DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Mixed discrete-continuous Bayesian Optimization for AutoTuning

Jan Nguyen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Mixed discrete-continuous Bayesian Optimization for AutoTuning

Gemischte diskret-kontinuierliche Bayes'sche Optimierung für AutoTuning

Author:	Jan Nguyen
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Gratl, M.Sc.
Submission Date:	15.10.2020

I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15.10.2020 Jan Nguyen

Abstract

Molecular dynamics simulations are used to analyze physical motion at the molecular level. A numerical approach is often used, where we assume that for small time intervals, the acting force remains constant. So we calculate the trajectory of each particle step by step by repeatedly using this assumption. In each of these steps, we have to calculate the pairwise forces between all particles. Due to this, simulations with a high number of particles could take a considerable amount of time, since every particle exert forces on every other particle. However, there are many different algorithms and corresponding parameters to calculate these forces efficiently. Which combination of parameters is most effective depends on the structure of the simulation and is generally hard to predict. To avoid having to test every combination, we use Bayesian optimization, which should give us a good result with only a few tests. Since we have discrete and continuous parameters typical Bayesian optimization can only be used to a limited extent. This is why we are testing an approach where we consider discrete values using a cluster model. As a result, we were able to observe significant improvements compared to other methods. We were often able to make an optimal selection automatically. A person would need sufficient expert knowledge to make a similarly efficient choice.

Contents

1 Introduction

Bayesian optimization (BO) is used to optimize arbitrary black-box functions. The only way to get information about these functions is to evaluate them at certain points. In Bayesian optimization, special emphasis is placed on keeping the number of evaluations low but still finding a value close to the optimum. For this purpose, after each evaluation, the information collected so far is used to decide which point is most suitable to be sampled next. The idea here is to estimate the output of the function using a probabilistic model. A prior probability is chosen in a very relaxed and general way. The posterior thus has enough flexibility to adapt to the collected data. The algorithm incrementally tries to derive the inner structure of the black-box function. With expert knowledge, this could be speeded up, but the algorithm often achieves a good result even without aid. Therefore this method can be used when almost no information about the black-box function is known. BO therefore has many different fields of application [Ngu19] [Liz+07] [Ulm+15]. We use BO for auto-tuning algorithm parameters. The algorithms we consider have a fixed outcome, independent from the chosen parameters. But the selection has other side effects, like affecting the algorithm's overall time needed. This is a property that we can consider as a black-box function. Each configuration leads to a different total runtime. But we can find out this value by running the algorithm and measuring the time required. Using BO we can optimize this mapping from configuration to time. However, we need several test runs to reliably find an optimal configuration. So this method is not suitable if all needed calculations are already completed after one measurement. It must therefore be possible to perform and measure only a part of the calculations. Naturally, the measured times still have to be somewhat representative of the remaining computations.

2 Problem Statement

2.1 Molecular Dynamics

In this thesis, we mainly focus on molecular dynamics simulations. But the underlying idea of tuning can also be applied to other areas. In molecular dynamics, we analyze the interaction of small particles like atoms. In a simulation of p particles, each of these particles exerts different types of forces on all the other particles. Many simplifications of the model are based on the Lennard-Jones 12-6 potential [All04]. Such models summarize the different forces using the following formula.

$$U_{LJ}(r) = 4\epsilon \left(\left(\frac{\sigma}{r} \right)^{12} - \left(\frac{\sigma}{r} \right)^{6} \right) \tag{2.1}$$

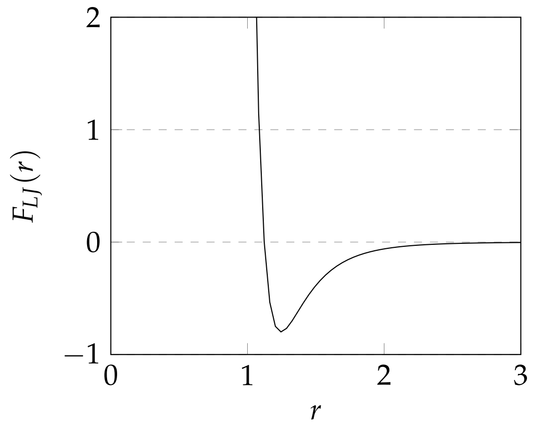


Figure 2.1: Example of Lennard Jones Potential

ϵ and σ are constants which are chosen depending on the type of particles observed. r is the Euclidean distance between the two particles. The direction and magnitude of the force exerted by one particle on another are therefore only dependent on their relative position. So if we are given the position of all particles at a given time t we can calculate their current acceleration by using Newton's second laws of motion. Thus we can also calculate their future speed and position.

$$a(t) = \frac{F(t)}{m} \tag{2.2}$$

$$v(t) = v_0 + \int_0^t a(u)\,du \tag{2.3}$$

$$x(t) = x_0 + \int_0^t v(u)\,du \tag{2.4}$$

So the acceleration $a(t)$ also depends on the mass of the particle m, which is constant and therefore no problem. We obtain the current speed $v(t)$ by integrating over the acceleration over the entire time period up to t and adding this to the initial speed v_0. Something similar applies to the current position $x(t)$, where we integrate $v(t)$ over the time period and add it to the starting position x_0. Thus to calculate the position of a particle at a time t we have to integrate the total acting force twice. However, the change of the positions again has an influence on the acting forces. This leads to an analytically nearly impossible to solve differential equations with an increasing number of particles. In fact, there is no general analytical solution for the problem with more than two particles [MQ14]. Hence we resort to a numerical method. For this purpose, Störmer-Verlet is often used, which uses the following approximations for a sufficiently small time step Δt [HLW03].

$$v(t + \Delta t) \approx v(t) + \frac{\Delta t}{2}(a(t) + a(t + \Delta t)) \tag{2.5}$$

$$x(t + \Delta t) \approx x(t) + \Delta t v(t) + \frac{\Delta t^2}{2} a(t) \tag{2.6}$$

The smaller we choose Δt, the smaller is the final error of the approximation. But to calculate the position of the particles at a time t we have to use these equations repeatedly until the time steps add up to t. Thus we need to do more calculations for the same simulation time. In each time step, the highest computational load results from the forces between the particles. Since each particle interacts with every other, these calculations grow quadratically with p. To work around this we take advantage of the fact that short-range forces converge towards 0 with increasing distance. We define a cutoff radius r_c and assume that particles beyond this distance have no influence on each other. With this trick, force calculations still take the most time, but they don't get out of hand as the number of particles increases. Figure 2.2 shows the portion of the total runtime which is used for force calculation for a different amount of particles.

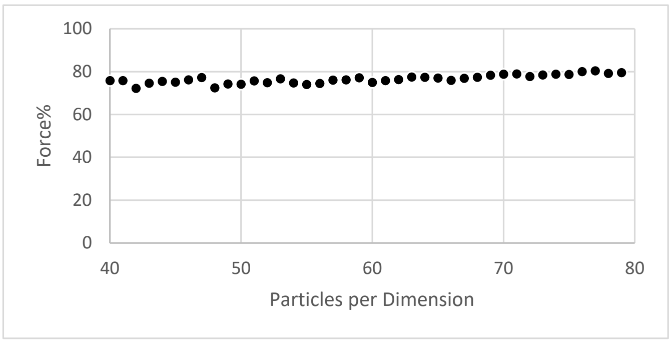


Figure 2.2: Percentage of total execution time for force calculations for different number of particles. Particles are arranged in a 3D grid.

There are now many different sub-algorithms we can choose from to exploit the cutoff radius. Many of them also share some adjustable parameters.

Parameter	Type	Examples
Subalgorithm	Discrete	Linked Cells, Verlet Lists
Cell size factor	Continuous	0.5, 1.0, 1.5
Data Layout	Discrete	AoS, SoA
Newton3	Discrete	On, Off

Table 2.1: Parameter space

Some sub-algorithms divide the considered area into cells. Cell size factor can be adjusted to generate a lot of smaller cells or fewer but larger cells. Data layout specifies how the particles are stored in memory to enable SIMD operations when appropriate. Finally, some algorithms can use Newton's third law of motion to halve the number of force calculations. So we have a wide range of options. But which configuration is optimal for a given simulation case is difficult to predict and requires sufficient expert knowledge.

2.2 General Algorithm Selection Problem

We now want to find the settings that optimize a given goal. In our case, we want to minimize the runtime of the simulation. To generalize our problem we assume that we have an unknown objective function f that maps every configuration to a time value. The time value is not directly the measured time but its negation. This leads to a maximization problem instead, which will be of use to us later. We call the set of all allowed configurations the search space. For simplicity, we map discrete parameters to integers. So for the data layout dimension, we do not use the set $\{AoS, SoA\}$ but $\{0, 1\}$. Our problem is therefore illustrated as follows.

$$f\colon \mathcal{X} \mapsto \mathbb{R} \tag{2.7}$$

$$\mathcal{X} := \mathcal{X}_d \times \mathcal{X}_c \tag{2.8}$$

$$\mathcal{X}_d := [N_1] \times [N_2] \times ... \times [N_d] \tag{2.9}$$

$$[N] := \{1, 2, ..., N\} \tag{2.10}$$

$$\mathcal{X}_c := [l_1, u_1] \times [l_2, u_2] \times ... \times [l_c, u_c] \tag{2.11}$$

Meaning we have a discrete search space $\mathcal{X}_d$ and a continuous search space $\mathcal{X}_c$. N_i for any i represents the number of possible values for a particular discrete parameter. For the continuous parameters, we assume that the allowed values can be described by a lower limit of l_i and an upper limit of u_i. Thus $\mathcal{X}_d$ is a regular d-dimensional grid and $\mathcal{X}_c$ is a c-dimensional hyperrectangle. If the choice of some parameters limits the choice of others, this can be achieved by modifying f or the acquisition function which we will discuss in Section 3.2. We can now find the optimal configuration by maximizing the objective function.

$$x^* = \arg\max_{x \in \mathcal{X}} f(x) \tag{2.12}$$

However we assume that we have no expert knowledge, so we don't know anything about f. To us, it is an arbitrary black-box function. The only thing we can do is to evaluate the function at a location x to get its value $f(x)$. In our particular instance, we accomplish this by running some iterations of the simulation with this configuration and measuring the time taken. Since our main goal is to keep the overall runtime low we should avoid evaluating the function too often.

3 Bayesian Optimization

3.1 Generative Model

To estimate the form of f with the help of some evaluations, we have to assume a model of how the values of $f(x)$ are generated. First, we say that f is continuous because points in the search space which are close to each other should have similar results. Besides, evaluations do not always have to deliver the same result but can deviate. This can be due to the distribution of the particles or hardware induced. Therefore, the following model is suitable [Sha+16].

$$f(x) = \Phi(x)^T \boldsymbol{w} + m \tag{3.1}$$

$$y(x) = f(x) + \epsilon \tag{3.2}$$

The function Φ maps from the search space into an unknown space, in which we can map linearly to f using the weights $\boldsymbol{w}$ and y-intercept m. The deviations are modelled by a noise ϵ. We assume a prior normal distribution on that noise and the weights.

$$\epsilon \sim \mathcal{N}(0, \sigma^2) \tag{3.3}$$

$$\boldsymbol{w} \sim \mathcal{N}(0, V_0) \tag{3.4}$$

Consider any number of configurations $x_1, ...x_n$ and their corresponding evaluations $\boldsymbol{y} := (y(x_1), ..., y(x_n))^T$ we see that $\boldsymbol{y}$ is also normally distributed.

$$\boldsymbol{\Phi} := \begin{bmatrix} \Phi(x_1)_1 & \dots & \Phi(x_1)_a \\ \vdots & \ddots & \vdots \\ \Phi(x_n)_1 & \dots & \Phi(x_n)_a \end{bmatrix} \tag{3.5}$$

$$\boldsymbol{y} \sim \mathcal{N}(m\mathbf{1}, \boldsymbol{\Phi} V_0 \boldsymbol{\Phi}^T + \sigma^2 \boldsymbol{I}) \tag{3.6}$$

y is therefore a so-called Gaussian process [Ebd15]. The function Φ which we have ignored so far is unfortunately not trivial to define. Instead we approximate the matrix $A := \boldsymbol{\Phi} V_0 \boldsymbol{\Phi}^T$. Each element A_{ij} corresponds to the inner product of $\Phi(x_i)$ and

$\Phi(x_j)$ scaled by V_0. Intuitively each element describes the similarity between these configurations. Therefore we approximate each element A_{ij} by $k(x_i, x_j)$. This is called the kernel trick and k is accordingly called kernel function. A commonly used function is the squared exponential kernel [Sha+16].

$$r(x_i, x_j)^2 := (x_i - x_j)^T \mathrm{diag}(\theta_1, ..., \theta_c)(x_i - x_j) \tag{3.7}$$

$$k_{SE}(x_i, x_j) := \theta_0^2 \exp(-\frac{1}{2} r(x_i, x_j)^2) \tag{3.8}$$

$\theta_0, ..., \theta_c$ are hyperparameters, which can be used to set the degree of influence of all or certain dimensions on the kernel value. These and m must be adjusted for each specific case. This can be done manually or automatically. For the former, some prior knowledge is necessary though.

3.2 Acquisition Functions

Gaussian processes have the convenient property that conditional probabilities can be expressed in closed form. If we have already measured some runtimes y_i for configurations x_i, we can use them to estimate runtimes of not yet tested configurations. Let us call D_n the evidence set, which contains all measurements made so far. Given D_n our model infers that the value y_* for configuration x_* is normally distributed.

$$D_n := \{(x_1, y_1), ..., (x_n, y_n)\} \tag{3.9}$$

$$y_* | D_n \sim \mathcal{N}(\mu(x_* | D_n), \sigma^2(x_* | D_n)) \tag{3.10}$$

$$\boldsymbol{k}(x) := (k(x, x_1), ..., k(x, x_n))^T \tag{3.11}$$

$$K(\boldsymbol{x}) := \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix} \tag{3.12}$$

$$\mu(x_* | D_n) = m + \boldsymbol{k}(x_*)^T (K(\boldsymbol{x}))^{-1} (\boldsymbol{y} - m) \tag{3.13}$$

$$\sigma^2(x_* | D_n) = k(x_*, x_*) - \boldsymbol{k}(x_*)^T (K(\boldsymbol{x}))^{-1} \boldsymbol{k}(x_*) \tag{3.14}$$

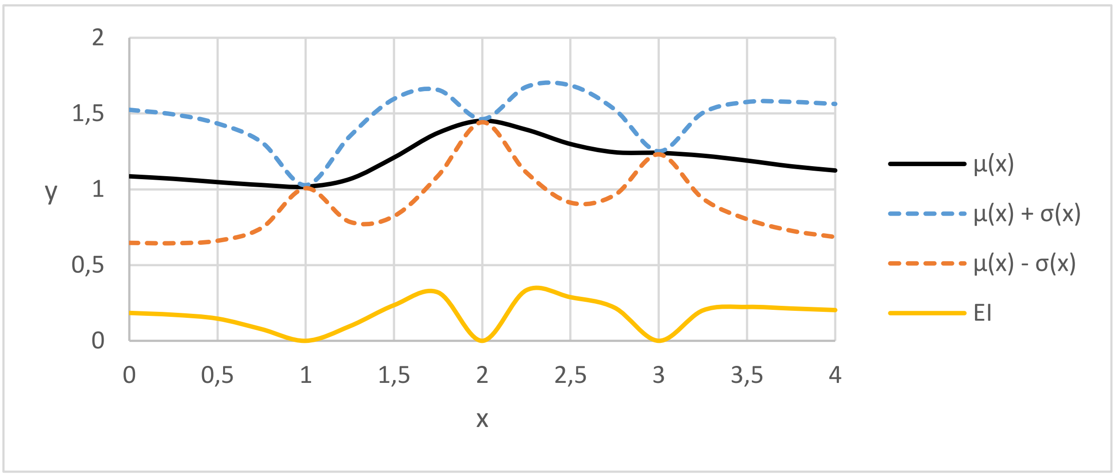


Figure 3.1: 1D examle function evaluated at 1, 2 and 3. Plot shows mean, confidence interval and expected improvement.

We use these distributions to estimate which configuration will give us the most information when we test it. The expected information gain is described using so-called acquisition functions [WHD18]. For this purpose upper confidence bound (UCB), probability of improvement (PI), or expected improvement (EI) are often used.

$$\alpha_{UCB}(x|D_n) := \mu(x|D_n) + \beta \cdot \sigma(x|D_n) \tag{3.15}$$

$$\alpha_{PI}(x|D_n) := \mathbb{P}[y(x) > y_{max}|D_n] \tag{3.16}$$

$$\alpha_{EI}(x|D_n) := \mathbb{E}[\max\{y(x) - y_{max}, 0\}|D_n] \tag{3.17}$$

So to decide which configuration x_{n+1} we test next, we maximize the selected acquisition function.

$$x_{n+1} = \arg\max_{x \in \mathcal{X}} \alpha(x|D_n) \tag{3.18}$$

Figure 3.1 shows the expected improvement of a 1D example function. We can also infer some information from the plot ourselves. We can easily see that we already have 3 evidence because exactly at these points the variance is low. In a normal distribution, the probability of deviating from the mean by less than one standard deviation is approximately 68%. So we expect that in most cases the actual function will be within the range limited by the dashed lines. We see that the EI avoids values near the evaluated points, as these probably do not deviate much from the respective values.

Nevertheless, we expect equally good values near the current optimum. Accordingly, EI assigns the highest acquisition value to points near this optimum but with a slight spacing.

3.3 Hyperparameter Update

The evidence can also be used to adjust the hyperparameter $\boldsymbol{\theta} := (\theta_0, ..., \theta_c, m)$ from Section 3.1. The conditional probability of the vector of evaluations $\boldsymbol{y}$ given the vector of all corresponding configurations $\boldsymbol{x}$ is normally distributed.

$$\boldsymbol{y}|\boldsymbol{x} \sim \mathcal{N}(m\mathbf{1}, K_{\boldsymbol{\theta}}(x) + \sigma^2 \boldsymbol{I}) \tag{3.19}$$

If we assume a constant prior over every hyperparameter we can calculate the probability that given hyperparameters fit the evidence set using Bayes' theorem.

$$\begin{aligned} p(\boldsymbol{\theta}|D_n) =& \frac{p(\boldsymbol{y}|\boldsymbol{x},\boldsymbol{\theta})p(\boldsymbol{\theta})}{p(D_n)} \\ =& \kappa \frac{1}{\sqrt{|K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2 \boldsymbol{I}|}} \exp\left(-\frac{1}{2}(\boldsymbol{y} - m\mathbf{1})^T (K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2 \boldsymbol{I})^{-1} (\boldsymbol{y} - m\mathbf{1})\right) \end{aligned} \tag{3.20}$$

κ is a constant factor that can often be ignored because we only compare these probabilities with each other. We could just use the hyperparameters that maximize this probability, but this would result in the loss of all other hyperparameters which may only have a slightly lower probability. To make sure they are taken into account we use a set of hyperparameters Θ. The acquisition function is now computed by evaluating it for all hyperparameters of the set and summing up the results weighted by their respective probabilities.

$$\nu := \sum_{\boldsymbol{\theta} \in \Theta} p(\boldsymbol{\theta}|D_n) \tag{3.21}$$

$$\alpha(x|D_n, \Theta) = \frac{1}{\nu} \sum_{\boldsymbol{\theta} \in \Theta} p(\boldsymbol{\theta}|D_n) \cdot \alpha(x|D_n, \boldsymbol{\theta}) \tag{3.22}$$

3.4 Discrete Dimensions

All previous calculations would also work with our discrete values. But if we simply use the transformation from Section 2.2 we notice, that the difference of two options is affected by their position in the set. This is an issue when calculating the distance from Equation 3.7. If we have for example the options $\{1, 2, 3\}$, options 1 and 2 will have a smaller distance than options 1 and 3 even if we have not specified anything to that effect. One way to solve this problem is to convert the discrete dimension using one-hot-encoding. Thus each parameter gets its own dimension which can take the values 0 or 1. When converting, we set the corresponding dimension of the parameter to 1 and the rest to 0. As a result, the difference between any two options is only zero if both options are the same. But the method has some shortcomings. Since each option correlates to a dimension, each option also generates a hyperparameter θ_i in the distance function, which weights this dimension more or less. Not only do additional hyperparameters make the search for the optimal values more difficult, but they can only be used to a limited extent. θ_i scales only the distance between two options if one uses option i and the other does not use option i. For example, we cannot state that option 1 and option 2 are strongly correlated without affecting their correlation to option 3. So we generally cannot use the hyperparameters to divide the options into groups in which they are similar to each other. This would only work if at most one group contains more than one option, which generally is of no use.

4 Cluster Model

As discrete values can not be treated simply like continuous values, we separate both types. The following idea is based on a paper from Anh Tran, Minh Tran, and Yan Wang [TTW19]. We create for each discrete value its own Gaussian process. As a result, we have a grid of Gaussian processes. We call each grid point a cluster. So the clusters are each associated with a possible discrete tuple $x_d \in \mathcal{X}_d$. Each cluster models the continuous search space if the discrete search space is fixed to the corresponding discrete tuple. A cluster l thus has its own evidence set E_l and can accordingly estimate the runtime of a continuous tuple x_c independently of the other cluster. Figure 4.1 shows the steps of the model and how it is used in connection with the simulation.

4.1 Correlation between Clusters

If we optimize each cluster individually we would ignore any correlation between discrete values. But we also cannot let every cluster interact with all others. This would lead to calculations that grow quadratic with the number of clusters. So we restrict the interaction of cluster x_d to a subset $\mathcal{B}(x_d)$, which we call its neighborhood. This method is ideal for excluding cluster pairs that are known to be unrelated. In our case, we decided to include all clusters in the neighborhood of x_d, which have a manhattan distance of 1 to x_d in the configuration encoding. Thus if only one parameter is set differently, we assume that they still show some correlation. Therefore the number of neighbors grow in $O(N_1 + ... + N_d)$. Without this restriction, the neighbors would grow in $O(N_1 \cdot ... \cdot N_d)$, which rapidly becomes unmanageable. After the clusters have been roughly sorted out, we can fine-tune the influence between them using weights. The idea is that when calculating the acquisition of a point we not only include the information in the corresponding cluster but also its neighbors in a weighted way. We use two types of weights, which we multiply to obtain the final weight. The first is completely based on prior knowledge and assigns a fixed value to each pair before any evidence is added. The second tries to determine the correlation using the evidence collected so far. Theoretically, there are no restrictions on these weight functions. But we orientate ourselves on the following intuitive properties. Let us assume we want to calculate the acquisition of the point (x_d, x_c). We call the cluster of x_d for this calculation main cluster and each neighbor in $\mathcal{B}(x_d)$ subcluster. If the main cluster and a subcluster

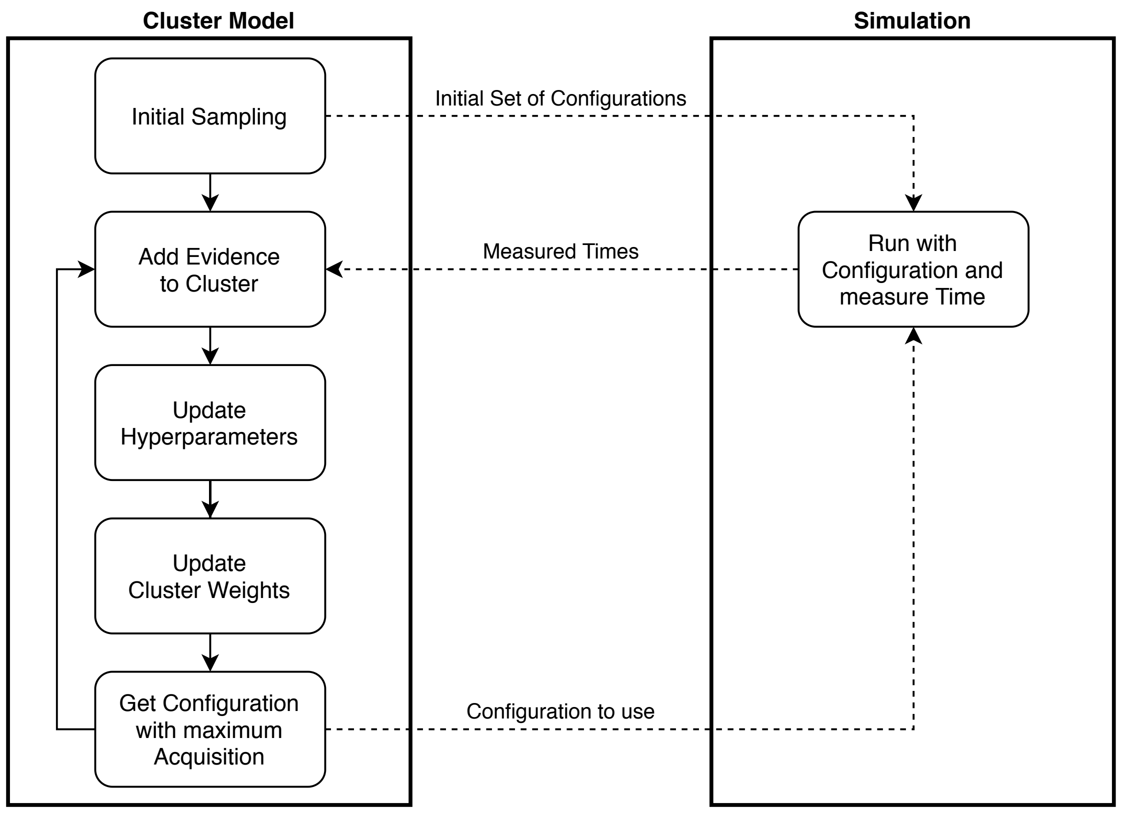


Figure 4.1: General workflow of the cluster model. We start with an initial sampling, without which our cluster model cannot yet work properly. We then enter a cycle where we use the measured times to update our model and the model then suggests the next configuration for testing.

have evidence at the same continuous position, a significant difference in runtimes should indicate low similarity between the configurations and thus should lead to a low weight. If the variance of x_c in the main cluster is low, the point should be relatively accurate. Therefore, subclusters should not have much influence anymore, i.e. they should have a low weight. Accordingly, if the variance is high, all weights should be high, because the main cluster is rather uncertain about this point.

4.1.1 Wasserstein Distance

The original paper uses the Wasserstein metric to compare the similarity of the clusters [TTW19]. Intuitively, both probability density functions are seen as piles of earth. The height of the pile in a point corresponds to the corresponding probability density at that point. We now want to get from one heap of earth to the other. For this purpose, we can move the earth from each point x_1 to another point x_2. The work we do depends on the amount we move and the distance $|x_1 - x_2|$. The Wasserstein distance is the minimum amount of work that must be done to get to the other pile. Therefore this distance is also known as the earth mover's distance [RTG98]. If m is the main cluster and l a subcluster, the following weighting is used.

$$w(m,l) = \left(\sigma_m^2 + W_2(\mathcal{N}(\mu_m, \sigma_m^2), \mathcal{N}(\mu_l, \sigma_l^2))\right)^{-1} \tag{4.1}$$

Where μ_k and σ_k^2 are the expected value and variance of the normal distribution $\mathcal{N}(\mu_k, \sigma_k^2)$ when the runtime of x_c is estimated using the cluster k. The Wasserstein distance of two normal distributions can be calculated quite easily.

$$W_2(\mathcal{N}(\mu_m, \sigma_m^2), \mathcal{N}(\mu_l, \sigma_l^2)) = ||\mu_m - \mu_l||^2 + ||\sigma_m - \sigma_l||^2 \tag{4.2}$$

Since the Wasserstein distance from the main cluster to itself is zero, its weight is the inverse of the variance. For low variances, this can lead to very high values. To avoid numeric instabilities we rescale the weights such that the weight of the main cluster is 1.

$$w'(m,l) = \frac{\sigma_m^2}{\sigma_m^2 + W_2(\mathcal{N}(\mu_m, \sigma_m^2), \mathcal{N}(\mu_l, \sigma_l^2))} \tag{4.3}$$

4.1.2 Evidence Fitting

We propose another way of weight calculation, which uses the posterior probability of the Gaussian processes. Each cluster can assign every point x_c a normal distribution for the expected runtime. We can also use this normal distribution to estimate how likely it is that an evidence pair (x, y) fits into a cluster. The idea is to take the evidence of a subcluster and evaluate if this evidence also suits the main cluster. For this we use the corresponding probability density function, scaling its maxima to 1.

$$p_m(y|x) = \exp\left(-\frac{1}{2}\left(\frac{\mu_m - y}{\sigma_m}\right)^2\right) \tag{4.4}$$

Similar to the Wasserstein distance, we use the rescaling so that the weight of the main cluster is fixed at 1 to avoid too large values. This also avoids the problem that high variances lead to generally low values of the probability density function. To get the final weight, we combine the values for all evidence of the subcluster using a mean value function of our choice. The geometric mean can be rewritten in a particularly elegant way.

$$\begin{aligned} w(m,l) &= \left(\prod_{(x,y)\in E_l} \exp\left(-\frac{1}{2}\left(\frac{\mu_m(x) - y}{\sigma_m(x)}\right)^2\right)\right)^{\frac{1}{|E_l|}} \\ &= \exp\left(-\frac{1}{2|E_l|}\sum_{(x,y)\in E_l}\left(\frac{\mu_m(x) - y}{\sigma_m(x)}\right)^2\right) \end{aligned} \tag{4.5}$$

Where E_l is the evidence set of cluster l and $|E_l|$ the corresponding number of evidence. We see that for the geometric mean we only have to calculate the exponential function once.

4.2 Acquisition using Neighbors

Next, we have to decide how to use the neighbors and weights to calculate the acquisition of a point (x_d, x_c).

4.2.1 Weighted Sum of Random Variables

One possibility is to estimate the weighted sum of the runtimes of all neighbors given the continuous part x_c. Since all these times are normally distributed, their sum is also normally distributed with the following mean and variance.

$$\mu(x_d, x_c) = \frac{1}{\nu}\sum_{l\in\mathcal{B}(x_d)} w(x_d, l)\cdot\mu_l(x_c) \tag{4.6}$$

$$\sigma^2(x_d, x_c) = \frac{1}{\nu^2}\sum_{l\in\mathcal{B}(x_d)} w(x_d, l)^2\cdot\sigma_l^2(x_c) \tag{4.7}$$

$$\nu := \sum_{l \in \mathcal{B}(x_d)} w(x_d, l) \tag{4.8}$$

We can simply replace the normal distribution of the single main cluster with this combined normal distribution when calculating the acquisition function. The mean of the distribution is the weighted sum of the individual means. But for the variance, we have to square the individual weights and the normalization factor ν. Unfortunately ν^2 can be significantly larger than the squared weights. In the worst-case, all weights have the same value w. Then the factor between ν^2 and the sum of all squared weights can be calculated as follows.

$$\frac{\nu^2}{\sum\limits_{l \in \mathcal{B}(x_d)} w(x_d, l)^2} = \frac{(|\mathcal{B}(x_d)| w)^2}{|\mathcal{B}(x_d)| w^2} = |\mathcal{B}(x_d)| \tag{4.9}$$

The more neighbors we have the lower the final variance. Hence, the value can become negligible with many neighbors. Additionally, we encounter a problem, if different clusters contain different numbers of neighbors. In general, a high variance leads to a high acquisition, so this method would prefer clusters with a low number of neighbors for no reason.

4.2.2 Weighted Sum of Acquisitions

Alternatively, the acquisition of each cluster can be calculated individually and the weighted sum of these values can be calculated.

$$\alpha(x_d, x_c) = \frac{1}{\nu} \sum_{l \in \mathcal{B}(x_d)} w(x_d, l) \cdot \alpha_l(x_c) \tag{4.10}$$

Unlike the previous method, the final acquisition is not directly influenced by the number of neighbors. So this method should be preferred, if not all clusters have the same or a too high number of neighbors. Interestingly, this approach sometimes coincides with the calculation of the corresponding mixture distribution. A mixed distribution results from a random selection from a set of random variables. So in our case, we randomly select a cluster with probabilities according to their weights. Then the runtime is generated by the normal distribution of this cluster. The probability density function of this generated value of y equals the weighted sum of the probability densities of the normal distributions.

$$p(y|x_d, x_c) = \frac{1}{\nu} \sum_{l \in \mathcal{B}(x_d)} w(x_d, l) \cdot p_l(y|x_c) \tag{4.11}$$

For the acquisition functions probability of improvement (Equation 3.16) and expected improvement (Equation 3.17), we can thus prove that the corresponding formulas for the mixture distribution equal Equation 4.10.

4.3 Extended Hyperparameter Update

We have now determined how we can use the neighbors to improve our acquisition calculation. Since each cluster has its own Gaussian process, we have to adjust the hyperparameter $\boldsymbol{\theta}$ for each of them. For example, these provide the default mean and variance for the case that a Gaussian process has no evidence yet. But in the end, mainly the evidence set dictates the estimations if these cover the search space well. So the hyperparameters help to estimate points in whose vicinity there is hardly any evidence so far. Therefore it makes sense that every cluster uses the same hyperparameters because without evidence each cluster should make the same estimates. So if we want to use a set of hyperparameters as described in Section 3.3, we have to assign a global weight to each hyperparameter across all clusters. We want to give a high weight to the hyperparameters, which fit well to many clusters. To do so we calculate the same weights for each cluster as in Section 3.3. Each weight indicates how well this hyperparameter fits in the corresponding cluster. Accordingly, a sensible global weight is the sum of all individual weights.

$$p(\boldsymbol{\theta}|D_n) = \sum_{l \in \mathcal{X}_d} p_l(\boldsymbol{\theta}|D_l) \tag{4.12}$$

4.4 Initial Sampling

If we now start with an empty evidence set and use our algorithm to build it incrementally we will encounter some problems. First of all, some clusters do not have any evidence yet, so we cannot use our evidence fitting to determine the neighbors' weights. But we also have a problem using other weight functions when most clusters are empty. With so little evidence, it is difficult to make a statement about the actual variance of the runtimes. The default mean, on the other hand, will probably lie between the minimum and maximum of the current evidence. A cluster, which contains evidence whose runtime is significantly higher than the rest, thus could lower the default mean heavily. Most acquisition functions work with a tradeoff between exploration and exploitation. Generally, this means that points with high mean or high variance are preferred. So it can occur that empty clusters are never tested because the default mean they use is too low. Therefore each cluster must contain some evidence before we can use our optimization algorithm. We collect one evidence for every discrete configuration before we use our model. In our case one evidence in each cluster was sufficient, but of course, this can vary from case to case.

4.5 Evidence Decay

Our black-box function f is special for molecular dynamics simulations. Every time we evaluate the function, we jump one iteration step forward. This changes the simulation with each evaluation. So it can happen that over time the runtime of each configuration changes. As a result, old evidence can become untrustworthy. So we have to modify our algorithm in a way, such that the relevance of each evidence becomes less with each iteration step. We see that in the calculation of the distribution of y given an input x and evidence set D_n the evidence set mainly affects the kernel. Intuitively the similarity of x is calculated with each evidence in D_n and similar x_i indicate that y is similar to y_i. So to weaken this implication, we can artificially increase the similarity to x_i, if the evidence is old. If expert knowledge is available to assign a relevance weight $v(a)$ to each difference in age a, the kernel can easily be modified accordingly.

$$k(x_i, t_i, x_j, t_j) = v(|t_i - t_j|)k(x_i, x_j) \tag{4.13}$$

So we additionally remember for each evidence x_i its creation time t_i. To find the next configuration to be tested, we, as before, look for the x_*, which maximizes the acquisition function. Since we know, that the next evidence will have the time of creation t_{n+1}, we fix $t_* = t_{n+1}$ in the calculations. If we do not know how to precisely define $v(a)$ we can only conclude that the kernel must decrease with an increasing age difference. If we take a closer look at our Equation 4.13, we notice, that the times of creation are behaving analogously to the other continuous dimensions. The more similar the continuous dimensions are, the higher the kernel value. The more similar the times of creation are, the higher $v(a)$. We have therefore decided to add the times of creation to the continuous search space.

$$\mathcal{X}_{c+} := [l_1, u_1] \times [l_2, u_2] \times ... \times [l_c, u_c] \times \mathbb{R} \tag{4.14}$$

This also gives us an additional hyperparameter, which controls the influence of the time differences. Since these are updated automatically as described in Section 4.3, we do not need any expert knowledge. In contrast to the other dimensions, no restriction of the new dimension is necessary. The restriction of the continuous area is only necessary because otherwise, it is impossible to search it properly. But since the time value for the next evidence is always fixed, we do not encounter any problems during the optimization.

$$x_{n+1} = \arg\max_{x \in \mathcal{X}_n} \alpha(x|D_n) \tag{4.15}$$

$$\mathcal{X}_n := \mathcal{X}_d \times \Big(\mathcal{X}_c \times \{t_{n+1}\} \Big) \tag{4.16}$$

5 Tests

The following tests were run on the Linux Cluster of the LRZ Rechenzentrum[1]. For each test, we use one node of the CoolMUC-2 segment. These Haswell-based nodes each have 64 GB DDR4 memory and 28 cores, each supporting 2 hyperthreads. We implemented our auto-tuning algorithms in the library AutoPas [Gra+19].

5.1 AutoPas

AutoPas[2] is an open-source C++ library for molecular dynamic simulations. AutoPas offers interfaces for custom particles and force functors and thus can also be used for arbitrary n-body problems. Special containers that define a fixed region are used for the particles, which make it easier to implement boundary conditions. For example, temporary particles can be added to the edge of the region to simulate a particle setup that repeats infinitely in each dimension. But the centerpieces of the library are the numerous possibilities for the calculation of pairwise short-range interactions like Linked Cells and Verlet Lists. Accordingly, there are many parameters to set. Whereby the choice can significantly affect the total runtime. But also users without a scientific background can easily use the library, as it also supports automatic tuning.

5.2 Algorithms

We compare the following auto-tuning algorithms implemented in the particle simulation library. All algorithms were generalized so that we have two phases. The first one is called the tuning phase. This corresponds to the optimization problem we discussed throughout the thesis. Each algorithm can specify different configurations, with which the simulation is then executed and the measured time is returned. The algorithms decide for themself how many configurations should be tested. When an algorithm is satisfied, it selects a configuration and we switch to the non-tuning phase. In this phase, we simply use the selected configuration for a fixed number of iterations. Many subalgorithms consume more time in the first iteration because they have to build

[1] `https://doku.lrz.de/display/PUBLIC/Linux+Cluster`
[2] `https://github.com/AutoPas/AutoPas`

certain data structures. So with the non-tuning phase, we avoid unnecessary rebuilds when we change the configuration. Since the optimal configuration still can change over time, we start a new tuning phase after each non-tuning phase. Consequently, we always alternate between the two phases.

5.2.1 Full Search (FS)

Full Search simply searches the entire search space exhaustively. Therefore, it cannot work with continuous values and must limit them to a finite number. We expect long tuning phases, but in return, we get a close to an optimal configuration. This approach can encounter some problems with large search spaces. The tuning phases could take up most of the time. They could last so long that the first evidence is no longer representative at the end of the phase. This brute force method is more suitable for small search spaces.

5.2.2 Bayesian Search (BS)

For Bayesian Search, we use the default Bayesian optimization described in Chapter 3. We apply the one-hot encoding mentioned in Section 3.4 to handle discrete values. With some acquisition functions, we could stop the tuning phase if the expected information gain falls below a set limit. But a proper limit depends on the simulation and the used acquisition function. For ease of use, we instead define a fixed number of configurations to be tested each tuning phase.

5.2.3 Bayesian Cluster Search (BCS)

Bayesian Cluster Search uses the cluster model described in Chapter 4. Like BS we limit our number of tested configurations for the same reasons. The other algorithms assume that after each non-tuning phase, the previously collected evidence are no longer trustworthy and therefore discard them for the next tuning phase. Since we already consider this problem using the evidence decay from Section 4.5, we keep all our evidence between phases. The initial sampling from Section 4.4 is therefore only necessary for the first tuning phase. So the relative difference of tuning iterations compared to BS becomes negligible with an increasing number of phases.

5.2.4 Random Search (RS)

Random Search samples a fixed number of configurations, which we select randomly. This would correspond to BS if we let the used acquisition function simply return a random value. RS will therefore collect as much evidence as our Bayesian approaches and thus is a good reference for the quality of our acquisition function.

5.3 Performance

For the tests, we have chosen a homogeneous and an inhomogeneous simulation. Figure 5.1 shows the performance of each algorithm when the particles are arranged in a homogeneous 3D grid. Figure 5.2 shows the performances in a simulation where the positions of the particles are generated using a 3-variate normal distribution. This leads to a strong concentration of particles at one point and the particle density decreases with increasing distance from it. Figure 5.3 visualizes both setups in 2 dimensions. We ran the different algorithms for 400 iterations multiple times and measured the time each run took. For each algorithm, we sampled 20 full runs. In all cases, FS took the longest. On average, RS, BS, and then BCS followed. So in general our algorithm performs best. Figure 5.4 shows the average relative advantage of BCS for each individual case. We see that FS constantly leads to the same runtime because it always tests the same configuration. The Bayesian approaches may vary because sometimes different results are obtained when maximizing the acquisition function. BCS is a bit more consistent than BS. But this also means that BS can be faster if it finds a good configuration by luck.

5.4 Analysis

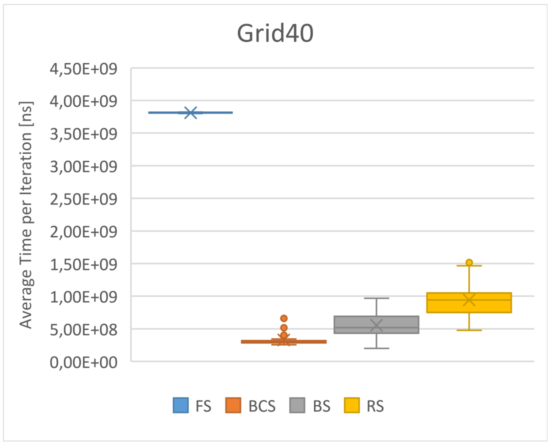
We have done some analysis to find out the reason for the lower runtimes. This could provide a better understanding and may help to find possible improvements.

5.4.1 Overhead

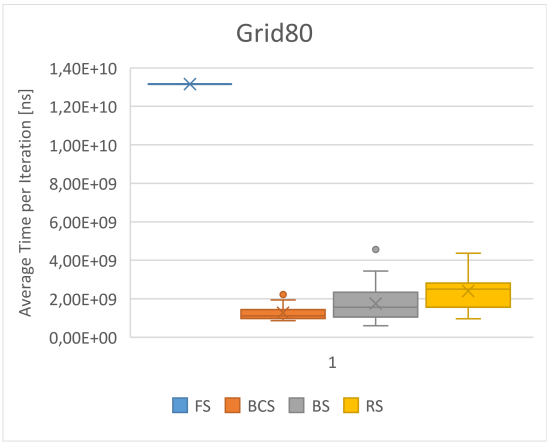
In Figure 5.5 we see the total time each tuning algorithm used to find the next configuration to test. The Bayesian approaches consume significantly more time here in relation to the other algorithms. But the force calculations exceed these times by far, so this overhead of each algorithm is negligible.

5.4.2 Tuning and non-Tuning Times

Figure 5.7 shows us the total time each algorithm spends in the tuning phase. FS thus consumes the main time in this phase as expected, whereas the other algorithms have relatively similar times. Their difference mainly results from their selection for

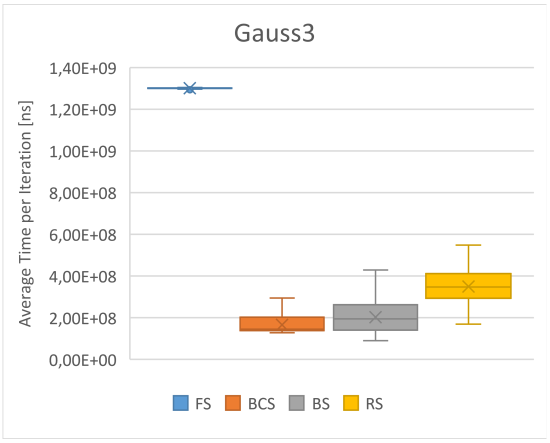


(a) Grid of 40x40x40 particles.

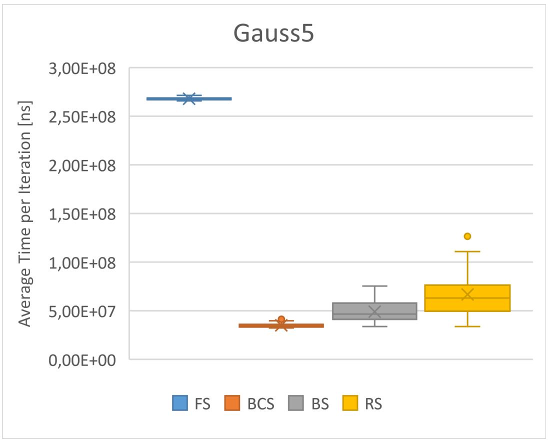


(b) Grid of 80x80x80 particles.

Figure 5.1: Box-and-whiskers plot of the average time per iteration of different tuning algorithms. The particles are generated into a 3D grid and each simulation runs for 400 iterations. Each algorithm was sampled 20 times.

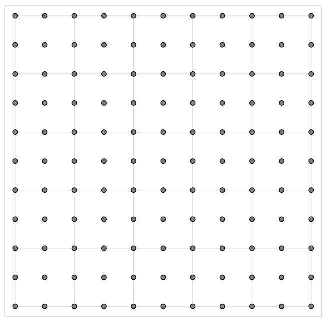


(a) 60000 particles are normally distrbuted using a standard deviation of 3.0.

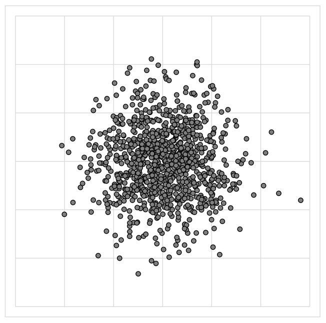


(b) 60000 particles are normally distrbuted using a standard deviation of 5.0.

Figure 5.2: Box-and-whiskers plot of the average time per iteration of different tuning algorithms. The particles' positions are generated using a 3-variate normal distribution and each simulation runs for 400 iterations. Each algorithm was sampled 20 times.



(a) Particles aligned in a grid. This is a homogeneous setup where the density of particles is roughly the same everywhere.



(b) Particles' positions generated using a multivariate normal distribution. This is a inhomogeneous setup where most particles are located in the center

Figure 5.3: Example particle setups in 2D.

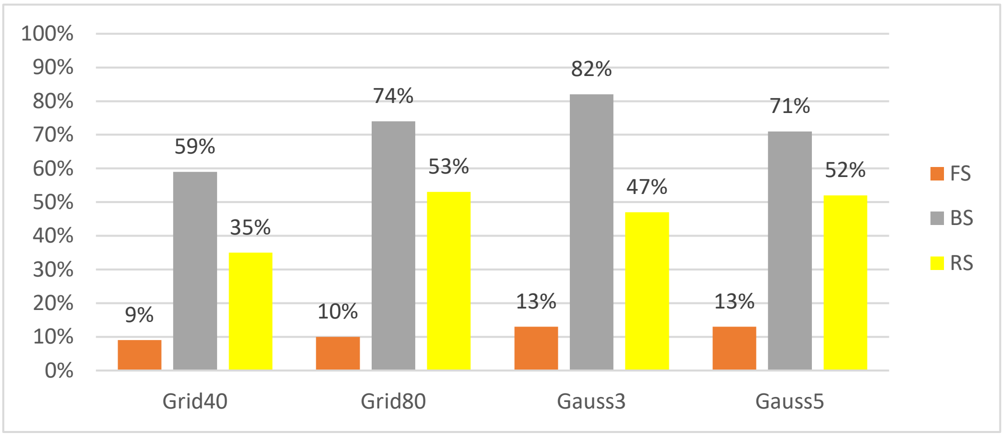
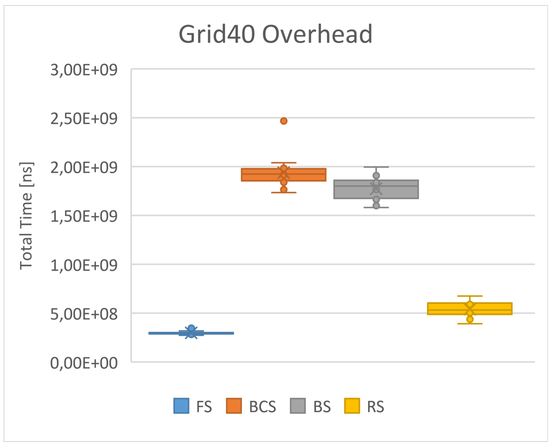
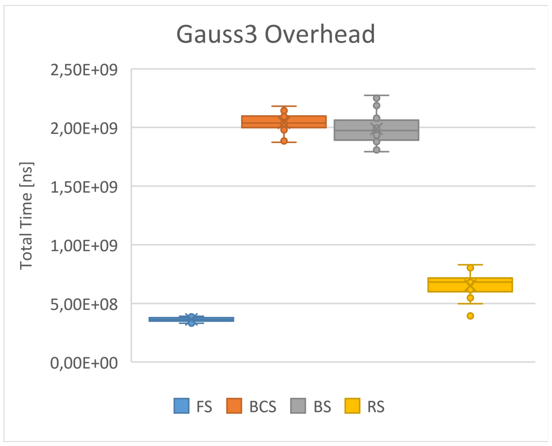


Figure 5.4: Average portion of total time that BCS needs in comparison to other algorithms.



(a) Grid of 40x40x40 particles.



(b) 60000 particles are normally distrbuted using a standard deviation of 3.0.

Figure 5.5: Box-and-whiskers plot of the total time used by different tuning algorithms to choose the next configuration. Each simulation runs for 400 iterations and each algorithm was sampled 20 times.

the non-tuning phases we see in Figure 5.8. The plot shows the average time of non-tuning iterations, which indicates the quality of the selected configuration from the corresponding tuning phase. Unsurprisingly FS mostly achieves a good time value, whereas the quality of the selection of RS fluctuates strongly. BS is a bit better than Random Search and our cluster-based algorithm even performs as well as Full Search. So our algorithm usually achieves an almost optimal selection without having to seek the entire search space. Figure 5.6 lists the number of tuning and non-tuning iterations for the different algorithms. BCS reduces the number of tuning iterations considerably while maintaining an equally good choice.

5.4.3 Force Directed Graph

To get a feel for how the evidence in our cluster algorithm is selected we use force-directed graphs (FDG) [Ban+13]. The creation of such a graph is similar to a molecular dynamics simulation. First, we have a weighted graph, which consists of nodes and weighted edges. In our case, the nodes are all our clusters and the edge weights are the corresponding cluster weights from Section 4.1. We distribute the nodes randomly over a space. We now pretend that all nodes are particles that exert forces on each other. The repulsive forces are adjusted so that the particles do not overlap too much or all form a lump. The attractive forces are determined using the edge weights. A larger weight

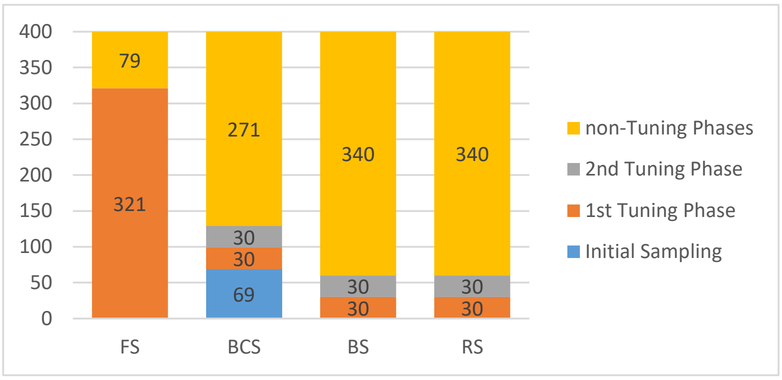
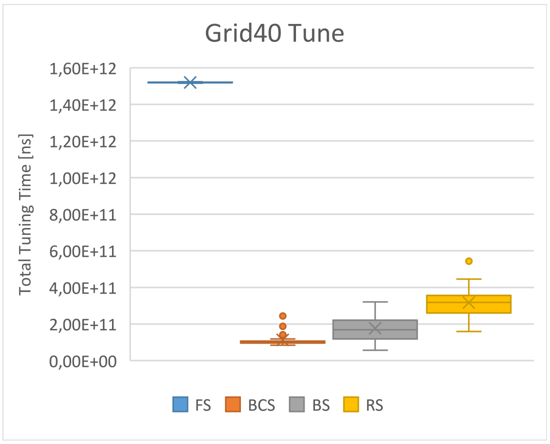
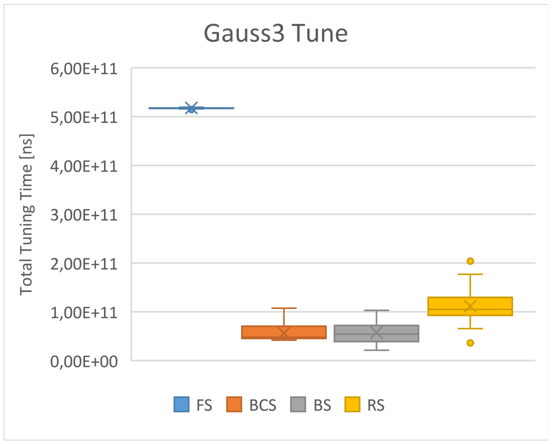


Figure 5.6: Number of tuning and non-tuning Iterations of tested algorithms. The search space contains 107 configurations. For one evidence we measure 3 iterations. We collect 10 evidence per tuning phase for the BCS, BS and RS.

should generate a larger force. The idea of our cluster weights is to give more weight to clusters, from whom we expect similar runtimes. Accordingly, this method should ensure that these clusters are grouped together in space. Figure 5.9 shows the FDG at the end of an example run. We have colored the nodes according to their runtime and see that indeed the resulting groups have similar runtimes. Each number *i* shown in the graph corresponds to the *i*-th sampled evidence of the last tuning phase. We see here that only a maximum of one element of each group is sampled here. Since we expect similar times for the rest of the group, we can consider them as sampled as well. This allows us to cover the search space just as well as Full Search, but with significantly less evidence.

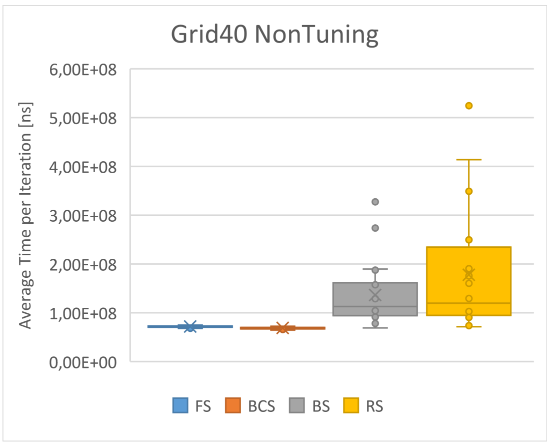


(a) Grid of 40x40x40 particles.

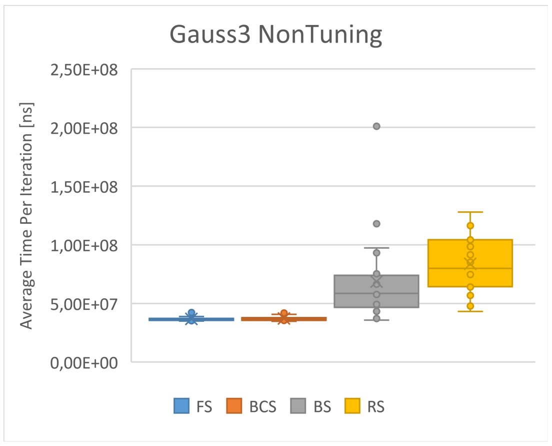


(b) 60000 particles are normally distrbuted using a standard deviation of 3.0.

Figure 5.7: Box-and-whiskers plot of the total time of different tuning algorithms during tuning phases. Each simulation runs for 400 iterations and each algorithm was sampled 20 times.



(a) Grid of 40x40x40 particles.



(b) 60000 particles are normally distrbuted using a standard deviation of 3.0.

Figure 5.8: Box-and-whiskers plot of the average time per iteration of different tuning algorithms during non-tuning phases. Each simulation runs for 400 iterations and each algorithm was sampled 20 times.

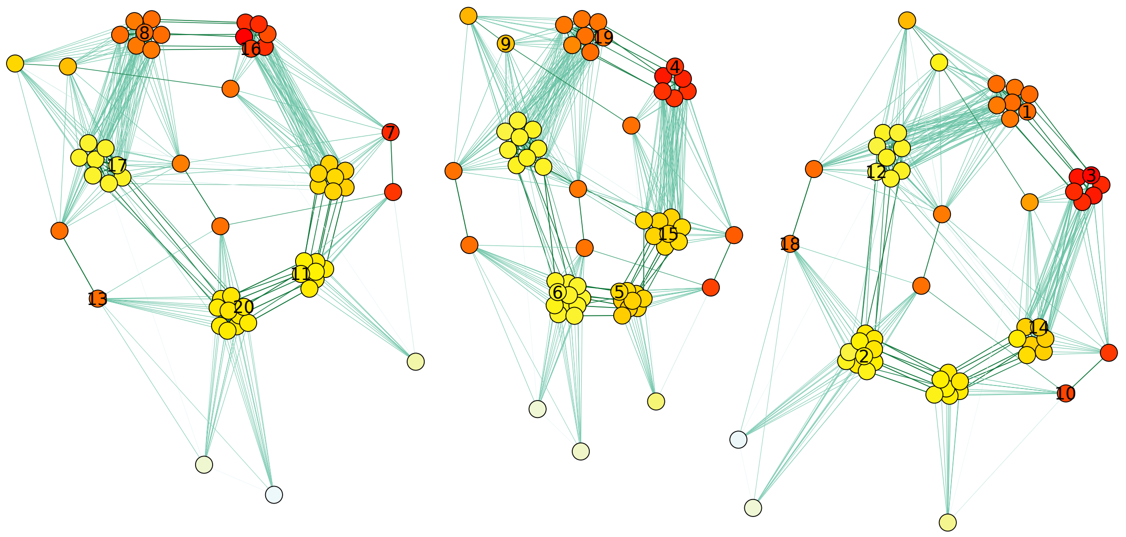


Figure 5.9: Force directed graph using allowed configuration as nodes. The weights between nodes equal the weights between the corresponding cluster. The numbers show the order in which the evidence was selected in the last tuning phase. The nodes are colored according to their runtime, where red corresponds to a low runtime and white to a high runtime. We have limited the selection of the cell size factor to 3 values. Each CSF corresponds to one of the connected graphs.

6 Conclusion

In summary, we showed how Bayesian optimization enables us to optimize a black-box function without having to evaluate the function too often. This is especially helpful if each evaluation is associated with some cost, which in our case was time. With the help of acquisition functions, we estimate the information gain of each point and could correspondingly extract adequate knowledge with a limited number of evaluations. But this method is not well suited for discrete spaces. To get around this, we consider each discrete value separately. Each value thus forms a cluster in which we only had to optimize the continuous values. But the clusters can also show similarities to each other, which we want to exploit. Using the evidence collected so far, we estimate the similarity between the clusters and incorporate this knowledge into the calculations. With this, we can often achieve a similarly good result as an exhaustive search, while reducing the number of needed evidence dramatically. This is especially advantageous for auto-tuning since the goal is to keep the overall runtime low. Our solution is appropriate for this and we have kept it as general as possible so that it can be applied to many other problems as well.

Bibliography

[All04] M. P. Allen. *Introduction to molecular dynamics simulation*. 2004.

[Ban+13] M. J. Bannister, D. Eppstein, M. T. Goodrich, and L. Trott. "Force-Directed Graph Drawing Using Social Gravity and Scaling." In: *Graph Drawing*. Ed. by W. Didimo and M. Patrignani. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 414–425. ISBN: 978-3-642-36763-2.

[Ebd15] M. Ebden. *Gaussian Processes: A Quick Introduction*. 2015. arXiv: `1505.02965 [math.ST]`.

[Gra+19] F. A. Gratl, S. Seckler, N. Tchipev, H. Bungartz, and P. Neumann. "AutoPas: Auto-Tuning for Particle Simulations." In: *2019 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. May 2019, pp. 748–757. DOI: `10.1109/IPDPSW.2019.00125`.

[HLW03] E. Hairer, C. Lubich, and G. Wanner. "Geometric numerical integration illustrated by the Störmer–Verlet method." In: *Acta Numerica* 12 (2003), pp. 399–450. DOI: `10.1017/S0962492902000144`.

[Liz+07] D. Lizotte, T. Wang, M. Bowling, and D. Schuurmans. "Automatic Gait Optimization with Gaussian Process Regression." In: Jan. 2007, pp. 944–949.

[MQ14] Z. E. Musielak and B. Quarles. "The three-body problem." In: *Reports on Progress in Physics* 77.6 (June 2014), p. 065901. ISSN: 1361-6633. DOI: `10.1088/0034-4885/77/6/065901`.

[Ngu19] V. Nguyen. "Bayesian Optimization for Accelerating Hyper-Parameter Tuning." In: *2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*. 2019, pp. 302–305.

[RTG98] Y. Rubner, C. Tomasi, and L. J. Guibas. "A metric for distributions with applications to image databases." In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)*. 1998, pp. 59–66.

[Sha+16] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. de Freitas. "Taking the Human Out of the Loop: A Review of Bayesian Optimization." In: *Proceedings of the IEEE* 104.1 (Jan. 2016), pp. 148–175. ISSN: 1558-2256. DOI: `10.1109/JPROC.2015.2494218`.

[TTW19] A. Tran, M. Tran, and Y. Wang. "Constrained mixed-integer Gaussian mixture Bayesian optimization and its applications in designing fractal and auxetic metamaterials." In: *Structural and Multidisciplinary Optimization* (Jan. 2019). DOI: `10.1007/s00158-018-2182-1`.

[Ulm+15] D. Ulmasov, C. Baroukh, B. Chachuat, M. P. Deisenroth, and R. Misener. *Bayesian Optimization with Dimension Scheduling: Application to Biological Systems*. 2015. arXiv: `1511.05385 [stat.ML]`.

[WHD18] J. T. Wilson, F. Hutter, and M. P. Deisenroth. *Maximizing acquisition functions for Bayesian optimization*. 2018. arXiv: `1805.10196 [stat.ML]`.