Utrecht University

Department of Information and Computing Sciences

Tree-GP: A Scalable Bayesian Global Numerical Optimization algorithm

February 2015

Author
Gerben van Veenendaal
ICA-3470792

Supervisor
dr. ir. D. Thierens

Abstract

This paper presents the Tree-GP algorithm: a scalable Bayesian global numerical optimization algorithm. The algorithm focuses on optimizing evaluation functions that are very expensive to evaluate. It models the search space using a mixture model of Gaussian process regression models. This model is then used to find new evaluation points, using our new CMPVR acquisition criteria function that combines both the mean and variance of the predictions made by the model. Conventional Gaussian process based Bayesian optimization algorithms often do not scale well in the total amount of function evaluations. Tree-GP resolves this issue by using a mixture model of Gaussian process regression models stored in a vantage-point tree. This makes the algorithm almost linear in the total amount of function evaluations.

Contents

1 Introduction **3**
1.1 Motivation . . . 3
1.2 Goals . . . 4
1.3 Contributions . . . 4
1.4 Thesis outline . . . 5

2 Preliminaries **6**
2.1 Global numerical optimization . . . 6
2.1.1 Definition . . . 6
2.1.2 Domain . . . 6
2.2 Bayesian optimization . . . 7
2.2.1 Posterior distribution . . . 7
2.2.2 Initial sample . . . 7
2.2.3 Acquisition criteria function . . . 7
2.2.4 Algorithm . . . 9
2.3 Gaussian process regression . . . 10
2.3.1 Gaussian process . . . 10
2.3.2 Gaussian process regression . . . 10
2.3.3 Mean function . . . 11
2.3.4 Covariance functions . . . 11
2.3.5 Making predictions . . . 12
2.3.6 Maximum likelihood estimation . . . 12
2.4 Vantage-point tree . . . 13
2.4.1 Definition . . . 13
2.4.2 Representation . . . 13
2.4.3 k-nearest-neighbor queries . . . 14

3 Related Work **16**
3.1 Bayesian optimization . . . 16
3.2 Response surface fitting . . . 16
3.3 Radial basis functions . . . 16
3.4 Gaussian process regression . . . 17

4 Algorithm **19**
4.1 Initial sample . . . 19
4.2 Posterior distribution . . . 21
4.3 Maximum likelihood estimation . . . 22
4.4 Acquisition criteria function . . . 23

5 Scaling the algorithm **29**
5.1 Time complexity . . . 29
5.2 Mixture model . . . 30
5.2.1 Definition of local model . . . 30
5.2.2 Vantage-point tree . . . 30
5.2.3 Adding a point . . . 30
5.2.4 Making predictions . . . 31
5.2.5 Splitting a leaf node . . . 34
5.3 Algorithm . . . 37

6 Experiments **39**
6.1 Experiment setup . . . 39
6.2 Test suite . . . 40
6.3 Results . . . 44

7 Discussion **51**
7.1 GP performance . . . 51
7.2 Tree-GP performance . . . 52
7.3 AMaLGaM performance . . . 53

8 Conclusions and Future Work **55**
8.1 Exploration versus exploitation . . . 55
8.2 Scalability . . . 55
8.3 Future work . . . 56

9 Acknowledgments **57**

Appendices **58**

A Result values **59**

Chapter 1

Introduction

1.1 Motivation

Much research has already been done in the field of global numerical optimization. That is because global optimization plays a major role in applied science. A lot of problems can be reduced to finding the optimum of some function. Examples are finding the best parameters of an algorithm, doing a simulation with many free variables, or maximizing profit. The problem definition of global numerical optimization seems so simple: finding the arguments of a function that gives its minimal value. If this function, which is typically called the fitness or loss function, is cheap to evaluate and low-dimensional, we can evaluate lots of points in the search space. Evolutionary Algorithms (EA) are typically very suited for this. Evolutionary algorithms maintain a fixed-size population of the points with the best function values that have been found. This way, an evolutionary algorithm can focus its search on regions with low values. Mutation and recombination operators are used to evolve the population and create new evaluation points. Selection is then used to determine the new population.

Simple evolutionary algorithms quickly take too much time if the function is expensive to evaluate or its dimensionality is high. A possible solution for this is to simply evaluate less points. Because we still want a good end result, this means that the algorithm needs to put in more effort to determine the evaluation points. Estimation of Distribution Algorithms (EDA) are evolutionary algorithms that do exactly that by modeling the population with a density probability distribution [3]. The algorithm can then sample from this probability distribution to find points in the search space that characterize good points. By modeling the best points found, the algorithm can take more intelligent decisions about where to evaluate next. This in turn means that less evaluations are required to get a similar result of a naive evolutionary algorithm.

Another approach to modeling is called Bayesian optimization [12]. Bayesian optimization tries to find the next evaluation point by completely modeling the search space. It does this by putting a prior function distribution over the function. By conditioning this distribution on the points that have been found before we can predict values for other points. With these predictions we can then determine where to evaluate next. In this thesis, we will use this Bayesian approach to global optimization. Our focus will be on functions that are expensive to evaluate, which we will define to be from one minute up to several hours. If the function takes a minute, we can only do 1440 evaluations per day, or 10,080 per week. If it takes an hour, we can only do 24 per day or 168 per week. Because we can only evaluate the function such a small number of times, we really have to put in some effort to find potentially good points to evaluate. This is exactly what Bayesian optimization is good at.

1.2 Goals

In global optimization an algorithm has to make a consideration between exploration and exploitation. Exploration is finding new interesting regions of the search space, whereas exploitation is using the currently known interesting regions to find a local optimum. If an algorithm explores too much, but does not exploit, it does not find a good final point. On the other hand, if it exploits too early, it will most likely find a mediocre local minimum of the function. Our first goal is therefore to construct an algorithm that like simulated annealing first focuses on exploration and gradually exploits its knowledge of the search space.

A problem of current Bayesian optimization algorithms is that they do not scale very well with the number of function evaluations. Some of these algorithms require a matrix inversion per function evaluation, which is of order $\mathbb{O}(n^3)$, where n is the amount of function evaluations so far. Even though we target expensive functions and so the time spent is relative, this quickly becomes a bottleneck, especially if the function can be evaluated in the order of a minute to several minutes. Our second goal is therefore to construct an algorithm that does scale well with the total number of function evaluations.

A lot of real-life functions have a lot of parameters. These parameters often interact in a non-linear way to determine the function value. So the more parameters there are, the harder the function is to optimize, because we can not optimize the parameters independently. We will test functions with a dimensionality up to 30 that simulate real-life optimization problems. We will then compare the performance of our algorithm of these functions with an estimation of distribution algorithm named AMALGAM [3]. In addition to the performance of the algorithm, we also want to show the overhead in time of the algorithm. The overhead of the algorithm is the time spent to find new evaluation points relative to the amount of time the function takes to evaluate. We will again compare the results with AMALGAM.

1.3 Contributions

We have created a Bayesian optimization algorithm called TREE-GP that performs very well on expensive functions and scales almost linearly in the amount of evaluations. We use a regression technique called Gaussian process regression to put a posterior distribution over the function. To make the algorithm scalable, we use a mixture model of multiple Gaussian process regression models. Each individual model only models a local part of the search space. The models of the mixture model are put in a data structure called a vantage-point tree to make nearest-neighbor queries fast. Evaluated function points are only put in models close to it so only a few small models need to be updated. Likewise, when making predictions in the search space, only models close to the query point are used. The predictions of the different models are weighted according to the distances of the models to the predicted point.

Gaussian process regression predicts a Gaussian distribution for each point in space. The mean and standard deviation of this distribution are combined in order to make the consideration between exploration and exploitation. We have constructed an auxiliary function called the Cumulative Mean Probability to Variance Ratio (CMPVR) function that weights this mean and standard deviation of each point in space. This function is minimized in order to find the best point to evaluate next. At the start of the algorithm the weighting is chosen to favor high standard deviations and thus exploration. Slowly low mean values are given more weight so the algorithm will gear towards exploitation. If no improvement has been found in a while, the weighting will be reset to make the algorithm start exploring again.

1.4 Thesis outline

We will start with some preliminaries: chapter two contains theory that is not part of our research but needs to be understood to follow this thesis. This chapter will contain a formal definition of global numerical optimization, explain Bayesian optimization, Gaussian process regression and the vantage-point tree that we will be using in our algorithm. In chapter three we will delve into related work of other researchers. After that, in chapter four, we will present the details of our Tree-GP algorithm. In chapter five we will make the algorithm scalable. In chapter six we will explain our experiments to compare Tree-GP with the AMaLGaM algorithm and present the results. We will then discuss these results in chapter seven. Finally we have some conclusions and ideas for future work in chapter eight. Chapter nine contains some acknowledgments.

Chapter 2

Preliminaries

In this chapter we will explain existing theory, techniques and algorithms that are needed to understand our TREE-GP algorithm. This allows the reader to skip techniques that he or she already knows.

2.1 Global numerical optimization

2.1.1 Definition

Let us first give a formal definition of the global numerical optimization problem that we want to solve. Given a black box loss function $f : \mathbb{R}^D \mapsto \mathbb{R}$, we want to find

$$\begin{aligned} \mathbf{x}^* = \underset{\mathbf{x}}{\operatorname{argmin}} \quad & f(\mathbf{x}), \\ \text{subject to} \quad & \mathbf{x}^* \in [0,1]^D. \end{aligned}$$

A loss function is basically the opposite of a fitness function. Whereas a fitness function returns the fitness of certain parameters, and thus needs to be maximized, a loss function returns the loss that we obtain for a certain set of input parameters. So fitness functions are supported with this definition by simply negating the function value. We want to minimize because this makes the problem statement equivalent to those found in other literature.

2.1.2 Domain

We restrict the arguments of the loss function f to domain $[0,1]$ for our ease. If the black box function has a different finite domain, a linear mapping to $[0,1]$ can easily be made. If the function has an infinite domain, a mapping can also be made. Given a loss function $g(\mathbf{z})$ with $z_i \in [-\infty, \infty]$, the loss function f becomes

$$f(\mathbf{x}) = g(\mathbf{z}), \quad z_i = \frac{1}{3}\alpha_i \tan((x_i - \frac{1}{2})\ \pi),$$

where $\alpha_i \in \mathbb{R}^+$ is the typical range of the z_i. The range $[-\alpha_i, \alpha_i]$ of z_i is approximately mapped to $[0.1, 0.9]$ of x_i. See figure 2.1 for some example mapping for different values of α_i. Similar mappings can be made for loss functions that have only an infinite lower or upper bound, or have different types of bounds per argument. These mappings can be considered part of the black box loss function, which is why we can assume the domain to be $[0,1]$.

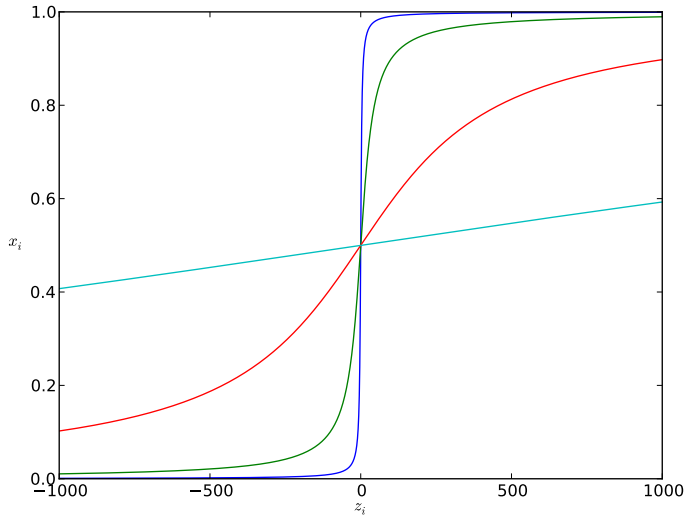


Figure 2.1: Mappings from z_i to $x_i \in [0, 1]$ for different values of α_i. The blue mapping has $\alpha_i = 10$, the green one $\alpha_i = 100$, red has $\alpha_i = 1000$ and light blue has $\alpha_i = 10000$.

2.2 Bayesian optimization

2.2.1 Posterior distribution

Like we said in the introduction, we will use an optimization technique called Bayesian optimization [12]. Bayesian optimization puts a prior function distribution over the loss function f. This prior function distribution is conditioned on the values found by evaluations of the loss function. We then obtain the posterior function distribution which we use to find the point of the next loss function evaluation. When we evaluate the loss function to find a new value at that particular point in space, we update the posterior distribution. This process is repeated until we have either reached an acceptable value or we have run out of evaluations when we give the algorithm a maximum number of loss function evaluations.

2.2.2 Initial sample

Before we can create our initial posterior distribution, we first need an initial sample of the loss function. This sample can be obtained in a number of ways, we might for example simply select random points. The size of this sample will depend on the way of putting a posterior distribution over f. There might for example be a minimum amount of points required for a model to be fitted. Once we have these points, we can construct the posterior distribution to bootstrap the algorithm.

2.2.3 Acquisition criteria function

Determining which point to evaluate given a posterior distribution over the search space is no easy task. By choosing this next evaluation point intelligently, we hope to converge faster to the minimum, so that less evaluations are required. The selected point has to satisfy certain criteria.

These criteria can be made explicit using an auxiliary function $h(\mathbf{x})$ called the acquisition criteria function [6]. Minimizing this function determines where to evaluate next given the posterior distribution. The point that we are going to evaluate at iteration i is given by

$$\begin{aligned} \mathbf{x}_i = \underset{\mathbf{x}}{\operatorname{argmin}} \quad & h(\mathbf{x} \mid p_{i-1}, i, N), \\ \text{subject to} \quad & \mathbf{x}_i \in [0,1]^D. \end{aligned}$$

Here p_{i-1} denotes the posterior distribution calculated at the previous iteration and N denotes the total number of evaluations. The previous posterior distribution p_{i-1}, the current iteration i and the total number of evaluations N is the only information needed by an acquisition criteria function to determine the next evaluation point. The values of i and N can for example be used to let the evaluated point depend on how many evaluations are still left. A simple example of an acquisition criteria function is the predicted value of the posterior distribution. This will then be minimized and the algorithm will evaluate the predicted minimum of the posterior distribution. The function h has to be differentiable because we will be using a gradient-based optimization algorithm to minimize it. It might not be very important to find the exact minimum of the function, as long as a good evaluation point can be obtained. This is a tradeoff between time spent minimizing the acquisition criteria function and the quality of the evaluation point according to the acquisition criteria function.

2.2.4 Algorithm

Algorithm 1 describes the full Bayesian optimization algorithm.

Algorithm 1 Bayesian optimization

```
procedure BayesianOptimization(f, N, M)
    Let f be the function to be optimized.
    Let N be the number of function evaluations.
    Let M be the number of initial samples.
    Let D be the dimensionality of f.
    Let h be the acquisition criteria function.
    Let i be the current iteration.
    Let X be the set of all evaluated points, X = {x_1, x_2, x_3, ...}.
    Let x_i be the position vector of the evaluation at iteration i.
    Let T be the set of the values of all evaluated points, T = {t_1, t_2, t_3, ...}.
    Let t_i be the value of f at x_i.
    Let t* be the minimum value found so far.
    Let x* be the position vector of the value t*.
    Let p_i be the posterior distribution over f at iteration i.

    (X, T) ← GetInitialSample(f, M)

    t* ← min(T)
    x* ← argmin_{x_i ∈ X} t_i

    i ← M + 1

    p_i ← CreateInitialPosterior(X, T)

    while i ≤ N do
        x_i ← argmin_{x ∈ [0,1]^D} h(x | p_{i-1}, i, N)        ▷ Minimize acquisition criteria function.

        t_i ← f(x_i)                                           ▷ Evaluate function.

        if t_i < t* then
            t* ← t_i
            x* ← x_i
        end if

        p_i ← UpdatePosterior(p_{i-1}, x_i, t_i)

        i ← i + 1
    end while

    return x*
end procedure
```

2.3 Gaussian process regression

2.3.1 Gaussian process

Our algorithm puts a prior distribution over the loss function by modeling it with Gaussian process regression. Gaussian process regression is a powerful regression technique that models the loss function with a Gaussian process. A Gaussian process is formally defined as a probability distribution over functions $y(\mathbf{x})$ which, when evaluated at n D-dimensional points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^D$, together give a joint dimensional multivariate Gaussian distribution [18]. A Gaussian process $y(\mathbf{x})$ is completely specified by a mean function $m : \mathbb{R}^D \mapsto \mathbb{R}$ and a covariance function $k : \mathbb{R}^D \times \mathbb{R}^D \mapsto \mathbb{R}$. These functions are defined over all functions $y(\mathbf{x})$ drawn from the Gaussian process as

$$\begin{aligned} m(\mathbf{x}) &= \mathbb{E}[y(\mathbf{x})], \\ k(\mathbf{x}, \mathbf{x}') &= \mathbb{E}[(y(\mathbf{x}) - m(\mathbf{x})) \times (y(\mathbf{x}') - m(\mathbf{x}'))]. \end{aligned}$$

That is, the expected value over all functions $y(\mathbf{x})$ drawn from the process at a certain $\mathbf{x}$ and the expected covariance between all function values $y(\mathbf{x})$ and $y(\mathbf{x}')$. Together we can write this as

$$y(\mathbf{x}) \sim \mathcal{GP}(m(\mathbf{x}), k(\mathbf{x}, \mathbf{x}')).$$

In sections 2.3.3 and 2.3.4 we will explain the meaning of these two functions.

2.3.2 Gaussian process regression

Gaussian process regression uses a posterior Gaussian process that has been conditioned on n D-dimensional input points $\mathbf{x}_1, \ldots, \mathbf{x}_n \in \mathbb{R}^D$ with their corresponding values $\mathbf{t} = \{t_1, \ldots, t_n \in \mathbb{R}\}$. Here conditioning means that all functions drawn from the Gaussian process have to go through the values $\mathbf{t}$ at their corresponding input points. This posterior Gaussian process is a model of the loss function based on points that have already been evaluated. From this posterior Gaussian process we can make a prediction for a new point which gives us a Gaussian distribution. The predicted Gaussian distribution at a point $\mathbf{x}$ is the probability distribution over all values of functions $y(\mathbf{x})$ drawn from the posterior Gaussian process. This is done by adding another input point to the conditioned Gaussian process and calculating the values that it can take on. This turns out to be a Gaussian distribution of values. Because the prediction is a Gaussian distribution, we have a measure of uncertainty, namely its standard deviation. This is a big advantage of Gaussian process regression.

See figure 2.2 for an example of three samples drawn from a prior Gaussian process and three samples drawn from a Gaussian process conditioned on some input points. Each sample is function $y(\mathbf{x})$ drawn from a Gaussian process. In the conditioned Gaussian process these samples have to go through the input points. The predicted values are also plotted for every $\mathbf{x}$. As these are Gaussian distributions, both their mean value and twice their standard deviation have been plotted. The mean value is indicated by the dotted line, and the gray area covers twice the standard deviation of the prediction. Note the assumed zero mean in the prior. Also note that the uncertainty that we have of the predicted values has been modeled with the standard deviation of the Gaussian process. The standard deviation becomes larger the more we move away from input points: the uncertainty increases. Also note how smooth the samples drawn from the Gaussian process are. This is determined by the covariance function, which we will explain later on.

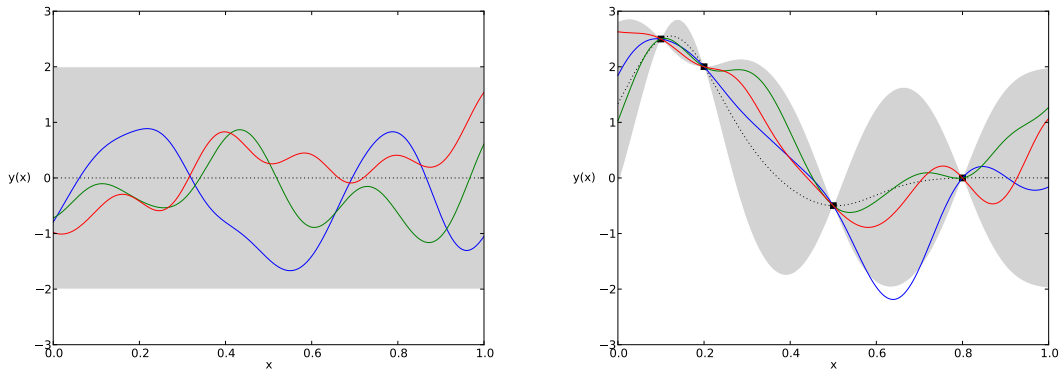


Figure 2.2: Three samples drawn from a prior Gaussian process on the left and a posterior Gaussian process on the right. Note that for each x the prediction is a Gaussian distribution. The dotted line indicates its mean and the gray area covers twice its standard deviation.

2.3.3 Mean function

Like we said before, a Gaussian process is completely defined by a mean and a covariance function. We will assume the mean function to be zero, that is $m(\mathbf{x}) = 0$. This assumption means that we believe the mean over all functions drawn from the Gaussian process to be zero. This in turn means that we believe the mean of the modeled function to be zero. As this will in general not be the case, we subtract the average of the values $\mathbf{t}$ from every value t_i. This process is called centering the data. The average is added again to the predictions made by the model. This makes it so that we believe the mean of the modeled function is the average of the input data, which is a reasonable assumption to make.

2.3.4 Covariance functions

The only thing left to determine is the covariance function. The covariance function $k(\mathbf{x}, \mathbf{x}')$ is a measure of how similar the function value of two points $\mathbf{x}$ and $\mathbf{x}'$ are. This function is also referred to as the kernel function. It describes our prior beliefs of the function that we are modeling. It determines whether samples drawn from the Gaussian process are smooth, linear or periodic. A number of covariance functions have been suggested for Gaussian process regression. Popular covariance functions include:

$$k(\mathbf{x}, \mathbf{x}') = \mathbf{x}^T \mathbf{x}' \qquad \text{(linear)}$$

$$k(\mathbf{x}, \mathbf{x}') = \|\mathbf{x} - \mathbf{x}'\|_2^2 \ \ln \|\mathbf{x} - \mathbf{x}'\|_2 \qquad \text{(thin plate spline)}$$

$$k(\mathbf{x}, \mathbf{x}') = \theta_0 \ \exp(-\sum_{d=1}^{D} \theta_d (x_d - x'_d)^2) \qquad \text{(squared exponential)}$$

Covariance functions that are a function of $\mathbf{x} - \mathbf{x}'$ are called stationary and are invariant to translations of the input space. The linear covariance function is not stationary and by using it we belief that the covariance of two points depend on their absolute positions, not on their relative ones. Using this kernel is the same as doing Bayesian linear regression which can be done much

more efficiently than Gaussian process regression. However, it is commonly combined with other covariance functions. The thin plate spline kernel results in a D-dimensional spline. A spline is a smooth piecewise defined polynomial. It has the advantage that it has no hyper-parameters, but this may result in a poorer fit. The squared exponential can be seen as the default covariance function for Gaussian process regression and works well in practice. It is stationary and functions drawn from a Gaussian process with this covariance function are smooth and continuous. It has $D + 1$ hyper-parameters: a scaling parameter θ_0 and D parameters that give each dimension a weight. The hyper-parameters make the function more generic so that it fits many forms of covariances between input points.

2.3.5 Making predictions

We can use the conditioned Gaussian process to make predictions for new points in space. Here we will give the equations to do this, but not how we came to these results. The results described here are explained in detail in [18], and therefore we refer the reader to this very well-written book for an in-depth explanation.

First we define the covariance matrix $\mathbf{C}$ to be an $n \times n$ matrix with the covariances between all points,

$$C_{i,j} = k(\mathbf{x}_i, \mathbf{x}_j).$$

Let $\mathbf{x}$ be the point for which we want to make a prediction. Furthermore, let $\mathbf{k}$ be a vector of covariances of the prediction point with all other points,

$$k_i = k(\mathbf{x}_i, \mathbf{x}), 1 \leq i \leq n.$$

Finally let c be the covariance of the predicted point with itself, so $c = k(\mathbf{x}, \mathbf{x})$. Then the mean and variance of the predicted Gaussian distribution at $\mathbf{x}$ are given by

$$\mu(\mathbf{x}) = \mathbf{k}^T \mathbf{C}^{-1} \mathbf{t},$$
$$\sigma^2(\mathbf{x}) = c - \mathbf{k}^T \mathbf{C}^{-1} \mathbf{k}.$$

2.3.6 Maximum likelihood estimation

We have already seen that covariance function can have hyper-parameters. These parameters have to be determined in order to provide the best fitting model. In order to find these hyper-parameters we use maximum likelihood estimation. Maximum likelihood estimation gives us the hyper-parameters that make the data most likely. This can be done by maximizing the likelihood function of the data given the parameters to be found. Often, the log-likelihood function is used because it is easier to calculate and maximizing it is equivalent to maximizing the likelihood function. The log-likelihood function of a Gaussian Process regression model is given by the standard log-likelihood function of multivariate Gaussian distributions, which is given by

$$\ln p(\mathbf{t}|\theta) = -\frac{1}{2} \ln |\mathbf{C}| - \frac{1}{2} \mathbf{t}^T \mathbf{C}^{-1} \mathbf{t} - \frac{N}{2} \ln(2\pi).$$

Maximizing this function in θ gives us our hyper-parameters. This optimization problem is aided by the fact that we can compute the partial derivative of the function in each hyper-parameter θ_i,

$$\frac{\partial}{\partial \theta_i} \ln p(\mathbf{t}|\theta) = -\frac{1}{2} \operatorname{tr}\left(\mathbf{C}^{-1} \frac{\partial \mathbf{C}}{\partial \theta_i}\right) + \frac{1}{2} \mathbf{t}^T \mathbf{C}^{-1} \frac{\partial \mathbf{C}}{\partial \theta_i} \mathbf{C}^{-1} \mathbf{t}.$$

Here tr is the trace operator which is the sum of the diagonal of a square matrix. These partial derivatives make it possible to use a derivative-based optimization algorithm, like gradient descent. A more advanced algorithm such as L-BFGS can be used to find the maximum a lot quicker than the basic gradient descent algorithm.

2.4 Vantage-point tree

2.4.1 Definition

Our algorithm uses a mixture model of multiple Gaussian process regression models. Each of these models are put into a vantage-point tree. A vantage-point tree is a data partitioning structure to speed up nearest-neighbor queries [19]. A data partitioning structure makes a partitioning of the data itself, in contrast to a space partitioning algorithm like the k-d tree which partitions all of space. This makes data partitioning algorithms often more efficient than space partitioning structures when the data is clustered. The vantage-point tree is a particular data partitioning structure that has been tailored for doing nearest-neighbor queries. A property that makes the vantage-point tree stand out is that it only needs to evaluate the distance metric on the input points themselves. Any distance metric can be used to measure distance between input points. The distance between points is also the only thing that it needs to know in order to operate. This can be a very useful feature if a distance can only be calculated between points, and new points cannot easily be constructed from other input points.

The vantage-point tree partitions data points by selecting a point from the data, the vantage-point. This can simply be done at random, but we will be using a more elaborate method that will be explained later on. The vantage-point is then removed from the list of input points, and the algorithm goes on by partitioning the rest of the points. It does this by calculating all distances to the selected vantage-point with the given distance metric. The median distance is then selected from these distances. The input points are split according to this median distance. If the distance of a point is smaller or equal than the median distance, it goes in the left child node. Likewise, if it is larger it goes into the right child node. After creating the root tree node with this method, the partitioning process is repeated for every node of the tree that still contain more than a certain maximum number of points.

2.4.2 Representation

Each node can be seen as a hypersphere if the data are points from $\mathbb{R}^D$ and the distance metric is the Euclidean distance, L_2. The left child node contains everything that falls into the hypersphere, whereas the right child node contains all the remaining nodes. See figure 2.3 for an example of a partitioning of points in the two-dimensional plane.

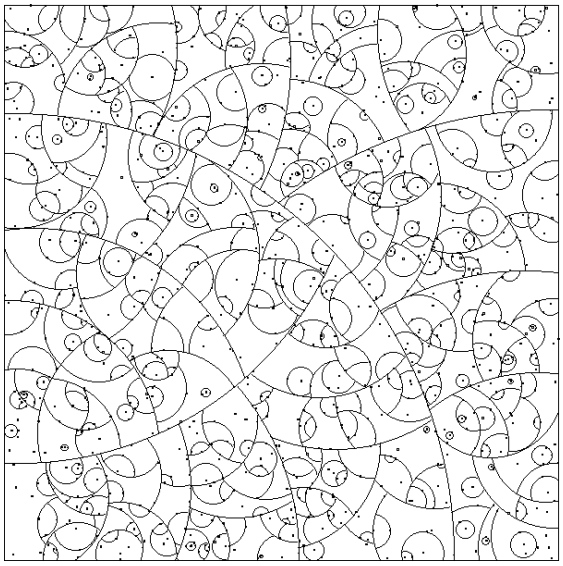


Figure 2.3: An example of a vantage-point tree of points in $\mathbb{R}^2$. Figure taken from [19].

Internal tree nodes store a reference to a vantage point and the minimum, median and maximum distance of their child nodes. Leaf nodes store a list of references of its points. The total storage is thus $\mathbb{O}(n)$ where n is the total amount of data points. The selection function for the vantage-point can be chosen to simply select a vantage point at random, in which case construction time is $\mathbb{O}(n \log n)$. This is the case for any vantage-point selection function that is at most linear.

2.4.3 k-nearest-neighbor queries

Ultimately the data structure is used for k-nearest-neighbor queries. These queries are in practice very fast as the tree is fully balanced. The k-nearest-neighbor algorithm can make deductions about which branches of the tree to visit by using the triangle-inequality property of the distance metric. It makes use of a fixed size max-heap to store the current k nearest points by their distance to a query point. It keeps track of the maximum distance in the heap, which is its first element if the heap is full, or ∞ if there are less than k points in the heap. For internal nodes we can use this maximum distance to determine whether a point in the left or right node can possibly contain a point closer to the query point than any of the points in the max-heap. If so, the algorithm recurses into the child node. The vantage point is pushed on the heap if its distance is smaller than the maximum distance. If a leaf node is reached, the algorithm walks over its points and pushes the points whose distance are smaller than the maximum distance. An implementation of the algorithm is given by algorithm 2.

Algorithm 2 Vantage-point tree k-nearest-neighbor querying

```
procedure VPTreeKNearestNeighbor(tree, x, k)
    heap ← CreateFixedSizeMaxHeap(k)

    VPTreeKNearestNeighborInNode(tree.root, x, heap, ∞)

    return FixedSizeMaxHeapToSet(heap)
end procedure

procedure VPTreeKNearestNeighborInNode(node, x, heap, d_max)
    if node.isLeaf then
        for p ∈ node.points do
            d ← ‖p − x‖                                          ▷ Calculate distance.

            if d < d_max then
                FixedSizeMapHeapInsert(heap, d, (node, p))

                d_max ← FixedSizeMaxHeapGetMaxKey(heap)
                                    ▷ Returns ∞ if the heap contains less than k items.
            end if
        end for
    else
        d ← ‖node.vantagePoint − x‖                              ▷ Calculate distance.

        if d < node.d_med then
            d_max ← VPTreeKNearestNeighborInNode(node.left, x, heap, d_max)

            if node.d_med − d < d_max then
                d_max ← VPTreeKNearestNeighborInNode(node.right, x, heap, d_max)
            end if
        else
            if d − node.d_max < d_max then
                d_max ← VPTreeKNearestNeighborInNode(node.left, x, heap, d_max)
            end if

            if node.d_med − d < d_max then
                d_max ← VPTreeKNearestNeighborInNode(node.right, x, heap, d_max)
            end if
        end if
    end if

    return d_max
end procedure
```

Chapter 3

Related Work

This chapter contains previous work made by other researchers related to our research.

3.1 Bayesian optimization

Bayesian optimization started with work from Mockus in the 1970s en 1980s [12]. After that, many ways of putting a prior distribution on the loss function have been suggested. The easiest way of putting a prior on the function is by fitting low-order polynomials on the function. Early work that describes how to do this include work of Box et al. [4], Khuri et al. [10] and Myers et al. [11]. Work by Powell expands on this by using a linear model in 1994 [13] and quadratic polynomials in 2002 and 2003 [14] [15] with a trust region. A trust region is the region of the search space that is used to fit the polynomial. If the fit is good, the trust region can be expanded. If the polynomial does not fit the function well, the trust region is contracted. Ultimately, the model is used to approximate the derivative of the function which is used to obtain the next optimization step.

3.2 Response surface fitting

Fitting low-order polynomials is an example of (response) surface fitting: using regression to create a surface that predicts values of other points in the search space [8] [17]. Response surfaces can be divided into non-interpolating surfaces, where some function is fitted onto the points but it is not required to pass through those points, and interpolating surfaces where the surface goes through each point [8]. A quadratic model is thus interpolating if it has been fitted on at most three points, but may be non-interpolating if more points are used. Jones shows that non-interpolating surfaces are unreliable because they do not sufficiently capture the shape of the loss function [8]. Instead, he advises to use interpolating methods based on basis functions such as radial basis functions or Gaussian process regression.

3.3 Radial basis functions

Radial basis functions have been used by Ishikawa et al. [7], Björkman et al. [2], Guttman [5] and Regis et al. [16]. Radial basis functions are real functions whose only parameter is a distance from a center point. So if $\mathbf{c}$ is a center point, a radial basis function is defined as

$$\phi(\mathbf{x}, \mathbf{c}) = \phi(\|\mathbf{x} - \mathbf{c}\|).$$

The distance norm is usually taken to be the Euclidean distance. An approximation of the landscape of n input points is created by taking a linear combination of n radial basis functions, one for each input point which is its center:

$$y(\mathbf{x}^*) = \sum_{i=1}^{n} w_i \, \phi(\|\mathbf{x}^* - \mathbf{x}_i\|).$$

Such a fit can easily be made by using linear least squares fitting, as only the weights have to be determined.

3.4 Gaussian process regression

Lately Gaussian process regression has been used a lot for surface fitting. Jones uses Gaussian process regression with the squared exponential covariance function for his DACE algorithm [9]. He proposes the Expected Improvement (EI) acquisition criteria function for Bayesian optimization:

$$\begin{aligned} h_{ei}(\mathbf{x}) &= \mathbb{E}[\max(t^* - \mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})), 0)], \\ h_{ei}(\mathbf{x}) &= (t^* - \mu(\mathbf{x})) \times \Phi\left(\frac{t^* - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right) + \sigma(\mathbf{x}) \times \phi\left(\frac{t^* - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right). \end{aligned}$$

Here, $\mathcal{N}$ is the normal distribution, ϕ denotes the standard normal distribution, and Φ denotes the standard normal cumulative distribution. Also note that the standard deviation $\sigma(\mathbf{x})$ is used, not the variance $\sigma^2(\mathbf{x})$.

This acquisition criteria function maximizes expected improvement, that is, it finds the point of the model that is expected to give the most improvement on the current best found value t^*. This acquisition function takes advantage of the predicted variance of the predicted distribution by incorporating the standard deviation. The function however has the problem that it samples a lot around the current best point before sampling more globally [8]. Because of this, it might take the algorithm a long time to converge to the global optimum.

Jones gives an overview of some more acquisition criteria functions in his 2001 taxonomy paper [8]. The first acquisition criteria function that he looks at is the Statistical Lower Bound (SLB) function, also called the Upper Confidence Bound (UCB) [6]. For a minimization problem, it is defined as

$$h_{slb}(\mathbf{x}) = \mu(\mathbf{x}) - \kappa\sigma(\mathbf{x}),$$

where κ is an adjustable parameter. This function tries to minimize the lower bound of the predicted value. It does this by subtracting the standard deviation from the predicted mean κ times. Jones takes $\kappa = 5$. This function has the problem that it quickly discards parts of the search space even if they contain the global optimum. This is caused by the Gaussian process regression, which generally underestimates the variance. This might give the global optimum a higher statistical lower bound value than a local minimum. If this is the case, the algorithm will find the local minimum instead of the global optimum. Therefore, it is important not to be dependent on the scale of the predicted variance.

The second acquisition criteria function that he gives is the Probability of Improvement (PI) function. This function tries to maximize the probability of improvement on the current best

found value t^*. It is defined as the probability that the predicted value at a point is smaller or equal to a certain target value. This target is chosen to be smaller than the current best value t^*. Jones initially takes the target to be $t^* - 0.25|t^*|$, based on the idea that he wants to find a 25% improvement on the current fitness value.

The full definition of the PI acquisition criteria function is

$$h_{pi}(\mathbf{x}) = P(\mathcal{N}(\mu(\mathbf{x}), \sigma^2(\mathbf{x})) \leq t^* - 0.25|t^*|)$$
$$h_{pi}(\mathbf{x}) = \Phi\left(\frac{t^* - 0.25|t^*| - \mu(\mathbf{x})}{\sigma(\mathbf{x})}\right),$$

where $\mathcal{N}$ is the normal distribution and Φ denotes the standard normal cumulative distribution. Because Φ is a strictly increasing function, using

$$h_{pi}(\mathbf{x}) = \frac{t^* - 0.25|t^*| - \mu(\mathbf{x})}{\sigma(\mathbf{x})}$$

will give the same result. Note that we need to maximize this function. For minimization we can use

$$h_{pi}(\mathbf{x}) = \frac{\mu(\mathbf{x}) - (t^* - 0.25|t^*|)}{\sigma(\mathbf{x})}.$$

This acquisition criteria function function has two problems. The first one is that we have a constant that we need to guess, the percentage of improvement wanted. It turns out that the PI function is very sensitive to this constant. Jones solves this by trying out different values for this constant and clustering the resulting points. He then evaluates all of the found points. This obviously involves a lot more work than using a single acquisition criteria function. Moreover, Jones suggests using a very inefficient clustering algorithm that takes $\mathbb{O}(n^4)$ time in the amount of points. Though this problem could be solved more efficiently, there is an even bigger problem with this acquisition criteria function. The problem is that the PI function is not translation independent in the fitness function values. If we would add a constant value to our fitness function value, the improvement wanted would increase, even though the scale of our fitness function remains the same. Also, if the fitness function value goes towards zero, the wanted improvement will decrease. At zero the wanted improvement will be zero and below zero the wanted improvement will increase again. This is obviously strange behavior which Jones has probably not noticed because he only tests on positive functions. Indeed, when using a positive function the improvement gradually decreases, which is exactly the wanted behavior for such a function. Therefore, it is important that the acquisition criteria function is independent of the scale and translation of fitness function values.

The inherent problem of using Gaussian process regression is the time taken to calculate the model. For Gaussian process regression this is $\mathbb{O}(n^3)$ in the amount of evaluation points. The model is calculated at every evaluation, at which n is increased by one. This will make the total algorithm use at least $\mathbb{O}(N^4)$ time, where N is the total amount of evaluation points. This makes the algorithms using plain Gaussian process regression not scalable in the total amount of function evaluations.

Chapter 4

Algorithm

In this chapter we will explain the bases of our Tree-GP algorithm. First we will explain how we do the initial sampling. Then we give the details of our posterior distribution and how we find its hyper-parameters. Finally we explain our acquisition criteria function.

4.1 Initial sample

In order to bootstrap Tree-GP, it first needs an initial sample of the loss function f. There are various ways to get such an initial sample. One way of doing this is simply selecting uniformly distributed random points in the search space and evaluating the function at those points. This is known as random sampling. The disadvantage of this approach is however that points may be clustered in space. We want to prevent this as sampling evenly over the search space will get us a better initial regression model.

The way we choose to sample is using Latin hypercube sampling. This method divides the search space evenly in ranges for every dimension and requires that every range of every dimension contains at least one point. For example the two-dimensional plane is divided into rows and columns of the same height and width, and every row and column must contain at least one point. Within a range, we can just uniformly select a value per dimension. See figure 4.1 for an example of random sampling versus Latin hypercube sampling in the two-dimensional plane. This method ensures that points are evenly distributed per dimension, but not over multiple dimensions. Such additional requirements are much harder to incorporate into an algorithm. Orthogonal sampling is an example of a sampling method that requires that points are evenly distributed over multiple dimensions. Latin hypercube sampling however has a balance between time complexity and how evenly distributed the samples are. We choose the number of divisions to be the number of sample points.

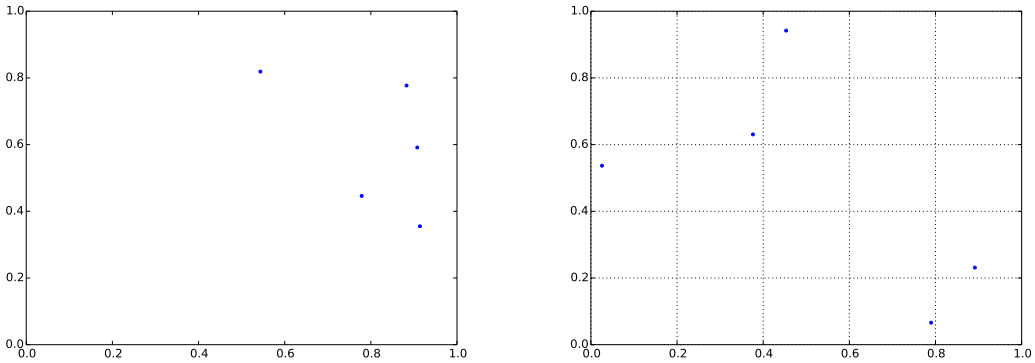


Figure 4.1: On the left a random sample is shown in the two-dimensional plane, whereas on the right a Latin hypercube sampling is shown. Note the ranges of the Latin hypercube sampling which have been indicated by dotted lines. Note that this random sample is clustered in contrast to a Latin hypercube sample, which can never be clustered.

The implementation of Latin hypercube sampling is very straightforward. Let's say we need M sample points and we have D dimensions. Per dimension $d \in [1, D]$ we split d its domain $[0, 1]$ into M even ranges. We then get a uniformly distributed random value within each range. After that we shuffle these values and assign them to the d-th coordinates of the M initial sample points. Technically the shuffling is not even required, but it ensures that the points are more randomly divided over the search space. Algorithm 3 describes Tree-GP in detail. The total run time of the algorithm is $\mathbb{O}(M \times D)$, as shuffling can be done in $\mathbb{O}(M)$.

Algorithm 3 Initial sample

```
procedure GetInitialSample(f, M)
    Let f be the function to be optimized.
    Let M be the number of initial samples.
    Let D be the dimensionality of f.
    Let X be the set of all evaluated points, X = {x_1, x_2, x_3, ...}.
    Let x_i be the position vector of the evaluated at iteration i.
    Let T be the set of the values of all evaluated points, T = {t_1, t_2, t_3, ...}.
    Let V be a matrix of dimensions M × D.

    for d ∈ [1, D] do
        for i ∈ [1, M] do
            r ← GetRealRandomNumber(0, 1)   ▷ Get real random division offset in [0, 1].
            j ← GetNaturalRandomNumber(1, i)   ▷ Get random point index in [1, i].

            v ← (i−1+r)/M

            V_{i,d} ← V_{j,d}   ▷ Swap value with random other point.
            V_{j,d} ← v   ▷ Assign value to random other point.
        end for
    end for

    for i ∈ [1, M] do
        x_i ← V_{i,⋆}   ▷ Set evaluation point.
        t_i ← f(x_i)   ▷ Evaluate function.
    end for

    return (X, T)
end procedure
```

4.2 Posterior distribution

The next step in Bayesian optimization is choosing the way we create the posterior distribution over the loss function f. This posterior distribution is first calculated over the initial sample. The posterior distribution is then recalculated over all points after each function evaluation. We will make use of Gaussian process regression, described in the preliminaries. We use the squared exponential covariance function,

$$k(\mathbf{x}, \mathbf{x}') = \theta_0 \ \exp(-\sum_{d=1}^{D} \theta_d (x_d - x'_d)^2).$$

This kernel function has some nice properties. First of all functions drawn from a Gaussian process with this kernel function are smooth and continuous. Because we know nothing about the black box loss function this seems like a reasonable assumption to make. It is stationary which makes the Gaussian process translation independent, which is also a nice property to have. Finally it provides us with $D + 1$ hyper-parameters, which we can use to obtain a better fit to the landscape of the loss function.

Like explained in the preliminaries, we will assume a zero mean function and center the value $\mathbf{t}$. In order to prevent numerical problems, we also divide the centered values by the standard deviation of $\mathbf{t}$. When making predictions, the reverse transformations are applied.

4.3 Maximum likelihood estimation

In order to fit the Gaussian process regression model to the input points we need to do maximum likelihood estimation to find the hyper-parameters θ of the kernel function, like we explained in section 2.3.6. As explained in the preliminaries, this involves a derivative-based optimization algorithm. A derivative or gradient based optimization algorithm can make use of the fact that we can calculate the first-order partial derivatives of the function in each of its parameters. Usually gradient based optimization algorithms only find a local optimum of a function, and so they need a starting point in the search space. A strategy that runs the algorithm multiple times with different starting points can be used to make it more likely to find the global optimum. Gradient descent is an example of a very simple algorithm that takes a step in the direction of the function gradient at each iteration. The size of the step is determined by the magnitude of the gradient multiplied by a small constant.

We have chosen the Limited-memory Broyden-Fletcher-Goldfarb-Shanno or L-BFGS algorithm to do derivative based optimization. This algorithm is a very popular quasi-Newton optimization method. In order to understand what a quasi-Newton optimization algorithm is, we first need to explain Newton's method. Newton's method for gradient-based optimization uses both the first derivative and the second derivative to approximate the region around the optimum with a quadratic function. This quadratic approximation can then be used to quickly step towards the optimum. The second-order partial derivatives of a multidimensional function is its Hessian matrix. The Hessian matrix consists of the second-order partial derivatives of all combinations of input variables. As this matrix is usually heavy to compute, quasi-Newton optimizers where developed. These optimizers do not compute the Hessian matrix directly. They approximate it instead by analyzing previously computed first-order partial derivatives of the function.

The Broyden-Fletcher-Goldfarb-Shanno or BFGS algorithm is such a quasi-Newton optimizer. It updates the inverse Hessian matrix at each gradient evaluation of the function to be optimized. The inverse of the Hessian matrix is initialized at the identity matrix. Starting from a given starting point, the BFGS algorithm calculates the step direction from the inverse Hessian matrix. After this, it determines the step size with a line search algorithm. Based on the step direction and size, the current point is updated. Then, both the function itself and its partial derivative are calculated. This information is used to update the inverse Hessian matrix. After the update the algorithm is repeated until the gradient is close enough to zero.

Limited-memory BFGS is a variant of BFGS that does not store the entire inverse Hessian matrix. It only stores the last H position and gradients of the function and implicitly constructs the inverse matrix at every iteration with this information. The amount of updates that are stored is determined by the history parameter H and is usually chosen to be around 10 to 15. For normal BFGS with a D-dimensional function memory usage is of order $\mathbb{O}(D^2)$. For L-BFGS it is only $\mathbb{O}(H \times D)$ which is a lot less if the function has a lot of parameters.

We have chosen not to run the algorithm multiple times, as this would be too expensive. Instead, we start from a single starting point and run the algorithm once to find the values of θ. The initial values of θ were all chosen to be 1, which seems to work reasonable well for most loss functions. To limit computation power used by L-BFGS, we have also set a maximum number of iterations. This makes the overhead of finding the hyper-parameters $\mathbb{O}(1)$. Note however,

that for each iteration a complete Gaussian process regression needs to be calculated, which involves a matrix inversion. This makes a single iteration very expensive, so in practice it is very important that we do as little iterations as possible. Because L-BFGS is very efficient, good hyper-parameters are usually found in 20 to 30 iterations.

4.4 Acquisition criteria function

We have seen in section 3.4 that the existing acquisition criteria function all have weaknesses. That's why we decided to construct a new acquisition criteria function. Minimizing the acquisition criteria function determines which point is going to be evaluated and so it is very important. We want a function that at first explores the search space, and will later on exploit interesting regions. After some experimentation we came up with a function that does both, depending on a constant that can be varied. It combines both the predicted mean and the predicted variance of the posterior distribution. We call this new acquisition criteria function the Cumulative Mean Probability to Variance Ratio (CMPVR). It is completely independent of scale and translation of the loss function. It is also independent of the scale of the posterior distribution its predicted error. The function is defined as a ratio of which the numerator is the cumulative probability of the predicted mean. The denominator is the predicted variance, which is weighted by an exponent c. Let g be the probability distribution over the values of $f(\mathbf{x})$. Furthermore, let G be the cumulative probability distribution over the values of $f(\mathbf{x})$. Then the full definition of the acquisition criteria function is

$$h_{cmpvr}(\mathbf{x}) = \frac{G(\mu(\mathbf{x}))}{\sigma^2(\mathbf{x})^c}.$$

Here $c \in \mathbb{R}_{\geq 0}$ is the exploration constant. If c is taken to be 0, the acquisition criteria function will only contain the cumulative probability of the predicted mean, and so minimizing it will find the minimum predicted mean. The algorithm will then just be exploiting. If however c is taken positive, the variance will be weighted in. High variances usually occur at places in the search space that do not have evaluated point nearby and so this will result in exploitation. Larger values of c will result in more exploitation, so this constant determines the balance between low mean values and high variances. Figure 4.2 shows the evaluated points of the two-dimensional sphere problem for different c values.

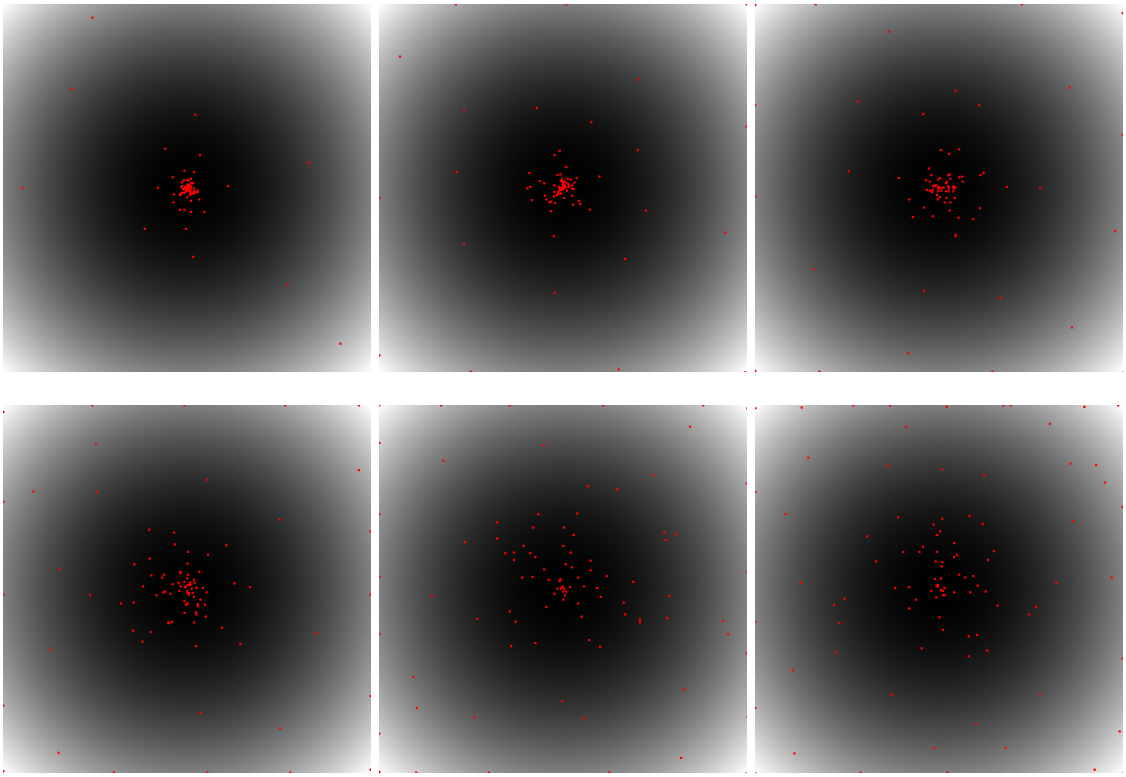


Figure 4.2: From top to bottom, left to right: the evaluated points of the two-dimensional sphere problem, for c values 0.1, 0.25, 0.5, 1, 2.5 and 5. Note the consideration between exploration and exploitation.

Until now we have assumed the probability distribution g to be known. However, in general this will not be the case. Therefore we will assume the distribution to be a Gaussian distribution. We can approximate the mean and standard deviation of this Gaussian distribution from the evaluated function values, the population. The cumulative probability distribution of a Gaussian distribution can be calculated using the error function $\operatorname{erf}(x)$. Given a population mean μ_{pop} and standard deviation σ_{pop}, $G(x)$ is given by

$$G(x) = \frac{1}{2}\left(1 + \operatorname{erf}\left(\frac{x - \mu_{pop}}{\sigma_{pop}\sqrt{2}}\right)\right).$$

The error function cannot be evaluated directly as it is defined as an integral,

$$\operatorname{erf}(x) = \frac{2}{\sqrt{\pi}} \int_0^x e^{-t^2}.$$

We can however approximate it with

$$\operatorname{erf}(x) \approx 1 - (a_1 t + a_2 t^2 + \cdots + a_5 t^5)e^{-x^2}, \quad t = \frac{1}{1 + px},$$

where $p = 0.3275911$, $a_1 = 0.254829592$, $a_2 = -0.284496736$, $a_3 = 1.421413741$, $a_4 = -1.453152027$ and $a_5 = 1.061405429$, which has a maximum error of 1.5×10^{-7} [1]. This approximation is only valid for $x \geq 0$. Because erf is an odd function, we can use $-\operatorname{erf}(-x)$ for $x < 0$.

Like we said before, the acquisition criteria function needs to be differentiable, otherwise we are left with a minimization problem that is just as hard to solve as the original problem. Taking the partial derivative of $h_{cmpvr}(\mathbf{x})$ with respect to a single coordinate x_d we get

$$\frac{\partial}{\partial x_d} h_{cmpvr}(\mathbf{x}) = \sigma^2(\mathbf{x})^{-1-c} \times \left(\sigma^2(\mathbf{x}) \times g(\mu(\mathbf{x})) \times \frac{\partial \mu(\mathbf{x})}{\partial x_d} - c \times G(\mu(\mathbf{x})) \times \frac{\partial \sigma^2(\mathbf{x})}{\partial x_d} \right),$$

where $g(x)$ is the Gaussian distribution with mean μ_{pop} and standard deviation σ_{pop},

$$\begin{aligned} g(x) &= \mathcal{N}(x, \sigma_{pop}, \mu_{pop}) \\ g(x) &= \frac{1}{\sigma_{pop}\sqrt{2\pi}} \exp\left(-\frac{(x-\mu_{pop})^2}{2 \times (\sigma_{pop})^2} \right). \end{aligned}$$

Note that we need the partial derivatives of both the mean function and the variance function of the posterior distribution. For Gaussian process regression they are given by

$$\begin{aligned} \frac{\partial \mu(\mathbf{x})}{\partial x_d} &= \sum_i \frac{\partial k(\mathbf{x}, \mathbf{x}_i)}{\partial x_d} (\mathbf{C}^{-1}\mathbf{t})_i \\ \frac{\partial \sigma^2(\mathbf{x})}{\partial x_d} &= -\sum_{i,j} \left(\frac{\partial k(\mathbf{x}, \mathbf{x}_i)}{\partial x_d} k(\mathbf{x}, \mathbf{x}_j) + \frac{\partial k(\mathbf{x}, \mathbf{x}_j)}{\partial x_d} k(\mathbf{x}, \mathbf{x}_i) \right) \mathbf{C}^{-1}_{i,j}. \end{aligned}$$

The CMPVR function is completely independent of scale and translation of loss function values. The numerator is the cumulative probability of the predicted loss function value, which is invariant under scale and translation of loss function values. This makes the CMPVR function very generic, in contrast to the PI function. The CMPVR acquisition criteria function is also invariant under the scale of the predicted variance. The scale of the predicted variance does not matter in the ratio as we are minimizing the CMPVR function. This invariance is important because Gaussian process regression may underestimates the scale of the predicted variance which gave problems for the SLB function.

Algorithm 4 gives an implementation of the CMPVR acquisition criteria function.

Algorithm 4 Acquisition criteria function

```
procedure GetErrorFunctionValue(x)
    swap ← x < 0

    if swap then
        x ← −x
    end if

    p ← 0.3275911
    a_1 ← 0.254829592
    a_2 ← −0.284496736
    a_3 ← 1.421413741
    a_4 ← −1.453152027
    a_5 ← 1.061405429

    t ← 1/(1+px)
    v ← 1 − (a_1 t + a_2 t^2 + ··· + a_5 t^5) e^{−x^2}

    if swap then
        return −v
    else
        return v
    end if
end procedure

procedure GetErrorFunctionDerivative(x)
    return (2/√π) e^{−x^2}
end procedure

procedure GetNormalValue(x, μ, σ)
    return (1/(σ√(2π))) exp(−(x−μ)^2/(2σ^2))
end procedure

procedure GetCumulativeNormalValue(x, μ, σ)
    return (1/2) × (1+ GetErrorFunctionValue((x−μ)/(σ√2)) )
end procedure

procedure GetCumulativeNormalDerivative(x, μ, σ)
    return GetNormalValue(x, μ, σ)
end procedure
```

Algorithm 5 Acquisition criteria function (continued)

```
procedure GetCMPVR(x, μ_pop, σ_pop, c)
    m ← GetMean(x)
    v ← GetVariance(x)

    n ← GetCumulativeNormalValue(m, μ_pop, σ_pop)
    d ← v^c

    return n/d
end procedure

procedure GetCMPVRDerivative(x, μ_pop, σ_pop, c)
    m ← GetMean(x)
    v ← GetVariance(x)

    m' ← GetMeanDerivative(x)
    v' ← GetVarianceDerivative(x)

    n ← GetCumulativeNormalValue(mean, μ_pop, σ_pop)
    n' ← GetCumulativeNormalDerivative(mean, μ_pop, σ_pop)

    ∀i : r_i ← v^(−1−c) × (v × n' × m'_i − c × n × v'_i)

    return r
end procedure
```

In order to minimize this acquisition criteria function to find the next evaluation point, we pick 100 random points and run the L-BFGS algorithm with these points as the starting points. We also run the L-BFGS algorithm starting in the best point that has been found. Of each of these runs, we pick the point that has the lowest function value. The loss function is then evaluated in this point and the posterior distribution is updated. Because we only run the L-BFGS algorithm a constant number of times, and the L-BFGS algorithm uses a constant amount of iterations, the total time that we spend finding a new point is of the order of calculating a prediction and its derivative of a Gaussian process regression. Calculating a prediction and its derivative is of order $\mathbb{O}(D \times n^2)$, where n is the number of points of the Gaussian process regression.

The only thing left to decide is how to vary the exploration constant c over time. Based on the results of figure 4.2 we decided to initially set c to 0.25 and then letting it exponentially decay to 0.0001 at iteration 100. This is implemented by multiplying c with $(\frac{0.0001}{0.25})^{\frac{1}{100}}$ at each iteration, which is about 0.92472. This way, the search space is gradually exploited. The decay continues after 100 iterations, but if there is no improvement in 50 iterations, c will be reset to its initial value 0.25. This will make the acquisition criteria function explore again, which will hopefully result in finding new minima that can be exploited. The exact constants are not very important, but is important that the initial value of c is not too low, because the algorithm needs the exploration to find a good minimum. The exploration also helps with building a more accurate model, because the evaluated points are distributed over the whole search space. This in turn helps the algorithm with exploitation. The constant that c converges to should be very small, because in the end we want to sample at the predicted minimum of the model. That's because in the end the minimum of the model is the most likely the minimum of the loss function.

Figure 4.3 shows the evaluated points of the two-dimensional sphere problem with our decaying scheme.

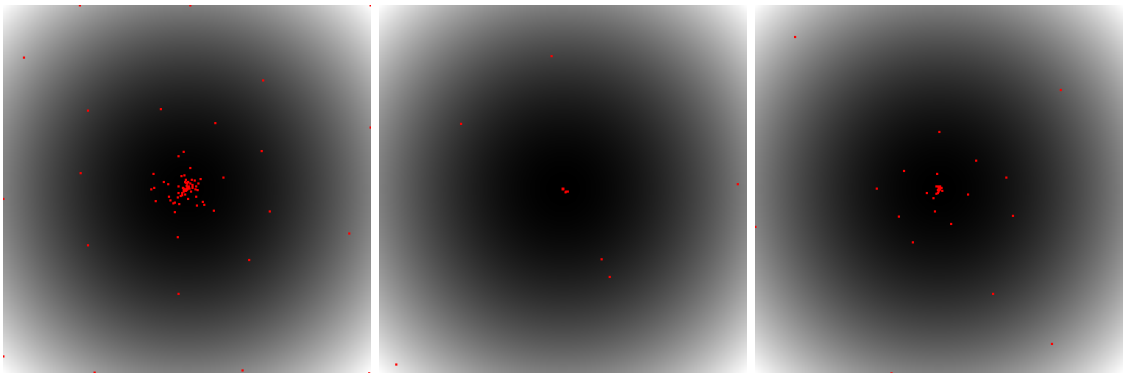


Figure 4.3: The left image shows the evaluated points of the two-dimensional sphere problem for $c = 0.25$. The image in the center shows the evaluated points for $c = 0.0001$. The right image shows the evaluated points with our decaying scheme.

Chapter 5

Scaling the algorithm

In this chapter we will try to scale the algorithm of the previous chapter in the amount of function evaluations. We will do this by using a mixture model of multiple Gaussian process regression models stored in a vantage-point tree.

5.1 Time complexity

The details given in the previous chapter are sufficient to give an implementation of a Bayesian optimization algorithm based on a Gaussian process regression model. However, the time complexity of the algorithm is very important. Even though we assume loss functions that take a lot of time to evaluate, we still want the overhead of our algorithm to be as little as possible. As is shown in the preliminaries, calculating a Gaussian process regression over n input points with dimensionality D requires a matrix inversion of the covariance matrix $\mathbf{C}$. The covariance matrix itself can be computed in $\mathbb{O}(D \times n^2)$ time and has size $n \times n$. The covariance matrix inversion takes $\mathbb{O}(n^3)$ time. This gives us a total of $\mathbb{O}(D \times n^2 + n^3)$ time to do a single Gaussian process regression. The regression is done multiple times to find the hyper-parameters θ, but the number of iterations of L-BFGS can be considered constant. The time taken to calculate a new evaluation point is $\mathbb{O}(D \times n^2)$. Because the Gaussian process regression is also recalculated for every function evaluation, the total time per function evaluation is $\mathbb{O}(D \times n^2 + n^3)$. In order to find the total complexity of the algorithm, we have to sum over all function evaluations. The algorithm does N function evaluations, so this gives us a total time of

$$
\begin{aligned}
&\mathbb{O}\left(\sum_{n=1}^{N}(D \times n^2 + n^3)\right) = \\
&\mathbb{O}\left(D\sum_{n=1}^{N} n^2 + \sum_{n=1}^{N} n^3\right) = \\
&\mathbb{O}\left(D \times \frac{N(N+1)(2N+1)}{6} + \left(\frac{N(N+1)}{2}\right)^2\right) = \\
&\mathbb{O}\left(D \times N^3 + N^4\right).
\end{aligned}
$$

5.2 Mixture model

The fact that the algorithm takes quartic time in the total amount of evaluations means that the algorithm is not scalable. We want the algorithm to be scalable, which ideally means that it uses linear time in the amount of iterations. In order to make the algorithm scalable, we came up with the idea to use a mixture model.

A mixture model models the whole search space with multiple local models. In our case, these local models are Gaussian process regression models. The models are local in the sense that they only model the region of space that is near the points that they consist of. We limit this amount of points, so that the time spent creating the local model is only dependent on the function dimensionality D. New evaluated points are only added to the models whose regions in space are the closest to that point. These models are then recalculated with the new point. When a model is bigger than the limit set, it has to be split into two models. Predictions made with a local model are also only dependent on D. When making a prediction, the models whose regions in space are the closest to that prediction point are used. The predictions made by each model at the prediction point are then weighted according to their distance to the model to make the final prediction. In the next sections, each of the parts of the mixture model will be explained in more detail.

5.2.1 Definition of local model

A local model of the mixture model is a normal Gaussian process regression model created from a limited number of points. We have chosen the maximum number of model points to be 50. Each local model has different hyper-parameters θ, so that the covariance function can be modeled differently per model. Because local models can have different hyper-parameters, local areas of the search space can be modeled better, resulting in a better overall model.

5.2.2 Vantage-point tree

When adding new points and making predictions we will be doing k-nearest neighbor queries on the existing points. In order to speed up these queries, all points are put into a vantage-point tree. Like explained in the preliminaries, a vantage-point tree speeds up nearest neighbor queries considerably by splitting the points recursively using a distance metric and a vantage-point. First all points are added to a root leaf node. Each leaf node it then split into two leaf nodes until all leaf nodes contain at most a certain maximum number of points. When a leaf node is split, a vantage-point is chosen from the its points and the remaining points are partitioned using the median of their distances to the vantage-point. We make two changes to the vanilla vantage-point tree algorithm in order to make it usable as a mixture model. First off, our tree is not static but dynamic. This means that the tree is not immediately fully created, but instead incrementally updated. How this is done will be explained later on. The second change is that our leaf nodes each contain a Gaussian process regression model of the points in it. The maximum number of leaf points is set to the maximum number of model points, which is 50. If the number of points in a leaf node exceeds this number of points, the leaf node is split into two leaf nodes. These two new leaf nodes are each given a new Gaussian process regression model of the points in it. The specifics of splitting a leaf node are explained later on.

5.2.3 Adding a point

The disadvantage of using local models is that they cannot use the information provided by points further away. To mitigate this problem somewhat, points are added to multiple local

models. When adding a point, we first find the k existing nearest points using the vantage-point tree. We then determine the leaf nodes of these nearest points in the vantage-point tree. Note that there may be less than k leaf nodes as nearest points may have the same leaf node. The new point is then added to each of the leaf nodes. When a leaf node exceeds the maximum number of points, it is split into two leaf nodes. The leaf nodes their Gaussian process regression models are then recalculated using their new set of points. We have chosen the constant k to be 5, which works very well in practice. Algorithm 6 gives an implementation.

Algorithm 6 Adding a point to a vantage-point tree

```
procedure VPTreeAddPoint(tree, x, t, k)
    N ← VPTreeKNearestNeighbor(tree, x, k)

    for node ∈ {n_i | (n_i, p) ∈ N, ∀j < i : n_j ≠ n_i} do        ▷ For all unique leaf nodes.
        node.x ← node.x ∪ {x}                                      ▷ Add point to node.
        node.t ← node.t ∪ {t}

        if |node.points| > 50 then                                 ▷ Check for split.
            (left, right) ←VPTreeSplitNode(tree, node)

            left.model ← GPCalculate(left.x, left.t)
            right.model ← GPCalculate(right.x, right.t)
        else
            node.model ← GPCalculate(node.x, node.t)
        end if
    end for
end procedure
```

5.2.4 Making predictions

When making a prediction in order to calculate the acquisition criteria function value, we also combine the values of at most k models. We first find the k nearest points to the prediction point, and make predictions with their corresponding models. We want points close by to have more impact on the final prediction than points far away. Therefore, we weight the points their predictions according to the distance to the prediction point. For the weighting, we use a modified version of the inverse distance weighting. Let $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ be the set of n points to weight. Let $V = \{v_1, \ldots, v_n\}$ be their corresponding predicted values. Let $\mathbf{x}$ be the prediction point. Let d_{max} be the maximum distance of $\mathbf{x}$ to P, given by

$$d_{max} = \max(\{d_i |\ \mathbf{p}_i \in P\}), \quad d_i = \|\mathbf{p}_i - \mathbf{x}\|.$$

Then the final predicted value v is given by

$$v = \begin{cases} \dfrac{\sum_{i=1}^{N} w_i v_i}{\sum_{i=1}^{N} w_i} & \text{if } \forall i : d_i \neq 0 \\ v_i & \text{if } \exists i : d_i = 0 \end{cases}$$

where

$$w_i = \left(\frac{d_{max} - d_i}{d_i} \right)^2, \quad d_i = \|\mathbf{p}_i - \mathbf{x}\|.$$

This weighting gives points close by a lot more weight than points far away. This is desirable as the models are local and so models close by a prediction point provide far better predictions for that point than models far away. We use the same weighting for predicting the mean, variance, the mean its derivative and the variance its derivative. Algorithm 7 gives an implementation.

Algorithm 7 Making predictions

```
Let tree be a vantage point tree.

procedure GetWeightedNearestModels(x, k)
    N ← VPTreeKNearestNeighbor(tree, x, k)

    d_max ← max({‖p − x‖ | (node, p) ∈ N})          ▷ Find maximum distance.

    W ← {}
    w_sum ← 0

    for (node, p_i) ∈ N do
        d_i ← ‖p_i − x‖                              ▷ Calculate distance.

        if d_i = 0 then
            return {(1, node.model)}
        end if

        w_i ← ((d_max − d_i) / d_i)^2                ▷ Calculate weight.
        w_sum ← w_sum + w_i

        W ← W ∪ {(w_i, node.model)}                  ▷ Add weighted model.
    end for

    for (w_i, model) ∈ W do                          ▷ Normalize weights.
        w_i ← w_i / w_sum
    end for

    return W
end procedure
```

Algorithm 8 Making predictions (continued)

```
procedure GetMean(x)
    W ← GetWeightedNearestModels(x, 5)

    v ← 0
    for (w_i, model) ∈ W do
        v ← v + w_i × GPGetMean(model, x)
    end for

    return v
end procedure

procedure GetVariance(x)
    W ← GetWeightedNearestModels(x, 5)

    v ← 0
    for (w_i, model) ∈ W do
        v ← v + w_i × GPGetVariance(model, x)
    end for

    return v
end procedure

procedure GetMeanDerivative(x)
    W ← GetWeightedNearestModels(x, 5)

    v ← 0
    for (w_i, model) ∈ W do
        v ← v + w_i × GPGetMeanDerivative(model, x)
    end for

    return v
end procedure

procedure GetVarianceDerivative(x)
    W ← GetWeightedNearestModels(x, 5)

    v ← 0
    for (w_i, model) ∈ W do
        v ← v + w_i × GPGetVarianceDerivative(model, x)
    end for

    return v
end procedure
```

5.2.5 Splitting a leaf node

The main change that we made to the vanilla vantage-point tree is the way that leaf nodes are split. Leaf nodes need to be split so that the local models do not become too large. The splitting boundary is determined by a hyper-sphere centered in a vantage-point, which has the median distance as the radius. Half of the points lie within the hyper-sphere, the other half lies outside of it. The trick is to select the vantage-point. Selecting a good vantage-point is a difficult task. Ideally, when the points are clustered into two clusters we want the split in between the two clusters. This way, the new models stay as local as possible. However, the situation may arise that there is only a single cluster. This happens for example when the algorithm exploits a local minimum. In that case, the density of the points will increase towards the local minimum. The split should definitely not split through this local minimum, as we want it to be as accurately modeled as possible. This is only possible if all points within a region of the local minimum are put into the same leaf node. This way, we zoom in on the solution.

The partitioning algorithm that we use does all of this. Our partitioning algorithm selects the vantage-point whose splitting hyper-sphere has the maximum average distance to all points. This is done by trying every point as the vantage-point, and calculating all distances of other points to the selected vantage-point. The median distance is then selected from these distances. This median distance will be the radius of the splitting hyper-sphere. The distance to the hyper-sphere of a point can then be calculated by taking the absolute value of the distance to the vantage-point minus the median distance. We can then take the average and select the vantage-point with the maximum average.

Let us also give a formal definition. Let $P = \{\mathbf{p}_1, \ldots, \mathbf{p}_n\}$ be all points to be split. Let the vantage-point that we are trying be $\mathbf{p}_{vp}$. Let d_{med} be the median distance of all points in P to $\mathbf{p}_{vp}$. The average distance d_{avg} of all points to this boundary is then given by

$$d_{avg} = \frac{1}{n} \sum_{i=1}^{n} |d_i - d_{med}|, \quad d_i = \|\mathbf{p}_i - \mathbf{p}_{vp}\|.$$

The optimal vantage-point $\mathbf{p}^*_{vp}$ is given by

$$\mathbf{p}^*_{vp} = \underset{\mathbf{p}_{vp} \in P}{\operatorname{argmax}}(d_{avg}).$$

Figure 5.1 shows the behavior of our partitioning algorithm on different sets of points in the plane. The hyper-sphere is shown in red, which is just a circle in two dimensions. Note the different behaviors with two clusters and a single cluster. The latter shows the zoom behavior that we are after. Hyper-spheres are ideal as splitting boundaries in this regard as they allow both behaviors. This is also one of the reasons that we selected the vantage-point tree as our nearest neighbor data structure instead of for example the k-d-tree, which has axis-aligned hyper-planes as splitting boundaries. A hyper-plane boundary will never be able to deliver this zoom behavior.

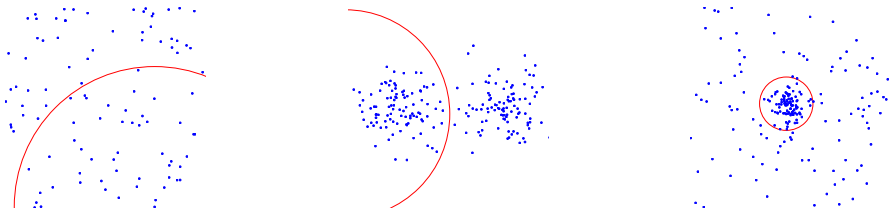


Figure 5.1: Three splits on leaf points: on uniformly distributed points, two clusters of points and on uniformly distributed points with a single high density cluster. Note the zoom behavior on the latter.

Algorithm 9 gives an implementation of our partitioning algorithm. The algorithm takes $\mathbb{O}(n^2)$ in the number of points of the leaf node to be split. This number is bound by a constant, so theoretically the time is only $\mathbb{O}(1)$.

Algorithm 9 Splitting a leaf node

```
procedure VPTreeSelectVantagePoint(points)
    d*_sum ← 0
    for p_vp ∈ points do
        D ← {}
        for p_i ∈ points do                          ▷ Calculate distances.
            d_i = ‖p_i − p_vp‖

            D ← D ∪ {d_i}
        end for

        d_med ← GetMedian(D)                         ▷ Get median distance.

        d_sum ← 0
        for p_i ∈ points do
            d_sum ← d_sum + |d_i − d_med|
        end for

        if d_sum > d*_sum then
            p*_vp ← p_vp
            d*_sum ← d_sum
            d*_med ← d_med
        end if
    end for

    return (p*_vp, d*_med)
end procedure
```

Algorithm 10 Splitting a leaf node (continued)

```
procedure VPTreeSplitNode(tree, node)
    (p_vp, d_med) ← VPTreeSelectVantagePoint(node.points)

    left.points ← {}
    right.points ← {}

    left.isLeaf ← T
    right.isLeaf ← T

    d_max ← 0
    for p_i ∈ node.points do
        d_i = ‖p_i − p_vp‖

        if d_i < d_med then
            left.points ← left.points ∪ {p_i}
        else
            right.points ← right.points ∪ {p_i}
        end if

        if d_i > d_max then
            d_max ← d_i
        end if
    end for

    node.isLeaf ← F
    node.vantagePoint ← p_vp
    node.d_med ← d_med
    node.d_max ← d_max
    node.left ← left
    node.right ← right

    return (left, right)
end procedure
```

5.3 Algorithm

Algorithm 11 describes the full TREE-GP algorithm.

Algorithm 11 The TREE-GP algorithm

```
Let tree be a vantage point tree.
Let μ_pop be the population mean.
Let σ_pop be the population variance.

procedure UpdatePosterior(x, t)
    VPTreeAddPoint(tree, x, t, 5)

    Update μ_pop and σ_pop with t.
end procedure

procedure CreateInitialPosterior(X, T)
    tree.root.isLeaf ← T                                  ▷ Create initial root node.
    tree.root.x ← {}
    tree.root.t ← {}

    for x_i ∈ X do                                        ▷ Add initial sample.
        UpdatePosterior(x_i, t_i)
    end for
end procedure

procedure GetNextEvaluationPoint(c)
    v* ← ∞

    for i ∈ [1, 100] do
        for d ∈ [1, D] do                                 ▷ Generate a random point.
            x_d ←GetRealRandomNumber(0, 1)                ▷ Get a real random number in [0, 1].
        end for

        (x, v) ← L-BFGS (GetCMPVR, GetCMPVRDerivative, x, μ_pop, σ_pop, c)

        if v < v* then                                    ▷ Find best point.
            v* ← v
            x* ← x
        end if
    end for

    return x*
end procedure
```

Algorithm 12 The TREE-GP algorithm (continued)

```
procedure OurAlgorithm(f, N)
    Let f be the function to be optimized.
    Let N be the number of function evaluations.
    Let M be the number of initial samples.
    Let D be the dimensionality of f.
    Let i be the current iteration.
    Let i_imp be the iteration of the last improvement.
    Let c be the exploration constant.
    Let X be the set of all evaluated points, X = {x_1, x_2, x_3, ...}.
    Let x_i be the position vector of the evaluated at iteration i.
    Let T be the set of the values of all evaluated points, T = {t_1, t_2, t_3, ...}.
    Let t_i be the value of f at x_i.
    Let t_min be the minimum value found so far.
    Let x_min be the position vector of the value t_min.

    M ← 5

    (X, T) ← GetInitialSample(f, M)

    t* ← min(T)
    x* ← argmin_{x_i ∈ X} t_i

    i ← M + 1
    i_imp ← i
    c ← 0.25

    CreateInitialPosterior(X, T)

    while i ≤ N do
        x_i ← GetNextEvaluationPoint(c)

        t_i ← f(x_i)                                   ▷ Evaluate function.

        if t_i < t* then                               ▷ Check for improvement.
            t* ← t_i
            x* ← x_i
            i_imp ← i
        end if

        UpdatePosterior(x_i, t_i)

        if i − i_imp ≥ 50 then
            c ← 0.25                                   ▷ Reset exploration constant.
        else
            c ← c × (0.0001/0.25)^(1/100)              ▷ Update exploration constant.
        end if

        i ← i + 1
    end while

    return x*
end procedure
```

Chapter 6

Experiments

In this chapter we will compare our Tree-GP algorithm to a variant called GP and with the AMaLGaM algorithm. We will do this by running the algorithm against two test suites which have been created for the IEEE Congress on Evolutionary Computation.

6.1 Experiment setup

Now that we have presented the Tree-GP algorithm, we obviously need to see how well it performs. In order to do this, we will have to run the algorithm against black box test functions. It is hard to do this with real-life test functions because we often know nothing about their properties or their true global minimum. That is why we will be running the algorithm against artificial test functions. In the literature, all kinds of such artificial test functions with known properties and global minima have been proposed. Over time, these test functions have been perfected and put into various standard test suites. One such a suite is the Black-Box Optimization Benchmarking (BBOB) test suite. This suite is updated every year and a lot of authors use this test suite to compare their algorithms with others. This is the most important aspect of such test suites, they can be used to give a fair comparison of two algorithms. The 2009 version of the BBOB test suite has been used to test the performance of the AMaLGaM algorithm [3].

Another such a test suite is provided each year by the IEEE Congress on Evolutionary Computation (IEEE CEC). The recently released 2014 version of the test suites provided by them is particularly interesting, as they now have a test suite specifically for expensive function optimization. They also have a normal test suite for other kinds of global numerical optimization algorithms. Because they provide this expensive function test suite and their test suites are the most recent, we will be using their test suites to test our algorithm.

The way we will be measuring the performance of our algorithm and that of AMaLGaM is twofold. The first performance measure is the minimum function value that has been reached at each function evaluation. The second is the amount of time required to get to this result. Both performances are very important in order to choose which algorithm is best for a certain evaluation function.

In order to be able to compare the results of Tree-GP, we will also run the same performance tests on the estimation of distribution algorithm AMaLGaM. This algorithm has not specifically been designed for expensive test functions. It does however, put in quite a lot of effort to get new points in the search space to evaluate. It does this in a totally different way than our algorithm does. Instead of modeling the values of the optimization function, it models the distribution of points in the search space that have the best values. It does this by keeping a population of points

whose density in space is modeled by a multivariate Gaussian distribution [3]. It then samples this density model and evaluates the sampled points. After this, the population is updated by keeping the best points and a new density model is created. We will be using the iAMaLGaM-Full-Free version of the algorithm for testing. This version determines the population size of its model incrementally and models all second-order interactions between variables. It also determines its parameters automatically, so that no configuration is required.

We will also compare the Tree-GP algorithm to a variant of the algorithm that does not use the vantage-point tree, but uses a normal Gaussian process regression model. This way we can see how well our mixture model idea scales the algorithm and the impact that it has on the values found. We will indicate this algorithm as GP. As this algorithm is very inefficient, we will limit the total amount of function evaluations to 500.

To do statistical sound experiments, we will repeat each experiment five times. For the minimum function value that has been found at each function evaluation, we will take the median of the five values. This is because we want to show a value that has actually been found, which an average value would not be. The timing values that we will show are averages over the five experiments. We will show both the total run time of each algorithm excluding the function evaluations and the overhead that each algorithm has per individual evaluation. This way we can clearly see how well the algorithms scale with the amount of function evaluations. As AMaLGaM is a lot faster than our algorithm, we will let it do three times as much function evaluations. This way we can see better how AMaLGaM its best function value converges, relative to our Tree-GP and GP algorithms. We do three times as much evaluations because this still allows us to plot the timing in the same graph.

All the tests will be run on machines with an Intel Xeon E3-1270 v3 3.5 GHz quadcore processor and 8 GiB of memory. Each test will use a single core, so we can run four tests at the same time on a machine. No other applications will be run on the machines during the tests.

6.2 Test suite

The test suite consists of various test functions that are assumed to be expensive. These functions are not actually expensive to evaluate, otherwise the experiments would take a very long time. The CEC expensive test suite contains 8 functions, each of which is used in three different dimensionalities. This makes for a total of 24 test functions. The used function dimensionalities are 10, 20 and 30. The suite also indicates how much function evaluations are allowed. This is 500 evaluations for the 10-dimensional functions, 1000 for the 20-dimensional functions and 1500 for the 30-dimensional functions.

The test functions themselves are just mathematical formulas, which makes them easy to analyze. Each of the functions has different properties. One of these properties is whether the function has a single global minimum, that is uni-modal. A function can also be multi-modal, and then is interesting to know how much local minima it has. Another property is separability. If a function is separable, each of it dimensions can be optimized independently of the other dimensions. This may reduce a 10-dimensional function optimization to 10 1-dimensional optimizations. The optimum values of all the test functions are set at zero, in order to easily compare the results. We will now discuss each test function is the suite.

1. **Shifted sphere function**: The sphere function is the easiest test function as it is just a multi-dimensional parabola. It has a single minimum and is separable. It has been shifted so that the optimum is not at the origin. Its definition before shifting is

$$g_1(\mathbf{x}) = \sum_{i=1}^{D} x_i^2.$$

The three versions of this function tested are

$$\begin{aligned} f_1(\mathbf{x}) &= g_1(\mathbf{x} - \mathbf{o_1}), \quad \mathbf{o_1} \in \mathbb{R}^D, \quad D = 10, \\ f_2(\mathbf{x}) &= g_1(\mathbf{x} - \mathbf{o_2}), \quad \mathbf{o_2} \in \mathbb{R}^D, \quad D = 20, \\ f_3(\mathbf{x}) &= g_1(\mathbf{x} - \mathbf{o_3}), \quad \mathbf{o_3} \in \mathbb{R}^D, \quad D = 30, \end{aligned}$$

where $\mathbf{o_1}$, $\mathbf{o_2}$ and $\mathbf{o_3}$ are the optimum positions.

2. **Shifted ellipsoid function**: The ellipsoid function function is slightly harder to optimize than the sphere function, because it scales the dimensions differently. The function still has a single minimum and is separable. Its definition is

$$g_2(\mathbf{x}) = \sum_{i=1}^{D} i x_i^2.$$

The function has been shifted to prevent the minimum to be at the origin. The three versions of this function tested are

$$\begin{aligned} f_4(\mathbf{x}) &= g_2(\mathbf{x} - \mathbf{o_4}), \quad \mathbf{o_4} \in \mathbb{R}^D, \quad D = 10, \\ f_5(\mathbf{x}) &= g_2(\mathbf{x} - \mathbf{o_5}), \quad \mathbf{o_5} \in \mathbb{R}^D, \quad D = 20, \\ f_6(\mathbf{x}) &= g_2(\mathbf{x} - \mathbf{o_6}), \quad \mathbf{o_6} \in \mathbb{R}^D, \quad D = 30, \end{aligned}$$

where $\mathbf{o_4}$, $\mathbf{o_5}$ and $\mathbf{o_6}$ are the optimum positions.

3. **Shifted and rotated ellipsoid function**: We also test the rotated ellipsoid function. This is basically the ellipsoid function, except its input has been rotated by a randomly generated rotation matrix. It also has been shifted so that the minimum is not at the origin. The function has a single mimimum, but is not separable anymore. The versions that we test are

$$\begin{aligned} f_7(\mathbf{x}) &= g_2(\mathbf{M_7}(\mathbf{x} - \mathbf{o_7})), \quad \mathbf{o_7} \in \mathbb{R}^D, \quad \mathbf{M_7} \in \mathbb{M}(D, D), \quad D = 10, \\ f_8(\mathbf{x}) &= g_2(\mathbf{M_8}(\mathbf{x} - \mathbf{o_8})), \quad \mathbf{o_8} \in \mathbb{R}^D, \quad \mathbf{M_8} \in \mathbb{M}(D, D), \quad D = 20, \\ f_9(\mathbf{x}) &= g_2(\mathbf{M_9}(\mathbf{x} - \mathbf{o_9})), \quad \mathbf{o_9} \in \mathbb{R}^D, \quad \mathbf{M_9} \in \mathbb{M}(D, D), \quad D = 30, \end{aligned}$$

where $\mathbf{o_7}$, $\mathbf{o_8}$ and $\mathbf{o_9}$ are the optimum positions and $\mathbf{M_7}$, $\mathbf{M_8}$ and $\mathbf{M_9}$ are rotation matrices.

4. **Shifted step function**: The shifted step function is the discontinuous version of the shifted sphere function. It has a single minimum and is separable. Its definition before shifting is

$$g_3(\mathbf{x}) = \sum_{i=1}^{D} (\lfloor x_i + 0.5 \rfloor)^2.$$

The three versions of this function tested are

$$\begin{aligned} f_{10}(\mathbf{x}) &= g_3(\mathbf{x} - \mathbf{o_{10}}), \quad \mathbf{o_{10}} \in \mathbb{R}^D, \quad D = 10, \\ f_{11}(\mathbf{x}) &= g_3(\mathbf{x} - \mathbf{o_{11}}), \quad \mathbf{o_{11}} \in \mathbb{R}^D, \quad D = 20, \\ f_{12}(\mathbf{x}) &= g_3(\mathbf{x} - \mathbf{o_{12}}), \quad \mathbf{o_{12}} \in \mathbb{R}^D, \quad D = 30, \end{aligned}$$

where $\mathbf{o_{10}}$, $\mathbf{o_{11}}$ and $\mathbf{o_{12}}$ are the optimum positions.

5. **Shifted Ackley's function**: Ackley's function is a popular testing function. The optimum is basically a giant hole in the search space, whereas the rest of the search space is fairly flat and has a lot of local minima that are hard to get out of. This makes it very hard to find the global optimum. The problem is separable and the number of local minima is exponential in its dimensionality. The definition is

$$g_4(\mathbf{x}) = -20 \exp(-0.2 \sqrt{\frac{1}{D} \sum_{i=1}^{D} x_i^2}) - \exp(\frac{1}{D} \sum_{i=1}^{D} \cos(2\pi x_i)) + 20 + e.$$

We test the shifted versions of the function

$$\begin{aligned} f_{13}(\mathbf{x}) &= g_4(\mathbf{x} - \mathbf{o_{13}}), \quad \mathbf{o_{13}} \in \mathbb{R}^D, \quad D = 10, \\ f_{14}(\mathbf{x}) &= g_4(\mathbf{x} - \mathbf{o_{14}}), \quad \mathbf{o_{14}} \in \mathbb{R}^D, \quad D = 20, \\ f_{15}(\mathbf{x}) &= g_4(\mathbf{x} - \mathbf{o_{15}}), \quad \mathbf{o_{15}} \in \mathbb{R}^D, \quad D = 30, \end{aligned}$$

where $\mathbf{o_{13}}$, $\mathbf{o_{14}}$ and $\mathbf{o_{15}}$ are the optimum positions.

6. **Shifted Griewank's function**: Griewank's function is basically the sphere function with a high frequency low amplitude cosine added to it. The cosine has a different period per dimension. This gives the function a factorial amount of local minima in its dimensionality. It is not separable which makes it harder as well. The definition is

$$g_5(\mathbf{x}) = \sum_{i=1}^{D} \frac{x_i^2}{4000} - \prod_{i=1}^{D} \cos(\frac{x_i}{\sqrt{i}}) + 1.$$

The three shifted versions of this function tested are

$$\begin{aligned} f_{16}(\mathbf{x}) &= g_5(\mathbf{x} - \mathbf{o_{16}}), \quad \mathbf{o_{16}} \in \mathbb{R}^D, \quad D = 10, \\ f_{17}(\mathbf{x}) &= g_5(\mathbf{x} - \mathbf{o_{17}}), \quad \mathbf{o_{17}} \in \mathbb{R}^D, \quad D = 20, \\ f_{18}(\mathbf{x}) &= g_5(\mathbf{x} - \mathbf{o_{18}}), \quad \mathbf{o_{18}} \in \mathbb{R}^D, \quad D = 30, \end{aligned}$$

where $\mathbf{o_{16}}$, $\mathbf{o_{17}}$ and $\mathbf{o_{18}}$ are the optimum positions.

7. **Shifted and rotated Rosenbrock's function**: The Rosenbrock function is one of the most popular testing functions as it is very non-linear. It is very hard because it has a very narrow valley which leads from a local optimum to the global optimum. It's easy to find the valley but very hard to actually find the global optimum in it. The function is not separable and has a lot of local minima. The definition is

$$g_6(\mathbf{x}) = \sum_{i=1}^{D-1} (100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2).$$

To make this function even harder we shift and rotate the search space. The versions that we test are

$$\begin{aligned}
f_{19}(\mathbf{x}) &= g_6(\mathbf{M_{19}}(\mathbf{x} - \mathbf{o_{19}})), \quad \mathbf{o_{19}} \in \mathbb{R}^D, \quad \mathbf{M_{19}} \in \mathbb{M}(D, D), \quad D = 10, \\
f_{20}(\mathbf{x}) &= g_6(\mathbf{M_{20}}(\mathbf{x} - \mathbf{o_{20}})), \quad \mathbf{o_{20}} \in \mathbb{R}^D, \quad \mathbf{M_{20}} \in \mathbb{M}(D, D), \quad D = 20, \\
f_{21}(\mathbf{x}) &= g_6(\mathbf{M_{21}}(\mathbf{x} - \mathbf{o_{21}})), \quad \mathbf{o_{21}} \in \mathbb{R}^D, \quad \mathbf{M_{21}} \in \mathbb{M}(D, D), \quad D = 30,
\end{aligned}$$

where $\mathbf{o_{19}}$, $\mathbf{o_{20}}$ and $\mathbf{o_{21}}$ are the optimum positions and $\mathbf{M_{19}}$, $\mathbf{M_{20}}$ and $\mathbf{M_{21}}$ are rotation matrices.

8. **Shifted and rotated Rastrigin's function**: Finally we test Rastrigin's function which is very much like Griewank's function, but the cosine added has a very high amplitude. This makes it very hard to find the global optimimum, as it is barely different from all the local minima. The cosine does have the same period per dimension unlike Griewank's function. It is not separable however, because it has been rotated. Rastrigin's function definition is

$$g_7(\mathbf{x}) = \sum_{i=1}^{D} (x_i^2 - 10\cos(2\pi x_i) + 10).$$

We will test the shifted and rotated versions

$$\begin{aligned}
f_{22}(\mathbf{x}) &= g_7(\mathbf{M_{22}}(\mathbf{x} - \mathbf{o_{22}})), \quad \mathbf{o_{22}} \in \mathbb{R}^D, \quad \mathbf{M_{22}} \in \mathbb{M}(D, D), \quad D = 10, \\
f_{23}(\mathbf{x}) &= g_7(\mathbf{M_{23}}(\mathbf{x} - \mathbf{o_{23}})), \quad \mathbf{o_{23}} \in \mathbb{R}^D, \quad \mathbf{M_{23}} \in \mathbb{M}(D, D), \quad D = 20, \\
f_{24}(\mathbf{x}) &= g_7(\mathbf{M_{24}}(\mathbf{x} - \mathbf{o_{24}})), \quad \mathbf{o_{24}} \in \mathbb{R}^D, \quad \mathbf{M_{24}} \in \mathbb{M}(D, D), \quad D = 30,
\end{aligned}$$

where $\mathbf{o_{22}}$, $\mathbf{o_{23}}$ and $\mathbf{o_{24}}$ are the optimum positions and $\mathbf{M_{22}}$, $\mathbf{M_{23}}$ and $\mathbf{M_{24}}$ are rotation matrices.

We will be using the shifted sphere function f_2 with $D = 20$ for the algorithm timing experiments.

6.3 Results

The results of our experiments can be seen below. The values shown in the graphs of figure 6.1 are the minimum values found by the algorithms at each function evaluation. That is why the graphs are always monotonically decreasing. The values are calculated by taking the median value of the minimum values at each function evaluation of the 5 runs that we did for each function. The values are in logarithmic scale. The graphs are shown in a grid where each row is a different test function and the columns are dimensionalities 10, 20 and 30 respectively.

Figure 6.2 shows the performance results. The graphs show the total time taken of the algorithms from the start at each function evaluation. The times shown are in seconds and are averages over the 5 runs that we did. The times do not include the actual function evaluation itself, but the time spend in the algorithm. Figure 6.3 shows the overhead time of the algorithms at each function evaluation. The times are again in seconds and are averages over the 5 runs. Again, the times do not include the function evaluation itself.

A cubic function has been fitted to the total run time of GP. It is given by

$$f(x) = -0.0473590576 + 0.00289585665\ x + 0.0000353808647\ x^2 + 2.76400425 \times 10^{-8}\ x^3$$

where x is the current evaluation starting at 1. The fit has a Mean Absolute Error (MAE) of 0.10911594. The summation over x from 1 to n is given by

$$g(n) = \sum_{x=1}^{n} f(x)$$
$$g(n) = -0.0459052\ n + 0.00146563\ n^2 + 0.0000118074\ n^3 + 6.91001 \times 10^{-9}\ n^4$$

where n is the total amount of function evaluations. This function gives an approximation of the total run time of the algorithm excluding function evaluations. The order of the total run time of GP based on the fitted function is given by

$$g(x) = \mathbb{O}(n^4).$$

A logarithmic function has been fitted to the times of the Tree-GP algorithm. It is given by

$$f(x) = 0.11766902\ \log(x + 39.26508884) - 0.40674665$$

where x is the current evaluation starting at 1. It has a MAE of 0.01368722. The summation over $x$from 1 to n which gives the total run time is given by

$$g(n) = \sum_{x=1}^{n} f(x)$$
$$g(n) = -0.40674665\ n + 0.11766902\ \log\left((1 + 39.26508884)_n\right)$$

where x is the current evaluation starting at 1 and $(x)_n$ is the Pochhammer symbol, given by

$$(x)_n = x \times (x+1) \times (x+2) \times \cdots \times (x+n-1) = \frac{(x+n-1)!}{(x-1)!}.$$

The order of the total run time of the TREE-GP based on this fitted function is given by

$$
\begin{aligned}
g(n) &= \mathbb{O}\left(-0.40674665\ n + 0.11766902\ \log((40+n)!) - \log(40!)\right) \\
g(n) &= \mathbb{O}\left(n + \log(n!)\right) \\
g(n) &= \mathbb{O}\left(n + \log(n^n)\right) \\
g(n) &= \mathbb{O}\left(n + n\log n\right) \\
g(n) &= \mathbb{O}\left(n\log n\right).
\end{aligned}
$$

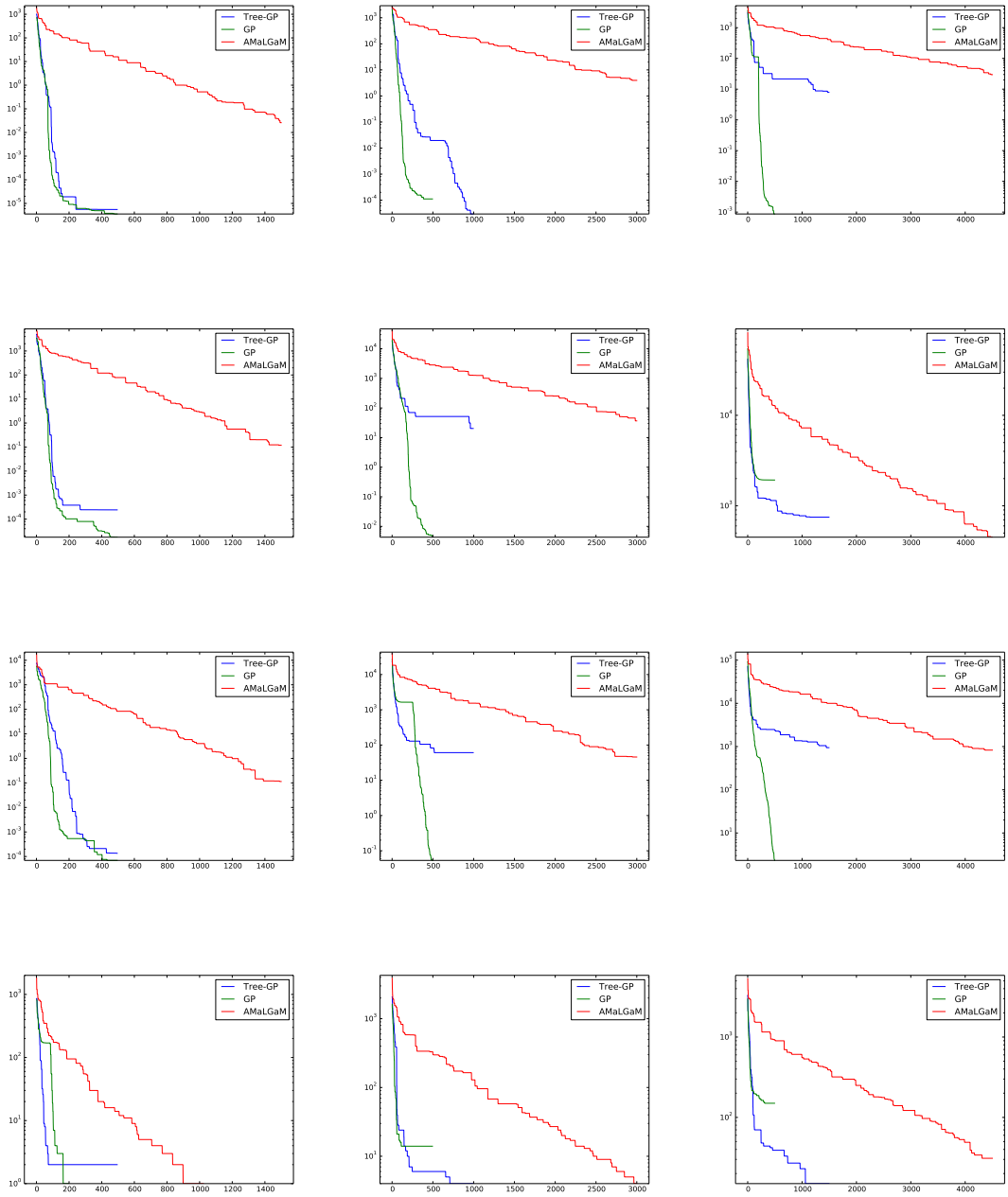
An nth root function has also been fitted to the TREE-GP algorithm times. It is given by

$$f(x) = 0.10250573x^{1/4.16360816} - 0.1282584$$

where x is the current evaluation starting at 1. It has a MAE of 0.01189511. The summation over x from 1 to n which gives the total run time is given by

$$g(n) = \sum_{x=1}^{n} f(x).$$

The total run times of the algorithm predicted from the fitted functions are shown in figure 6.4. They are calculated by using the summations above. The predicted algorithm overhead per function evaluations is also shown in figure 6.5. The times are in seconds and do not include the function evaluations themselves.


Tree-GP
GP
AMaLGaM

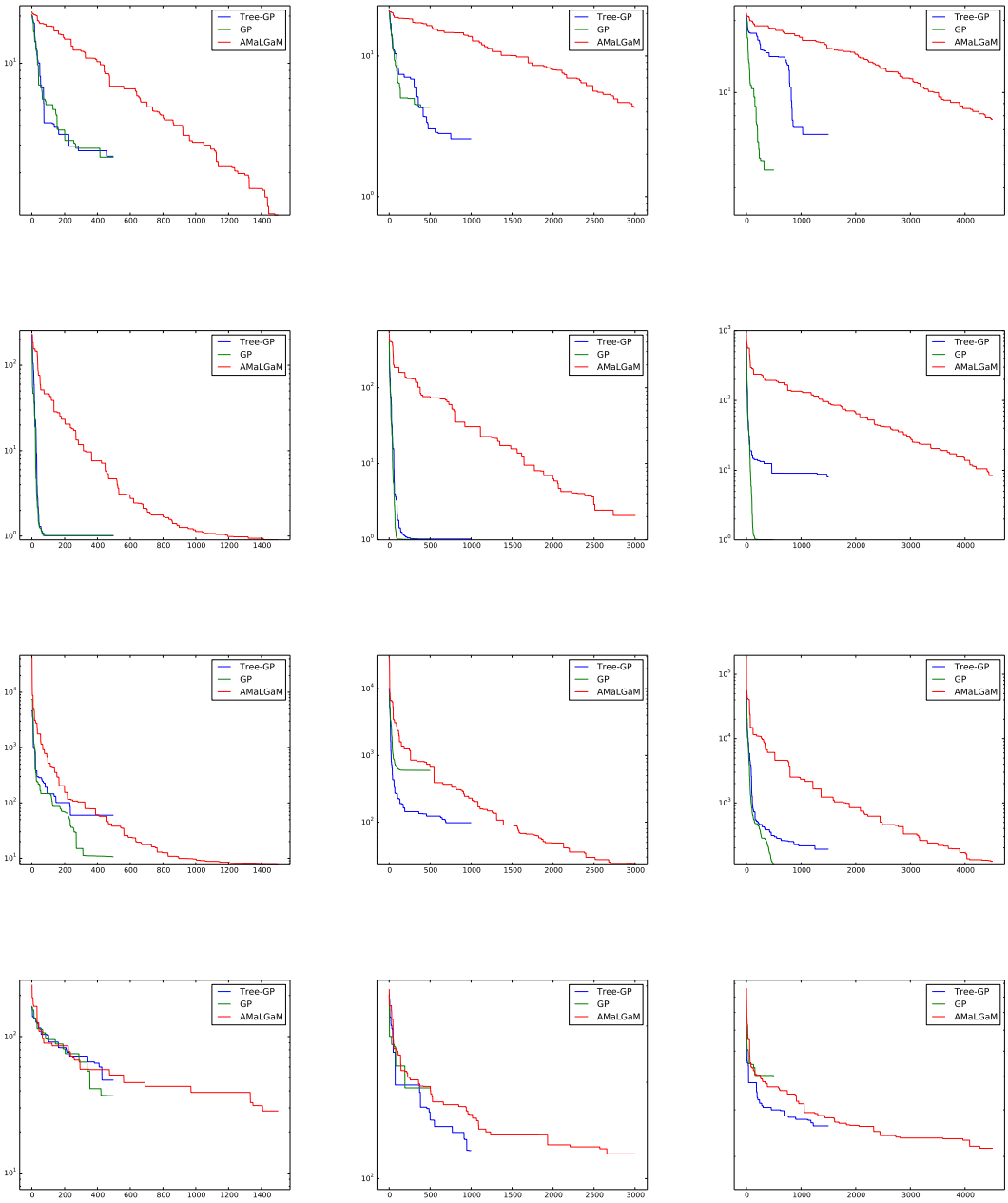


Figure 6.1: From top to bottom, left to right: the average minimum evaluation value of f_1 to f_{24} found per iteration. Evaluation values are averages over 5 runs. Each row contains the graphs of the same function in three different dimensionalities: 10, 20 and 30.

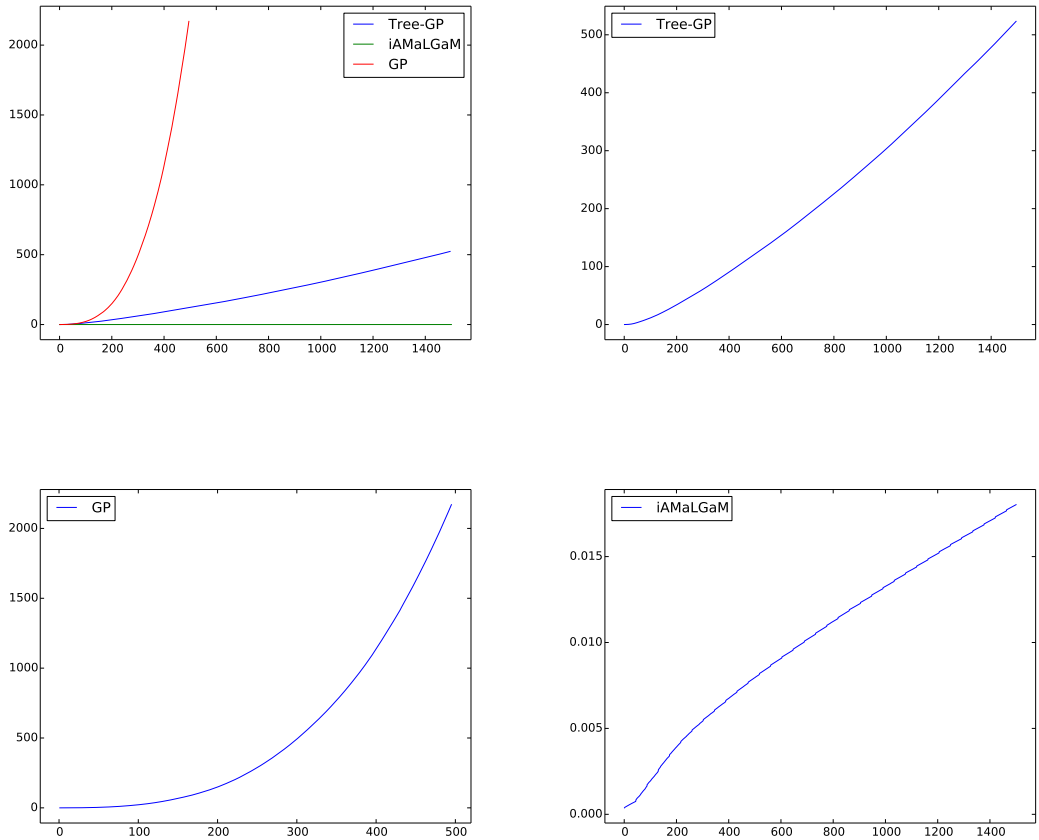


Figure 6.2: The average time taken in seconds since the start of the algorithm at each function evaluation. The times shown are not including the function evaluations. The values are averages over 5 runs. The first graphs shows all algorithms so you can compare their scaling behavior. The latter graphs show the times of each individual algorithm.

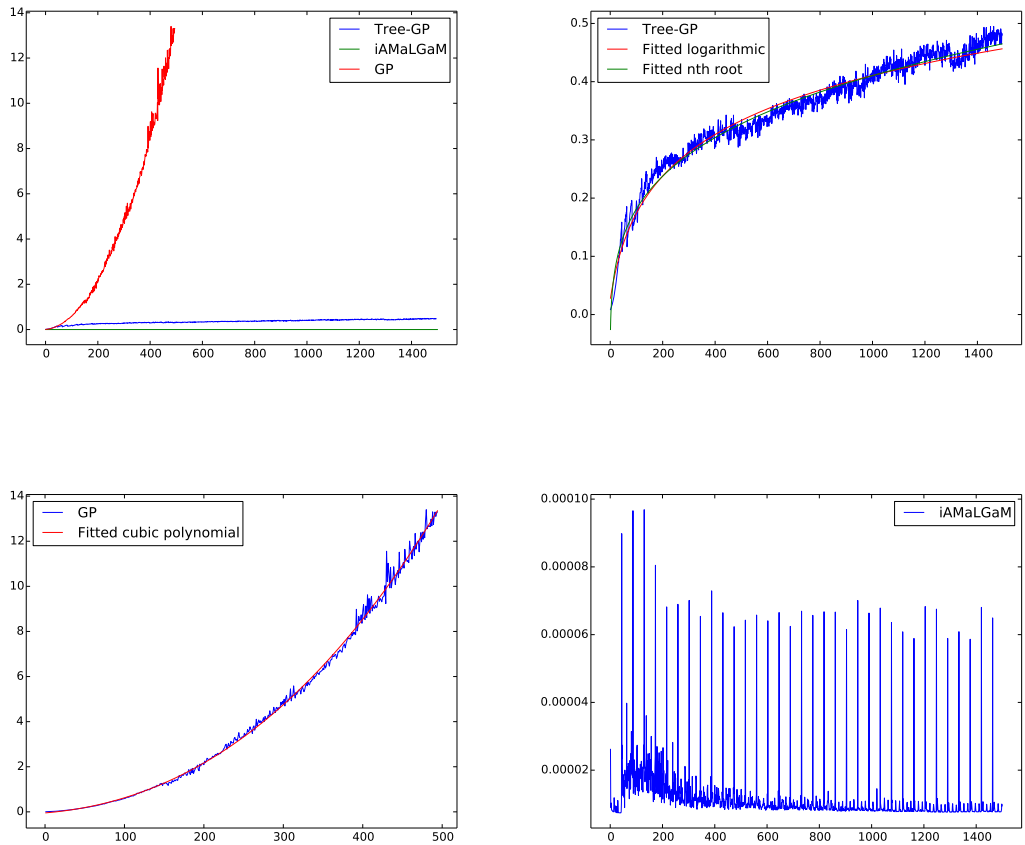


Figure 6.3: The average algorithm overhead in seconds per function evaluation. The times shown are not including the function evaluation. The values are averages over 5 runs. The first graphs shows all algorithms so you can compare their scaling behavior. The latter graphs show the times of each individual algorithm. Note the cubic function that has been fitted to the GP algorithm times and the logarithmic function that has been fitted to the Tree-GP algorithm.

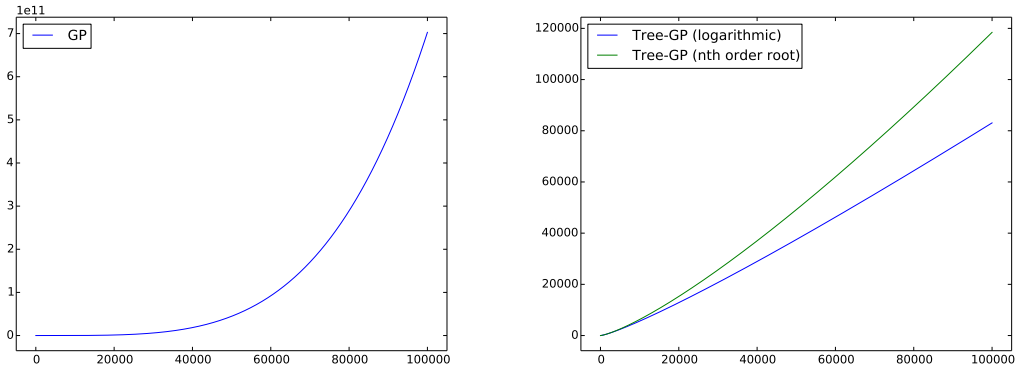


Figure 6.4: The predicted time taken in seconds since the start of the algorithm at each function evaluation. The values are calculated from the fitted functions. The times shown are not including the function evaluations.

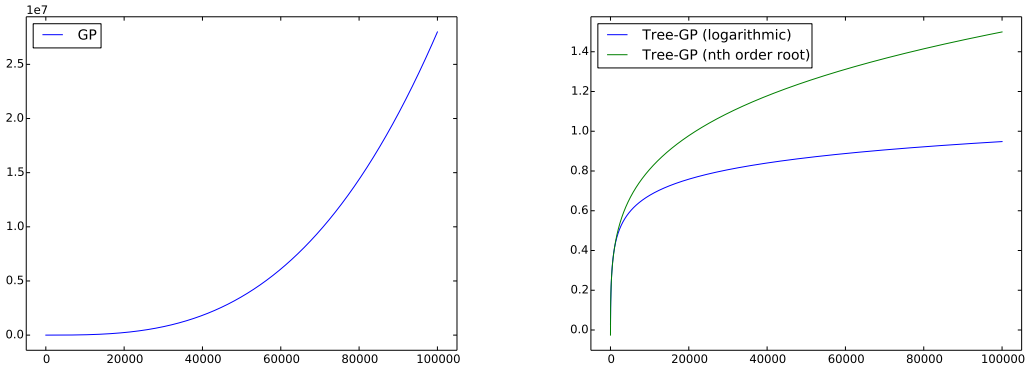


Figure 6.5: The predicted algorithm overhead in seconds per function evaluation according to the fitted functions. The times shown are not including the function evaluations. Note the difference between the fitted logarithmic and the fitted nth root function in their predictions.

Chapter 7

Discussion

In this chapter we will discuss the results of experiments that we did in the previous chapter. We will do this by comparing the performance of the three algorithms that we tested, GP, Tree-GP and AMaLGaM.

7.1 GP performance

GP is our basic algorithm, which is why we will analyze its performance first. This algorithm does not use the mixture model to scale well in the total amount of function evaluations. It uses just a plain Gaussian process regression model as the posterior distribution. The performance of this algorithm is very important, as it provides us with a baseline for our scaling improvements. The GP algorithm shows us how well our acquisition criteria function CMPVR performs in combination with a Gaussian process regression model.

Looking at the performance of the GP algorithm against the number of function evaluations we can indeed say that it performs very well in this regard. The graphs show that GP descends very quickly with the very few function evaluations that it was given. AMaLGaM only beats GP for the very last problem, and even there they are basically on par. Because we assume our loss function to be very expensive, we can afford to spend more time looking for good candidate points to evaluate. It seems like that investment does pay off, as we find very good values with very little function evaluations. GP only finds the minimum for problem 4 (shifted step function) with $D = 10$, but this is actually quite an accomplishment given that only 500 function evaluations were allowed.

For almost all problems we see the same behavior in the value graph, first it descends quickly and then it slowly converges to its final value. This behavior is probably due to the decay of exploration constant, which will cause the algorithm to exploit more and more towards the end. The graph of problem 3 (shifted and rotated ellipsoid function) with $D = 20$ clearly shows the advantage of resetting this constant so that the algorithm explores again. The algorithm is stuck in a local minimum for quite some evaluations. This causes the exploration constant to be reset, resulting in exploration of the search space. This in turn causes the algorithm to find new local minima, which it then exploits. We can clearly see the quick descend that follows. With problem 7 (shifted and rotated Rosenbrock's function), $D = 20$ we are not so lucky however, as the algorithm does not find a better minimum even after resetting the exploration constant.

Though the performance versus the number of function evaluations of the GP algorithm is very good, the total run time of the algorithm is very dramatic. As we can see in the results of the timing experiments in figure 6.3, the time taken per function evaluation quickly increases.

The fitted function is a cubic polynomial, which gives the algorithm a total run time order of $\mathbb{O}(n^4)$ as we expected. If we would have run the tests with $D = 30$ and 1500 evaluations, we could have expected the last iteration of the algorithm to take about 177 seconds or 2 minutes 57 seconds. The full run time of the algorithm would be a whopping 78,060 seconds, or 21 hours 41 minutes, excluding the function evaluations themselves. That is quite a lot of overhead for just 1500 function evaluations. We can see in figure 6.4 that at a hundred thousand function evaluations, the full run time of the algorithm is about 7×10^{11} seconds, which is more than 22182 years! These numbers are of course only true if the extrapolation of the fitted polynomial holds. This is fair assumption however, because the fitted function fits very well to the timing results.

7.2 Tree-GP performance

The Tree-GP algorithm improves upon the GP algorithm by trying to make it scalable. It is the main algorithm of this thesis, and so its performance is very important. We will compare the performance and timing experiment results with the GP algorithm, which is our baseline.

If we examine the value performance of the Tree-GP algorithm, we can see that the results for $D = 10$ are very similar to the GP algorithm, which are very good. On the higher dimensional problems, Tree-GP performs slightly worse than GP. This might be because of the curse of dimensionality: the higher the dimensionality of the model, the less are the differences in distances between evaluation points. This is bad for the mixture model, because it relies on models being local. If models are not local anymore, one global model performs a lot better than multiple overlapping global models.

Still, the overall results are very good and almost always better than the results of AMaLGaM. The minimum values found by Tree-GP quickly decrease at first and then converge towards the end. The convergence behavior is the same as GP, and is due to the decaying exploration constant. The algorithm is sometimes stuck, which can be seen by the plateaus in the graphs. It often starts descending again after the exploration constant has been reset, but this may take some time. We can clearly see that the algorithm is stuck in the second problem (the shifted ellipsoid function) with $D = 20$. At the end of the graph the algorithm gets stuck and after quite a few evaluations it starts descending again. Even though it takes some time, without our exploration constant reset it would probably not have started descending again.

Because we test both the shifted and the shifted and rotation ellipsoid problems (problems 2 and 3), we can see how well the algorithm handles rotation. It looks like the algorithm handles rotation of the loss function well. There is not much difference in performance between the two problems. We can not draw hard conclusions from these results however, because the ellipsoid function is a fairly easy test function.

Let us finally examine how well our algorithm does in the timing results. It is after all one of the main goals of this thesis to scale existing Bayesian optimization algorithms in the number of function evaluations. If we look at the total time taken by the algorithm it is immediately clear that our algorithm performs much better than GP does. In the top-left graph of figure 6.2 we can see how well the algorithms perform relative to each other. Whereas GP shows cubic behavior, Tree-GP seems to be linear in the number of function evaluations. Zooming in on the times of just GP in the top-right graph, we can see that the times are slightly worse than linear. This is probably because doing queries on the vantage-point tree becomes more expensive over time. In the worst case, the algorithm walks through all previously evaluated points in the vantage-point tree to find the models that are closest to a new evaluation point. This worst case scenario would take quadratic time. Fortunately however, we can see that the behavior tends

more toward linear time than quadratic time.

The top-right graph of figure 6.3 shows the overhead of the algorithm per function evaluation. It confirms that the overhead is not constant which would give a linear total run time. The overhead seems to be either a logarithmic or an nth root function. Because we were not sure which one it was, we fitted both of them. Both functions give a very good fit, which is why we included both extrapolations in figure 6.5. As we can see, the difference is not that large, even at a 100,000 function evaluations.

If we examine the total run time in figure 6.2, we can see that at 1500 evaluations, the total overhead is about 520 seconds or 8 minutes 40 seconds. This is not much overhead on an evaluation function that takes 5 minutes to compute. The total evaluation time is then $1500 \times 5 = 7500$ minutes or 5 days and 5 hours. In fact, it is only about 0.12%. For a function that takes an hour the overhead is only 0.0019%. One that takes a minute has 0.58% overhead. An evaluation functions that takes a second to compute would have 6.9% overhead, which is significantly more. Moreover, you could evaluate that function a lot more than 1500 times in a reasonable amount of time. In a day you could evaluate that function 86,400 times. Let's say that we evaluate it 100,000 times, which would take about 27 hours 47 minutes. Let us assume that our worst prediction, namely the fitted nth root function is correct. In figure 6.4 we can see that our algorithm takes about 120,000 seconds. The overhead amounts to 120% of the time taken to evaluate the loss function 100,000 times.

We can clearly see that our algorithm is not very suited for loss functions that only takes a second. Tree-GP was designed for loss functions that are expensive, and so should be used for those functions. Let us look at a more expensive loss function that takes a minute. Evaluating it 100,000 times would take more than 3 months, which is a pretty long time. If we would do it however, the total overhead of the algorithm would still be 120,000 seconds or 33 hours and 20 minutes. This is 2%, which is very reasonable. Evaluation functions that take longer than one minute have even less overhead and evaluating them 100,000 times becomes even more infeasible. As a conclusion we can say that our algorithm is very suited for evaluation functions that take a minute or longer.

7.3 AMaLGaM performance

Finally we tested with the AMaLGaM algorithm. This algorithm has not been made for loss functions that are expensive to evaluate. Therefore it can permit doing a lot more function evaluations. Testing AMaLGaM is important because it puts our algorithm in perspective. We can compare the value performance and timings with Tree-GP in order to make a better decision when to use which algorithm.

In each of the value performance graphs we can clearly see the steady exponential decline of AMaLGaM. The algorithm is very robust and predictable, which is a good thing. It models its population with a density probability distribution. Because the population always contains the best values found, the model provides valuable information about which regions to exploit. It does not compute a best point to evaluate from this model, but simply samples the probability distribution. This gives it the steady decline in the minimum value found. The algorithm does not do very well on the last problem, the shifted and rotated Rastrigin's function. The algorithm is often stuck for quite some time. All algorithms basically perform equally bad on this problem. Rastrigin's function is the hardest problem of the test suite because it is not separable, it has a lot of local minima and it is very hard to model. In the end, the algorithms just jump from one local minimum to the next. Modeling Rastrigin's function is very hard because it has a large periodic component. None of the algorithms that we tested can model periodicity in the loss

function, which is why they all perform badly.

Time wise AMALGAM performs very well. All of experiments were done in almost no time at all. As we can see in figure 6.2 it takes AMALGAM about 19 milliseconds to do 1500 evaluations. In the graph of the times per function evaluation in figure 6.3 we can see that after some time the overhead per evaluation goes to about 10 microseconds with peaks of 65 microseconds. At the peaks the probability distribution is recalculated from the current population. When the probability distribution has been calculated the algorithm samples about 43 points from that distribution and evaluates them. This makes the average overhead per evaluation about 11.3 microseconds. A million evaluations only has about 11.3 seconds of overhead. This makes AMALGAM a very good general purpose algorithm, because the evaluation function is likely to take a lot more time than the algorithm itself.

Chapter 8

Conclusions and Future Work

At the start of this thesis we had two goals. The first one was to construct an algorithm that first explores the search space and gradually exploits it. The second goal was to make the algorithm scalable in the total number of function evaluations. These two goals are evaluated in this chapter. We will also give some examples of future work that can be done.

8.1 Exploration versus exploitation

We have created a numerical optimization algorithm called Tree-GP that focuses on optimizing expensive evaluation functions. It uses Bayesian optimization with Gaussian process regression to model the search space. In order to find a point to evaluate in this model of the search space, we have created an acquisition criteria function called the Cumulative Mean Probability to Variance Ratio (CMPVR). This function combines both the predicted mean and variance of the model. The weight of each is determined by the exploration constant. By varying this constant we determine whether to explore the search space or exploit it. We have created an exponential decay scheme to gradually change this constant over time. If no improvement is found within a number of evaluations, the constant is reset to its initial value and the decaying scheme is run again.

We have tested our algorithm with a test suite created by the IEEE CEC designed specifically for expensive function optimization. The CMPVR function and the exploration constant decaying scheme turned out to work well. The Tree-GP algorithm outperformed AMaLGaM on most test problems. Tree-GP was also almost as good as GP, which is a variant of the Tree-GP algorithm that does not use the mixture model. The value graphs show that Tree-GP rapidly descends at first and later on converges more slowly towards the minimum. The algorithm gets stuck sometimes, but due to the exploration constant reset starts descending again after some evaluations. The algorithm seems to handle rotations of the loss functions well.

8.2 Scalability

Our second goal was to make the algorithm scalable in the total number of function evaluations. In order to do this we made a mixture model of multiple local Gaussian process regression models. We have put these models in a vantage-point tree in order to find the local models close to a new point. The vantage-point tree also handles splitting local models. It does this with a custom partitioning function that allows a local model to zoom in on an interesting region. The local

models are kept small and only a few local models are updated. This makes the algorithm more scalable than an algorithm that uses one global Gaussian process regression model. We also did not have to give up much to make the algorithm scalable, because TREE-GP performs almost as well as GP, which is a version of the algorithm without the mixture model.

We have done some extensive timing experiments on GP, TREE-GP and AMALGAM. Whereas GP uses cubic time in the total amount of function evaluations, TREE-GP uses almost linear time. This means that we have made a big improvement in run time. The cubic behavior of GP quickly becomes unacceptable. TREE-GP has been designed for expensive evaluation functions and it delivers on that promise. If an evaluation function takes a minute and we let the algorithm do 100,000 evaluations, the overhead of the algorithm is only 2%. For evaluation functions that take longer than a minute the overhead is even less. For evaluations functions that take much less than a minute, for example only a second, AMALGAM is more suited. AMALGAM is very robust and fast, but takes longer to converge toward the function minimum. AMALGAM has almost no overhead, and so can be used for evaluation functions that are cheap to evaluate. The overhead of AMALGAM becomes constant over time and is on average about 11.3 microseconds per function evaluation.

8.3 Future work

Let us finally see what the possibilities are for future work. An improvement on our work can be accomplished by improving the performance of TREE-GP. This is always useful, because the less overhead an optimization algorithm has, the more time can be spend evaluating the loss function. A significant performance gain can be obtained by parallelizing TREE-GP. The current algorithm minimized the acquisition criteria function by doing derivative-based optimization starting in one hundred random points. This is now done sequentially, but can easily be done in parallel. The local models that are stored in the vantage-point tree can also be recalculated in parallel when a new point is added to them. You can even evaluate multiple promising points of the acquisition criteria function in parallel and update the model afterward. Or finding a good model of the search space while evaluating the loss function.

Another possible performance improvement is using another regression algorithm for the local models. Gaussian process regression is very time consuming and there are other regression algorithm that can be used. In order to use our acquisition criteria function it needs to not only predict a value, but also the variance in that value.

Further research may also focus on the effects of varying the exploration constant. We have shown some figures of evaluation points of the sphere problem for different values of the exploration constant. An investigation can be done about the effects of various schemes to alter the constant over the number of function evaluations.

Another possibility for an investigation is to see which acquisition criteria function performs best under various circumstances. We have given some of the popular acquisition criteria functions that are given in the literature. More acquisition criteria functions can be found and a comparison between them would be very valuable.

Chapter 9

Acknowledgments

First of all I want to thank my supervisor Dirk Thierens for assisting me in making this master thesis. He provided useful tips, remarks, knowledge and constructive criticism during the whole process. I also want to thank Linda van der Gaag for examining my work. Furthermore I want to thank my friends, my two brothers and my parents for supporting me throughout the entire process. I want to thank Thomas Bayes for creating solutions to solve the problem of Bayesian inference, which is the very basis of Bayesian optimization. I also want to thank Carl Friedrich Gauss for writing down the formula of the wonderful Gaussian distribution, which is of course the basis for Gaussian process regression. Furthermore I want to thank Bjarne Stroustrup for creating the most powerful programming language in existence today, namely C++, which also quite possibly has the worst syntax conceivable and requires an awful lot of boilerplate code. Finally I want to thank the Chinese for inventing Kung fu, which gave me the much needed distraction from my work on this thesis.

Appendices

Appendix A

Result values

The tables below show the minimum / median / maximum values that have been reached by the tested algorithms over the 5 runs that were done. Note that the GP algorithm has been tested with a maximum number of evaluations of 500.

	Tree-GP		
	10D (500 evs)	20D (1000 evs)	30D (1500 evs)
Function 1	1.833×10^{-6} / 5.543×10^{-6} / 1.891×10^{-5}	3.922×10^{-6} / 2.958×10^{-5} / 7.067×10^{-2}	3.978 / 7.869 / 5.595×10^{2}
Function 2	3.944×10^{-5} / 2.417×10^{-4} / 7.140×10^{-4}	9.895 / 2.033×10^{1} / 1.015×10^{2}	3.061×10^{2} / 7.493×10^{2} / 8.679×10^{2}
Function 3	3.497×10^{-6} / 1.349×10^{-4} / 2.237	5.773 / 6.077×10^{1} / 9.756×10^{1}	7.452×10^{2} / 9.437×10^{2} / 1.332×10^{3}
Function 4	1.000 / 2.000 / 3.000	1.000 / 4.000 / 6.000	1.000×10^{1} / 1.500×10^{1} / 3.300×10^{1}
Function 5	2.033 / 2.546 / 3.510	1.995 / 2.576 / 1.467×10^{1}	5.892 / 6.684 / 1.374×10^{1}
Function 6	9.661×10^{-1} / 1.009 / 1.027	9.678×10^{-1} / 1.017 / 1.081	3.798 / 8.011 / 9.103
Function 7	4.823×10^{1} / 6.002×10^{1} / 1.459×10^{2}	5.322×10^{1} / 9.814×10^{1} / 1.631×10^{2}	1.444×10^{2} / 1.905×10^{2} / 3.784×10^{2}
Function 8	4.583×10^{1} / 4.792×10^{1} / 7.986×10^{1}	5.641×10^{1} / 1.224×10^{2} / 1.394×10^{2}	2.293×10^{2} / 2.609×10^{2} / 2.847×10^{2}

	AmalGam		
	10D (500 evs)	20D (1000 evs)	30D (1500 evs)
Function 1	7.323 / 1.587×10^{1} / 3.074×10^{1}	1.534×10^{2} / 1.679×10^{2} / 1.804×10^{2}	3.007×10^{2} / 4.095×10^{2} / 4.654×10^{2}
Function 2	4.645×10^{1} / 7.821×10^{1} / 1.449×10^{2}	9.520×10^{2} / 1.266×10^{3} / 1.667×10^{3}	4.408×10^{3} / 4.983×10^{3} / 5.441×10^{3}
Function 3	5.569×10^{1} / 8.333×10^{1} / 1.550×10^{2}	1.302×10^{3} / 1.536×10^{3} / 2.567×10^{3}	7.179×10^{3} / 9.997×10^{3} / 1.085×10^{4}
Function 4	9.000 / 1.400×10^{1} / 2.600×10^{1}	5.900×10^{1} / 1.290×10^{2} / 1.720×10^{2}	3.130×10^{2} / 3.900×10^{2} / 4.310×10^{2}
Function 5	6.441 / 7.135 / 9.551	1.269×10^{1} / 1.379×10^{1} / 1.414×10^{1}	1.499×10^{1} / 1.529×10^{1} / 1.590×10^{1}
Function 6	4.474 / 4.683 / 7.151	2.252×10^{1} / 3.076×10^{1} / 4.586×10^{1}	8.947×10^{1} / 9.389×10^{1} / 1.180×10^{2}
Function 7	3.304×10^{1} / 3.799×10^{1} / 6.040×10^{1}	1.689×10^{2} / 2.293×10^{2} / 3.064×10^{2}	8.639×10^{2} / 1.232×10^{3} / 1.728×10^{3}
Function 8	3.283×10^{1} / 5.222×10^{1} / 6.519×10^{1}	1.400×10^{2} / 1.587×10^{2} / 1.712×10^{2}	2.518×10^{2} / 2.800×10^{2} / 2.854×10^{2}

	GP		
	10D (500 evs)	20D (500 evs)	30D (500 evs)
Function 1	5.876×10^{-7} / 3.577×10^{-6} / 7.081×10^{-5}	4.918×10^{-5} / 1.117×10^{-4} / 4.715×10^{-3}	5.507×10^{-4} / 8.690×10^{-4} / 3.856
Function 2	1.037×10^{-5} / 1.768×10^{-5} / 2.548×10^{-5}	2.701×10^{-3} / 4.316×10^{-3} / 2.296	2.221×10^{-1} / 1.927×10^{3} / 5.171×10^{3}
Function 3	2.264×10^{-5} / 6.935×10^{-5} / 1.062×10^{-3}	2.104×10^{-3} / 5.321×10^{-2} / 1.652×10^{1}	3.024×10^{-1} / 2.323 / 1.367×10^{4}
Function 4	0.000 / 0.000 / 3.000	4.000 / 1.400×10^{1} / 4.650×10^{2}	2.000×10^{1} / 1.500×10^{2} / 5.510×10^{2}
Function 5	2.306 / 2.512 / 3.052	4.000 / 4.327 / 4.945	4.111 / 4.742 / 1.759×10^{1}
Function 6	7.683×10^{-1} / 1.012 / 1.017	9.094×10^{-1} / 9.920×10^{-1} / 1.005	9.728×10^{-1} / 1.005 / 9.385×10^{1}
Function 7	5.103 / 1.067×10^{1} / 6.453×10^{1}	1.896×10^{1} / 5.974×10^{2} / 2.536×10^{3}	4.819×10^{1} / 1.094×10^{2} / 8.819×10^{2}
Function 8	1.877×10^{1} / 3.682×10^{1} / 6.475×10^{1}	1.548×10^{2} / 1.921×10^{2} / 2.254×10^{2}	3.405×10^{2} / 4.033×10^{2} / 4.485×10^{2}

Bibliography

[1] Milton Abramowitz and Irene A Stegun. *Handbook of mathematical functions: with formulas, graphs, and mathematical tables.* Number 55. Courier Dover Publications, 1972.

[2] Mattias Björkman and Kenneth Holmström. Global optimization of costly nonconvex functions using radial basis functions. *Optimization and Engineering*, 1(4):373–397, 2000.

[3] Peter AN Bosman, Jörn Grahl, and Dirk Thierens. Benchmarking parameter-free AMaLGaM on functions with and without noise. *Evolutionary computation*, 21(3):445–469, 2013.

[4] George EP Box and Norman R Draper. *Empirical model-building and response surfaces.* John Wiley & Sons, 1987.

[5] H-M Gutmann. A radial basis function method for global optimization. *Journal of Global Optimization*, 19(3):201–227, 2001.

[6] Matthew D Hoffman, Eric Brochu, and Nando de Freitas. Portfolio allocation for Bayesian optimization. In *Uncertainty in Artificial Intelligence*, pages 327–336, 2011.

[7] Takeo Ishikawa and Michio Matsunami. An optimization method based on radial basis function. *IEEE Transactions on Magnetics*, 33(2):1868–1871, 1997.

[8] Donald R Jones. A taxonomy of global optimization methods based on response surfaces. *Journal of global optimization*, 21(4):345–383, 2001.

[9] Donald R Jones, Matthias Schonlau, and William J Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[10] André I Khuri and John A Cornell. *Response surfaces: designs and analyses*, volume 152. CRC press, 1996.

[11] Response Surface Methodology. Process and product optimization using designed experiments. *Myers, RH and Montgomery, DC, John Wiely & Sons, New York*, 1995.

[12] Jonas Mockus. *Bayesian approach to global optimization.* Springer, 1989.

[13] Michael JD Powell. A direct search optimization method that models the objective and constraint functions by linear interpolation. In *Advances in optimization and numerical analysis*, pages 51–67. Springer, 1994.

[14] Michael JD Powell. UOBYQA: unconstrained optimization by quadratic approximation. *Mathematical Programming*, 92(3):555–582, 2002.

[15] Michael JD Powell. On trust region methods for unconstrained minimization without derivatives. *Mathematical programming*, 97(3):605–623, 2003.

[16] Rommel G Regis and Christine A Shoemaker. Constrained global optimization of expensive black box functions using radial basis functions. *Journal of Global Optimization*, 31(1):153–171, 2005.

[17] Timothy W Simpson, Timothy M Mauery, John J Korte, and Farrokh Mistree. Comparison of response surface and kriging models for multidisciplinary design optimization. *American Institute of Aeronautics and Astronautics*, 98(7):1–16, 1998.

[18] Christopher KI Williams and Carl Edward Rasmussen. *Gaussian processes for machine learning*, volume 2. 2006.

[19] Peter N Yianilos. Data structures and algorithms for nearest neighbor search in general metric spaces. In *Proceedings of the fourth annual ACM-SIAM Symposium on Discrete algorithms*, pages 311–321. Society for Industrial and Applied Mathematics, 1993.