DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Interdisciplinary Project report

AutoTuning using Bayesian Statistics in AutoPas

Jan Nguyen

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Interdisciplinary Project report

AutoTuning using Bayesian Statistics in AutoPas

AutoTuning via Bayessche Statistik in AutoPas

Author:	Jan Nguyen
Supervisor:	Univ.-Prof. Dr. Hans-Joachim Bungartz
Advisor:	Fabio Gratl, M.Sc.
Submission Date:	26.01.20

Abstract

In many cases, a program can have many configuration options. The choice can have a large impact on the performance, but may not always be trivial for the layman. It is possible to recommend options that are efficient in most cases. However, if the individual use case leads to significant differences in the optimal choice, automation is preferable. We have analyzed how Bayesian statistics can be applied here. Such an algorithm uses a probabilistic model to generate a good configuration by observing some test runs. For the case of molecular dynamics simulations, we implemented this idea into the C++ library AutoPas. This achieved on average significantly better results than brute force and purely random methods.

Contents

1 Introduction

In high-performance computing (HPC) we try to solve various computing-intensive problems. The calculation time for normal computers would be way too high in most cases. Instead, more powerful supercomputers are used. For them, it is important how their resources are optimally used as they are most often based on parallel computing. How we use multiple processor cores can greatly affect the overall performance. One example, which we also discuss extensively in this paper, are molecular dynamics simulations. There are different algorithms to run the simulation, but they can use the resources with varying degrees of efficiency depending on the simulation scenario. This is an instance of the often encountered problem that expert knowledge is necessary. Here the expert is needed to evaluate the possible choices of the simulation parameters. To enable non-professionals to use such an algorithm, we try to transfer the responsibility of parameter selection to the computer. Bayesian optimization as described in Chapter 4 is ideal for this purpose as its usage does not require any prior knowledge. This paper shows the general underlying idea and how we tailored it for our purposes.

2 Problem statement

Given is an unknown objective function f which we want to optimize.

$$f\colon \mathcal{X} \mapsto \mathbb{R} \tag{2.1}$$

In our case, the function returns the time a run of the analyzed algorithm needs. The space of all possible inputs $\mathcal{X}$ is called search space. An input of the algorithm may consist of d independent choices. So the search space may be written as a Cartesian product of d sets.

$$\mathcal{X} := \mathcal{X}_1 \times \mathcal{X}_2 \times ... \times \mathcal{X}_d \tag{2.2}$$

f is a black-box function as we do not know how a configuration x relates to its runtime $f(x)$. We search for a configuration x^* which leads to a minimal runtime.

$$x^* = \arg\min_{x \in \mathcal{X}} f(x) \tag{2.3}$$

A simple solution would be to pick a sufficient amount of elements from $\mathcal{X}$ to cover it well and determine $f(x)$ for each element. Although it is not quite obvious how a good cover is defined. If $\mathcal{X}$ is finite, we would have the trivial option to check the whole space. But in both cases, it is problematic if the evaluation of the function is considered expensive in terms of time, money or the like. This forces us to limit the number of evaluations, making the choice of each tested configuration crucial.

3 Gaussian process

3.1 Multivariate normal distribution

The main idea is to approximate the black-box function using a probabilistic model. A good base is the multivariate normal distribution which is a generalization of the well-known (univariate) normal distribution. A vector of random variables is (multivariate) normal distributed if each linear combination of the variables is normally distributed. Consequently a multivariate normally distribution $\mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma})$ is fully described by its mean vector $\boldsymbol{\mu}$ and its covariance matrix $\boldsymbol{\Sigma}$. Let $\boldsymbol{X} = (X_1, X_2, ..., X_d)$ a d-variate normally distributed random vector.

$$\boldsymbol{X} \sim \mathcal{N}(\boldsymbol{\mu}, \boldsymbol{\Sigma}) \tag{3.1}$$

The corresponding density function for a d-variate normal distribution is:

$$f_{\boldsymbol{X}}(x_1, ..., x_d) = \frac{1}{\sqrt{(2\pi)^d |\boldsymbol{\Sigma}|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - \boldsymbol{\mu})^T \boldsymbol{\Sigma}^{-1} (\boldsymbol{x} - \boldsymbol{\mu})\right) \tag{3.2}$$

Whereby $|\boldsymbol{\Sigma}|$ and $\boldsymbol{\Sigma}^{-1}$ are the determinant and inverse of $\boldsymbol{\Sigma}$ respectively. Normal distributions are a popular choice in statistics because of the central limit theorem. The multivariate variant of this theorem states that the componentwise sum of k independent and identical distributed random vectors will tend toward a multivariate normal distribution if k is chosen sufficiently high. The theorem supposedly applies to arbitrary distributions of the random vectors [Vaa98]. Therefore if the distribution is unknown a normal distribution is a decent assumption most of the time.

3.2 Infinite dimensional generalization

The multivariate normal distribution can be extended to a model with infinite random variables $\{h(x) : x \in \mathcal{X}\}$. We call $\mathcal{X}$ the index set which may be infinite. The function h maps each index to a random variable. If all finite subsets of the variables are normally distributed we call the model a Gaussian process [Do07]. To describe this model completely we only need a mean value function m and a covariance function k.

$$m\colon \mathcal{X} \mapsto \mathbb{R} \tag{3.3}$$

$$k\colon \mathcal{X} \times \mathcal{X} \mapsto \mathbb{R} \tag{3.4}$$

For vectors of indices $\boldsymbol{x} \in \mathcal{X}^n$ of any length n we define a random vector $h(\boldsymbol{x})$, a mean vector $m(\boldsymbol{x})$ and covariance matrix $K(\boldsymbol{x})$.

$$h(\boldsymbol{x}) := (h(x_1), ..., h(x_n))^T \tag{3.5}$$

$$m(\boldsymbol{x}) := (m(x_1), ..., m(x_n))^T \tag{3.6}$$

$$K(\boldsymbol{x}) := \begin{bmatrix} k(x_1, x_1) & \dots & k(x_1, x_n) \\ \vdots & \ddots & \vdots \\ k(x_n, x_1) & \dots & k(x_n, x_n) \end{bmatrix} \tag{3.7}$$

Let $\mathcal{GP}(\mathcal{X}, m, k)$ be a Gaussian process. Each random vector we can generate from the index set is normally distributed.

$$\forall n \in \mathbb{N} : \forall \boldsymbol{x} \in \mathcal{X}^n : h(\boldsymbol{x}) \sim \mathcal{N}(m(\boldsymbol{x}), K(\boldsymbol{x})) \tag{3.8}$$

Meaning our naming of the functions m and k is accurate.

$$m(x) = \mathbb{E}\Big[h(x)\Big] \tag{3.9}$$

$$k(x_1, x_2) = \mathbb{E}\Bigg[\Big(h(x_1) - \mathbb{E}\Big[h(x_1)\Big]\Big)\Big(h(x_2) - \mathbb{E}\Big[h(x_2)\Big]\Big)\Bigg] \tag{3.10}$$

We may choose m without limitation but k has to lead to a valid covariance matrix $K(\boldsymbol{x})$ for any $\boldsymbol{x}$. Since every covariance matrix is positive semi-definite, the same must apply to k.

The main argument in favor of Gaussian processes is the possibility to represent conditional distributions in closed form. Assuming we have a set of evidence $D_n = \{(x_1, y_1), ..., (x_n, y_n)\}$. The set contains pairs (x, y) where each y is a measurement performed on the corresponding random variable $h(x)$. For example each variable $h(x)$ represents the runtime of a certain configuration x. To measure this variable we run the algorithm with this configuration and time it. Since time depends on various factors, different results may be obtained at random. Given these measurements D_n the distribution of an index x_* is normally distributed.

$$h(x_*|D_n) \sim \mathcal{N}(\mu(x_*|D_n), \sigma^2(x_*|D_n)) \tag{3.11}$$

If we define $\boldsymbol{k}(x) := (k(x, x_1), ..., k(x, x_n))^T$ we can represent the expected value and the variance as follows.

$$\mu(x_*|D_n) = m(x_*) + \boldsymbol{k}(x_*)^T (K(\boldsymbol{x}))^{-1} (\boldsymbol{y} - m(\boldsymbol{x})) \tag{3.12}$$

$$\sigma^2(x_*|D_n) = k(x_*, x_*) - \boldsymbol{k}(x_*)^T (K(\boldsymbol{x}))^{-1} \boldsymbol{k}(x_*) \tag{3.13}$$

This allows us to estimate the runtime of untested configurations and lets us weigh how certain our estimate is. The most computationally intensive part is the calculation of the inverse of the covariance matrix. But since these computations are independent of x_*, we can pre-calculate them once and use them for estimates of different configurations. For the mean function this also applies to $(\boldsymbol{y} - m(\boldsymbol{x}))$, allowing us to reformulate Equation 3.12.

$$\mu(x_*|D_n) = m(x_*) + \boldsymbol{k}(x_*)^T \boldsymbol{\lambda} \quad \text{with } \boldsymbol{\lambda} := (K(\boldsymbol{x}))^{-1} (\boldsymbol{y} - m(\boldsymbol{x})) \tag{3.14}$$

3.3 Regression

To perform a regression on a continuous function f we analyze following generative model.

$$f(x) = \Phi(x)^T \boldsymbol{w} \tag{3.15}$$

$$y(x) = f(x) + \epsilon \quad \text{with } \epsilon \sim \mathcal{N}(0, \sigma^2) \tag{3.16}$$

The function Φ maps elements from the index set to a feature space. For vectors $\boldsymbol{x}$ we define $\boldsymbol{\Phi}$ as the feature matrix such that $\boldsymbol{\Phi}_{ij} = \Phi(x_i)_j$. In the feature space we assume that we can represent the output of the function using a linear regression with weights $\boldsymbol{w}$. Additionally we account for noise with ϵ. Assuming a normal prior distribution on the weights it can be shown that any output vector $\boldsymbol{Y} = (y(x_1), ..., y(x_n))^T$ is also normally distributed.

$$\boldsymbol{w} \sim \mathcal{N}(0, V_0) \tag{3.17}$$

$$\boldsymbol{Y} \sim \mathcal{N}(0, \boldsymbol{\Phi} V_0 \boldsymbol{\Phi}^T + \sigma^2 \boldsymbol{I}) \tag{3.18}$$

This means y is a Gaussian process. Unfortunately, it is often hard to define the feature mapping. To avoid this problem we redefine $A := \mathbf{\Phi} V_0 \mathbf{\Phi}^T$ instead of finding a suitable Φ. This is called the kernel trick. The values of A are inner products in the feature space which are scaled by V_0. Intuitively, each value A_{ij} describes how similar x_i and x_j are. So we replace the specific calculation of A_{ij} with a kernel function $k(x_i, x_j)$.

3.4 Kernels

Kernels may be chosen arbitary but to keep y a Gaussian process we have to use a symmetric positive-definite function as they correspond to a covariance function. Common choices are the Matérn kernels [RW06]. We represent the squared distance between arbitary x_1 and x_2 as r^2 using scaling factors $\boldsymbol{\theta}$.

$$r^2 := (x_1 - x_2)^T \mathrm{diag}(\theta_1, ..., \theta_d)(x_1 - x_2) \tag{3.19}$$

Using these distances Matérn defined a class of kernels to generate covariances.

$$k_{1/2}(r) = \theta_0^2 \exp(-r) \tag{3.20}$$

$$k_{3/2}(r) = \theta_0^2 \exp(-\sqrt{3}r)(1 + \sqrt{3}r) \tag{3.21}$$

$$k_{5/2}(r) = \theta_0^2 \exp(-\sqrt{5}r)(1 + \sqrt{5}r + \frac{5}{3}r^2) \tag{3.22}$$

$$k_{SE}(r) = \theta_0^2 \exp(-\frac{1}{2}r^2) \tag{3.23}$$

Note that the scaling $\boldsymbol{\theta}$ is freely configurable and needs to be tailored to the problem. Without prior knowledge, there is also the possibility to utilize the evidence collected so far to derive reasonable values for the hyperparameter as we will see in Section 4.4.

4 Bayesian optimization

4.1 Algorithm outline

With the help of Gaussian process regression we can estimate any black-box function f [Sha+16]. In our case, this also includes the specific problem of estimating the runtime for each configuration. More evidence may lead to more accurate approximations. But since a piece of evidence corresponds to a test run of the algorithm, these are expensive in our sense. To choose the most informative evidence, we use a one-step method. At every step n we have n evidence D_n. Using all currently gathered evidence we update our model. With the updated model, we estimate the input x_{n+1} which yields the highest estimated informational gain $\alpha(x)$. We evaluate the function once to augment the evidence set and then repeat this process.

Algorithm 1 Bayesian optimization

```
1: Choose first input $x_1$: Randomly or with prior knowledge
2: $D_1 = \{(x_1, f(x_1))\}$
3: for $n$ = 1, 2, ... do
4:     Update model with evidence $D_n$
5:     Find optimal input: $x_{n+1} = \arg\max_{x \in \mathcal{X}} \alpha(x|D_n)$
6:     Add to evidence: $D_{n+1} = D_n \cup \{(x_{n+1}, f(x_{n+1}))\}$
```

4.2 Parameters

To convert a concrete problem to a Gaussian process we only need to specify the distance $r(x_1, x_2)$ for any two element x_1 and x_2 in the search space. We simplify this problem by choosing an encoding $e : \mathcal{X} \mapsto \mathbb{R}^d$ and applying Equation 3.19.

$$r(x_1, x_2)^2 := (e(x_1) - e(x_2))^T \text{diag}(\theta_1, ..., \theta_d)(e(x_1) - (x_2)) \tag{4.1}$$

Using Equation 2.2 we can encode each dimension of an element $x \in \mathcal{X}$ separately and concatenate them together to a real vector. Values that are already numeric can directly be converted or just copied. To handle finite sets whose element has no clear correlation to each other, we use a one-hot encoding. For example, a dimension of X is the choice between the algorithms A, B, and C. We then map this dimension to three values, each associated with its respective algorithm. A becomes $(1, 0, 0)$, B becomes $(0, 1, 0)$ and C becomes $(0, 0, 1)$. This encoding can be extended to any size and it always ensures that the difference between the two algorithms is 0 if and only if they are the same.

4.3 Acquisition functions

With this model in place, we still have to clarify how we estimate the informational gain of a given input. As described in the Section 3.2 we can represent the output of $f(x)$ given D_n as a normal distribution. With this base, we may define different acquisition functions α. These functions map each encoded input $v \in R^d$ to a value $\alpha(x|D_n)$ which implements our approximations of gain in different ways.

4.3.1 Variance

An obvious choice is using the variance or the standard deviation of the normal distribution as an acquisition function. Both cases lead to the same outcome, as they trivially always have the same optimum.

$$\alpha_{Var}(x|D_n) := \sigma^2(x|D_n) \tag{4.2}$$

$$\alpha_{SD}(x|D_n) := \sigma(x|D_n) \tag{4.3}$$

The main observation is that an evidence (x, y) minimizes the variance near x. Thus at every step, this function "removes" the point of highest variance. Repeating this process reduces the maximal variance over the whole search space. So we can make more confident approximation at each point. As regions that do not contain evidence have high variance the resulting evidence set often covers the search space thoroughly.

4.3.2 Lower Confidence Bound

Instead of simply searching for the whole area, we should focus on promising sections. By analyzing Equation 3.14, we realize that the expected value is obtained by adding up individual kernels scaled by the elements of the vector $\boldsymbol{\lambda}$. As similar inputs lead to similar kernels, we also expect similar outputs. So $\mu(x|D_n)$ can be used to search near already discovered low runtimes. As we want to maximize acquisition functions, we may choose $-\mu(x|D_n)$. Such an approach often gets stuck in a local minimum. That is why we used a mixture of $-\mu(x|D_n)$ and the standard deviation. The expected value is used for exploitation and the standard deviation for exploration. We could scale both values before adding them together, but because we compare all outputs of the acquisition function in relation to each other the whole function can be scaled by an arbitrary factor. Using this fact we can normalize the scaling factor from the expected value to 1 and thereby obtain the following function.

$$\alpha_{LCB}(x|D_n) := -\mu(x|D_n) + \beta \cdot \sigma(x|D_n) \tag{4.4}$$

$\beta \in (0, \infty)$ is a hyperparameter that determines how much we weight the standard deviation component. For large values of β, this function degenerates to α_{SD}. This acquisition function is called lower confidence bound (LCB), since the maximum of this function corresponds to the minimum of $\mu(x|D_n) - \beta \cdot \sigma(x|D_n)$. In our model, the probability that x given D_n is greater than this value is fixed for any β. As an example for $\beta = 2$, the probability is always approximately 98%. So we are confident that we will not get a value below this bound, but we are optimistic that we get close to it.

4.3.3 Probability of decrease

Using the cumulative distribution function of the normal distribution, we can determine the probability that the random variable is smaller than an arbitrary target value. If we use the lowest runtime already observed y_{min} as the target, this results in the probability of observing a lower runtime.

$$Z(x|D_n) := \frac{y_{min} - \mu(x|D_n)}{\sigma(x|D_n)} \tag{4.5}$$

$$\alpha_{PD}(x|D_n) := \mathbb{P}[f(x) < y_{min}|D_n] = \Phi(Z(x|D_n)) \tag{4.6}$$

Where Φ is the cumulative distribution function of the standard normal distribution $\mathcal{N}(0,1)$. As this function is monotonically increasing we want to maximize $Z(x|D_n)$. As the standard deviation is always positive, this acquisition function always prefers x such that $\mu(x|D_n) < y_{min}$ because of other x lead to a $Z(x|D_n)$ smaller or equal zero. If $\mu(x|D_n) > y_{min}$ we want $\sigma(x|D_n)$ to be high. This behaves similarly to LCB. If $\mu(x|D_n) > y_{min}$ we would like a low standard deviation. This means that if we expect an improvement, we want to be sure of it. This can again lead to the problem of a local minimum. PD prefers small improvements in the vicinity of already tested places because of the variance there is low.

4.3.4 Expected decrease

To overcome the shortcomings of PD we also take into account how much exactly we expect to improve. This acquisition function is called the expected decrease (ED). In comparison to PD, a small difference $y_{min} - f(x)$ will likely yield a low acquisition.

$$\begin{aligned}\alpha_{ED}(x|D_n) :=& \mathbb{E}[\max\{y_{min} - f(x), 0\}|D_n] = \\ =& \Big(y_{min} - \mu(x|D_n)\Big)\Phi(Z(x|D_n)) + \sigma(x|D_n)\phi(Z(x|D_n))\end{aligned} \tag{4.7}$$

Where ϕ is the probability density function of $\mathcal{N}(0,1)$. When we observe how a change of $\mu(x|D_n)$ and $\sigma(x|D_n)$ affects the acquisition function, we notice that it behaves like LCB.

$$\frac{d}{d\mu}\ \alpha_{ED}(x|D_n) = -\Phi(Z(x|D_n)) < 0 \tag{4.8}$$

$$\frac{d}{d\sigma}\ \alpha_{ED}(x|D_n) = \phi(Z(x|D_n)) > 0 \tag{4.9}$$

Decreasing μ increases α_{ED} and increasing σ increases α_{ED}. The difference to LCB is that the changes to α_{ED} are not linear to the changes of μ and σ.

4.4 Update step

4.4.1 Acquisition maximization

With the acquisition function $\alpha(x|D_n)$ in place we want to find the input x that maximizes it. The problem cannot easily be solved analytically, instead, we evaluate the acquisition function at some chosen points and select the optimum from this subset. If we do not evaluate all possible points, we have at least to make sure that the parameter space is well covered. If we choose a grid that uses c points for each dimension, we get c^d points for d dimensions. Hence polynomial many in relation to c. We want many values within each individual dimension, but the total number of points should be manageable. Latin hypercube sampling (LHS) allows us to fulfill this, generating only c points in total. For each dimension, we choose c values. If the dimension is continuous, we can select evenly spaced points in the interval. For dimensions with finitely many values, we try to take each element equally often. If this does not add up, we choose the rest randomly. We want each element at least once so c should be chosen accordingly. This gives us d lists of c values. From each list, we take out one element at random and combine them into one point in the parameter space. We repeat this until the lists are empty, which will result in c parameter points. For each of these points, we calculate their acquisition and return the highest result as an estimated optimum.

4.4.2 Hyperparameter

The kernels we introduced in Section 3.4 led to new hyperparameters $\boldsymbol{\theta}$. We define $K_{\boldsymbol{\theta}}(\boldsymbol{x})$ as the covariance matrix using these hyperparemeters. Additionaly we simplify the mean value function $m(x)$ of our Gaussian process to a new hyperparameter c_m.

$$m(x) = c_m \tag{4.10}$$

Thus the output vector $\boldsymbol{y}$ given the input vector $\boldsymbol{x}$ has following a normal distribution.

$$\boldsymbol{y}|\boldsymbol{x} \sim \mathcal{N}(c_m\mathbf{1}, K_{\boldsymbol{\theta}}(x) + \sigma^2\boldsymbol{I}) \tag{4.11}$$

The corresponding probability density function can be obtained by inserting this into Equation 3.2.

$$p(\boldsymbol{y}|\boldsymbol{x}, c_m, \boldsymbol{\theta}) = \frac{1}{\sqrt{(2\pi)^n|K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I}|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - c_m\mathbf{1})^T(K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I})^{-1}(\boldsymbol{x} - c_m\mathbf{1})\right) \tag{4.12}$$

Assuming a constant prior to the hyperparameters, we can use Bayes' theorem to represent the posterior of the hyperparameters up to a constant factor of κ.

$$\begin{aligned} p(c_m, \boldsymbol{\theta}|D_n) =& \frac{p(\boldsymbol{y}|\boldsymbol{x}, c_m, \boldsymbol{\theta})p(c_m, \boldsymbol{\theta})}{p(D_n)} \\ =& \kappa\frac{1}{\sqrt{|K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I}|}} \exp\left(-\frac{1}{2}(\boldsymbol{x} - c_m\mathbf{1})^T(K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I})^{-1}(\boldsymbol{x} - c_m\mathbf{1})\right) \end{aligned} \tag{4.13}$$

Both the inverse and the determinant can be calculated using Cholesky decomposition. As the covariance matrix $K_{\boldsymbol{\theta}}(\boldsymbol{x})$ is positive semi-definite, $(K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I})$ is positive definite. Thus we can represent it with a lower triangular matrix L.

$$K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I} = LL^T \tag{4.14}$$

The inverse of a triangular matrix can be calculated easily. We also see that the eigenvalues of a triangular matrix equal to its diagonal elements. This can be used as the determinant of a matrix equals the product of its eigenvalues.

$$(K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2\boldsymbol{I})^{-1} = (L^{-1})^TL^{-1} \tag{4.15}$$

$$\sqrt{|K_{\boldsymbol{\theta}}(\boldsymbol{x}) + \sigma^2 \boldsymbol{I}|} = |L| = \prod_{i=1}^{n} L_{ii} \tag{4.16}$$

Now that we can calculate the probability that certain θ and c_m matches our evidence, we could maximize it. But then we would ignore all hyperparameters, which could have only a slightly lower probability. Instead, we use a set of hyperparameters Θ. If we want to calculate an acquisition α, we calculate them for all hyperparameters in Θ and sum them up weighted according to their respective probabilities.

$$\nu := \sum_{(c_m,\boldsymbol{\theta})\in\Theta} p(c_m, \boldsymbol{\theta}|D_n) \tag{4.17}$$

$$\alpha(x|D_n, \Theta) = \frac{1}{\nu} \sum_{(c_m,\boldsymbol{\theta})\in\Theta} p(c_m, \boldsymbol{\theta}|D_n) \cdot \alpha(x|D_n, c_m, \boldsymbol{\theta}) \tag{4.18}$$

The normalization factor ν cancels out the factor κ in Equation 4.13, so its value does not have to be determined. In summary, we can use a set Θ of hyperparameters instead of choosing a single one. The set can be constructed using LHS and each weight $p(c_m, \boldsymbol{\theta}|D_n)$ can be precalculated as it is independent of x. Since the overhead of each acquisition calculation is linear to the number of elements in the set Θ, one also may cut off the elements with weights significantly lower than the rest.

5 Performance tests

The following tests were performed on the CoolMUC-2 system of the LRZ Linux Cluster. Each node consists of 28-core Haswell nodes and has 64 GB RAM available.

5.1 Experiment

This work deals specifically with the application of the n-body problem. We want to calculate the pairwise interactions between n objects. The n-body problem is for example encountered in molecular dynamics (MD) as molecules apply different forces to each other depending on their distance. The different forces are often modeled using the Lennard-Jones potential which leads to the following formula to calculate the force between any two molecules.

$$F_{LJ} = \frac{24\epsilon}{r}\left(2\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right) \tag{5.1}$$

The variables ϵ and σ depend on the type of molecules we are examining and are therefore constant over time. The distance r between two molecules on the other hand generally changes over time. As the movement depends on the force acting on the molecules we have to solve a differential equation if we want to predict the exact position of the molecules at a given time. If the number of molecules is too high this approach is unfortunately analytically intractable because the number of interactions grows quadratically with the molecule count. That is why we resort to a numerical approach instead. If we consider only a small time step, we can approximate the changes in the system relatively well. To calculate the status of the system for any time we make incremental steps. Force, acceleration, speed, and position can be approximated for the small-time step and can this be repeated until the time steps add up to the specified time. This makes the calculation much simpler, but we still have the problem that n molecules lead to $n(n-1)$ interactions. There are several ways to deal with this problem, most of them are based on the fact that the majority of forces converge towards zero with increasing distance. We define a cutoff radius of r_c, for which we assume that molecules that are further apart than this radius do not exert forces on each other. With this, we can choose from a variety of sub-algorithm to calculate one timestep. These sub algorithms also often share the three hyperparameters cell size factor, data layout and Newton3. Some algorithms divide the observed region into cells with a side length of at least r_c. The cell size factor is multiplied by r_c beforehand to obtain smaller or larger cells. For the data layout, we can choose between an Array of Structures (AoS) or Structure of Arrays (SoA). This influences how the data of the molecules are arranged in the memory and consequently have a strong influence on SIMD operations. The Newton3 parameter refers to Newton's third law of motion which states that each force produces a counterforce of equal strength. Some algorithms can use this to halve the number of force calculations. We have implemented the Bayesian optimization algorithm in the C++ library AutoPas [Gra+19]. In AutoPas we can re-select the hyperparameters in each iteration step, which leads to the following parameter space.

Parameter	Type	Number of elements
Subalgorithm	Discrete	9
Cell size factor	Continuous	∞
Data Layout	Discrete	2
Newton3	Discrete	2

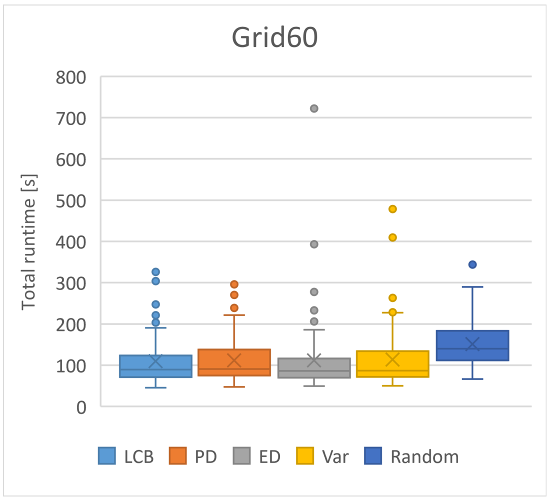
Table 5.1: Parameter space

The parameters may have a significant influence on the runtime but it is not possible to test all parameter combinations as the cell size factor is continuous. Our Bayesian approach, however, works under these conditions. The Gaussian process decides which parameters to select and we perform a couple of iterations with them. If we assume that the time per iteration for given parameters only changes negligibly in the short run, it can be estimated by measuring the time these runs need. We pass the time to the Gaussian process as evidence to update it and then repeat this process. As all these tuning steps are used for the simulation, our Bayesian optimization algorithm could be used until the simulation ends. But because the matrices in the Gaussian process become accordingly large, we stop after a given number of evidence and use the evidence, which resulted in the lowest runtime, for the next iterations. Since the particles move over time and the optimal parameter setting can change in the long run, we define a tuning interval. The interval indicates when a new tuning phase is started.

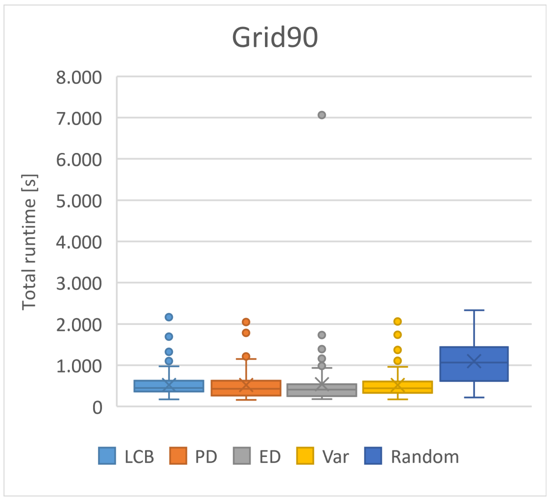
In our measurements, we compare different acquisition functions, but also the brute force method which tests the whole parameter space. As there are theoretically infinitely many points, we reduce the continuous variable to a few selected values. The idea between the comparisons is that the Bayesian optimization needs less tuning iterations, but only makes a possibly sub-optimal choice for the next iterations. The exhaustive search takes longer for tuning but should result in the best parameter choice. The expectation is that the choice of the Bayesian approach is only slightly inferior and therefore results in an overall lower runtime. The search space may also be so large that an exhaustive search might be spending all its time tuning. In these cases, this method is clearly unsuitable. Instead, we compare our method with an algorithm that randomly selects parameters. Like our method, the lowest runtime is selected from the evidence and its parameters are used until the next tuning interval.

5.2 Results

In the tests in Figure 5.1 a grid of particles was generated and in Figure 5.2 the positions were chosen using a normal distribution. The first is a very homogeneous scenario and the second is one where the particles are more concentrated in one point. The figures show the runtime of hundreds of sample runs using a box-and-whisker diagram. This visualization allows us to judge how the algorithms behave in most cases. In these simple scenarios, the difference between the acquisition functions is relatively small, but they all generally perform significantly better than the random approach. So our approach indeed makes good decisions, but how does it behave in a complicated scenario. Figure 5.3 shows the result of applying the algorithm on a scenario that simulates spinodal decompositions [Cah61]. In this experiment, a liquid is first equilibrated at a non-critical temperature. The particles should be homogeneously spread after this phase. Then the mixture is cooled down to a subcritical temperature, which leads to spots with a high concentration of particles and spots of low concentrations. This effect can only be seen on the microscopic level at first, but both spot types become larger over time. We focus only on the cooling phase where various particle densities are created in different areas over time. So at any point in time, other simulation-parameters may become the optimal ones. It is, therefore, more effective to set the parameters on the fly rather than committing to them in advance as the density of all regions for a given time is not trivial to predict. Our algorithm generally performed better than an exhaustive full search as seen in Figure 5.3. Lower confidence bound and expected decrease led to the lowest runtime on average.

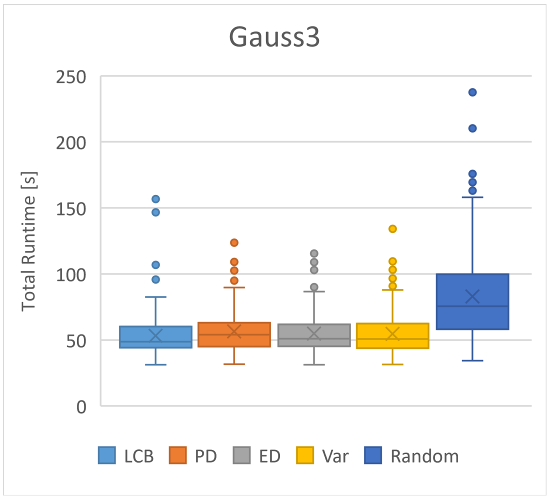


(a) Grid of 60x60x60 particles.

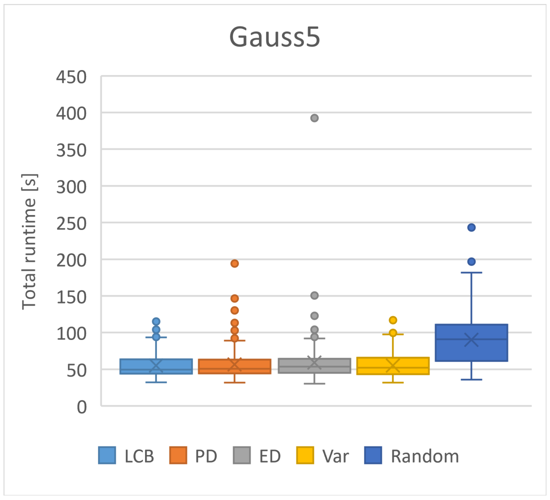


(b) Grid of 90x90x90 particles.

Figure 5.1: Total runtime box-and-whisker diagram of different acquistion function and a random parameter selection in seconds. Particles are aligned in a 3D-Grid. Each column consists of 100 samples.



(a) 150000 particles are normally distrbuted using a standard deviation of 3.0.



(b) 150000 particles are normally distrbuted using a standard deviation of 5.0.

Figure 5.2: Total runtime box-and-whisker diagram of different acquistion function and a random parameter selection in seconds. The positions of the particles are generated using a 3-variate normal distribution. Each column consists of 200 samples.

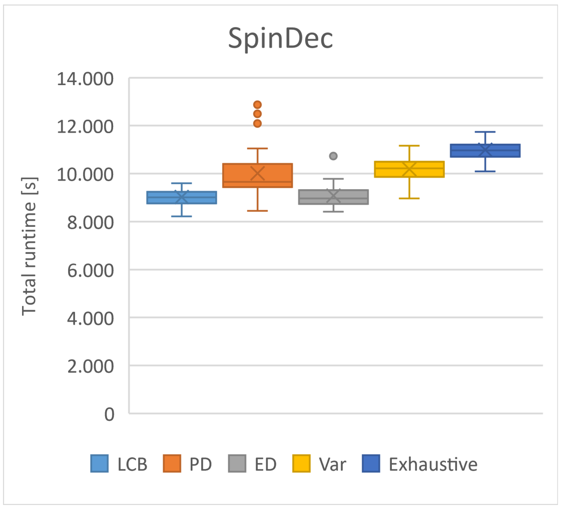


Figure 5.3: Runtime box-and-whisker diagram of different acquistion function and exhaustive search in seconds. The runtime consists of the time needed to calculate the forces in the cooling phase of a spinodal decomposition scenario. Each column consists of 40 samples.

6 Conclusion

Bayesian optimization can easily adapt to other optimization problems by adjusting the objective function. As an example, one can benchmark the performance per watt of a run and use this as evidence. Maximizing this score leads to a program run which tries to minimize the energy consumption of the used hardware. With the help of this method, the system "kukai" was the second most energy-efficient system of the top 500 supercomputers worldwide in June 2017 [MSS18]. Likewise, the Bayesian approach can be tailored to many other applications.

In summary, the Gaussian process model allows for regression of an arbitrary continuous function f. With the use of kernels we do not need any prior knowledge of the form of f, but a supply of evidence instead. Our Bayesian optimization algorithm generates this evidence by estimation of the informational gain through the use of an acquisition function. In our case, we use this algorithm to predict the runtime of a program for given input parameters. No expert knowledge of the program is required to select at least close-to-ideal input parameters. One mainly only has to choose one acquisition function and our algorithm finds good input parameters by performing a few test runs. Which acquisition function is a good choice depends on the given use case, but over several tested cases, LCB and ED were generally on par or superior to the other functions. Regardless of the choice, our method is generally better than brute force and random methods.

Bibliography

[Cah61] J. W. Cahn. "On spinodal decomposition." In: *Acta Metallurgica* 9.9 (1961), pp. 795–801. ISSN: 0001-6160. DOI: `https://doi.org/10.1016/0001-6160(61)90182-1`.

[CGM01] H. Chipman, E. I. George, and R. E. McCulloch. *The Practical Implementation of Bayesian Model Selection*. 2001.

[Do07] C. B. Do. *Gaussian processes*. 2007.

[Fra12] P. Frazier. "Tutorial: Optimization via simulation with Bayesian statistics and dynamic programming." In: *Proceedings of the 2012 Winter Simulation Conference (WSC)*. Dec. 2012, pp. 1–16. DOI: `10.1109/WSC.2012.6465237`.

[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`.

[Guo03] H. Guo. "A Bayesian Approach for Automatic Algorithm Selection." In: 2003.

[MSS18] T. Miyazaki, I. Sato, and N. Shimizu. "Bayesian Optimization of HPC Systems for Energy Efficiency." In: *ISC*. 2018.

[NOR18] F. M. Nyikosa, M. A. Osborne, and S. J. Roberts. "Bayesian Optimization for Dynamic Problems." In: 2018.

[Nyi18] F. M. Nyikosa. *Adaptive Bayesian Optimization for Dynamic Problems*. 2018.

[RW06] C. E. Rasmussen and C. K. I. Williams. *Gaussian Processes for Machine Learning*. 2006.

[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`.

[SN12] R. Suda and V. S. Nittoor. "Efficient Monte Carlo Optimiation with ATMathCoreLib." In: 2012.

[Vaa98] A. W. van der Vaart. “Asymptotic Statistics.” In: (1998).