relatively easy (cf. [4]).

For a general regression problem with a given training data set, there are infinitely many functions to fit the data (hypothesis class $|\mathcal{H}| = \infty$). The question is then which function fits the data the best. Gaussian processes address this by assigning a probability to each of these functions $h \in \mathcal{H}$.

A. From Gaussian random variables to Gaussian processes

A Gaussian random variable (GRV) is a random variable which is based on a normal (Gaussian) distribution. A GRV x is defined by $x \in \mathbb{R}^n, x \sim \mathcal{N}(\mu_n, \Sigma)$ with mean μ_n and covariance matrix Σ. Therefore, a Gaussian process is a generalization of a GRV and can be though of as an extension to it.

The main difference is that a GRV describes a probability distribution for vectors or scalars while stochastic processes as GPs define properties of functions.[3][6]

Definition 1. Let $m : \mathbb{X} \to \mathbb{R}$ be any function and let $k : \mathbb{X} \times \mathbb{X} \to \mathbb{R}$ be a valid covariance function[1]. A *Gaussian process* $f \sim \mathcal{GP}(m, k)$ is a probability distribution over the function space $f : \mathbb{X} \to \mathbb{R}$ such that every restriction to finitely many function values f_X is a GRV, $f_X \sim \mathcal{N}(m_X, \mathbf{K}_n)$. A GP is completely defined by its mean function $m(X)$ and its covariance function $k(x, x')$ with

$$\begin{aligned} X &:= [x_1, x_2, \dots, x_n] \in \mathbb{R}^{D\times n} \\ f(X) &:= [f(x_1), f(x_2), \dots, f(x_n)]^T \in \mathbb{R}^n \\ m(X) &:= \mathbb{E}[f(X)] \in \mathbb{R}^n \\ \mathbf{K}_n &\in \mathbb{R}^{n\times n}, [\mathbf{K}_n]_{ij} = k(x_i, x_j) \end{aligned}$$

[1]A function $k(\cdot,\cdot)$ is a valid covariance function if the corresponding matrix $k(\mathbf{X},\mathbf{X})$ is positive semidefinite ($\mathbf{K}_n = k(\mathbf{X},\mathbf{X}) \geq 0$) for all $\mathbb{X}$.

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

$\mathbf{K}_n$ is defined by the covariance function $k(x, x')$ which is evaluated pairwise between all training points. The kernel receives two points $x, x' \in \mathbb{R}$ as an input and outputs a scalar which is a similarity measure between those two points. Loosely speaking, the entry $[\mathbf{K}_n]_{ij}$ describes the influence of point i to point j. Since the kernel describes the similarity, the choice of the kernel defines the shape of the fitted function.

We can think about a GP as a very long vector where each entry is a GRV, so that a GP is a collection of random variables, any finite number of which have a joint Gaussian distribution.[1]

B. Academic Example of GPs

Consider a simple 1-d regression problem $f : \mathbb{R} \rightarrow \mathbb{R}, x \mapsto f(x)$. For Gaussian process regression, the first step is to set up a prior distribution over functions. This represents the knowledge about functions we expect to observe before seeing any data and is shown in figure 1 on the left. Without loss of generality, the mean can be set to zero which is a common assumption in machine learning [4]. Besides the mean, we can evaluate the variance at each sample point x_i, $i = 1, \ldots, n$ too.

Suppose further that we have been given a data set $\mathcal{D} = \{(x_1, y_1)\}$ consisting of one single data point. Then, we only want to consider functions from our hypothesis set which are passing through this point (or are sufficient close in case we assume additional noise). We can see that the variance is (almost) zero at the sample point since we assume that the ground truth function passes through our sample point (figure 1 (middle)). This combination of data and prior distribution is then called posterior distribution. The more samples are added, the better the mean fits the true function as seen in figure 1 (right).

For the covariance function, a squared exponential kernel (SE) is used. The SE kernel is the most widely-used kernel within the kernel machines field according to [1], but another kernel like the Matérn kernel which is used in [2] work too. The squared exponential kernel is given by $k_{SE}(x, x') = \sigma_{SE}^2 \exp\left(-\frac{(x-x')^2}{2l^2}\right)$ and includes the two hyperparameters variance σ_{SE} and the length scale l. Since the main objective of learning with Gaussian processes is optimizing the hyperparameters for the specific problem, we give a quick overview about what a change in the hyperparameters means.

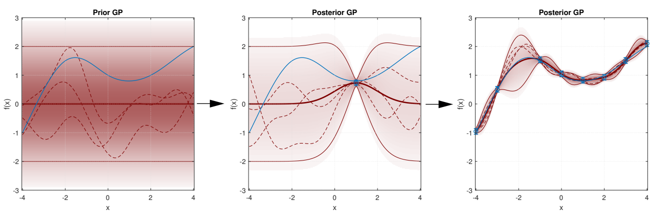

Fig. 1: 1d-example of a Gaussian process. The blue line indicates the unknown ground-truth function and the dashed red lines are sample functions. Prior GP (left), Posterior GP with 1 sample point (middle), final Posterior GP after 8 measurements (right)

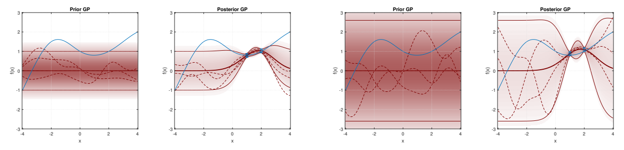

Fig. 2: Variance comparison. $\sigma_{SE} = 0.5$ (left) and $\sigma_{SE} = 1.3$ (right)

a) Variance σ_{SE}: Loosely speaking, the variance identifies how far a data point is allowed to deviate from the data points next to it (how many points are considered is specified by the length scale below).In figure 2, we can see the uncertainty for the kernel with a smaller σ_{SE} on the left is smaller compared to a kernel with larger σ_{SE} on the right.

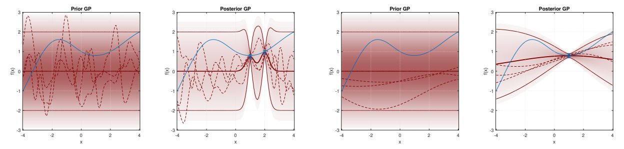

Fig. 3: Length scale comparison. $l = 0.3$ (left), $l = 4$ (right)

b) Length scale l: The length scale configures how much the considered functions may vary. Functions are allowed to vary rapidly if the characteristic length-scale is chosen short while increasing the length scale then increases the banding, as points further away from each other become more correlated which is shown in figure 3.

C. Updating procedure for GPs

For the next sections we continue considering a discrete view of the distribution. Instead of finding an implicit function we want to find function values at distinct test points x_i. With Bayesian inference we get to the posterior by combining the prior distribution with the measured data $\mathcal{D}_n = (\mathbf{X}, \mathbf{Y}) = \{(x_1, y_1), \ldots, (x_n, y_n)\}$ to

$$\mu_n(x^*) = \mathbf{k}_n(x^*)\left(\mathbf{K}_n + \sigma_\omega^2 \mathbf{I}\right)^{-1}\mathbf{Y} \tag{1}$$

$$\sigma_n^2(x^*) = k(x^*, x^*) - \mathbf{k}_n(x^*)\left(\mathbf{K}_n + \sigma_\omega^2 \mathbf{I}\right)^{-1}\mathbf{k}_n^T(x^*) \tag{2}$$

where x^* is the new input point, the covariance matrix $\mathbf{K}_n = k(\mathbf{X}, \mathbf{X}) \in \mathbb{R}^{n \times n}$ has entries $[\mathbf{K}_n]_{(i,j)} = k(x_i, x_j)$, $i, j \in \{1, \ldots, n\}$, the vector $\mathbf{k}_n(x^*) = [k(x^*, x_1), \ldots, k(x^*, x_n)]$ contains the variances between the new input point x^* and the observed data points $\mathcal{D}_n$. The identity matrix is denoted by $\mathbf{I} \in \mathbb{R}^{n \times n}$. A further extension in here is that it is assumed that the training vector observed $\mathbf{Y} \in \mathbb{R}^n$ is corrupted by Gaussian noise with mean zero and variance σ_ω, i.e. $\mathbf{Y} = f(\mathbf{X}) + \epsilon$.

III. Automatic controller tuning with Gaussian processes

This section introduces how to tune a controller automatically with incorporating Gaussian processes. First, a nonlinear, dynamic control law is given by $\mathbf{u}_k = \mathbf{g}(\mathbf{y}_k, \mathbf{r}_k, \mathbf{a}_n)$ which is parameterized by $\mathbf{a}_n \in \mathcal{A}$ at iteration n, where $\mathbf{y}_k$ are the (noisy) measurements, $\mathbf{r}_k$ is a reference signal, and $\mathbf{u}_k$ is control actions at time step k. At each iteration, the used controller parameters are evaluated by a performance measure, $J(a) : \mathcal{A} \mapsto \mathbb{R}$. The goal is to tune the parameters to maximize the performance measure for a safety-critical system. According to [2], there is no prior knowledge on the model of the dynamic system and on the performance measure, except an assumption that an initial set of the parameters guarantees stability and safety. The safety criterion is set as a performance threshold, J_{min}, so that $J(\mathbf{a}_n) \geq J_{min}$ must be satisfied with high probability. The intuition is that unstable controller gains will lead to a low performance and are then and are likely to be excluded through the threshold then. This guarantees safety and gives the algorithm its name *SafeOpt* [2].

Since the performance function is initially unknown, it is modeled by a GP to approximate $J(\mathbf{a})$. The GP framework allows to predict $J(\mathbf{a}^*)$, $\mathbf{a}^* \in \mathcal{A}$, based on n past observations, $\mathcal{D}_n = \{\mathbf{a}_i, \hat{J}(\mathbf{a}_i)\}_{i=1}^n$ where $\hat{J}(\mathbf{a}) = J(\mathbf{a}) + \omega$ with $\omega \sim \mathcal{N}(0, \sigma_\omega^2)$. From the observations, the mean and variance of the prediction are given as equation (1) and (2) with $\hat{J}_n$ and $\mathbf{a}^*$ instead of $\mathbf{Y}$ and $\mathbf{x}^*$, respectively.

To obtain the next sample locations, Bayesian optimization is then used. In this paper, GP-based Bayesian optimization methods choose new parameter sets for which the performance is the most uncertain, mostly within a safe set. Recalling the safety constraint from above, the safe set is defined by $\mathcal{S} = \{\mathbf{a} \in \mathcal{A} | J(\mathbf{a}) \geq J_{min}\}$. But, SafeOpt considers not only the safe set which includes the set of potential maximizers, $\mathcal{M}_n$, for exploitation, it also includes the set of potential expanders, $\mathcal{G}_n$, for exploration.[2]

In summary, the algorithm starts with an initial safe parameters and unknown performance measure. Then, a new evaluation point is computed via Bayesian Optimization, and the measurement is obtained. Finally, GP based performance function is updated with the new point and the measurement. It is iterated until either the process is aborted or a desired criterion is reached.

IV. Setup of Simulation environment (Quadrotor)

The quadrotor simulation environment used is strongly based on work of [11] which is available online. It provides a nonlinear 6 DOF model of a quadcopter with freedom about three translational and three rotational directions. Since SafeOpt aims to tune a controller without a distinct model of the system, a detailed description of the model is omitted.

For the controller, a basic PID-controller is chosen for each of the six dimensions. We define the linear control law for controller i, $i = 1, \ldots, 6$ at time k by

$$u_{i,k} = k_{i,p}(x_{i,k} - r_{i,k}) + k_{i,i}\int(x_{i,k} - r_{i,k})d\tau + k_{i,d}\dot{x}_{i,k}$$

with the controller gains $k_{i,p}$, $k_{i,i}$, and $k_{i,d}$ for the proportional, integrative and derivative part, respectively. In the following, the integrative part will be set to zero most of the time.

V. Quadrotor controller turning

With the theoretical foundations from the previous sections, the SafeOpt-algorithm can be applied to the given quadrotor controller. This is described in the following. While [2] uses a linear quadratic state and input cost and defines the performance as the cost improvement relative to the initial state cost, we use a performance metric based on the response of our system to a step input. This is feasible since the input trajectory used in this example is a step input only. The performance function is given by

$$P = \frac{100}{\mathbf{w} \cdot \mathbf{r}} = \frac{100}{0.1rt + 2os + 2ss + oc} \tag{3}$$

with the performance weights $\mathbf{w} = \begin{bmatrix} 0.1 & 2 & 2 & 1 \end{bmatrix}$ which have been manually tuned and the system response metric $\mathbf{r} = \begin{bmatrix} rt & os & ss & oc \end{bmatrix}^T$. A detailed explanation about how to compute the system response metric can be found in [8]. A summary of the entire algorithm we used is found in algorithm 1.

In the following, we applied this algorithm to the quadrotor simulation environment, tuned controller gains first just for one and then for multiple dimensions. Even tuning the controller for one dimensions leads to two controller gains to be estimated (k_p and k_d). As a last step we added noise to our model to include some more uncertainty and estimate the robustness of the algorithm by adding disturbance too.

Algorithm 1: Apply SafeOpt-algorithm to quadrotor controller

```
Result: Obtain improved controller gains
Input  : $k_{max}$ iterations for optimization;
         initial gains $z_0$;
create linearly space parameter set;
initialize kernel with GPy;
set initial gains to $z_0$;
simulate one time step and obtain $x_0$;
compute $rt$, $os$, $ss$, $oc$;
obtain performance $P$ to $z_0$;
create GP with $P$ and $z_0$;
add GP to optimization routine;
for $k < k_{max}$ do
    get next controller gain $z_{k+1}$ through
      Baysian Optimization;
    simulate one timestep and obtain $x_{k+1}$;
    compute $rt$, $os$, $ss$, $oc$;
    calculate $P$;
    add $z_{k+1}$ to optimization routine;
end
```

A. *Results of one dimensional tuning*

For the one-dimensional example, the quadrotor learns optimal controller gains for the position controller in x-direction. The other two directions are stabilized by separate controllers.

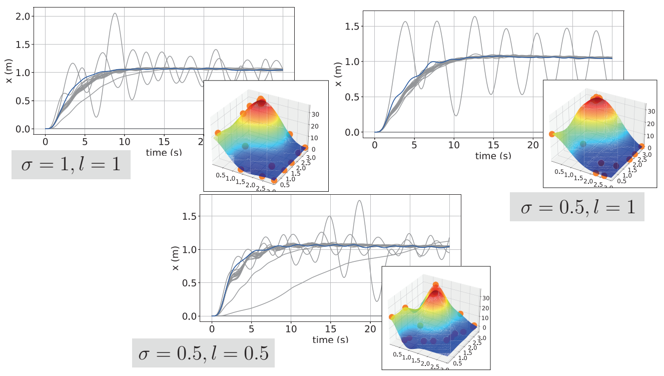

Fig. 4: Trajectories and performance function for 1d-example with different variance and length scales.

Each controller conducts 30 iterations to obtain the optimal controller gains. This is done for different hyperparameters (variance and length scale) of the kernel.

In figure 4 on the upper left and on the upper right the length scale is $l = 1$ and the variance is $\sigma = 1$ and $\sigma = 0.5$, respectively. Here, reducing the variance causes the cost function to hone in on the optimal gains more quickly. However, when reducing the length scale and keeping the variance constant, it allows for more peaks and valleys in the cost function, making the algorithm test more points that would otherwise lead to more oscillatory behavior. This is shown in figure 4 in the upper right with $l = 1$ and in the second row with length scale $l = 0.5$. The blue trajectory is the optimal one with the highest performance.

B. *Results from multi dimensional tuning*

For multi dimensional tuning, in case of 2-d tuning the x-, y-direction or x-, z-direction and in case of 3-d tuning the x-, y-, and z-direction are controlled each by a separate controller. For simplicity, we assume that each axis is independent. To optimize the parameters for this multi dimensional example, the performance measures are combined by averaging them. Figure 5 shows that the controller is optimized to the best trajectory for each direction among variations. The result for x-, z-direction is presented in the video.

Figure 6 shows the result for 3-dimensional tuning. To control 3 directions, 6 parameters are used. Initially each parameter set consists of a set of 100 values. However, this huge parameter space qickly causes

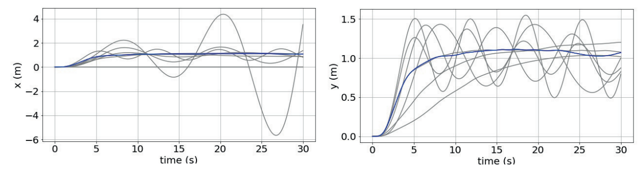

Fig. 5: Trajectories of quadrotor for 2-d(x, y) control.

memory error for multiple dimensions.This is, tuning was conducted with the reduced number of samples for each parameter resulting in a more rough resolution of the parameter space. Despite of the reduction, as shown in the figure 6, the algorithm works for this example too. Comparing to the 1-dimensional example, there are more unstable approximation during learning. However, these results presents the expandability of the algorithm to a n-dimensional case.

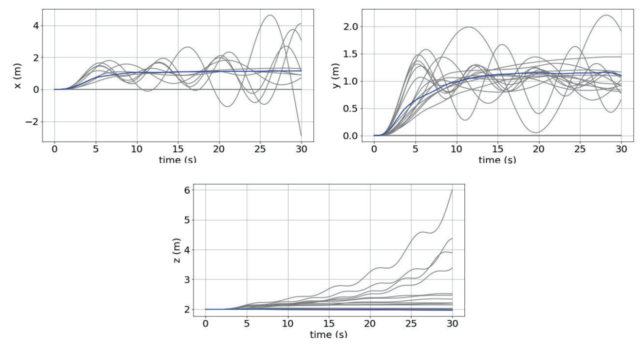

Fig. 6: Trajectory of quadrotor for 3-d(x, y, z) control with reduced parameter set.

C. *Adding noise to model*

One major issue when tuning a controller is noise in the measurements. For applying noise, we used the 1d-tuning example with two controller gains to be tuned. We added Gaussian noise with zero mean and standard deviation σ_{noise} to the x-position in front of the controller input. Recall that usually $\sigma_{noise} \neq \sigma_\omega$ since we have a distinction in measurement and process noise. We tried different noise and selected the biggest value for σ_{noise} which still leads to stable results. This is shown in figure 7.

Comparing to the tuning without noise, the algorithm needs more attempts to find optimal or even stable controller gains. Most of all, the algorithm explores a lot initially which leads more often to unstable configurations. This may be improved by choosing better boundaries of the performance function.

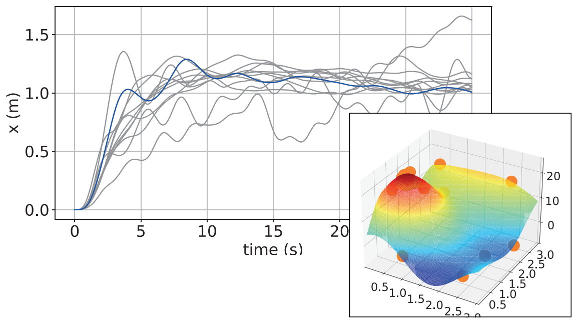

Fig. 7: Trajectories and performance function for variance $\sigma = 1$, length scale $l = 1$ (noisy setup).

At this point it need to be pointed out that the underlying controller was not designed with noise in mind. Since a PID-controller was used, reducing the derivative part may lead to better results. An investigation of this is not part of this paper.

D. *Adding disturbance to model*

Gaussian noise disturbance rejection is also something worth looking into for a quadrotor system to counter the disturbances present from wind and other environmental factors. To simulate this, 0-mean Gaussian noise was added to the position and heading angle of the quadrotor. The disturbed system was tested on a 1-d step response. The standard deviation of the added noise was increased in increments and the SafeOpt algorithm was run over 30 trials to optimize to the best performance at that disturbance level.

σ in m	**Performance**
0.000	182.5
0.003	73.3
0.005	57.1 (high oscillations)
0.01	5.35 (crashed)

TABLE I: Performance Metric for Disturbance of the form: $\mathcal{N}(0, \sigma^2)$

After adding the disturbance, the performance function also had to be slightly modified. The previous version of this function over-emphasized the negative effect of oscillations. This penalty turned out to be too strict since the addition of disturbance resulted in more oscillations. The peak response value was also redefined as the maximum deviation from the

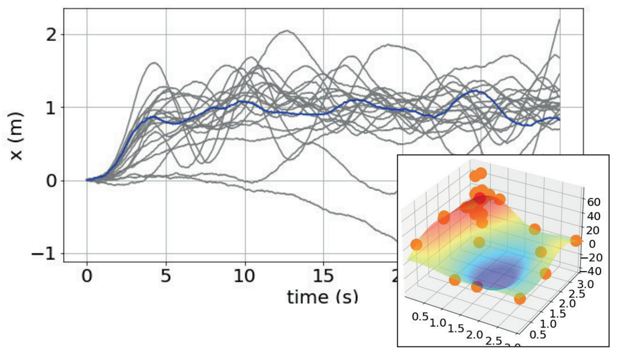

Fig. 8: Quadrotor Trajectory (optimal in blue) and Value function for 1d step response with $\sigma = 3$ mm as the disturbance

desired state to limit that quantity's overall effect on the performance. Overall, these changes rewarded low steady-state error without over-penalizing for oscillations and slower responses.

Table I outlines the performance metric for the optimal parameter set as a function of the disturbance added (changing choice of σ in the 0-mean Gaussian model). As expected, increasing disturbance decreases the overall performance of the system with $\sigma = 5$mm resulting in high oscillations and the $\sigma = 10$ mm disturbance resulting in a crash. Figure 8 shows the response for the 3 mm disturbance response.

VI. Conclusion

In this paper, we gave an introduction into Gaussian processes in machine learning and showed an application to robotics and control by considering a simulated quadrotor. The main idea was to automatically tune the controller gains of the quadroter with Gaussian processes without risking dangerous system failures nd without prior knowledge. Therefore, we utilized the SafeOpt Algorithm by [2], applied it to an one dimensional example, then expanded it to more dimensions and added noise to the model. In general the SafeOpt-algorithm turned out to be not as stable and robust as advertised. Above all, the major weaknesses besides specifying an initial feasible point is that reasonable boundaries for the performance function need to be specified a priori. This conflicts the major idea of this algorithm a bit, which is allowing to tune a controller with no or almost no prior knowledge. To conclude it can be said that Gaussian processes (as a part of machine learning) are a powerful tool, but should not solely replace traditional tuning techniques as its fullest.

VII. Outlook

An extension to tuning a PID controller as done in this paper would be considering more advanced controllers, like an LQR or MPC approach, and tune these. Especially for LQR and MPC controllers, an automatic tuning of the weight matrices without providing safety is already provided by [7] and [9].

Another major improvement is choosing a different performance measure which is not purely based on step input characteristics. Especially when focusing on very fast systems with small steps, a more continuous approach might be feasible.

References

[1] Carl Edward Rasmussen and Christopher K. I. Williams, 2005. *Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning)*. The MIT Press.

[2] Berkenkamp, Felix and Schoellig, Angela P. and Krause, Andreas, May 2016. *Safe controller optimization for quadrotors with Gaussian processes*, 9781467380263, http://dx.doi.org/10.1109/ICRA.2016.7487170, 2016 IEEE International Conference on Robotics and Automation (ICRA), IEE

[3] Sebastian Trimpe, November 2019 *Lecture Notes on Statistical Learning and Stochastic Control* Max Plank Institute for Intelligent Systems, University of Stuttgart, Germany. (unpublished)

[4] Görtler, et al., 2019 *A Visual Exploration of Gaussian Processes*, Distill.

[5] Shalec-Shwartz, S., Ben-David, S., May 2014 *Understanding Machine Learning: From Theory to Algorithms*. Cambridge University Press.

[6] Bishop, Christopher M., 2006. *Pattern Recognition and Machine Learning. (Information Science and Statistics)*. Springer-Verlag, Berlin, Heidelberg.

[7] A. Marco, P. Hennig, J. Bohg, S. Schaal and S. Trimpe, 2016. *Automatic LQR tuning based on Gaussian process global optimization. 2016 IEEE International Conference on Robotics and Automation (ICRA)*. Stockholm, 2016, pp. 270-277, doi: 10.1109/ICRA.2016.7487144.

[8] The MathWorks, Inc, 1994-2020. *stepinfo* Retrieved from https://www.mathworks.com/help/control/ref/stepinfo.html (11/18/2020)

[9] J. Kabzan, L. Hewing, A. Liniger and M. N. Zeilinger, *Learning-Based Model Predictive Control for Autonomous Racing* IEEE Robotics and Automation Letters, vol. 4, no. 4, pp. 3363-3370, Oct. 2019, doi: 10.1109/LRA.2019.2926677.

[10] N. Srinivas, A. Krause, S. M. Kakade, and M. Seeger, 2010. *Gaussian process optimization in the bandit setting: no regret and experimental design. Proc. of the International Conference on Machine Learning(ICML)*. pp. 1015-1022.

[11] Charles Tytler, *QuadcopterSim* https://github.com/charlestytler/QuadcopterSim