VILNIUS UNIVERSITY

ALBERTAS GIMBUTAS

NONCONVEX OPTIMIZATION ALGORITHM WITH A NEW BI-CRITERIA SELECTION OF POTENTIAL SIMPLICES USING AN ESTIMATE OF LIPSCHITZ CONSTANT

Doctoral dissertation
Physical sciences, informatics (09P)

Vilnius, 2018

The dissertation work was carried out at Vilnius University from 2013 to 2017.

Scientific supervisor:

prof. habil. dr. Antanas Žilinskas (Vilnius University, physical sciences, informatics – 09P)

VILNIAUS UNIVERSITETAS

ALBERTAS GIMBUTAS

NEIŠKILIOJO OPTIMIZAVIMO ALGORITMAS SU NAUJU BIKRITERINIU POTENCIALIŲJŲ SIMPLEKSŲ IŠRINKIMU NAUDOJANT LIPŠICO KONSTANTOS ĮVERTĮ

Daktaro disertacija
Fiziniai mokslai, informatika (09P)

Vilnius, 2018

Disertacija rengta 2013–2017 metais Vilniaus universitete.

Mokslinis vadovas:

prof. habil. dr. Antanas Žilinskas (Vilniaus universitetas, fiziniai mokslai, informatika – 09P)

Abstract

In this thesis, DIRECT (DIviding RECTangles) type algorithms based on Lipschitz objective function models with unknown Lipschitz constant, which are often applied for practical black-box optimization problems, are considered. The main goal of this thesis is set - to propose a global optimization algorithm for Lipschitz functions with unknown Lipschitz constants in order to efficiently spend potentially expensive function evaluations. For the considered class of algorithms, a new simplicial optimization algorithm LIBRE (LIpschitz Bound Rough Estimation) is proposed, which is based on DISIMPL (DIviding SIMPLices) algorithms. The novelty of the proposed algorithm is that a single estimate of the Lipschitz constant is used instead of a set to select potential simplices for division. Experimental analysis is conducted and the competitiveness of the proposed algorithm to other best algorithms from this algorithm class is shown. In addition, various modifications of the LIBRE algorithm are proposed and experimentally investigated; the impact of the tightness of the surogate Lipschitz bounds on the efficiency of the LIBRE algorithm is demonstrated. Moreover, a strategy to generalize Lipschitzian global optimization algorithms for multi-objective problems is proposed. LIBRE algorithm is generalized for multi-objective problems by applying the proposed strategy and then experimentally investigated.

Santrauka

Disertacijoje nagrinėjami DIRECT (DIviding RECTangles) tipo Lipšico tikslo funkcijos modeliais su nežinoma Lipšico konstanta pagrįsti optimizavimo algoritmai, kurie yra dažnai taikomi praktinių „juodosios dėžės“ tipo uždavinių sprendimui. To pagrindu išsikeltas pagrindinis disertacijos tikslas - pasiūlyti globaliojo optimizavimo Lipšico klasės su nežinoma Lipšico konstanta algoritmą, efektyviai išnaudojantį potencialiai brangius funkcijų skaičiavimus. Nagrinėjamai algoritmų klasei yra pasiūlytas naujas simpleksinis algoritmas LIBRE (LIpschitz Bound Rough Estimation), paremtas ankstesniais DISIMPL (DIviding SIMPLices) algoritmais. Pasiūlytojo algoritmo naujumas tame, kad naudojamas vienas Lipšico konstantos įvertis vietoje aibės Lipšico konstantų, parenkant potencialiuosius simpleksus dalinimui. Atlikta eksperimentinė analizė parodė pasiūlytojo algoritmo konkurencingumą su kitais geriausiais šios klasės algoritmais. Taip pat, yra eksperimentiškai ištiriamos įvairios LIBRE algoritmo modifikacijos; parodoma surogatinių Lipšico rėžių griežtumo įtaka pasiūlytojo algoritmo efektyvumui. Be to, yra pasiūlyta strategija, kaip Lipšico globaliojo optimizavimo algoritmus apibendrinti daugelio kriterijų optimizavimo uždavinių sprendimui. LIBRE algoritmas apibendrintas daugiakriteriniams uždaviniams, pritaikius pasiūlytąją strategiją, ir eksperimentiškai ištirtas.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Notation

D	feasible region
X	a set of trial points
Y	a set of objective function values obtained
Y^{Pareto}	discrete approximation of the Pareto front
$\overline{D}$	unit-hypercube got from scaling D
d	number of dimensions in feasible region, i.e. $D \in \mathbb{R}^d$
n	number of objectives
m	number of trials performed in an arbitrary iteration of an algorithm
$\boldsymbol{x}$	vector
$f(\cdot)$	objective function
$\boldsymbol{f}$	a vector of objective functions
f^m_{min}	lowest observed function value after m trials
$g(\cdot)$	function of Lipschitz lower bounds over objective function
$\boldsymbol{g}$	a vector of Lipschitzian minorants (one for each objective function)
$\boldsymbol{x}^*$	global minimizer
f^*	global minimum
$\mathbb{R}$	set of real numbers
L_p	the Lipschitz constant with respect to p-norm distance
L	the Lipschitz constant with respect to Euclidean distance (same as L_2)
$\widetilde{L}$	any kind of estimate of the Lipschitz constant L_2
S	a set of simplices
$V(\cdot)$	a set of vertices
$\Delta(\cdot)$	a diameter
$\lvert S \rvert$	size of a set S, i.e. number of elements in a set S
$\lVert \cdot \rVert_p$	distance with respect to p-norm, i.e. $\lVert \boldsymbol{x} \rVert_p = \left(\sum_{i=1}^{d} \lvert x_i \rvert^p \right)^{1/p}$
$\lVert \cdot \rVert$	Euclidean distance, also denoted as $\lVert \cdot \rVert_2$

Chapter 1

Introduction

1.1 Research Context

There are many engineering problems that pose the need to obtain a global optimum of a certain objective function with respect to a combination of parameters in the search space. Using the globally optimal solution instead of locally optimal ones has recongnized advantages, although obtaining such a solution is a much harder task, and, consequently, requires additional effort and resources. The field of global optimization deals with this complicated task and covers the related theory and implementation of algorithms. In many practical problems the analytical expression of the objective function is unavailable and evaluating such black-box objective function involves executing an expensive numerical experiment. Therefore in general not much can be assumed about the objective function, especially as the potential presence of multiple local minima must be taken into account. Due to the high cost of trials it is important to obtain a good approximation of the best possible objective function value within a minimal possible budget of trials. This goal justifies the effort to develop new global optimization algorithms of higher efficiency.

One of the natural assumptions to be made is that a limited change in input variables leads to a limited change in the objective function value. This is the central assumption in the Lipschitz global optimization formalized mathematically as the Lipschitz condition for the objective function. This assumption greatly fa-

cilitates the investigation and development of algorithms, e. g. proofs of convergence properties can be derived and guarantees of solution accuracy become possible. Moreover, the Lipschitz condition is usually exploited by deterministic algorithms, which have an advantage in comparison to stochastic ones that require multiple runs. However, the acknowledged problem of the algorithms in this class is the absence of the Lipschitz constant in realistic scenarios, which is either ignored or tackled by different estimation techniques. The univariate case of Lipschitzian global optimization is very well covered in the literature both from the theoretical and practical point of view. However, the multivariate case continues to pose challenges. The ongoing research in this direction exploits the branch-and-bound techniques in combination with a set of Lipschitz constants considered simultaneously.

1.2 Relevance of the Study

In many applied optimization problems the objective function evaluations are expensive due to the involved time-consuming numerical simulations. It is therefore important to increase the efficiency of the optimization algorithms in terms of the consumed trials. Among the Lipschitzian optimization algorithms those using an adaptive estimate of the Lipschitz constant have proved to be competitive. However, using any given estimate of the Lipschitz constant instead of the true one does not provide any guarantees regarding the precision of the obtained solution. On the other hand, a class of Lipschitzian optimization methods that considers a set of admissible Lipschitz constants at once instead of a single estimate allows to guarantee that the global optimum will be approximated with the required precision in a finite number of trials. It is therefore worth considering a combination of both approaches to create the algorithms that both guarantee a satisfactory solution and save expensive trials.

It seems natural that using tighter Lipschitz bounds translates into a more precise objective function model. The more appropriate to the actual situation the model is, the more informed decision about the next trial location the optimization algorithm can make. Consequently, it is important to investigate the effect that the tightness of the Lipschitz bounds has on the effectiveness of considered

optimization algorithms.

The multi-objective optimization problems are usually approached using metaheuristic algorithms. Algorithms in this category are randomized, therefore they require repeated runs and provide no guarantee that the obtained solutions are truly Pareto-optimal with a certain precision. On the other hand, it was demonstrated in the literature that deterministic algorithms are more efficient for simple optimization problems than the metaheuristic ones. This motivates to further explore the potential of the deterministic methods to expand the set of problems they can be applied to in order to obtain algorithms with performance similar to their metaheuristic counterparts and at the same time avoid their drawbacks.

1.3 Objectives and Tasks of the Thesis

The most general goal of this thesis is to propose Lipschitzian optimization algorithms without the Lipschitz constant, ensuring efficient usage of objective function evaluations.

The following specific objectives were raised:

1. Propose an algorithm combining the strengths of DIRECT-type Lipschitzian optimization algorithms with algorithms using estimates of the Lipschitz constant in order to ensure more efficient performance in terms of the number of objective function evaluations.
2. Investigate the effect that increasing the tightness of the surrogate Lipschitz bounds has on the optimization process of complex multi-modal objective functions.
3. Propose a way to adapt Lipschitz optimization algorithms for multi-objective problems in order to approach the efficiency of the state-of-the-art genetic algorithms.

The following tasks were identified:

1. Identify the strengths of DIRECT-type simplicial Lipschitz optimization algorithms and suggest an algorithm modification which would preserve these strengths and would use adaptively estimated Lipschitz constant estimate to obtain an improved optimization algorithm.
2. Test the performance of the suggested and popular algorithms using complex objective function classes generated artificially.
3. Experimentally compare the effect that strategies of defining the objective function surrogate Lipschitz bounds of different tightness within a simplex have on the performance of optimization algorithms.
4. Generalize the suggested algorithm to the multi-objective problems and experimentally evaluate the efficiency of the obtained algorithm in comparison to the state-of-the-art genetic algorithm.

1.4 Scientific Novelty and Results

In the context of single-objective simplicial optimization of hard multi-modal global optimization problems it was suggested to combine two criteria to select subregions for further partitioning. The first criterion is the minimum value of the surrogate Lipschitz bound for the simplex and the second one is the simplex diameter. An algorithm based on the proposed idea was experimentally demonstrated to outperform other considered algorithms in the worst-case scenario for hard global optimization problems.

For the implementation of the proposed algorithm several strategies regarding the computing of the surrogate Lipschitz bound for the simplex were considered. The main differences between them lie in the tightness of the reduced surrogate Lipschitz bounds for the simplex and the associated computational complexity. It was experimentally determined that increasing the surrogate Lipschitz bound tightness does not necessarily result in the improved algorithm efficiency in terms of the number of objective function evaluations.

A generalization strategy transforming a single-objective partition-based Lipschitzian global optimization algorithm into a multi-objective optimization algorithm was suggested, based on the ideas considered previously in one-step

worst-case optimal bi-objective Lipschitzian optimization algorithm. The suggested strategy was applied to generalize the proposed algorithm to the multi-objective case. The resulting multi-objective optimization algorithm was shown to compare similarly to the popular genetic algorithm for low-dimensional problems.

1.5 Statements Defended

The statements defended in this thesis are:

1. The proposed global optimization algorithm Libre is more efficient with respect to the number of objective function evaluations than other popular alternatives in the worst-case scenario for complex global optimization problems.
2. Using stricter surogate Lipschitz bounds does not necessarily improve the efficiency of Direct-type global optimization algorithms with respect to the number of objective function evaluations.
3. The multi-objective version of the proposed algorithm performs similarly to a popular genetic algorithm NSGA-II with respect to the number of function evaluations for low-dimensional multi-objective optimization problems.

1.6 Approbation of the Thesis Results

The main findings of this thesis are published in peer reviewed periodicals *Jaunųjų mokslininkų darbai*, *Journal of global optimization* and in peer reviewed proceedings of the international conferences *Numerical Computations: Theory and Algorithms* and *International Workshop on Optimization and Learning: Challenges and Applications*. In addition, the results of the thesis were presented in 9 international and national conferences. Full references of the publications and titles of the presentations are provided in the Chapter Publications by the Author.

Chapter 2

A Review of Global Optimization of Lipschitz-Continuous Functions

This chapter presents a review of the origins and recent advancements in the field of global optimization of functions complying with the Lipschitz objective function model. The fundamental assumption of this model is the bounded slope of the objective function with respect to the decision variables. The function might be given either as a black-box computer code computing values at the given points, or explicitly. This mathematical model has attracted a lot of attention due to its suitability for theoretical investigation as well as applicability to practical problems.

2.1 General Global Optimization Problem

Let a continuous function $f(\boldsymbol{x}) : D \subset \mathbb{R}^d \rightarrow \mathbb{R}$ be defined, such that D is a bounded, robust set. Then the general global optimization problem is to find

$$\min_{x \in D} f(\boldsymbol{x}), \tag{2.1}$$

$$D = [\boldsymbol{l}, \boldsymbol{u}] = \{\boldsymbol{x} \in \mathbb{R}^d : \, l_i \leqslant x_i \leqslant u_i, \, i = 1, ..., d\}. \tag{2.2}$$

The set D is called the feasible region (feasible set). The function $f(\boldsymbol{x})$ is called the objective function. It is customary to refer to $f(\boldsymbol{x})$ as a black-box if its val-

ues can be retrieved at arbitrary points of D, but explicit functional form of the function is unknown. The source of difficulty in the general problem statement (2.1) is the potential presence of multiple local minima of $f(x)$ in D, and the consequence is that local optimization techniques are not adequate for solving it.

The set of solutions to (2.1) is denoted $X^* = \{x^* : f(x^*) = \min_{x \in D} f(x)\}$. Each point $x^* \in X^*$ is called a global minimizer and value $f^* = f(x^*)$, $x^* \in X^*$ is called the global minimum.

The problem (2.1) is generally difficult to solve precisely, therefore numerical approximations are employed. For example, it might be sufficient to find a point located close enough to one of the points in X^*. Alternatively, the approximation of f^* might be sought within some predefined tolerance ϵ:

$$\widehat{x}^* \in D : f(\widehat{x}^*) \leqslant f^* + \epsilon,\ \epsilon > 0. \tag{2.3}$$

An ϵ-convergent algorithm will find an ϵ-optimal point $\widehat{x}^*$ in a finite number of trials.

Various specific cases of the general problem (2.1) have been addressed in the literature [69]. They differ in the assumptions concerning D and f, allowing to propose case-specific algorithms.

2.2 Lipschitz Global Optimization Problem

When the objective function $f(x)$ is assumed to be Lipschitz-continuous on D, i. e.

$$|f(x_1) - f(x_2)| \leqslant L\|x_1 - x_2\|, \forall x_1, x_2 \in D, \tag{2.4}$$

the specification of the general global optimization problem (2.1) is obtained. (2.4) is called the Lipschitz condition and function $f(x)$ is said to satisfy it. In this case constant L is called the Lipschitz constant. Usually the Euclidean norm is used for $\|\cdot\|$, but different ones have also been found in the literature [64].

The model (2.4) is very general. Condition (2.4) is fundamental in order to obtain deterministic estimates of the global minimum in a finite number of trials. Without it the global optimum could only be estimated using probabilistic tools or an infinite everywhere dense sequence of trial points [69]. It is important to note that for applied black-box optimization problems where close to none analytical information is available about the objective function, the assumption (2.4) is often useful and plausible. On the other hand, this assumption is attractive from the theoretical point of view and a solid volume of research has been devoted to investigation of theoretical properties of the Lipschitzian optimization algorithms.

Lipschitzian optimization algorithms often take advantage of the fact that lower bound on the objective function value might be established using condition (2.4):

$$g(\boldsymbol{z}) = \max_{\boldsymbol{x} \in X} \{f(\boldsymbol{x}) - L\|\boldsymbol{x}, \boldsymbol{z}\|\}, \boldsymbol{z} \in D, \tag{2.5}$$

where X is the set of already performed trial points. The bound allows to design stopping conditions and eliminate regions of D as unpromising.

It was proved in [64] that when an analytical expression for a function is known, it is possible to obtain the Lipschitz constant using this equation:

$$L_p = \sup\{\|\nabla f(\boldsymbol{x})\|_p : \boldsymbol{x} \in D\}, \tag{2.6}$$

$$\nabla f(\boldsymbol{x}) = \left(\frac{\partial f}{\partial x_1}, \ldots, \frac{\partial f}{\partial x_d}\right), \tag{2.7}$$

$\nabla f(\boldsymbol{x})$ is the gradient of $f(\cdot)$ and $1/p + 1/q = 1, 1 \leqslant p, q \leqslant \infty$. Varying the value of p, constants for various norms can be obtained.

However, the main disadvantageous aspect of the Lipschitz model is the absence of the constant L in practice. The minimal suitable constant L is usually not known, although its valid overestimates can often be obtained. Imprecise values of L mean that the lower bound (2.5) is unreliable. In addition, L might vary considerably over D and a fixed value would decrease the efficiency of algorithms or increase the risk of missing the global minimizer entirely. Besides using a fixed value of L, provided in advance, the literature suggests approaches to estimate it dynamically as the algorithm progresses or to consider at the same time a set of possible values for L.

2.3 Lipschitz Global Optimization Algorithms

In this section various algorithms for solving the global optimization problem specified for Lipschitz-continuous functions are discussed. For an in-depth presentation of the field, books [29, 36, 69, 75, 83] are recommended.

2.3.1 Global Optimization Algorithm

Let us consider a sequence of points $\boldsymbol{x}_i, i = 1, \ldots, m$, $\boldsymbol{x}_i \in D$, and define the record value as

$$f_{min}^m = \min_{i=1,\ldots,m} f(\boldsymbol{x}_i). \tag{2.8}$$

A global optimization algorithm could be defined as a generator of this sequence, ensuring that

$$f_{min}^m \to f^*, m \to \infty. \tag{2.9}$$

Value f_{min}^m is accepted as an approximation of the global minimum and a point $\boldsymbol{x}_j, j \in \{1, \ldots, m\}$, such that $f(\boldsymbol{x}_j) = f_{min}^m$, is accepted as an approximation of the global minimizer.

The points in the sequence $\boldsymbol{x}_i, y_i = f(\boldsymbol{x}_i), i = 1, \ldots, m$ are referred to as trials or objective function evaluations. When $D \subset \mathbb{R}$, additional indexing of the trial points is useful. When the points $\boldsymbol{x}_i, i = 1, \ldots, m$, are arranged in the increasing order, they are denoted $\boldsymbol{x}_i^m$ and satisfy $\boldsymbol{x}_1^m \leqslant \boldsymbol{x}_2^m \leqslant \cdots \leqslant \boldsymbol{x}_m^m$.

The sequence $\boldsymbol{x}_i, i = 1, \ldots, m$, might either be fixed even before the global optimization algorithm starts to run, or determined dynamically when the algorithm executes. In the first case such a global optimization algorithm is called passive, while in the second case it is called adaptive.

2.3.2 The Source of the Lipschitz Constant

Global optimization algorithms targeting Lipschitz-continuous objective functions can be categorized on the basis of the adapted approach of retrieving the Lipschitz constant [76]. Algorithms relying on a straightforward assumption of an a priori known Lipschitz constant are considered first. Then, some important cases in which the Lipschitz constant is estimated dynamically are examined. Finally, algorithms considering a set of all feasible Lipchitz constants are presented.

2.3.3 The Lipschitz Constant is Known

Let us start with the algorithms making the assumption that the Lipschitz constant L is known in advance. It should be noted that these algorithms are more of a theoretical interest, as the constant is usually unknown in practical situations.

2.3.3.1 Univariate Algorithms

Suppose that $D \subset \mathbb{R}$ and a global optimization algorithm has already produced trials x_i^m, $f(x_i^m) = y_i^m$, $i = 1, \ldots, m$, $x_1^m \leqslant x_2^m \leqslant \cdots \leqslant x_m^m$. Suppose further that a fixed value of L is provided. Then a Lipschitzian minorant of $f(x)$, $x \in D$ is equal to

$$g_m(x) = \max_{i=1,\ldots,m} (y_i^m - L|x - x_i^m|). \tag{2.10}$$

Figure 2.1 provides an illustration for (2.10).

Alternative names for (2.10) appear in the literature, like the lower Lipschitz bound for $f(x)$ or the saw-tooth cover of $f(x)$. The endpoints of each subinterval $[x_i^m, x_{i+1}^m]$ are referred to as the basis points of a particular tooth of the Lipschitz bound. Each tooth is characterized by its height, achieved at the peak point $\hat{x}_i$. The teeth of the lower bound $g(x)$ provide information regarding the regions of D, where it is still possible for x^* to exist, called the region of indeterminacy.

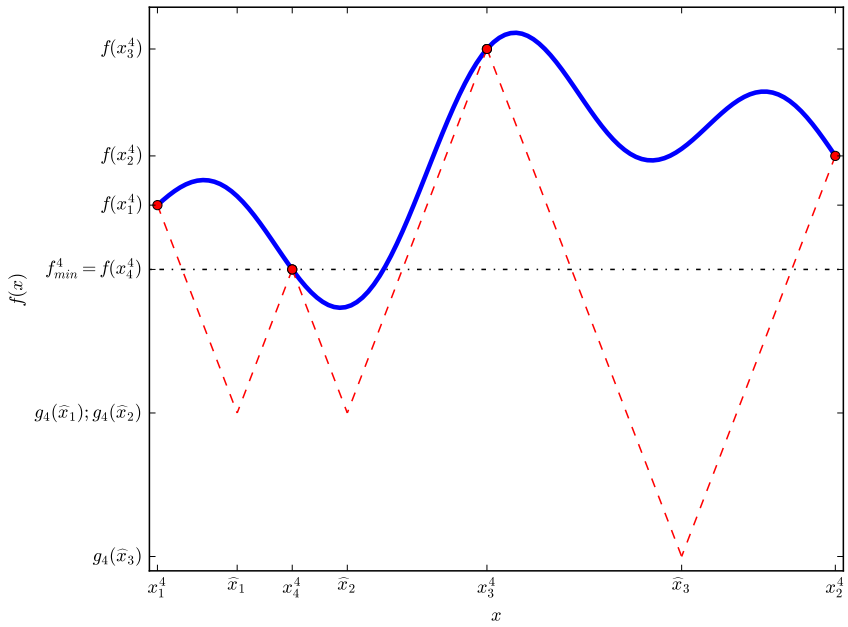


Figure 2.1: The objective function and its Lipschitzian minorant over a closed interval.

Intervals where the tooth is completely above the current best value f_{min}^m can safely be discarded from analysis.

A lot of univariate Lipschitzian optimization algorithms estimating the global minimum with accuracy ϵ have been reported in the literature. The most trivial, but also the least efficient one, is the uniform grid search. For $D = [\boldsymbol{l}, \boldsymbol{u}]$, the points $\boldsymbol{x}_i^m$ are equally spaced: $\boldsymbol{x}_i^m = l + \frac{i(\boldsymbol{u}-\boldsymbol{l})}{m}$, $i = 1, \ldots, m$. Their locations do not depend on the algorithm progress, so this is a passive strategy. It provides a guarantee that ϵ-convergence is reached in $m = \frac{L(\boldsymbol{u}-\boldsymbol{l})}{2\epsilon}$ steps. The performance of the uniform grid search algorithm is considered as a benchmark for the worst-case situation.

At the other end of the spectrum, exclusively in the univariate case there exists an ideal benchmark algorithm, which was called the best possible algorithm, proposed by Danilin [13, 14]. Since this algorithm uses a global minimum value f^* as an input parameter, it is practically not applicable. The algorithm produces a saw-tooth cover $g_m(\boldsymbol{x})$ where each tooth has a height equal to $f^* - \epsilon$ and requires the minimum possible number of trials for an estimate of f^* with ϵ precision. The algorithm is not designed to solve new optimization

problems, but helps to evaluate other algorithms with respect to trials required for ϵ-convergence.

It was established in [38, 84] that in case $f(\boldsymbol{x}) = c, \boldsymbol{x} \in D$, both uniform grid search and the best possible algorithms make the same number of function evaluations before ϵ-convergence is guaranteed. This is considered the worst-case scenario and for it there does not exist a sequential algorithm better than the passive one. However, sequential algorithms perform better than the passive ones on average.

Among globally ϵ-convergent sequential optimization algorithms, the Pijavskij-Shubert algorithm [67, 81] has been the most widely studied. The central idea of the algorithm is that at every iteration a point minimizing the current Lipschitzian minorant is found and the new objective function evaluation is made there. The pseudocode of the Pijavskij-Shubert algorithm is provided in Algorithm 1. The first step of the algorithm is to evaluate objective function values at the bounds of the feasible region:

$$\boldsymbol{x}_1 = \boldsymbol{l},\ \boldsymbol{x}_2 = \boldsymbol{u},\ y_1 = f(\boldsymbol{x}_1),\ y_2 = f(\boldsymbol{x}_2). \tag{2.11}$$

Further the main loop of the algorithm begins, which is terminated when the condition $f^m_{min} \leqslant f^* + \epsilon$ is satisfied. In each iteration a point $\boldsymbol{x}_{k+1}$ is selected to assess the objective function value

$$g(\boldsymbol{x}) = \max_{i \in \{1,\dots,k\}} (y_i - L|\boldsymbol{x} - \boldsymbol{x}_i|), \tag{2.12}$$

$$\boldsymbol{x}_{k+1} = \arg\min_{x \in [\boldsymbol{l},\boldsymbol{u}]} g(\boldsymbol{x}). \tag{2.13}$$

The lower Lipschitz bound (2.12) gets stricter with each evaluation of the objective function. Value of (2.12) for the first six iterations of the Pijavskij-Shubert algorithm is shown in Figure 2.2 .

The one-step worst-case optimality of the Pijavskij-Shubert algorithm was demonstrated in [37, 38, 84], in the sense that the next function evaluation maximally decreases the height of the saw-tooth cover. It is known that this algorithm could require 4 times the number of trials used by the best-possible algorithm to ensure ϵ-convergence [31]. For the worst-case function, the Pijavskij-Shubert

Algorithm 1: Pijavskij-Shubert

Input l - lower bound of the feasible region, u - upper bound of the feasible region, $f(\cdot)$ - objective function, L - Lipschitz constant of the objective function, ϵ - precision of the solution.
Function *PijavskijShubert*(l, u, f, L, ϵ):
    $x_1 = l,\ x_2 = u$
    $y_1 = f(x_1),\ y_2 = f(x_2),\ m = 2$
    $f^m_{min} = \min_{i \in \{1,\dots,m\}} y_i$
    $g(x) = \max_{i \in \{1,\dots,m\}} \{y_i - L|x - x_i|\}$
    while $f^m_{min} - \min_{x \in [l,u]} g(x) > \epsilon$ **do**
        $m = m + 1$
        $x_m = \arg\min_{x \in [l,u]} g(x)$
        $y_m = f(x_m)$
        $f^m_{min} = \min_{i \in \{1,\dots,m\}} y_i$
    return f^m_{min}

algorithm takes twice the number of trials required for a uniform grid search. The algorithm becomes similar to the uniform grid search when L is very large [40]. The underlying assumption of a fixed constant L does not allow to adaptively decrease it and speed up the convergence when the basin of the global minimum is found.

Attempts have been made to improve the efficiency of the Pijavskij-Shubert algorithm in the cases similar to the constant-function scenario. For example, approaches in [71, 85] allow for a mixture of a passive search strategy with the Pijavskij-Shubert algorithm. The passive strategy predetermines a set of potential trial locations and then the sequence $x_i, i = 1, \dots, m$, generated by the Pijavskij-Shubert algorithm, is replaced with a sequence of the closest points on the predermined grid. Another successful modification [30] suggested using the Pijavskij-Shubert algorithm to discard sugregions of D where improvement was not possible and exploring the rest of the search space via an approximated version of the best possible algorithm. It was demonstrated in [29] that for small ϵ the algorithm was close to the best possible in terms of efficiency.

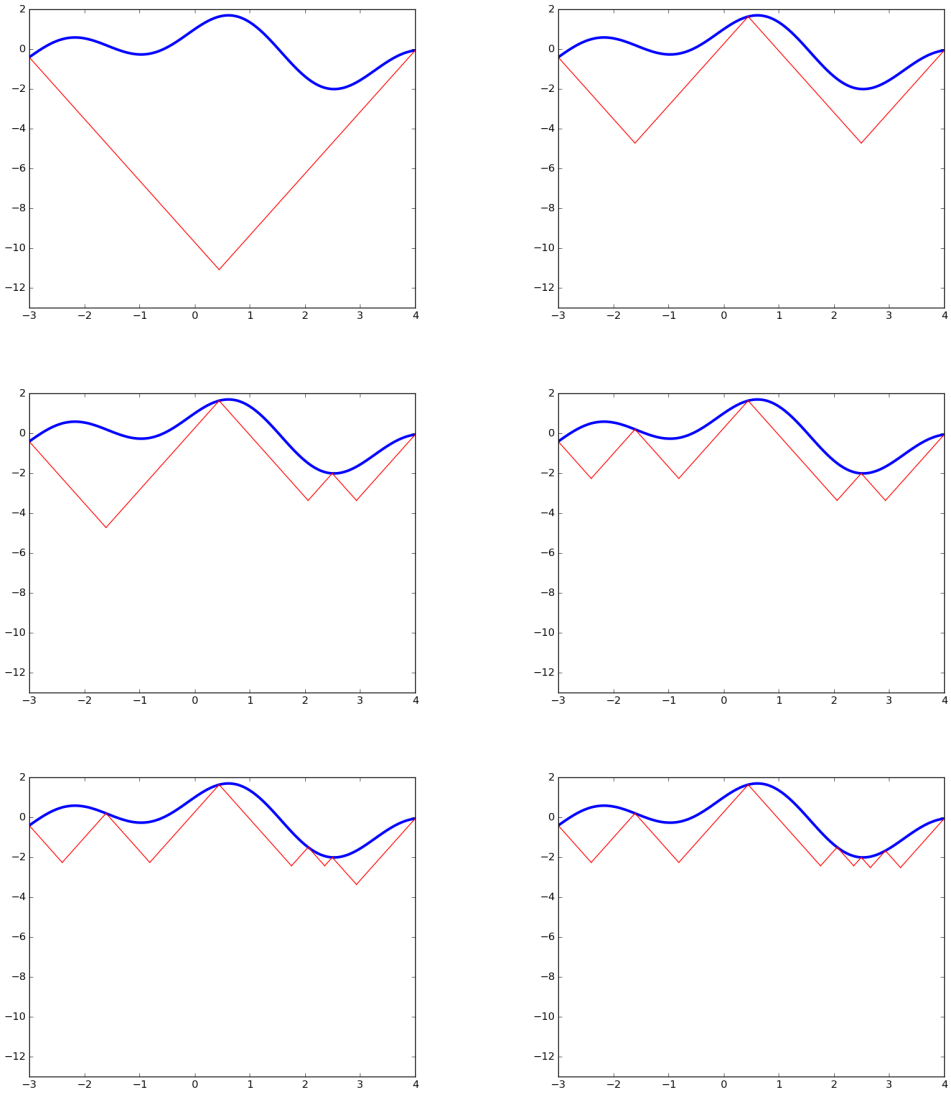


Figure 2.2: Illustration of the first six iterations of the Pijavskij-Shubert algorithm.

2.3.3.2 Multivariate Algorithms

In the considerably harder case $D \subset \mathbb{R}^d, d > 1$ there have been several directions of development.

The seminal paper by Pijavskij [67] proposed an approach where a multivariate optimization problem is replaced by a large series of univariate optimization problems. The variables are ordered and optimization is constructed as a set of

nested univariate problems corresponding to the ordered variables. The problem is solved at the innermost level using the univariate Pijavskij algorithm, when the values of higher-level variables are held fixed. This approach is mainly of a theoretical interest and has not received much attention after its introduction as the number of problems to be solved grows multiplicatively with problem dimension d.

Another approach to cast the multivariate problem as a univariate one was with the help of Peano curves [7, 82]. The feasible region D is covered by a one-dimensional curve of a desired granularity and the univariate optimization algorithm is executed over this curve. The downside of the approach is the increased number of local minima, arising because the Peano curve crosses the region of attraction of a particular optimum several times.

Besides the ideas of reusing single-objective optimization algorithms, there have been a number of solutions to generalize the original univariate Pijavskij algorithm for higher dimensions [3, 39, 56, 58, 67, 86]. All these algorithms suggested a way to construct a Lipschitzian minorant of $f(\boldsymbol{x})$, analogous to the saw-tooth cover in the univariate case. The teeth in this case are cones, and nonlinear systems of equations have to be solved at each iteration of an optimization algorithm in order to find their intersections and locations of all possible local minima of the minorant. Various simplifications of this complex process were the subject of research in this category. Algorithms in this category grow markedly slower with the increase of trials.

In comparison to other approaches in the multivariate case, the most promising one was based on the divide-and-conquer principle, embodied by the branch-and-bound framework [33, 36]. Algorithms defined within this framework share the same general outline: the optimization problem over the feasible region is partitioned into a set of subproblems over smaller subregions, for each subregion a lower bound of $f(\boldsymbol{x})$ is computed based on the bounds the next subproblem to be solved is selected and further divided. However, the particular definition of these steps differs [23, 27, 57, 68]. This approach is faster and easier to implement than the previous ones that use a single Lipschitzian minorant covering the whole feasible region [29].

The subproblems in the branch-and-bound framework might be defined over

subregions of different shape, for example, both rectangular [69, 74] and simplicial [11, 65] subsets were used. The simplicial subsets are preferable for certain types of problems, where the feasible region may be expressed as a moderate number of simplices, e. g. when $f(\boldsymbol{x})$ has symmetries or in case of linear constraints [9, 34, 35, 61, 92].

2.3.4 The Lipschitz Constant is Estimated

Having an accurately known Lipschitz constant in advance is questionably a realistic scenario. A far more practically justified approach would be to obtain a dynamic estimate of the Lipschitz constant L in the course of execution of the optimization algorithm, e. g. [48, 80]. Some algorithms maintain a single global estimate of L pertaining to the whole feasible region D, e. g. [36, 69, 83]. Although this is a more reasonable approach than to expect an a priori known constant L, it has a disadvantage that the obtained estimate is too general and is likely to misrepresent the rate of change of $f(\boldsymbol{x})$ over some of the subregions of D. This can occur when the objective function has a great slope in certain areas, while in others it is relatively flat. Such inaccuracy in estamation of L might lead to severe problems for an optimization process. When the estimate is too low, it is possible to miss the global minimum entirely. On the contrary, a too high estimate of L implies a complex structure of $f(\boldsymbol{x})$, characterized by marked oscillations and narrow regions of attraction of the minima. In such a situation the optimization algorithm would converge towards the global minimum slowly. To improve the correspondence of an estimate of L to the actual rate of change of the function over a certain region, approaches using local estimates of L were suggested [45, 55, 60, 73, 83]. In particular an approach called local tuning [45, 73, 79, 83] allows to adaptively estimate L over D, balancing local and global information during the search.

2.3.5 The Lipschitz Constant Varies over a Set

2.3.5.1 The DIRECT Algorithm

An original idea of simultaneously considering a set of Lipschitz constants instead of a single one first appeared in the DIRECT algorithm [40]. The rationale behind DIRECT was to provide a generalization of the Pijavskij-Shubert algorithm to the multivariate case, with the advantage of removing the limitation of using a fixed Lipschitz constant. The specific way in which a set of possible Lipschitz constants could be considered proved very successful in ensuring adaptive balance between local and global search, and sparked a lot of scientific interest. As its further advantages the applicability to black-box problems, a low number of parameters and satisfactory performance could be mentioned all of which enabled to use DIRECT to address modern engineering problems [2, 4, 8, 12, 32, 72].

The algorithm follows the general outline of the branch-and-bound techniques so that the original global optimization problem is divided into smaller subproblems, which are iteratively divided further until the maximum allowed number of function evaluations is reached. Initially the optimization problem is defined over a unit hyper-rectangle. After each iteration a number of new hyper-rectangles appear by dividing those from the previous iteration. An arrangement of all hyper-rectangles on a two-dimensional plane is proposed, allowing to spot the ones with the lowest Lipschitz bound with respect to at least one possible Lipschitz constant. This is achieved when the x coordinate of the plane corresponds to the half-diagonal of the hyper-rectangle and the y coordinate corresponds to the $f(\boldsymbol{x})$ value at the center of the hyper-rectangle (Figure 2.3).

The selection of the next set of hyper-rectangles to be divided takes place based on their lower Lipschitz bound, expressed in terms of potential optimality. That is, a hyper-rectangle S_j is called potentially optimal if there exists a constant $\widetilde{L} > 0$ and some $\epsilon > 0$ satisfying

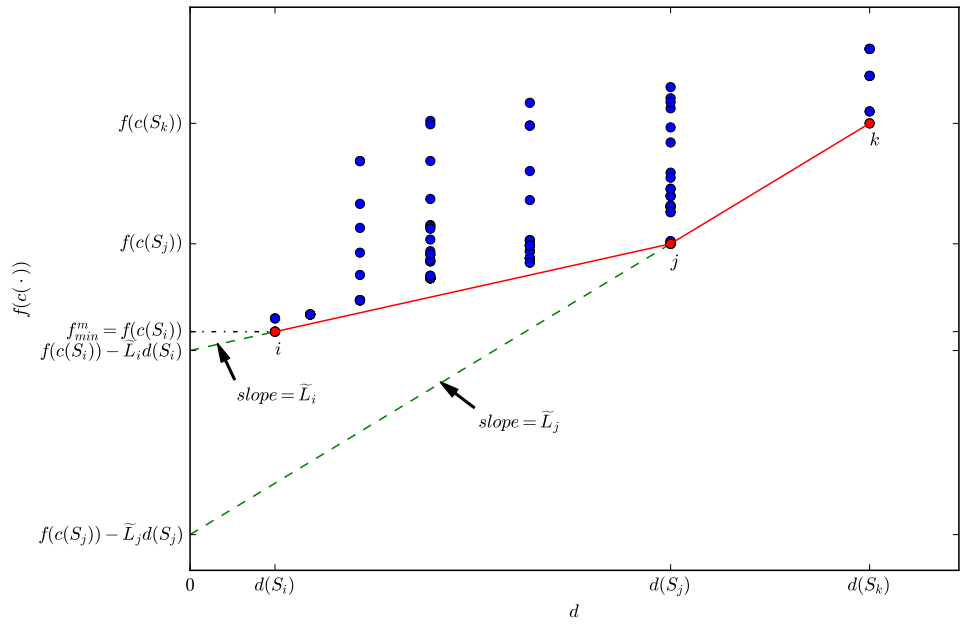


Figure 2.3: Illustration of the selection of hyper-rectangles in DIRECT.

$$f(\boldsymbol{c}(S_j)) - \widetilde{L}\, d(S_j) \leqslant f(\boldsymbol{c}(S_j)) - \widetilde{L} d(S_i),\ \forall S_i \in S, \tag{2.14}$$

$$f(\boldsymbol{c}(S_j)) - \widetilde{L}\, d(S_j) \leqslant f_{min} - \epsilon |f_{min}|, \tag{2.15}$$

where $c(\cdot)$ is the center point of the respective hyper-rectangle, $d(\cdot)$ is the distance from the center point to one of hyper-rectangle's vertices, f_{min} is the best observed objective function value and ϵ is a positive constant (usually $\epsilon = 0.0001$ value is used). The first condition ensures that with at least one constant $\widetilde{L}$ the lower bound of $f(\boldsymbol{x})$ over hyper-rectangle S_j is the lowest with respect to all other hyper-rectangles. The second condition requires a non-trivial improvement over the current record value. When a line with slope $\widetilde{L}$ is extended from a point j in Figure 2.3 towards the y axis, the intersection occurs at a height equal to the lower Lipschitz bound of $f(\boldsymbol{x})$ over hyper-rectangle S_j. For a potentially-optimal hyper-rectangle the rest of the points happen to be located above this line. All potentially optimal hyper-rectangles could be collected by identifying the points on the lower-right side of the convex hull of the point set in Figure 2.3. The selected potentially-optimal hyper-rectangles are divided employing a particular trisection scheme that ensures the grouping of points along the x axis in

the illustration. For more detailed description of the DIRECT algorithm refer to [6, 40, 65].

A great advantage of DIRECT is the dynamic balancing of local and global search as hyper-rectangles of various sizes are selected and divided at each iteration. This way refinement takes place both on a large and small scale. Furthermore, there is no need to have an analytical objective function definition for the algorithm to operate. On the other hand, the notable disadvantage is that the quick identification of the approximate location of the global solution is followed by a very slow refinement of its neighbourhood. The smallest hyper-rectangles that are near the global minimum are massively selected by conditions (2.14) and (2.15) which makes the algorithm inefficient as it progresses.

2.3.5.2 Modifications of the DIRECT Algorithm

The main weaknesses of the algorithm were pointed out in various subsequent modifications, e. g. [10, 19, 52]. Some of them suggested certain structural changes within the DIRECT itself. For example, in the version of DIRECT called aggressive [2] the requirement of potential optimality was loosened and at each iteration hyper-rectangles characterized by the lowest objective function value were selected. That way the number of divided hyper-rectangles could be increased. Another version, called locally-biased DIRECT [22], suggested reducing the number of horizontal groups in Figure 2.3 by applying a coarser grouping mechanism as well as limiting the number of selected hyper-rectangles per group. Thus the number of divided hyper-rectangles was decreased and more focus was given to refinement on a smaller scale.

As an alternative approach, an idea to combine a DIRECT-like base solver with an external mechanism for filtering the hyper-rectangles by size according to improvement monitored during the operation of the solver was initially raised in [77]. Instead of the original DIRECT as a base solver, its certain modification was used, employing an adaptive diagonal partitioning strategy. It was suggested to define two modes of operation - local and global - and allow the solver to work only with larger hyper-rectangles in the global mode and smaller ones in the local mode. Switching to the global mode happened when no considerable improvement occurred for some iterations, ensuring that the partitioning

of small hyper-rectangles does not consume too many resources. Similar dual-mode ideas were then extended to the case of Lipschitz simplicial optimization in [63].

Further developments in the direction of applying a level-based approach in combination with DIRECT appeared for two [50] and more levels [49]. The hyper-rectangles are taken into consideration based on the current level or scale of the algorithm. Starting from the coarsest scale, where all hyper-rectangles are visible to the algorithm, on finer scales a percentage of the larger hyper-rectangles is hidden, while on the finest scale only the very smallest hyper-rectangles can be divided. In such manner certain regions can be explored very accurately.

It was noticed that when a high precision of the global minimum approximation is required, DIRECT might be unable to reach it because the computer memory is overfilled with definitions of numerous hyper-rectangles and the algorithm cannot proceed. Therefore in [51, 53] it was suggested to apply a local optimization algorithm in combination with DIRECT as a global search algorithm to speed up convergence. At each iteration of DIRECT a local optimization algorithm is executed starting from the centers of all potentially optimal hyper-rectangles. In addition, repeated runs of the whole combined algorithm are made after the feasible region D has been transformed according to the results of previous runs. The achieved gain in precision is counterbalanced by this extremely expensive optimization process.

Further interesting extensions for DIRECT resulted from adapting it for objective functions with Lipschitz-continuous derivatives [46, 47] and exploiting the symmetry of the Lipschitz-continuous objective function [28].

An alternative interpretation of points in Figure 2.3 is presented in [59], where it was suggested to treat the selection of hyper-rectangles for partitioning as a bi-objective optimization problem. The objectives correspond to the half-diagonal of a hyper-rectangle and function value at its center location.

2.3.5.3 The DISIMPL-V Algorithm

Among the various modifications of DIRECT, the one called DISIMPL-V [66] is of special interest for the purposes of this thesis.

The DISIMPL algorithm [66] is based on DIRECT, but simplicial partitioning is used instead of rectangular. DISIMPL-C and DISIMPL-V versions are provided by the authors. The middle point of the simplex is used in DISIMPL-C to characterise the subregion, while the vertex with the smallest objective function value is used in the DISIMPL-V algorithm. The performance of the DISIMPL-V algorithm was shown to be better, therefore only DISIMPL-V is described in this section. The pseudocode of the DISIMPL-V algorithm is provided in Algorithm 3.

Initially, the feasible region is rescaled to the unit hyper-cube

$$\overline{D} = \left\{ \boldsymbol{x} \in \mathbb{R}^d : 0 \leqslant x_i \leqslant 1,\ i = 1, ..., d \right\}. \tag{2.16}$$

Then the feasible region is covered by face-to-face simplicial partition, which is obtained by applying the combinatorial vertex triangulation algorithm [65] (pseudocode provided in Algorithm 2). The initial partition produces $d!$ equal simplices which share 2^d vertices $\boldsymbol{v}_i$, $i = 1, \ldots, 2^d$, of the hyper-cube. The objective function values are computed at the vertices of the unit hyper-cube.

The main loop of the algorithm begins and at each iteration potentially optimal simplices are selected and divided. At the first iteration, all simplices are selected and bisected along their longest edges. In the following iterations, potentially optimal simplices are selected and each of them is bisected along its longest edge. The arbitrary simplex S_j is potentially optimal if there exists some rate-of-change constant $\widetilde{L}$ such that:

$$\min_{\boldsymbol{v} \in V(S_j)} f(\boldsymbol{v}) - \widetilde{L}\Delta(S_j) \ \leqslant \ \min_{\boldsymbol{v} \in V(S_i)} f(\boldsymbol{v}) - \widetilde{L}\Delta(S_i),\ \forall S_i \in S, \tag{2.17}$$

$$\min_{\boldsymbol{v} \in V(S_j)} f(\boldsymbol{v}) - \widetilde{L}\Delta(S_j) \ \leqslant \ f_{min} - \epsilon |f_{min}|, \tag{2.18}$$

where $\Delta(S_j)$ denotes the length of S_j longest edge (i. e. diameter of the simplex), $V(S_j)$ is a set of vertices of simplex S_j, f_{min} is the best observed objective function value in an arbitrary iteration and ϵ is a positive constant (usually $\epsilon = 0.0001$ value is used). The iterative process is repeated until the stopping criteria are satisfied, for example, stopping criterion can be exhausting the budget M_{max} of trials.

It was shown in [63] that DISIMPL-V method, which uses simplicial partition-

Algorithm 2: Combinatorial vertex triangulation algorithm for d-dimensional unit-cube

Input d - dimension of the unit hyper-cube.
Function *TRIANGULATE(d):*
  Initialize a set of simplices $S = \varnothing$.
  for t = one of all permutations of *{1, ..., d}* **do**
    for j = *1, ..., d* **do**
      $v_{1,j} = 0$
    for i = *1, ..., d* **do**
      for j = *1,..., d* **do**
        $v_{(i+1),j} = v_{i,j}$
      $v_{(i+1),t_i} = 1$
    $S = S \cup \{\boldsymbol{v}\}$
  return S

Algorithm 3: DISIMPL-V

Input $\boldsymbol{l}$ - vector containing lower bounds of the feasible region, $\boldsymbol{u}$ - vector containing upper bounds of the feasible region, $f(\cdot)$ - objective function, M_{max} - number of maximum function evaluations.
Function *DISIMPLV($\boldsymbol{l}$, $\boldsymbol{u}$, f, M_{max}):*
  Normalize the feasible region D to be the unit hyper-cube $\overline{D}$.
  Cover $\overline{D}$ by face-to-face simplices $S = \left\{S_i : \overline{D} = \cup S_i,\ i = 1, ..., d!\right\}$ using combinatorial vertex triangulation algorithm.
  Evaluate $\{f(\boldsymbol{v}_i) : \boldsymbol{v}_i$ unique vertex of $S, i = 1, ..., 2^d\}$. Find f_{min}, $m = 2^d$.
  while $m < M_{max}$ **do**
    Select a set of potentially optimal simplices for division. **foreach** $S_l \in P$ **do**
      Divide S_l into two new simplices S_l^1, S_l^2, by adding a vertex $\boldsymbol{v}$ in the middle of the longest edge of S_l.
      Update $S = S \backslash \left\{S_l\right\} \cup \left\{S_l^1, S_l^2\right\}$. If $\boldsymbol{v}$ is a new vertex, evaluate $f(\boldsymbol{v})$, set $m = m + 1$ and update f_{min}.
  return f_{min}.

ing, outperforms DIRECT and locally-biased DIRECT methods, which use hyper-rectangular partitioning strategies. Basically, the only difference between DIRECT and DISIMPL-V algorithms is the partitioning strategy. It was also demonstrated, that the methods using simplicial partitioning strategies perform particularly well when symmetries of the objective function may be taken into account

to reduce the search space. In this thesis, we have decided to design algorithms using simplicial partitioning in order to benefit from its advantages.

As a further development of DISIMPL-V algorithm, using it as a base solver in a dual-mode, algorithm GB-DISIMPL-V [63] was proposed. The GB-DISIMPL-V algorithm is a modification of DISIMPL-V achieved by adding a globally biased optimization phase. Therefore, GB-DISIMPL-V consists of the following two-phases: a usual phase (as in the original DISIMPL-V method) and a global one. The usual phase is performed until a sufficient number of subdivisions of simplices near the current best point has taken place. Once these subdivisions around the current best point have been executed, its neighborhood contains only small simplices and all the larger ones are located far away from it. Thus the two-phase approach forces the GB-DISIMPL-V algorithm to explore larger simplices and to return to the usual phase only when an improved minimal function value is obtained. Each of these phases can consist of several iterations.

In particular, during the usual phase the GB-DISIMPL-V algorithm tries to explore better the subregion around the current best point. This phase finishes when a relative (with a coefficient τ) improvement of the minimal function value is not reached in i_u^{max} iterations.

After the end of the usual phase the method is switched to the global phase. The global phase consists of subdividing mainly large simplices, located possibly far away from the current best point. It is performed until a function value improving the current minimal value by at least 1% is obtained. When this happens, the algorithm again switches to the usual phase during which the obtained new solution is improved. During its work the GB-DISIMPL-V algorithm can switch many times from the usual phase to the global one.

2.4 Deterministic versus Metaheuristic Approaches

To better understand the position of Lipschitzian optimization with respect to other algorithms, it is worth considering a distinction between deterministic and stochastic algorithms. Deterministic algorithms are characterized by the fact that for a given input they produce the same result irrespectively of the

number of times such algorithms are run. On the contrary, stochastic algorithms use some random elements during their execution and therefore multiple runs of such algorithms produce different results given the same initial input. Expectations regarding the solution that these two classes of algorithms can produce differ in the following way. The stochastic methods provide a guarantee that in infinite number of trials the exact global minimum will be found with probability 1, whereas the deterministic methods guarantee to find an approximation of the global minimum to be found with required precision in a finite sufficiently large number of trials. The large number of trials required by the deterministic approaches is compensated by the advantage of there being no need for repeated runs, which are necessary in the case of stochastic algorithms. Lipschitzian algorithms are usually deterministic.

Various classes of stochastic algorithms are known [5, 20, 62], among which the metaheuristic nature-inspired algorithms have gained great popularity, especially in the context of multi-objective optimization. These include evolutionary algorithms [15, 70], simulated annealing [1, 54], swarm intelligence algorithms [17, 41, 42, 87, 88]. All of these algorithms are widely used for solving applied large-scale optimization problems. However, it is important to note that the choice between metaheuristic and deterministic approaches to a particular practical problem should be made often careful consideration of the related implications. For example, the user of the algorithm has to keep in mind that the solution produced by a metaheuristic algorithm does not guarantee to be a globally optimal one. The only thing known is that the generated solutions do not dominate each other. Moreover, the metaheuristic approaches might be costly as the repeated runs consume a great amount of generally expensive objective function evaluations. To aid the user in making this choice, a systematic experimental comparison of algorithms in both categories is provided in a recent study [44]. Among the evolution-based algorithms, SPEA 2[95] and NSGA-II [15] have become de facto standard approaches.

Chapter 3

Proposed Algorithms and Their Experimental Investigation

3.1 Global Optimization Algorithm Using a Global Estimate of the Lipschitz Constant

In practice, the Lipschitz constant is usually not known for global optimization problems. Among the most commonly used Lipschitzian optimization algorithms to solve this problem is DIRECT algorithm (described in section 2.3.5.1.) and its modifications.

The first advantage of the DIRECT algorithm is that it combines both the global and local search by dividing subregions of different sizes in each iteration, and the global minimum point is guaranteed to be found when the number of trials tends to infinity. The second advantage is that simple Lipschitzian bounds, constructed only from the center points of subregions, are used. Hence, numerical computations to find the minimum of the lower Lipschitz bounds are very low, because it can be found analytically.

However, DIRECT algorithm has several disadvantages as well. For example, Lipschitz constant estimates are selected from a very large set ($[0, \infty)$). Hence, more accurate estimates may be used. Secondly, DIRECT-type algorithms often spend an excessive number of function evaluations on problems with many lo-

cal optima exploring suboptimal local minima, thereby delaying the discovery of the global minimum.

In this section, a Lipschitzian optimization algorithm for constrained global optimization problems (2.1),(2.4) with unknown Lipschitz constant is proposed. The idea to select and divide several subregions of different sizes is borrowed from DIRECT-type algorithms. The main differences between the proposed and DIRECT-type algorithms are:

1. a single Lipschitz constant estimate instead of a set of Lipschitz constants is used in each iteration;
2. simplicial instead of rectangular partition is used for decomposition, just like in DISIMPL-v and GB-DISIMPL-v;
3. the proposed algorithm has a single parameter $\alpha \in [0, 1]$, which allows to select the globality of the search. The higher α value, the more global the search is.

In addition, several modifications of the proposed algorithm are numerically compared. Firstly, different types of Lipschitz bounds are experimentally compared. Secondly, different strategies for selecting subregions for division are experimentally compared. Finally, different strategies to evaluate the Lipschitz constant estimate are compared.

3.1.1 Description of the LIBRE Algorithm

Initially, the feasible region is rescaled to the unit-hypercube; consequently it is assumed that partition of the feasible region is a standard face-to-face simplicial partition of the unit-hypercube by the combinatorial vertex triangulation [65] (pseudocode provided in Algorithm 2). The initial partition produces $d!$ equal simplices which share 2^d vertices v_i, $i = 1, \ldots, 2^d$, of the hypercube. The objective function values are computed at the vertices of the hypercube. Next, using the computed objective function values and the mutual distances between

the corresponding vertices, the initial estimate of the Lipschitz constant $\widetilde{L}$ is obtained as maximum of

$$|f(\boldsymbol{v}_i) - f(\boldsymbol{v}_j)|/||\boldsymbol{v}_i - \boldsymbol{v}_j||, \; i \neq j. \tag{3.1}$$

At the current iteration, $\widetilde{L}$ is updated in case the approximation of directional derivative (3.1), based on the newly computed function values and the corresponding edges of new simplices produced at that iteration, is larger than the current $\widetilde{L}$.

To select simplices for subdivision, two criteria are applied. The first criterion is the approximation of minimum of the surrogate Lipschitz lower bound found over the simplex in question. A surrogate Lipschitz lower bound for the simplex S_i is defined using the smallest of the function values at the vertices of the considered simplex and the current estimate of the Lipschitz constant $\widetilde{L}_k$:

$$G(S_i, \widetilde{L}_k) = \min_{\boldsymbol{v}_j \in V(S_i)} f(\boldsymbol{v}_j) - \widetilde{L}_k \Delta(S_i)\alpha, \tag{3.2}$$

where $V(S_i)$ is the set of vertices of S_i, $\Delta(S_i)$ is the diameter of S_i and $\alpha \geqslant 0$ is a parameter of the proposed algorithm by which the globality of the search can be regulated. The larger values of α lead to a more global search, while $\alpha = 0$ leads to a global optimization algorithm DISIMPL-V [66], which tends to perform an excessively local search.

The second selection criterion is the diameter of the simplex: $\Delta(S_i)$.

An algorithm of simplicial partition with the criterion of selection for subdivision $G(\cdot)$ can be seen as a modification of the multi variable PIJAVSKIJ-SHUBERT algorithm where the feasible region is partitioned by simplices, the validated Lipschitz constant is replaced by its estimate, and a computationally simple version of the Lipschitz lower bound is used.

The definition of the estimate of the Lipschitz constant is complicated since even the smallest true Lipschitz constant of the function in question does not necessarily adequately represent its overall behaviour. For example, in the case of steep growth of function values close to the boarder of the feasible region and relatively flat hyper surface of the function elsewhere, the Lipschitz constant

might not be an appropriate characteristic of the function over the majority of the feasible region.

However, normally the directional derivatives close to minimizers are considerably smaller than finite differences based estimate of the Lipschitz constant, despite the latter itself being smaller than a validated Lipschitz constant. Such a discrepancy can cause too frequent computation of the objective function in the relatively close neighborhood of the currently found best points. The density of sites for computation can be controlled heuristically adapting the estimate of the Lipschitz constant to different subregions of the feasible region. In this section a different method is proposed: in order to increase the density of computing sites in relatively sparse subregions the second selection criterion $\Delta(\cdot)$ is introduced strengthening the priority of selection of large simplices. As a result, the selection of a set of simplices for partitioning at the current iteration can be viewed as a two-criteria optimization problem, where the criteria are expressed by $G(\cdot)$ and $\Delta(\cdot)$, and one seeks to minimize $G(\cdot)$ and maximize $\Delta(\cdot)$:

$$\min_{S_i \in S} (G(S_i), -\Delta(S_i)). \tag{3.3}$$

A set of simplices that correspond to the supported Pareto optimal solutions to this problem is selected for division to ensure that the best compromises between (3.2) and $\Delta(\cdot)$ are captured and that the selected set is of a moderate size. The set of Pareto optimal solutions to (3.3) problem might be found using Algorithm by applying it to each simplex in partition. The selected simplices are subdivided by bisection of the longest edge of the selected simplices. The objective function values are computed at the middle points of the bisected edges. The function values are stored in a balanced tree data structure and are not computed repeatedly but read from the structure if computed at previous iterations.

The convergence of the proposed algorithm is implied by the fact that a simplex with the longest diameter is subdivided at each iteration. Such a simplex is selected at each iteration since it inevitably belongs to the set of supported Pareto optimal solutions: its two dimensional vector of objectives $G(\cdot)$ and $\Delta(\cdot)$ obviously belongs to the Pareto front as a vector with the maximum value of one of the objectives. The investigation of the rate of convergence is more difficult. Meanwhile the performance of the algorithm is assessed by testing experiments

presented in the next section.

Algorithm 4: LIBRE - the proposed algorithm

Input $\boldsymbol{l}$ - vector containing lower bounds of the feasible region, $\boldsymbol{u}$ - vector containing upper bounds of the feasible region, $\boldsymbol{f}(\cdot)$ - vector containing objective functions, M_{max} - number of maximum function evaluations.
Function *LIBRE*($\boldsymbol{l}$, $\boldsymbol{u}$, $\boldsymbol{f}$, M_{max}):
    Normalize the feasible region D to be the unit-hypercube $\overline{D}$.
    Cover $\overline{D}$ by face-to-face simplices $S = \{S_i : \overline{D} = \cup S_i,\ i = 1, ..., d!\}$ using combinatorial vertex triangulation algorithm.
    Evaluate $\{f(\boldsymbol{v}_i) : \boldsymbol{v}_i$ unique vertice of $S, i = 1, ..., 2^d\}$. Set $m = 2^d, \widetilde{L} = 0$.
    while *stopping condition not satisfied* **and** $m < M_{max}$ **do**
        foreach $S_l \in S$ **do**
            Find $\widetilde{L}_l = \max\left\{\frac{|f(\boldsymbol{v}_i)-f(\boldsymbol{v}_j)|}{||\boldsymbol{v}_i-\boldsymbol{v}_j||} : \boldsymbol{v}_i, \boldsymbol{v}_j \in V(S_l), \boldsymbol{v}_i \neq \boldsymbol{v}_j\right\}$.
        Update $\widetilde{L} = \max\left\{\widetilde{L}\right\} \cup \left\{\widetilde{L}_l : S_l \in S\right\}$.
        foreach $S_l \in S$ **do**
            find $G(S_l, \widetilde{L}) = \min_{\boldsymbol{v}_i \in V(S_l)} f(\boldsymbol{v}_i) - \widetilde{L}\Delta(S_l)\alpha$
        Identify a set of simplices for division:

        $P = \{S_i : S_i \in S,\ S_i$ is supported Pareto optimal solution to (3.3)$\}$
        foreach $S_l \in P$ **do**
            Divide S_l into two new simplices S_l^1, S_l^2, by adding a vertex $\boldsymbol{v}$ in the middle of the longest edge of S_l.
            Update $S = S \backslash \{S_l\} \cup \{S_l^1, S_l^2\}$. If $\boldsymbol{v}$ is a new vertex, evaluate $f(\boldsymbol{v})$ and set $m = m + 1$.
    return f_{min}

We have entitled the proposed algorithm LIBRE (LIpschitz Bounds Rough Estimation), since the approximation of the Lipschitz bounds (3.2) is constructed using only one vertex with the smallest function value and, in addition, it is defined using an estimate of the Lipschitz constant. The pseudocode of the proposed algorithm is presented in Algorithm 4. It has been implemented in C++, avoiding repetitive calculations, e.g. approximations (3.2) were updated only if the lower bound of the Lipschitz constant has changed. Only unique function evaluations were calculated. The source code of this algorithm is provided under Affero GPL v3 (or greater) licence in the Appendix B.

To illustrate the main properties of the proposed algorithm, some intermediate

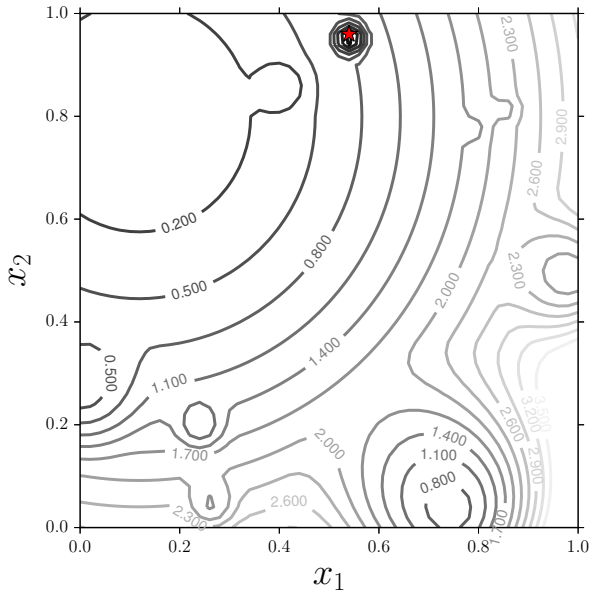


Figure 3.1: The values of the objective function presented by the level lines

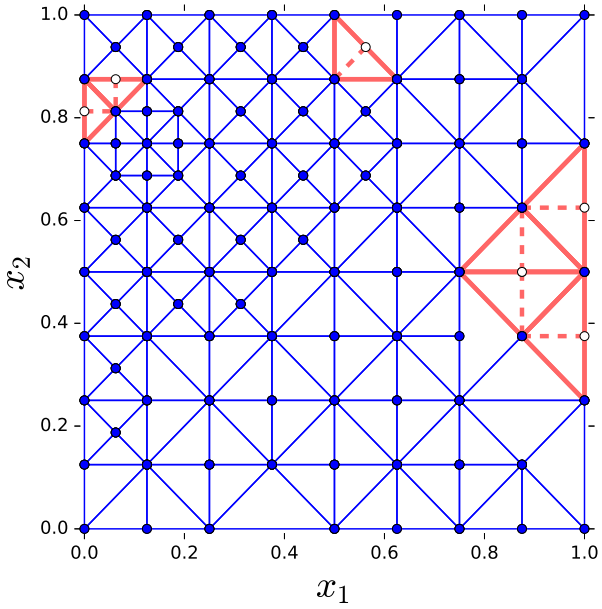


Figure 3.2: The subdivision of the feasible region by triangles. The triangles selected for the subdivision at the current iteration are indicated by thicker edges

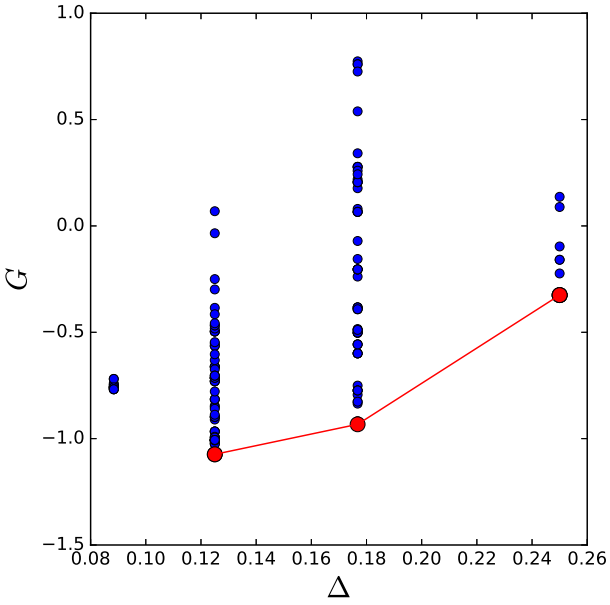


Figure 3.3: The two dimensional vectors of criteria computed for the triangles shown in Figure 3.2

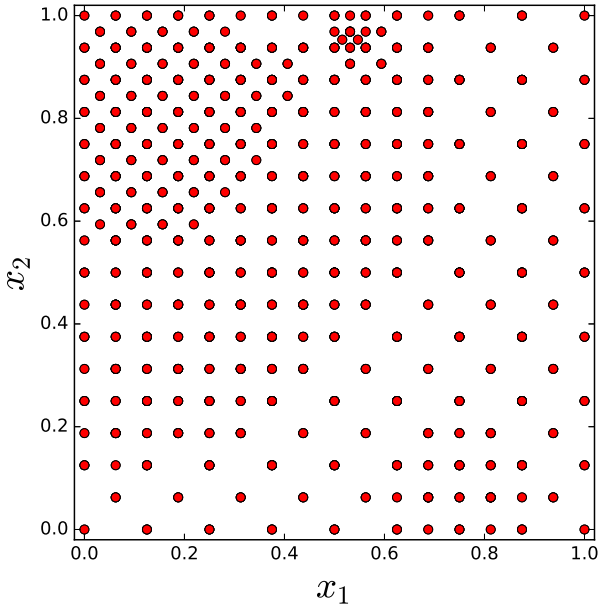


Figure 3.4: The distribution of points where the values of the objective functions were computed

and final results of the minimization are presented graphically. In this example, the objective function was generated using GKLS [24] test function generator; the problem class 2 and function ID 1 were selected. In Figure 3.1, the considered test function is presented by the level lines; the global minimizer is indicated by a star. The triangular (simplicial) partition of the feasible region after 35 iteration (101 objective function values are computed) is shown in Figure 3.2 where 7 non-dominated triangles are indicated by the thicker edges. These triangles correspond to three points which compose the Pareto front; see Figure 3.3. One of the points represents four largest triangles which share the vertex with the smallest (among the computed at vertices of these triangles) objective function value. Similarly, one point represents two smallest non-dominated triangles. A single triangle of the intermediate size is also represented in the Pareto front.

The optimization process was terminated according to the (3.4) rule which is used in testing experiments in Sect. 3.1.2 and described there. 97 iterations were performed, and 276 values of the objective function were computed. The distribution of points, where the function values were computed, is shown in Figure 3.4. The higher density is seen in the vicinity of the global minimizer.

3.1.2 Experimental Investigation

The algorithm was developed aiming at difficult optimization problems where the local minima are like spikes in a slowly changing landscape. Therefore its performance was tested using the GKLS-generator [24] which generates test functions with desirable characteristics. In our experiments, the same classes of test functions as in [63, 66, 78] were used, i.e. the randomly generated differentiable test functions with the parameters presented in Table 3.1. These functions are constructed as a (relatively flat) background hyper paraboloid with added "spikes" defined by polynomials. The complexity of problems depends on the extent of the region of attraction of the global minimizer and its distance from the minimum point of the background paraboloid.

The following parameters are common to all test functions: the number of local minima is equal to 10, and the global minimum is equal to -1. The specific

Table 3.1: Description of the GKLS test function classes

Class	Difficulty	d	Number of minima	f^*	ρ	r	δ
1	Simple	2	10	-1	0.90	0.2	10^{-4}
2	Hard	2	10	-1	0.90	0.1	10^{-4}
3	Simple	3	10	-1	0.66	0.2	10^{-6}
4	Hard	3	10	-1	0.90	0.2	10^{-6}
5	Simple	4	10	-1	0.66	0.2	10^{-6}
6	Hard	4	10	-1	0.90	0.2	10^{-6}
7	Simple	5	10	-1	0.66	0.3	10^{-7}
8	Hard	5	10	-1	0.66	0.2	10^{-7}

parameters of different classes of test functions are presented in Table 3.1. The following notation is used: ρ - the distance of global minimizer from the minimum point of the background paraboloid, r - the radius of the attraction region of global minimizer. Let us recall that d denotes the dimensionality of the problem. Experiments were performed using 8 different classes of the test functions. The classes are named "hard" in case their ρ is relatively large, and r is relatively small.

First, we aimed to evaluate an appropriate value of the parameter α. The experiments were planned using the testing methodology identical with that in [63, 66, 78]. The minimization process is stopped after m-th computation of the objective function value if the distance between the global minimizer and x_m, the closest site, where an objective function value was computed, falls below a predefined threshold $\epsilon = \sqrt[d]{\delta}$. The algorithm also was supposed to stop after 10^6 function evaluations in case the previous stopping condition has not been satisfied. However, during our experiments such a termination, meaning that the global minimizer is lost, did not take place.

To implement such a stopping condition, the global minimizer of the considered objective function should be known; let it be denoted by $\boldsymbol{x}^*$; let us note that $\boldsymbol{x}^* \in D$. The stopping condition is satisfied, when a point $\boldsymbol{x} \in D$ is found, which satisfies the following condition:

$$|x_i - x_i^*| \leqslant \sqrt[d]{\delta}(u_i - l_i),\ i = 1, ..., d, \tag{3.4}$$

the δ value is provided in Table 3.1.

Table 3.2: The influence of α on the performance of the proposed algorithm, i.e. on the numbers of function evaluations (average, median, largest) made before stopping; a hundred randomly generated test functions from every GKLS class were minimized

#	Libre α	Average	Median	Largest	#	Libre α	Average	Median	Largest
1	0.01	185.94	140	810	4	0.01	2322.49	2042	6960
	0.2	**136.18**	**116**	428		0.2	**1142.39**	**949**	**3337**
	0.4	151.92	145	**371**		0.4	1448.94	1386	3484
	0.6	217.16	216	425		0.6	2976.24	2865	7554
	0.8	280.92	274	505		0.8	4788.87	4528	12745
	1.0	330.93	323	616		1.0	6796.57	6292	16722
2	0.01	940.08	909	2765	5	0.01	9994.97	6486	63849
	0.2	513.26	483	1459		0.2	**4793.12**	**3437**	25048
	0.4	**431.53**	**397**	1117		0.4	5339.45	4572	**16968**
	0.6	475.18	453	**935**		0.6	8033.69	7228	19282
	0.8	531.93	495	966		0.8	10415.30	9469	26604
	1.0	581.62	579	1060		1.0	12593.80	10595	33889
3	0.01	975.21	737	4738	6	0.01	27599.00	23961	98295
	0.2	**621.48**	**523**	2393		0.2	11188.10	9153	39736
	0.4	1009.82	957	**2113**		0.4	**8965.54**	**8422**	**23348**
	0.6	2071.96	2030	4360		0.6	10431.40	9606	25871
	0.8	3461.76	3191	7285		0.8	12421.80	10856	31172
	1.0	5021.94	4638	11517		1.0	14581.50	11764	38221

The impact of α parameter value on the efficiency of the proposed algorithm was experimentally evaluated. The proposed algorithm was executed with a set of different α values $\alpha \in 0.01, 0.2, 0.4, 0.6, 0.8, 1.0$ on randomly generated test functions from six GKLS function classes. The results are presented in Table 3.2. The comparison of the average and worst case performance of the algorithm for different classes of test functions shows that the value $\alpha = 0.4$ is appropriate for the objective functions of various complexity.

To compare the performance of the proposed algorithm (where $\alpha = 0.4$) with the performance of other algorithms aimed at similar problems, the respective results from Table 3.2 are also presented in Table 3.3 together with the corresponding results from [63, 66] concerning the algorithms Disimpl-v and Gb-Disimpl-v; the best results are presented in bold.

The experimental results indicate that the proposed algorithm is advantageous

Table 3.3: Results obtained by solving 100 optimization problems from each of the GKLS test function classes, using the stopping condition with a known global minimizer

Class	Difficulty	Algorithm	Average	Median	Largest
1	Simple	DISIMPL-V	192.93	151.0	773
		GB-DISIMPL-V	174.25	151.0	472
		LIBRE α=0.4	**151.97**	**146.5**	**371**
2	Hard	DISIMPL-V	1003.56	1021.0	2683
		GB-DISIMPL-V	518.11	**511.0**	1547
		LIBRE α=0.4	**431.55**	515.0	**1117**
3	Simple	DISIMPL-V	1061.83	787.0	4740
		GB-DISIMPL-V	**751.25**	**720.0**	2694
		LIBRE α=0.4	1009.72	959	**2113**
4	Hard	DISIMPL-V	2598.91	2594.0	7354
		GB-DISIMPL-V	**1364.40**	**1283.0**	3723
		LIBRE α=0.4	1449.18	1390	**3484**
5	Simple	DISIMPL-V	10618.00	7334.0	58764
		GB-DISIMPL-V	**4579.24**	4615.0	**13825**
		LIBRE α=0.4	5340.43	**4575**	16968
6	Hard	DISIMPL-V	33985.20	29807.0	118482
		GB-DISIMPL-V	10700.20	10033.0	30759
		LIBRE α=0.4	**8965.73**	**8436**	**23348**
7	Simple	DISIMPL-V	11200.4	7252	48590
		GB-DISIMPL-V	**5997.37**	**4524**	**23126**
		LIBRE α=0.4	17305.2	13343	65622
8	Hard	DISIMPL-V	64751	42680	382593
		GB-DISIMPL-V	**28946**	**22416**	168067
		LIBRE α=0.4	44000.4	36306	**154277**

for the most inappropriate test functions of "hard" classes; see column "Largest" in Table 3.3.

3.1.3 Stopping Condition for Problems in Practice

In the previous 3.1.2 section stopping condition was associated with a global minimizer which is known in advance for the test problems. In other cases different stopping condition has to be used. In this section, a new stopping

condition, which is associated with the estimated potential improvement of the objective function, is proposed.

Worst-case one-step optimal algorithms [89–91] use a Lipschitz constant which is known in advance. However, this is usually not the case in practice and finding an accurate Lipschitz constant can be as hard as solving an optimization problem itself.

To overcome this problem, we suggest using a lower Lipschitz bound $\widetilde{L}$ on the objective function values as an estimate of the Lipschitz constant. This estimate tends to the true Lipschitz constant, as the number of function evaluations m tends to infinity, i. e. $\widetilde{L} \to L$, when $m \to \infty$. As a result, when the number of evaluations is relatively high, we can estimate L accurately enough. Moreover, in such a case the estimate of the potential improvement over the best known objective function value, obtained using $\widetilde{L}$, is practically sufficiently accurate as well.

Suppose that a global optimization algorithm has generated a simplicial decomposition of the feasible region S. Then the estimate of the Lipschitz constant in an arbitrary iteration can be defined as follows:

$$\widetilde{L} = \max\left\{\left\{\frac{|f(\boldsymbol{v}_i) - f(\boldsymbol{v}_j)|}{||\boldsymbol{v}_i - \boldsymbol{v}_j||} : \boldsymbol{v}_i, \boldsymbol{v}_j \in V(S_z), \boldsymbol{v}_i \neq \boldsymbol{v}_j,\ S_z \in S\right\} \cup \left\{\widetilde{L}\right\}\right\}. \quad (3.5)$$

The focal idea of the present section is that an estimated potential improvement, obtained using Lipschitz constant estimation technique (3.5), can be used in constructing a stopping criterion for a wide range of algorithms.

To compute a stopping condition, an algorithm needs to rely on a decomposition of the feasible region. Moreover, a method for estimating a Lipschitz lower bound over any part of the decomposition has to be chosen. We defined the stopping condition for the case when the decomposition is simplicial, but it could also be extended to the case of other possible decomposition types. In order for the results of several different algorithms to be comparable to each other they all have to use the same method for estimating the Lipschitz lower bound.

A method to estimate the Lipschitz lower bound over a simplex S_i has been

Table 3.4: Results obtained by solving 100 optimization problems from first four GKLS test function classes, using a proposed stopping condition

Class	Algorithm	Average	Median	Largest
1	DISIMPL-V	3569.93	3574	5766
	GB-DISIMPL-V	1283.23	1292.5	1839
	Proposed	**170.46**	**181**	**393**
2	DISIMPL-V	8756.84	9488	16286
	GB-DISIMPL-V	3246.03	3484.5	5464
	Proposed	**252.71**	**163**	**690**
3	DISIMPL-V	58113.4	58963	97345
	GB-DISIMPL-V	18435.9	18248	31296
	Proposed	**1185.68**	**1115.5**	**2989**
4	DISIMPL-V	100773	99300	174902
	GB-DISIMPL-V	32417.4	32660.5	58126
	Proposed	**1868.05**	**1899**	**3149**

taken from algorithm DISIMPL-V [66]. Namely, for a Lipschitz constant estimate $\widetilde{L}$,

$$G(S_i, \widetilde{L}) = \min_{v_j \in V(S_i)} f(v_j) - \widetilde{L}\Delta(S_i). \tag{3.6}$$

This criterion has the advantage that it is easy to compute for simplicial subregions of the feasible region in any dimension.

The stopping condition is associated with an estimated potential improvement and is defined as:

$$\max \left\{ f_{min} - G(S_i, \widetilde{L}) : S_i \in S \right\} \leqslant \epsilon, \tag{3.7}$$

where $f_{min} = \min_{v \in V(S_j), S_j \in S} f(v)$.

We have implemented the stopping criterion (3.7) in three algorithms: the proposed algorithm (described in Section 3.1.1, $\alpha = 0.4$), DISIMPL-V and GB-DISIMPL-V. All the algorithms track the Lipschitz constant estimate (3.5), moreover, compute (3.6) for simplices in the current decomposition and determine whether the maximum estimated potential improvement is smaller than chosen ϵ (3.7).

The comparison of the number of objective function evaluations before the con-

dition (3.4) is satisfied with $\epsilon = 0.5$ by the considered algorithms is presented in Table 3.4 for the test function classes with parameters in Table 3.1.

As can be observed from Table 3.4, the proposed algorithm performs considerably better than the alternatives, when estimated improvement (3.7) stopping condition is used. One of the explanations of this difference in performance could be that the stopping condition (3.7) is exclusively convenient for the Libre algorithm. This is because the same definition of the estimate of the Lipschitz constant is used both in stopping condition and in the Libre algorithm. In addition, the (3.6) expression used in stopping condition is very similar to the (3.2), which is used in the Libre algorithm to select simplices for the subdivision.

3.1.4 Modifications

In this section several modifications of the proposed algorithm are numerically compared. Firstly, different types of Lipschitz bounds are experimentally compared. Secondly, different strategies for selecting subregions for division are compared. Finally, a comparison is made between different strategies applied to evaluate the Lipschitz constant estimate. The execution of the experiments was automated by means of the utility tool described in Appendix Automation of Experiment Execution and Result Aggregation.

3.1.4.1 Lipschitz Bounds Estimation Strategies

A hypothesis was made that the more strict Lipschitz bounds are used, the better performance the algorithm demonstrates. To check this hypothesis in the context of the proposed algorithm (section 3.1.1), four different strategies to estimate lower Lipschitz bounds over subregions are described and compared.

In the proposed algorithm (section 3.1.1), lower Lipschitz bounds over subre-

gion are estimated using only one vertex with the lowest function value:

$$\widehat{L} = \alpha \widetilde{L}, \tag{3.8}$$

$$\boldsymbol{v}_{min}(S_i) = \arg \min_{\boldsymbol{v}_i \in V(S_i)} f(\boldsymbol{v}_i), \tag{3.9}$$

$$g(\boldsymbol{x}, S_i, \widehat{L}) = f(\boldsymbol{v}_{min}(S_i)) - \widehat{L}||\boldsymbol{x} - \boldsymbol{v}_{min}(S_i)||, \tag{3.10}$$

$$G(S_i, \widehat{L}) = \min_{\boldsymbol{x} \in S_i} g(\boldsymbol{x}, S_i, \widehat{L}) = f(\boldsymbol{v}_{min}(S_i)) - \widehat{L}\Delta(S_l). \tag{3.11}$$

This strategy was selected, because it was successfully used in DISIMPL-V and GB-DISIMPL-V [63, 66] algorithms. It requires a small number of computations, because the solution (3.11) is found analytically. In Table 3.5 the strategy (3.9-3.11) is denoted by v_{min}.

The second chosen strategy (3.12-3.14) which was chosen is the same as (3.9-3.11), except that the vertex with the maximum function value is used:

$$\boldsymbol{v}_{max}(S_i) = \arg \max_{\boldsymbol{v}_i \in V(S_i)} f(\boldsymbol{v}_i), \tag{3.12}$$

$$g(\boldsymbol{x}, S_i, \widehat{L}) = f(\boldsymbol{v}_{max}(S_i)) - \widehat{L}||\boldsymbol{x} - \boldsymbol{v}_{max}(S_i)||, \tag{3.13}$$

$$G(S_i, \widehat{L}) = \min_{\boldsymbol{x} \in S_i} g(\boldsymbol{x}, S_i, \widehat{L}) = f(\boldsymbol{v}_{max}(S_i)) - \widehat{L}\Delta(S_l). \tag{3.14}$$

In Table 3.5 the strategy (3.12-3.14) is denoted by v_{max}.

In this case, the number of computations persists the same as in the previous strategy, but stricter Lipschitz bounds are constructed. An example for univariate case is shown in Figure 3.5, where the green line shows the Lipschitz bound constructed using strategy v_{min} and the black line shows the Lipschitz bound constructed using strategy v_{max}. As can be seen, the bounds over interval of the latter strategy are stricter. Hence, the maximum improvement of the best known objective function value is adequately smaller.

The third strategy implies the usage of all the vertices to construct Lipschitz

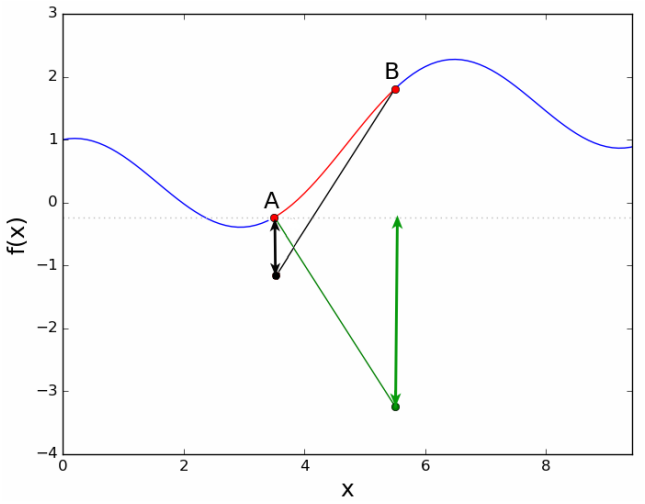


Figure 3.5: Visual comparison of strategy v_{min} (green line) and strategy v_{max} (black line) for univariate case

bounds, disregarding the additional computations:

$$g^{strict}(\boldsymbol{x}, S_i, \widehat{L}) = \max_{\boldsymbol{v} \in V(S_i)} \{f(\boldsymbol{v}) - \widehat{L}||\boldsymbol{v} - \boldsymbol{x}||\}, \ \boldsymbol{x} \in S_i, \tag{3.15}$$

$$G^{strict}(S_i, \widehat{L}) = \min_{\boldsymbol{x} \in S_i} g^{strict}(\boldsymbol{x}, S_i, \widehat{L}). \tag{3.16}$$

To find a solution for (3.16), an inner level optimization problem has to be solved. This strategy guarantees that the value of the strictest Lipschitz bounds is assessed with a bounded error. In our experiments, a grid-search strategy was employed. City block distance between the covering points was set to be less than 10% of simplex's diameter. This strategy is denoted by *all_verts*.

The final strategy makes an attempt to find a compromise between the number of computations and the accuracy of the Lipschitz bounds. The suggestion is to use Lipschitz bounds over the edges of a simplex as an approximation to (3.16) over the simplex

$$G^{strict}(S_i, \widehat{L}) \approx \min_{\boldsymbol{v}_i, \boldsymbol{v}_j \in V(S_i), i \neq j} G^{edge}(\boldsymbol{v}_i, \boldsymbol{v}_j, S_i, \widehat{L}) \ = \ \widetilde{G}^{strict}(S_i, \widehat{L}). \tag{3.17}$$

The minimum of the Lipschitz bounds over an edge $(\boldsymbol{v}_i, \boldsymbol{v}_j)$ can be found ana-

Table 3.5: Comparison of different Lipschitz bound estimation strategies using first six problem classes from GKLS test function generator

Class	Strategy	Calls average	Calls median	Calls largest	Avg. seconds per trial
1	v_{min}	**151.92**	**145**	**371**	**0.000039**
	v_{max}	283.92	247	1002	0.000063
	all_verts	280.74	222	952	0.000349
	edges	204.58	162	512	0.000156
2	v_{min}	**431.53**	**397**	**1117**	**0.000071**
	v_{max}	1085.13	947	2757	0.00024
	all_verts	1297.6	1196	3133	0.000512
	edges	911.78	764	2361	0.000284
3	v_{min}	**1009.82**	**957**	**2113**	**0.000323**
	v_{max}	2205.68	1886	8156	0.001023
	all_verts	1942.33	1324	11803	0.004319
	edges	1371.71	989	4238	0.001714
4	v_{min}	**1448.94**	**1386**	**3484**	**0.000517**
	v_{max}	4771.85	4157	17699	0.002511
	all_verts	4605.42	3699	18843	0.007346
	edges	3133.99	2632	10821	0.004411
5	v_{min}	**5339.45**	**4572**	**16968**	**0.012432**
	v_{max}	11855.8	9490	52130	0.061572
	all_verts	11385	8784	61363	0.155959
	edges	10435.2	7701	58035	0.173710
6	v_{min}	**8965.54**	**8422**	**23348**	**0.035610**
	v_{max}	26246.3	16769	113266	0.156995
	all_verts	36930.1	28451	159262	0.582037
	edges	35082.5	28524	113353	0.522561

lytically if they are constructed using only two vertices v_i, v_j forming the edge

$$G^{edge}(v_i, v_j, \widehat{L}) = \frac{f(v_i) + f(v_j) - \widehat{L}||v_i - v_j||}{2}, \tag{3.18}$$

$$X^{edge}(v_i, v_j, \widehat{L}) = v_i t + v_j(1-t), \;\; t = 0.5 + \frac{f(v_j) - f(v_i)}{2\widehat{L}||v_j - v_i||}, \tag{3.19}$$

where $X^{edge}(v_i, v_j, \widehat{L})$ is a point where $G^{edge}(v_i, v_j, \widehat{L})$ value can be observed. If $G^{edge}(v_i, v_j, \widehat{L}) == g(X^{edge}(v_i, v_j, \widehat{L}), \widehat{L})$, then $G^{edge}(v_i, v_j, S_i, \widehat{L}) = G^{edge}(v_i, v_j, \widehat{L})$. Otherwise, a one dimensional optimization problem has to be solved to approximately find $G^{edge}(v_i, v_j, S_i, \widehat{L})$. We used Pijavskij-Shubert algorithm [67, 81] with

the Lipschitz constant equal to $\widehat{L}$ and considered the problem solved when the maximum expected improvement was $\leqslant \epsilon = 10^{-4}$. In Table 3.5 this strategy is denoted by *edges*.

The performance of these four strategies was experimentally evaluated by solving 600 test problems from 6 classes generated by the GKLS-generator [24] (see section 3.1.2). The same stopping condition as in [63, 78] was used. To measure the performance, the average, median and largest number of trials needed to satisfy the (3.4) stopping condition was evaluated. To measure the computational complexity of the strategies, the average CPU time per trial was evaluated. The numerical results are provided in Table 3.5.

The $\alpha = 0.4$ was used for the experiments (just like in section 3.1.1). However, experiments with different α values are provided in Table 3.2.

Numerical results (Table 3.5) show that the usage of stricter lower Lipschitz bounds did not improve the performance of the proposed algorithm (section 3.1.1) for GKLS test problem set. Constructing lower Lipschitz bounds from only the vertex with the best function value is the best strategy (heuristic) both in respect of performance and the number of computations.

3.1.4.2 Selecting Potentially Optimal Simplices

In the proposed algorithm, two criteria (3.3) are used to characterise simplices and to select potentially optimal ones for the division. In this section the strategies on how to find the best compromises between these two criteria are compared.

The first strategy is to select simplices which represent Pareto optimal solutions in the two criteria (3.3) space

$$Y = \{(G(S_i), -\Delta(S_i)) : S_i \in S\}, \tag{3.20}$$

$$Y^{Pareto} = \left\{\boldsymbol{y} \in Y : \{\boldsymbol{y}' \in Y : \boldsymbol{y}' \succ \boldsymbol{y},\ \boldsymbol{y}' \neq \boldsymbol{y}\} = \varnothing\right\}, \tag{3.21}$$

$$P = \left\{S_i : S_i \in S, (G(S_i), -\Delta(S_i)) \in Y^{Pareto}\right\}, \tag{3.22}$$

where P is a set of potentially optimal simplices. This strategy is denoted by *Pareto*.

Table 3.6: Comparison of different strategies to select potentially optimal simplices using first six problem classes from GKLS test function

Class	Strategy	Calls average	Calls median	Calls largest	Avg. seconds per trial
1	*Pareto*	153.42	149	421	**0.000039**
	Supported	**151.92**	**145**	**371**	**0.000039**
2	*Pareto*	464.42	408	1453	0.000086
	Supported	**431.53**	**397**	**1117**	**0.000071**
3	*Pareto*	**1004**	**953**	2200	0.000348
	Supported	1009.82	957	**2113**	**0.000323**
4	*Pareto*	1454.22	**1380**	3571	0.00055
	Supported	**1448.94**	1386	**3484**	**0.000517**
5	*Pareto*	5826.71	4994	**16870**	0.016318
	Supported	**5339.45**	**4572**	16968	**0.012432**
6	*Pareto*	9409.04	8947	26536	**0.029641**
	Supported	**8965.54**	**8422**	**23348**	0.035610

The second strategy is to select simplices which represent *supported* Pareto optimal solutions in the two criteria (3.3) space

$$Y = \{(G(S_i), -\Delta(S_i)) : S_i \in S\}, \tag{3.23}$$

$$Y^{Supported} = Y \cap \{\boldsymbol{y} \in Conv(Y) : \{\boldsymbol{y}' \in Conv(Y) : \boldsymbol{y}' \succ \boldsymbol{y},\ \boldsymbol{y}' \neq \boldsymbol{y}\} = \varnothing\}, \tag{3.24}$$

$$P = \left\{S_i : S_i \in S, (G(S_i), -\Delta(S_i)) \in Y^{Supported}\right\}, \tag{3.25}$$

where $Conv(Y)$ is a convex hull of a set Y. This strategy is denoted by $Supported$.

The numerical results (Table 3.6) show, that the performance is better for hard problems when supported Pareto optimal solutions are used, i.e. strategy $Supported$.

3.1.4.3 Globality of the Lipschitz Constant Estimate

If a whole partition of the feasible region is used to estimate the Lipschitz constant, a *global Lipschitz constant estimate* is obtained. Otherwise, if a subset of the partition is used, a *local Lipschitz constant estimate* is obtained. When the Lipschitz constant is estimated locally, it might represent the local region more accurately than a global Lipschitz constant estimate. For example, if the function values are shallow in a region, the local Lipschitz constant would be small.

Table 3.7: Comparison of different strategies to estimate Lipschitz constant using first six problem classes from GKLS test function

Class	Strategy	Calls average	Calls median	Calls largest	Avg. seconds per trial
1	*global*	151.92	145	371	**0.000039**
	local	**122.04**	**116**	265	0.000049
	mixed	126.87	121	**257**	0.000047
2	*global*	431.53	397	1117	**0.000071**
	local	**290.16**	**261**	**888**	0.000086
	mixed	292.25	269	1011	0.000089
3	*global*	**1009.82**	**957**	**2113**	**0.000323**
	local	1153.69	1087	2587	0.000693
	mixed	1093.65	1066	2304	0.000713
4	*global*	1448.94	1386	3484	**0.000517**
	local	**1181.18**	**1158**	2439	0.000753
	mixed	1186.78	1156	**2163**	0.000834
5	*global*	**5339.45**	**4572**	**16968**	**0.012432**
	local	12715.1	12599	31046	0.083735
	mixed	11023.1	10797	24078	0.075724
6	*global*	**8965.54**	**8422**	**23348**	**0.035610**
	local	11594.5	11198	27905	0.078067
	mixed	10989.4	10771	23630	0.066359

Hence, this would lead to reduced priority of this region's investigation and might reduce the number of unnecessary trials. In this section a hypothesis is made that the usage of the local Lipschitz constant should improve the performance of the proposed algorithm (section 3.1.1). In order to verify this hypothesis, two additional strategies how to evaluate and use the Lipschitz constant were suggested.

In the proposed algorithm (section 3.1.1), only a global Lipschitz constant is used, which is defined as:

$$\widetilde{L}_{global} = \max\left\{\frac{|f(\boldsymbol{v}_i) - f(\boldsymbol{v}_j)|}{||\boldsymbol{v}_i - \boldsymbol{v}_j||} : \boldsymbol{v}_i, \boldsymbol{v}_j \in V(S_l), \boldsymbol{v}_i \neq \boldsymbol{v}_j, S_l \in S\right\} \tag{3.26}$$

where S is a set of simplices in current partition and $V(S_l)$ is a set vertices of S_l. In Table 3.7 this strategy is denoted by *global*.

The second strategy is to use only the local Lipschitz constant estimate. When

Lipschitz bounds are calculated for an arbitrary simplex S_i, its local Lipschitz constant is used

$$\widetilde{L}_{local}(S_i) = \max\left\{\frac{|f(\boldsymbol{v}_k) - f(\boldsymbol{v}_l)|}{||\boldsymbol{v}_k - \boldsymbol{v}_l||} : \boldsymbol{v}_k, \boldsymbol{v}_l \in V(S_j),\ V(S_j) \cap V(S_i) \neq \varnothing,\ S_j \in S\right\}. \tag{3.27}$$

The region in which the local Lipschitz constant is estimated consists of a simplex and its neighbouring simplices. In this case, simplices are considered to be neighbours if they have at least one common vertex. In Table 3.7 this strategy is denoted by *local*.

The third strategy is to combine the global and local Lipschitz constant estimates. The same way was chosen to combine the estimates as in the Information global optimization algorithm with local tuning [73]:

$$\widetilde{L}_{mixed}(S_i) = \max\{\widetilde{L}_{local},\ \widetilde{L}_{global}\frac{\Delta(S_i)}{\Delta_{max}}\}, \tag{3.28}$$

$$\Delta_{max} = \max_{S_j \in S} \Delta(S_j). \tag{3.29}$$

In this case, the globality of the Lipschitz constant estimate used is associated with the relative size of the simplex. $\widetilde{L}_{global}$ is always used for the biggest simplices of the partition (because their relative size is 1 and $\widetilde{L}_{local} \leqslant \widetilde{L}_{global}$). And for smaller simplices the value $\geqslant \widetilde{L}_{local}$ and $< \widetilde{L}_{global}$ is used. In Table 3.7 this strategy is denoted by *mixed*.

To compare the strategies, the same experimental methodology as in the previous section was applied. I.e. 600 test problems generated with GKLS-generator [24] were solved; the average, median, largest number of trials and the average CPU time per trial were measured. The numerical results are provided in Table 3.7. The $\alpha = 0.4$ was used for the experiments. Experiments with different α values are provided in Table 3.2.

The numerical results (Table 3.7) demonstrate, that none of these three strategies stood out as more efficient for all the test problem classes. However, the performance of the strategy *global* proved to be better for both 5'th and 6'th classes, which have the highest number of variables among the problems examined. Furthermore, it can be observed, that the usage of the local Lipschitz

constant increased the number of computations, since the average duration per trial increased.

3.2 Global Optimization Algorithm Using A Local Estimate of Lipschitz Constant

Independently of Libre algorithm, another similar algorithm [26], which uses local Lipschitz constant estimate, instead of global one, was proposed. These are the main differences between this algorithm and a modification of Libre algorithm denoted as *local* and described in Section 3.1.4.3:

1. An estimate of the Lipschitz constant used is a maximum of these values: (3.1) values calculated for pairs of vertices of neighbouring simplices and simplicial gradient [43] norm value at the vertex with the lowest function value. Simplicial gradient norm L' can be found for a vertex v_1 of a simplex S_k as follows:

$$B(S_k) = (\boldsymbol{v}_2 - \boldsymbol{v}_1, \boldsymbol{v}_3 - \boldsymbol{v}_1, ..., \boldsymbol{v}_{d+1} - \boldsymbol{v}_1), \tag{3.30}$$

$$F(S_k, f) = (f(\boldsymbol{v}_2) - f(\boldsymbol{v}_1), f(\boldsymbol{v}_3) - f(\boldsymbol{v}_1), ..., f(\boldsymbol{v}_{d+1}) - f(\boldsymbol{v}_1))^T, \tag{3.31}$$

$$\boldsymbol{v}_i \in V(S_k),\ i = 1, ..., d+1, \tag{3.32}$$

$$L' = ||B(S_l)^{-T} F(S_l, f)||. \tag{3.33}$$

2. Surrogate Lipschitz bounds are constructed using all vertices of a simplex:

$$g(\boldsymbol{x}) = \max_{\boldsymbol{v} \in V(S)} \{f(\boldsymbol{v}) - L||\boldsymbol{v} - \boldsymbol{x}||, \boldsymbol{x} \in S\} \tag{3.34}$$

   (just like in (3.15)). An inner optimization problem is solved in order to minimize these bounds.

3. Simplices are considered neighbouring if no more than two vertices are different.

4. Neither α, nor any other similar parameter is used in order to reduce the value of the Lipschitz constant estimate.

Algorithm 5: Description of the proposed global optimization algorithm using local Lipschitz constant estimate

Cover feasible region D by face-to-face simplicial partition $S = \{S_i : D = \cup S_i,\ i = 1, ..., d!\}$ using combinatorial vertex triangulation algorithm.
Evaluate $\{f(v_i) : v_i$ unique vertices of $S,\ i = 1, ..., 2^d\}$.
while *stopping condition is not satisfied* **do**
    foreach $S_l \in S$ **do**
        Find $\widehat{L}_l = \max\left\{\frac{|f(v_i)-f(v_j)|}{||v_i-v_j||} : v_i, v_j \in V(S_l), v_i \neq v_j\right\}$.
    foreach $S_l \in S$ **do**
        Find $B(S_l) = (v_2 - v_1, v_3 - v_1, ..., v_{d+1} - v_1),\ v_1 = v_{min},\ v_i \in V(S_l)$,
        $F(S_l, f) = (f(v_2) - f(v_1), f(v_3) - f(v_1), ..., f(v_{d+1}) - f(v_1))^T$,
        $\widetilde{L}_l =$
        $\max\left\{\widehat{L}_j : S_j, S_l\ has\ \leqslant\ 2\ different\ vertices\right\} \cup \left\{||B(S_l)^{-T}F(S_l, f)||\right\}$.
        Find $G_l = G(S_l, \widetilde{L})$ by solving inner optimization problem.
    Select a set of simplices for partitioning: $P = \{S_i : S_i \in S,\ S_i$ is a supported Pareto optimal solution to $\min(G_l, -\Delta(S_l))\}$
    foreach $S_l \in P$ **do**
        Divide S_l into two new simplices S_l^1, S_l^2, add a new vertex v in the middle of the longest edge of S_l.
        Update $S = S\backslash\{S_l\} \cup \{S_l^1, S_l^2\}$. If v is a new vertex, evaluate $f(v)$.
        Update f_{min}.
return f_{min}

The pseudocode of the proposed algorithm is provided in Algorithm 5. The inner level optimization problem was chosen to be solved using Algorithm 6, because the feasible region of the inner level optimization problem is a simplex and (3.34) function evaluations are computationally cheap.

In order to investigate the efficiency of the proposed algorithm, the same experimental methodology as described in Section 3.1.2 was chosen. Results for the first four GKLS test problem classes are provided in Table 3.8. As can be observed from this table, the performance of the algorithm proposed in this section is better than LIBRE with respect to average and median number of objective function evaluations for all four GKLS test problem classes. However, this is not true for problems with higher dimension than 3. It can be concluded that the performance of the algorithms using local Lipschitz constant estimate

Algorithm 6: Optimization algorithm of the inner level optimization problem

Input S_k - simplex over which Lipschitz bounds are minimized, $\widetilde{L}$ - Lipschitz constant estimate, g - function of surrogate Lipschitz bounds, M_{max} - maximum number of iterations (default value is 10).
Function *INNER*(S_k, $\widetilde{L}$, g $M_{max} = 10$)
    $S = \{S_k\}$
    $g_{min} = \min\{g(\boldsymbol{v}) : \boldsymbol{v} \in V(S_k)\}$
    $m = 1$
    while $m < M_{max}$ **do**
        foreach $S_l \in S$ **do**
            Find $G(S_l) = max\{g(l_1) - L\Delta(S_l), g(l_2) - L\Delta(S_l)\}$, where l_1, l_2 - vertices of the longest edge of S.
        Select a simplex for bisection: $S_p = \arg\min_{S_l \in S} G(S_l)$.
        Bisect S_p into two new simplices S_p^1, S_p^2, by adding a vertex $\boldsymbol{v}$ in the middle of the longest edge of S_p.
        Update $S = S \backslash \{S_p\} \cup \left\{S_p^1, S_p^2\right\}$. If $\boldsymbol{v}$ is a new vertex, evaluate $g(\boldsymbol{v})$ and update g_{min}.
        $m = m + 1$
    return g_{min}.

is impacted by the definition of the neighbouring simplices. Hence, more different strategies to define neighbouring simplices should be investigated.

3.3 Strategy for Generalizing Single-Objective Lipschitzian Optimization Algorithms to Multi-Objective Case

In this section, a general strategy for generalizing single-objective Lipschitzian global optimization algorithms to the multi-objective case is described. This strategy is applicable to Lipschitzian global optimization algorithms, which maintain an iteratively updated partition S of the feasible region D:

$$S = \{S_i : D = \cup S_i, i = 1, ..., k\}, \tag{3.35}$$

Table 3.8: Results obtained by solving 100 optimization problems from four GKLS test function classes, using the stopping condition with a known global minimizer

Class	Difficulty	Algorithm	Average	Median	Largest
1	Simple	DISIMPL-V	192.93	151.0	773
		GB-DISIMPL-V	174.25	151.0	472
		LIBRE α=0.4	151.97	146.5	371
		PROPOSED	**103.68**	**91**	**277**
2	Hard	DISIMPL-V	1003.56	1021.0	2683
		GB-DISIMPL-V	518.11	511.0	1547
		LIBRE α=0.4	431.55	515.0	**1117**
		PROPOSED	**384.02**	**298.5**	1714
3	Simple	DISIMPL-V	1061.83	787.0	4740
		GB-DISIMPL-V	**751.25**	**720.0**	2694
		LIBRE α=0.4	1009.72	959	2113
		PROPOSED	963.21	866.5	**2162**
4	Hard	DISIMPL-V	2598.91	2594.0	7354
		GB-DISIMPL-V	1364.40	1283.0	3723
		LIBRE α=0.4	1449.18	1390	3484
		PROPOSED	**1079.56**	**1029.5**	**2859**

such that one or more subregions are selected for partitioning in each iteration based on the minimum value of the lower Lipschitz bound over the subregion (see Figure 3.6 for general scheme). The lower the Lipschitz bound, the lower (better) function values can be obtained by partitioning that subregion. In this study, the following notation is used for the lower Lipschitz bounds and its minimum value over an arbitrary subregion S_i:

$$g(\boldsymbol{x}, S_i), \ \boldsymbol{x} \in S_i, \tag{3.36}$$

$$G(S_i) = \min_{\boldsymbol{x} \in S_i} g(\boldsymbol{x}, S_i). \tag{3.37}$$

An example of $g(\boldsymbol{x}, S_i)$ and the respective underestimate of $G(S_i)$ are used in [63,

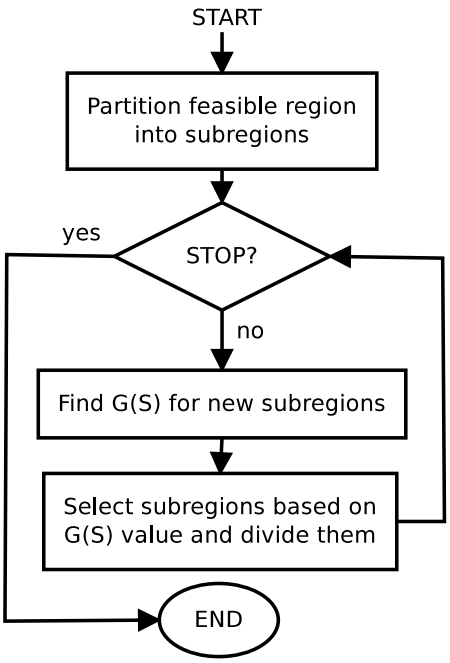


Figure 3.6: General Lipschitzian global optimization algorithm scheme

66] and in the proposed algorithm:

$$g^{single}(\mathbf{x}, S_i) = \min_{\mathbf{v} \in V(S_i)} f(\mathbf{v}) - L||\mathbf{x} - \arg\min_{\mathbf{v} \in V(S_i)} f(\mathbf{v})||, \ \mathbf{x} \in S_i. \tag{3.38}$$

$$G^{single}(S_i) \approx \min_{\mathbf{v} \in V(S_i)} f(\mathbf{v}) - L\Delta(S_i) \quad \leqslant \quad \min_{\mathbf{x} \in S_i} g^{single}(\mathbf{x}, S_i). \tag{3.39}$$

The largest possible improvement of the best currently known objective function value f_{min} by dividing the S_i is equal to $f_{min} - G(S_i)$. The largest possible improvement has to be maximized and if we want to minimize, $-(f_{min} - G(S_i)) = G(S_i) - f_{min}$ criterion is obtained. It is sufficient to use only $-G(S_i)$ or $G(S_i)$ to compare the subregions to each other, because f_{min} is fixed during the comparison in each iteration.

In the multi-objective case, the criterion defining the suitability of a subregion for further partitioning is not so trivial, because several contradicting objectives have to be considered at once. A vector of Lipschitz lower bounds for each of the objectives is defined as follows:

$$\mathbf{g}(\boldsymbol{x}, S_i) = (g^1(\boldsymbol{x}, S_i), ..., g^n(\boldsymbol{x}, S_i)). \tag{3.40}$$

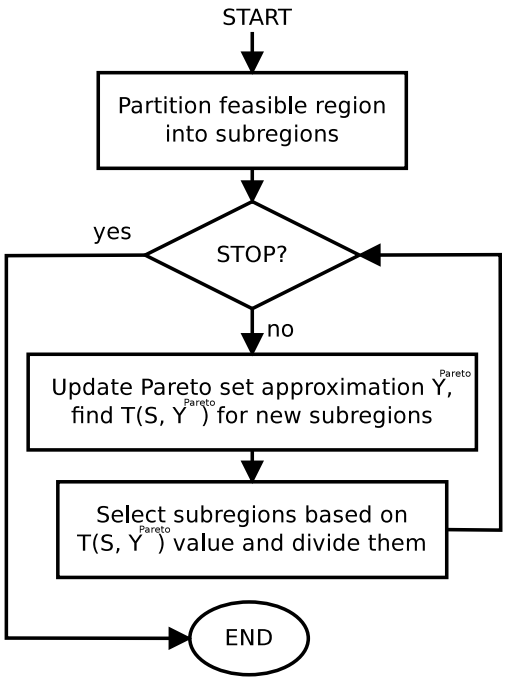


Figure 3.7: General Lipschitzian multi-objective optimization algorithm scheme after applying the suggested strategy to generalize Lipschitzian global optimization algorithm for multi-objective case

There have been some attempts to define the subregion selection (for division) criterion as the tightness of the multi-objective Lipschitz bounds over that subregion [89, 90]. In order to tighten the Lipschitz bounds, the subregions with maximum tightness values have to be selected for the division, i.e. tightness criterion has to be maximized. In the univariate case, when S_i is a line segment and $V(S_i)$ are its endpoints, the definition of the tightness in question using our notation is expressed as:

$$T(S_i) = \max \left\{ \min_{\xi \in \boldsymbol{G}^{strict}(S_i)} ||\xi - \boldsymbol{f}(\boldsymbol{v})|| : \boldsymbol{v} \in V(S_i) \right\}, \tag{3.41}$$

$$\boldsymbol{G}^{strict}(S_i) = \min_{\boldsymbol{x} \in S_i} \boldsymbol{g}^{strict}(\boldsymbol{x}, S_i), \tag{3.42}$$

$$\boldsymbol{g}^{strict}(\boldsymbol{x}, S_i) = (g^{strict,1}(\boldsymbol{x}, S_i), \ldots, g^{strict,n}(\boldsymbol{x}, S_i)), \tag{3.43}$$

$$g^{strict}(\boldsymbol{x}, S_i) = \max_{\boldsymbol{v} \in V(S_i)} \{f(\boldsymbol{v}) - L||\boldsymbol{v} - \boldsymbol{x}||\}, \ \boldsymbol{x} \in S_i, \tag{3.44}$$

where $\|\cdot\|$ is the Euclidean norm. An illustration of this definition for the uni-

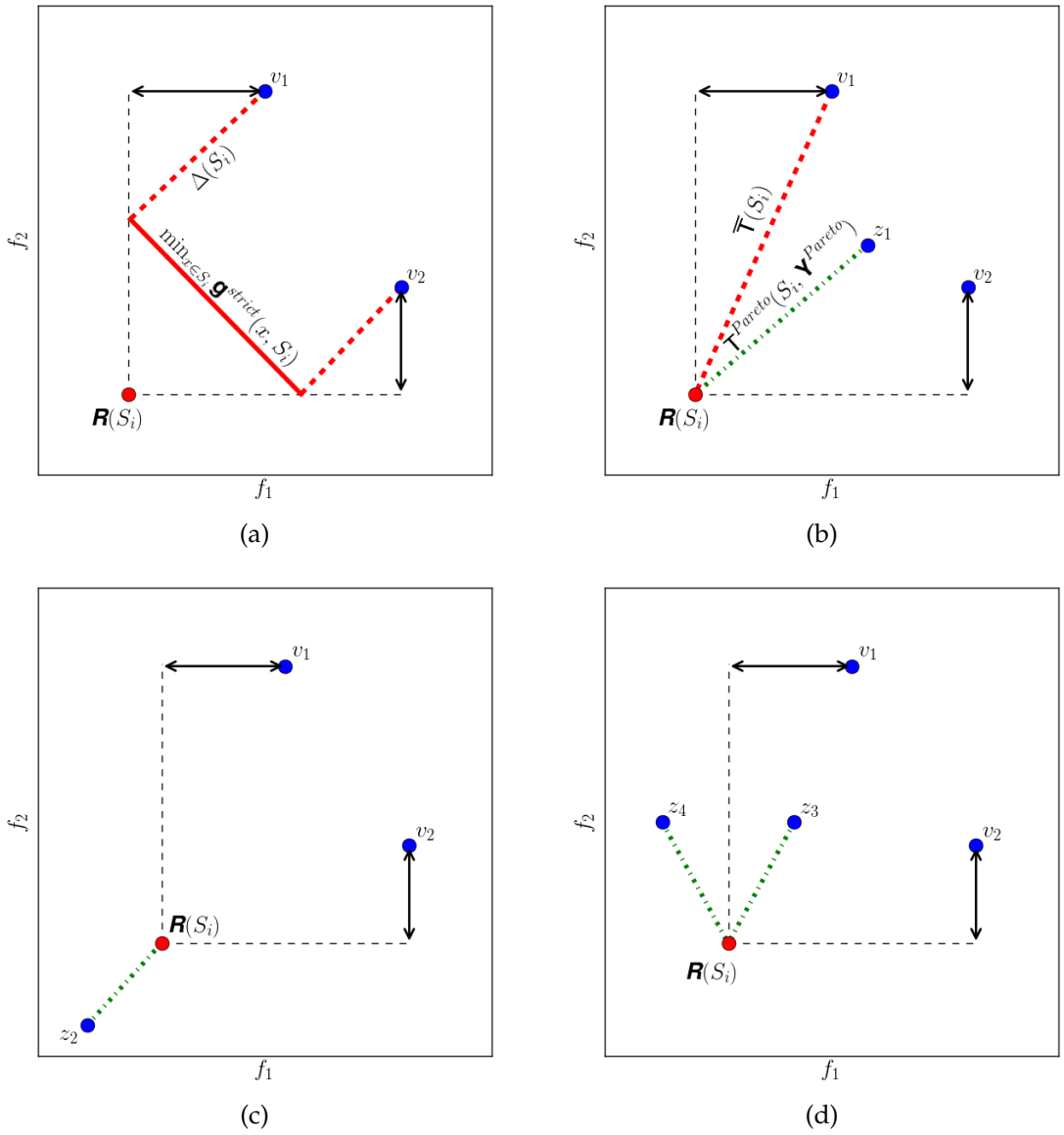


Figure 3.8: The local lower Lipschitz bounds over a univariate bi-objective subregion S_i visualized in objective space; v_1, v_2 are vertices of S_i

variate bi-objective case is provided in Figure 3.8a, where $V(S_i) = \{\boldsymbol{v}_1, \boldsymbol{v}_2\}$. In this particular case the set of points in the objective space (3.42) is a line segment. The bound tightness is then understood as the largest distance between this line segment and the objective space points corresponding to $V(S_i)$. However, for a higher number of objectives the set (3.42) is a hyper-surface and its complexity makes it hard to use criterion (3.41).

In order to overcome this limitation, a single point can be used to underestimate the local Lipschitz bounds. Specifically, the minimum of the lower Lipschitz bound can be found for each objective separately, so the bounding point is defined as:

$$\boldsymbol{R}(S_i) = (G^1(S_i), ..., G^n(S_i)) = (\min g^1(S_i), ..., \min g^n(S_i)). \tag{3.45}$$

The updated definition of the tightness for this case is:

$$\overline{T}(S_i) = \max_{\boldsymbol{v} \in V(S_i)} ||\boldsymbol{f}(\boldsymbol{v}) - \boldsymbol{R}(S_i)||. \tag{3.46}$$

See Figure 3.8b for an illustration of the univariate bi-objective case, where the discrete approximation of the Pareto front is $Y^{Pareto} = \{\mathbf{f}(\mathbf{v}_1), \mathbf{f}(\mathbf{v}_2), \mathbf{z}_1\}$ and $V(S_i) = \{\mathbf{v}_1, \mathbf{v}_2\}$. The tightness $\overline{T}(S_i)$ is equal to the largest distance between the bounding point $\boldsymbol{R}(S_i)$ and the objective space points corresponding to $V(S_i)$.

Definitions (3.41) and (3.46) reflect the largest possible improvement over the objective values at $V(S_i)$ that could result from partitioning subregion S_i. However, the definitions do not account for the circumstance that the improvement over some of the points in the whole Y^{Pareto} could be considerably smaller. For example, in Figure 3.8b it can be observed that $\overline{T}(S_i) < T^{Pareto}(S_i, Y^{Pareto})$. To address this issue, we propose to express the tightness in terms of the largest possible improvement of Y^{Pareto}:

$$T^{Pareto}(S_i, Y^{Pareto}) = \min_{\boldsymbol{y} \in Y^{Pareto}} ||\boldsymbol{y} - \boldsymbol{R}(S_i)||. \tag{3.47}$$

The minimization in (3.47) reflects the fact that the partitioning of S_i is supposed to improve upon the point in the Pareto front approximation that is closest to $\boldsymbol{R}(S_i)$. Figure 3.8b presents an illustration of definition (3.47), where the tightness $T^{Pareto}(S_i, Y^{Pareto})$ is equal to the distance between the bounding point $\boldsymbol{R}(S_i)$ and $\boldsymbol{z}_1 \in Y^{Pareto}$.

Further, we suggest two improvements of definition (3.47). First, let us consider the case, when there exists $\boldsymbol{x}_2 \in Y^{Pareto}$ that dominates $\boldsymbol{R}(S_i)$ (see Figure 3.8c). Expression (3.47) is positive in this case, implying that an improvement over the Pareto front approximation is possible by dividing S_i, which is not true. An updated definition (3.47) uses negative values to indicate that $\boldsymbol{R}(S_i)$ is dominated:

$$T'(S_i, Y^{Pareto}) = \begin{cases} \min\limits_{\boldsymbol{y} \in Y^{Pareto}} \|\boldsymbol{y} - \boldsymbol{G}(S_i)\|, & \text{if } \nexists \boldsymbol{y}' \succ \boldsymbol{G}(S_i),\ \boldsymbol{y}' \in Y^{Pareto}, \\ -\min\limits_{\boldsymbol{y} \in Y^{Pareto}} \|\boldsymbol{y} - \boldsymbol{G}(S_i)\|, & \text{if } \exists \boldsymbol{y}' \succ \boldsymbol{G}(S_i),\ \boldsymbol{y}' \in Y^{Pareto}. \end{cases} \tag{3.48}$$

Second, let us consider the case, when two different points $z_3, z_4 \in Y^{Pareto}$ are equidistant to $\boldsymbol{R}(S_i)$ (see Figure 3.8d). The value of (3.48) for S_i is the same for both z_3 and z_4. However, $\boldsymbol{R}(S_i)$ is better in all objectives than z_3, whereas only a single objective value is better in $\boldsymbol{R}(S_i)$ than in z_4. This example shows, that (3.48) should be further modified to account for the improvements of the objective values offered by $\boldsymbol{R}(S_i)$. Moreover, if at least one of the objective values is improved by $\boldsymbol{R}(S_i)$, the deterioration in other objective values should not be penalized. The required effect could be ensured by replacing the Euclidean norm in (3.48) by an asymmetric norm $|||\cdot|||$ which would ignore the deterioration of a value. We propose several variants of the considered asymmetric norm:

$$|||\boldsymbol{a} - \boldsymbol{b}|||_1 = \sum_{j=1}^{k} p(a_j, b_j), \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^k, \tag{3.49}$$

$$|||\boldsymbol{a} - \boldsymbol{b}|||_2 = \sqrt{\sum_{j=1}^{k} p(a_j, b_j)^2}, \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^k, \tag{3.50}$$

$$|||\boldsymbol{a} - \boldsymbol{b}|||_\infty = \max_{j=1,\ldots,k} p(a_j, b_j), \quad \boldsymbol{a}, \boldsymbol{b} \in \mathbb{R}^k, \tag{3.51}$$

$$p(a, b) = \max(0,\ a - b), \quad a, b \in \mathbb{R}. \tag{3.52}$$

An updated version of the definition (3.48) follows:

$$T(S_i, Y^{Pareto}) = \begin{cases} \min\limits_{\boldsymbol{y} \in Y^{Pareto}} |||\boldsymbol{y} - \boldsymbol{R}(S_i)|||_2, & \text{if } \nexists \boldsymbol{y}' \succ \boldsymbol{R}(S_i),\ \boldsymbol{y}' \in Y^{Pareto}, \\ -\min\limits_{\boldsymbol{y} \in Y^{Pareto}} |||\boldsymbol{R}(S_i) - \boldsymbol{y}|||_2, & \text{if } \exists \boldsymbol{y}' \succ \boldsymbol{R}(S_i),\ \boldsymbol{y}' \in Y^{Pareto}, \end{cases} \tag{3.53}$$

which might be interpreted as a minimal distance of $\boldsymbol{R}(S_i)$ from Y^{Pareto}.

To generalize a Lipschitzian global optimization algorithm to the multi-objective case, the Pareto front approximation Y^{Pareto} has to be updated in each itera-

tion. Furthermore, subregions for further partitioning should be selected using $T(S_i, Y^{Pareto})$ (3.53) criterion, instead of $G(S_i)$, estimating the largest possible improvement of the currently known best solution, which could result from dividing subregion S_i. It can be observed, that if only one objective function is used after applying this strategy to an algorithm, the result persists the same as before applying it. This is because for one objective $Y^{Pareto} = \{f_{min}\}$ and $T(S_i, Y^{Pareto}) = f_{min} - G(S_i)$. $T(S_i, Y^{Pareto})$ has to be maximized and if we want to minimize $-T(S_i, Y^{Pareto}) = G(S_i) - f_{min}$ the same criterion as before applying the generalization strategy is obtained. As it was mentioned before, f_{min} is constant in each iteration of an algorithm, so $G(S_i) - f_{min}$ is $G(S_i)$ value shifted by a constant, which does not make any impact on the result of comparing two values of the criterion.

3.4 Multi-Objective Version of the Libre Algorithm

3.4.1 Differences Between Single-Objective and Multi-Objective Versions of the Libre Algorithm

The strategy (described in the previous section 3.3) to generalize a single-objective algorithm for the multi-objective case was applied to the proposed Lipschitzian global optimization algorithm Libre (see section 3.1.1). The algorithm does not change for global optimization problems, after generalization, however, it gains the ability to solve multi-objective problems. Bearing this in mind, we preserve the same title Libre for multi-objective version of the algorithm. The pseudocode of the resulting multi-objective algorithm is provided in Algorithm 8. The remaining part of this section is devoted to highlighting some of the aspects of the multi-objective version of the algorithm.

The lower bounds of the Lipschitz constants have to be updated for each objective function separately. Lipschitz bounds are constructed only from a vertex with the best function value (see (3.38)) and overestimate (3.39) is used in order to find $\boldsymbol{R}(\cdot)$ (3.45) analytically.

The globality of the search can be chosen by setting the value of the Libre pa-

Algorithm 7: Algorithm to update Pareto front approximation

```
Input y - new solution, Y^Pareto - Pareto front approximation.
Function UpdateParetoFrontApproximation( y, Y^Pareto ):
    foreach p ∈ Y^Pareto do
        if p ≻ y then
            return Y^Pareto
    foreach p ∈ Y^Pareto do
        if y ≻ p then
            Y^Pareto = Y^Pareto \ p
    Y^Pareto = Y^Pareto ∪ {y}
    return Y^Pareto
```

rameter $\alpha \in [0, \infty)$. Two criteria (3.3) are used to select a set of simplices for division in a single-objective version of the algorithm. The first criterion is the largest expected improvement of the currently known best solution, which for a single-objective case is the lowest observed function value f_{min} and for multi-objective case it is a Pareto set approximation Y^{Pareto}. The second criterion is the diameter of the simplex $\Delta(\cdot)$. If for multi-objective case, the (3.53) definition of the largest expected improvement of the currently known best solution is used and $\alpha\widetilde{\boldsymbol{L}}$ is used instead of $\boldsymbol{L}$, then a set of simplices for division can be identified by solving:

$$\max_{S_i \in S}(T(S_i, Y^{Pareto}, \alpha\widetilde{\boldsymbol{L}}),\ \Delta(S_i)), \tag{3.54}$$

where $\widetilde{\boldsymbol{L}}$ is a vector of Lipschitz constant estimates. It was shown in [25], that for a single-objective version of the algorithm the higher the α value is the more globally biased the search gets. Higher α value causes only the simplices with higher diameter values to be selected for the division. The same effect of α persists also for multi-objective case.

Multi-objective version of the Libre algorithm has to track and constantly update Pareto front approximation Y^{Pareto}. It can be achieved by firstly initializing $Y^{Pareto} = \varnothing$ and then applying Algorithm 7 for each new trial to update the Y^{Pareto} with nondominated solutions.

The stopping condition might be associated with the maximum expected improvement of the Pareto front approximation (as an alternative to the stopping

Algorithm 8: Description of multi-objective version of the Libre algorithm

Normalize the feasible region D to be the unit-hypercube $\overline{D}$.
Cover $\overline{D}$ by face-to-face simplices $S = \{S_i : \overline{D} = \cup S_i,\ i = 1, ..., d!\}$ using combinatorial vertex triangulation algorithm.
Evaluate $\{\boldsymbol{f}(\boldsymbol{v}_i) : \boldsymbol{v}_i$ unique vertice of $S,\ i = 1, ..., 2^d\}$. Set $m = 2^d$, $\widetilde{\boldsymbol{L}} = 0$.
Find Pareto front approximation Y^{Pareto}.
while *stopping condition not satisfied* **and** $m < M_{max}$ **do**
  for $k = 1$ ***to*** $|S|$ **do**
    find $\widetilde{\boldsymbol{L}}_k = (\widetilde{L}_k^1, ..., \widetilde{L}_k^n)$,
    $\widetilde{L}_k^i = \max\left\{\frac{|f^i(\boldsymbol{v}_i) - f^i(\boldsymbol{v}_j)|}{||\boldsymbol{v}_i - \boldsymbol{v}_j||} : \boldsymbol{v}_i, \boldsymbol{v}_j \in V(S_k), \boldsymbol{v}_i \neq \boldsymbol{v}_j\right\}$.
  Update $\widetilde{\boldsymbol{L}} = \left(\max_{k\in\{1,...,|S|\}} \widetilde{L}_k^1,\ ...,\ \max_{k\in\{1,...,|S|\}} \widetilde{L}_k^n\right)$.
  for $k = 1$ ***to*** $|S|$ **do**
    find $T_k = T(S_k, Y^{Pareto}, \widetilde{\boldsymbol{L}}\alpha)$ based on (3.53), where (3.38), (3.39) are used to find Lipschitz bounds in (3.45) and $\widetilde{\boldsymbol{L}}\alpha$ is used instead of $\boldsymbol{L}$.
  Identify a set of simplices for division:
    $P = \{S_i : S_i \in S,\ S_i$ - supported Pareto optimal solution to (3.54)$\}$.
  foreach $S_k \in P$ **do**
    Divide S_k into two new simplices S_k^1, S_k^2, by adding a vertex $\boldsymbol{v}$ in the middle of the longest edge of S_k.
    Update $S = S\backslash\{S_k\} \cup \{S_k^1, S_k^2\}$. If $\boldsymbol{v}$ is a new vertex, evaluate $\boldsymbol{f}(\boldsymbol{v})$, set $m = m + 1$ and update Pareto front approximation Y^{Pareto}.

condition (3.7) for global optimization problems):

$$\max_{S_i \in S} T(S_i, Y^{Pareto}) \leqslant \epsilon. \tag{3.55}$$

3.4.2 Experimental Investigation

3.4.2.1 Performance Comparison to the NSGA-II

The resulting multi-objective Libre algorithm was numerically evaluated employing two popular multi-objective test problem sets ZDT [93] and DTLZ [16]. The results were compared with one of the most popular genetic algorithms NSGA-II [15].

An implementation of the NSGA-II algorithm from DEAP framework [21] with default parameter values was used. Libre algorithm with $\alpha = 0.1$ was used as well. Dimension of the problems was set to 4 (the same as in GKLS test problem generator [24] 5'th and 6'th classes). The budget of 15000 trials was used. Change of the hyper volume was measured (see Figures 3.9a-3.9l), where the X and Y axes indicate the number of trials and the hyper volume, respectively. The blue line denotes the results of the Libre algorithm and the green area denotes the results (from minimum to maximum) obtained from 10 different NSGA-II runs.

The results show that the proposed multi-objective Libre algorithm performs better (reaches best observed hyper volume value faster) than NSGA-II in 7 cases out of 12. The conclusion to be drawn is that the algorithm obtained by applying the suggested strategy demonstrated the performance comparable to that of the NSGA-II, so both the suggested strategy and the Libre algorithm should be further investigated.

3.4.2.2 Performance Comparison to Lipschitzian Optimization Algorithms

The Libre algorithm was compared with OSWCO (One-Step Worst-Case Optimal) [91] and NUC (Nonuniform Covering Method) [18] algorithms. The same two bi-objective test problems and the same two quantitative characteristics as in [18, 91] were used for experimentation. The characteristics are Hyper volume and Uniformity of Y^{Pareto} points distribution which were proposed in [94].

The definition of the first bi-objective test problem (PE1) is:

$$\min_{x \in [0,2]^2} \boldsymbol{f}(x), \quad f^1(x) = x_1, \quad f^2(x) = \min(|x_1 - 1|, 1.5 - x_1) + x_2 + 1. \tag{3.56}$$

Pareto front of PE1 is disconnected and according to [18] Uniformity characteristic is not suitable for this problem. Nevertheless, we have calculated it for PE1, having ensured the covering of Pareto front to be dense enough, i. e. $\forall y \in Y^{Pareto}$, $\forall y' \in Y^{Pareto}$ if y and y' are nearest neighbours $\implies$ the nearest points from Pareto front to y, y' belong to the same disconnected part of Pareto front.

Table 3.9: Algorithm performance comparison on two bi-objective problems

Problem	Algorithm	Calls	$\lvert Y^{Pareto}\rvert$	Hyper volume	Uniformity	Nadir
PE1	Libre $\alpha = 0.1$	438	**163**	**3.61328**	-	(2,3)
	OSWCO	435	92	3.60519	-	
	NUC	490	36	3.42	-	
PE2	Libre $\alpha = 0.1$	484	**221**	**0.325086**	**0.086283**	(1,1)
	OSWCO	498	68	0.308484	0.174558	
	NUC	515	29	0.306	0.210	

The definition of the second bi-objective test problem (PE2) is:

$$\min_{x\in[0,1]^2} \boldsymbol{f}(x), \quad f^1(x) = (x_1 - 1)x_2^2 + 1, \quad f^2(x) = x_2. \tag{3.57}$$

The numerical results are provided in Table 3.9. $\alpha = 0.1$ was used in Libre algorithm for the experimental comparison. As can be seen from the Table 3.9, the performance of the Libre algorithm is better than the performance of the alternatives.

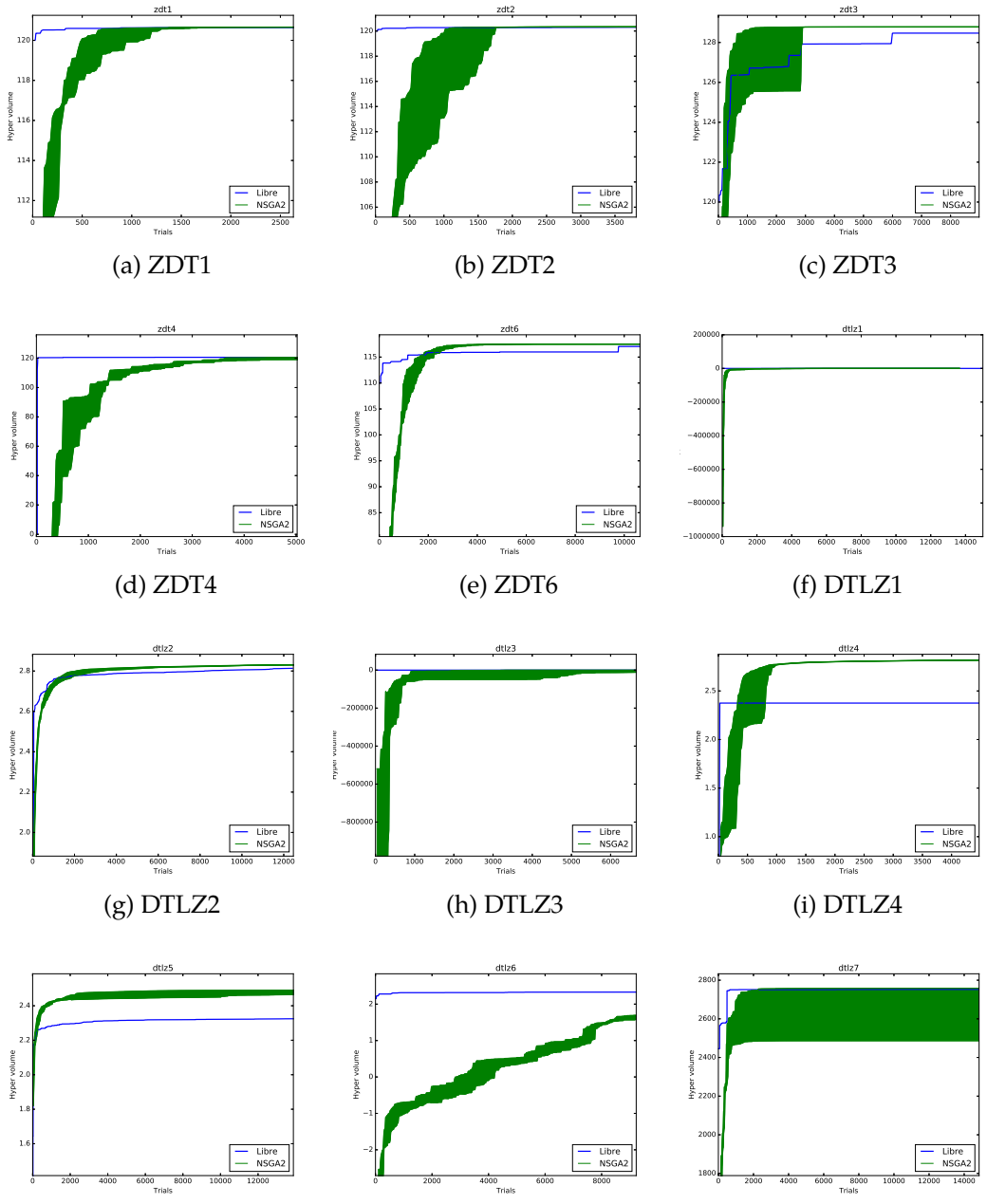
3.4.2.3 Insights about Libre Algorithm Parameter α

As mentioned previously, it was shown in [25], that higher α value causes only the simplices with higher diameter values to be selected for the division. A hypothesis can be made that in multi-objective case α parameter regulates the ration between Pareto front approximation's hyper volume and its covering uniformity. The impact of α value on the result was examined for a multi-objective Libre algorithm. Results with a set of different α values are provided in Table 3.10.

As can be observed from Table 3.10, the hypothesis about α value's impact on the results is not always true. It is obvious that when α value is increased, the Hyper volume decreases and Uniformity of Y^{Pareto} increases only for PE1 and only up to $\alpha = 0.8$.

Table 3.10: LIBRE algorithm results with different α values for two bi-objective optimization problems

	PE1				PE2			
α	Calls	$\lvert Y^{Pareto}\rvert$	Hyper vol	Uniformity	Calls	$\lvert Y^{Pareto}\rvert$	Hyper vol	Uniformity
0.01	463	181	3.61328	0.408823	483	221	0.325200	0.084739
0.1	486	195	3.61328	0.410380	484	221	0.325086	0.086283
0.2	492	178	3.60156	0.240059	487	202	0.324590	0.091654
0.3	487	139	3.60156	0.188248	482	191	0.319475	0.111888
0.4	474	130	3.60156	0.153485	481	186	0.317870	0.112702
0.5	476	130	3.60156	0.153485	482	184	0.317862	0.112672
0.6	482	120	3.60156	0.151383	481	176	0.317862	0.111554
0.7	480	106	3.60156	0.147161	481	173	0.317862	0.106292
0.8	499	106	3.60156	0.147161	481	173	0.317862	0.105668
0.9	475	101	3.60156	0.325707	482	173	0.317862	0.105668
1.0	474	92	3.60156	0.322507	482	169	0.317862	0.113096



(a) ZDT1 (b) ZDT2 (c) ZDT3

(d) ZDT4 (e) ZDT6 (f) DTLZ1

(g) DTLZ2 (h) DTLZ3 (i) DTLZ4

(j) DTLZ5 (k) DTLZ6 (l) DTLZ7

Figure 3.9: Comparison of Libre and NSGA-II [15] algorithms using continuous test from problems from ZDT and DTLZ test problem sets

Chapter 4

General Conclusions

An algorithm that combines the elements of the DISIMPL-V algorithm with an adaptive Lipschitz constant estimation strategy was proposed. Specifically, the simplicial decomposition, the sub-region selection strategy and the Lipschitz bounds computation were reused from the original DISIMPL-V algorithm. The proposed algorithm was experimentally compared to other DIRECT-type Lipschitz optimization algorithms using 800 objective functions produced by the GKLS test function generator. The experimental analysis of the algorithm's single parameter α, which regulates the globality of the search, was conducted.

Four strategies to estimate the Lipschitz lower bound within a simplex were considered in this thesis. They were applied to the proposed algorithm, and experiments were conducted to select the best one.

A strategy generalizing a single-objective Lipschitzian optimization algorithm to the multi-objective case was proposed. It is based on estimating the potential improvement over the current approximation of the Pareto front that might be obtained after dividing any simplex in the current decomposition. The strategy was applied to the LIBRE algorithm and the resulting multi-objective algorithm was experimentally compared to the NSGA-II algorithm.

Conclusions:

1. It was experimentally demonstrated that the proposed LIBRE algorithm performs the best among other investigated alternatives for complex test

function classes in the worst-case scenario. The worst-case results for the proposed algorithm were not the best only for 2 classes of simple functions out of the considered 8. Moreover, the LIBRE algorithm improves upon the original DSIMPL-V algorithm in 7 classes out of 8 both in worst and average cases.

2. The experiments showed that the most efficient strategy to estimate the Lipschitz bounds within a simplex in terms of the number of function evaluations is v_{min}, corresponding to using a single vertex with the best function value. The application of this strategy results in the best average function evaluation time and the lowest number of function evaluations across all test function classes considered. The bounds obtained with this strategy are the least tight ones among the considered alternatives, therefore it can be concluded that tighter surogate Lipschitz bounds do not necessarily lead to increased efficiency of the DIRECT-type optimization methods.

3. The comparative experiments revealed that the generalized multi-objective version of the proposed LIBRE algorithm performs similarly to the NSGA-II algorithm for low-dimensional optimization problems. For 7 out of 12 functions in the ZDT and DTLZ test suites, the proposed algorithm reached not worse hyper-volume values after any given number of function evaluations.

Bibliography

[1] E. Aarts, J. Korst, and W. Michiels. "Simulated annealing". In: *Search methodologies*. Springer, 2005, pages 187–210.

[2] C. A. Baker, L. T. Watson, B. Grossman, W. H. Mason, and R. T. Haftka. "Parallel global aircraft configuration design space exploration". In: (2000). (Preprint).

[3] G. R. Baoping Z. Wood and W. P. Baritompa. "Multidimensional bisection: the performance and the context". In: *Journal of Global Optimization* 3.3 (1993), pages 337–358.

[4] M. C. Bartholomew-Biggs, S. C. Parkhurst, and S. P. Wilson. "Using DIRECT to solve an aircraft routing problem". In: *Computational Optimization and Applications* 21.3 (2002), pages 311–323.

[5] R. Battiti, M. Brunato, and F. Mascia. *Reactive search and intelligent optimization*. Volume 45. Springer Science & Business Media, 2008.

[6] M. Björkman and K. Holmström. "Global Optimization Using the DIRECT Algorithm in Matlab". In: *Matlab. Advanced Modeling and Optimization 1 (2)*. Citeseer. 1999, pages 17–37.

[7] A. R. Butz. "Space filling curves and mathematical programming". In: *Information and Control* 12.4 (1968), pages 314–330.

[8] R. G. Carter, J. M. Gablonsky, A. Patrick, C. T. Kelley, and O. J. Eslinger. "Algorithms for noisy problems in gas transmission pipeline optimization". In: *Optimization and engineering* 2.2 (2001), pages 139–157.

[9] L. G. Casado, E. M. T. Hendrix, and I. García. "Infeasibility spheres for finding robust solutions of blending problems with quadratic constraints". In: *Journal of Global Optimization* 39.4 (2007), pages 577–593.

[10] L. Chiter. "DIRECT algorithm: A new definition of potentially optimal hyperrectangles". In: *Applied Mathematics and computation* 179.2 (2006), pages 742–749.

[11] J. Clausen and A. Žilinskas. *Global optimization by means of branch and bound with simplex based covering*. IMM, Department of Mathematical Modelling, Technical Universityof Denmark, 1994.

[12] Steven E Cox, Raphael T Haftka, Chuck A Baker, Bernard Grossman, William H Mason, and Layne T Watson. "A comparison of global optimization methods for the design of a high-speed civil transport". In: *Journal of Global Optimization* 21.4 (2001), pages 415–432.

[13] Yu. M. Danilin. "Estimation of the efficiency of an absolute-minimum-finding algorithm". In: *USSR Computational Mathematics and Mathematical Physics* 11.4 (1971), pages 261–267.

[14] Yu. M. Danilin and S. A. Pijavskij. "An algorithm for finding the absolute minimum". In: *Theory of Optimal Decisions* 2 (1967), pages 25–37.

[15] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. "A fast and elitist multiobjective genetic algorithm: NSGA-II". In: *IEEE transactions on evolutionary computation* 6.2 (2002), pages 182–197.

[16] K. Deb, L. Thiele, M. Laumanns, and E. Zitzler. "Scalable multi-objective optimization test problems". In: *Evolutionary Computation, 2002. CEC'02. Proceedings of the 2002 Congress on*. Volume 1. IEEE. 2002, pages 825–830.

[17] R. C. Eberhart, Y. Shi, and J. Kennedy. "Swarm intelligence (The Morgan Kaufmann series in evolutionary computation)". In: (2001).

[18] Yu. G. Evtushenko and M. A. Posypkin. "Nonuniform covering method as applied to multicriteria optimization problems with guaranteed accuracy". In: *Computational Mathematics and Mathematical Physics* 53.2 (2013), pages 144–157.

[19] D. E. Finkel and C. T. Kelley. "Additive scaling and the DIRECT algorithm". In: *Journal of Global Optimization* 36.4 (2006), pages 597–608.

[20] C. A. Floudas and P. M. Pardalos. *Encyclopedia of Optimization, second edn.* Springer Science & Business Media, 2009.

[21] F. A. Fortin, F. M. De Rainville, M. A. Gardner, M. Parizeau, and C. Gagné. "DEAP: Evolutionary algorithms made easy". In: *Journal of Machine Learning Research* 13.Jul (2012), pages 2171–2175.

[22] J. M. Gablonsky and C. T. Kelley. "A locally-biased form of the DIRECT algorithm". In: *Journal of Global Optimization* 21.1 (2001), pages 27–37.

[23] E. A. Galperin. "The Cubic Algorithm". In: *Journal of Mathematical Analysis and Applications* 112.2 (1985), pages 635–640.

[24] M. Gaviano, D. E. Kvasov, D. Lera, and Ya. D. Sergeyev. "Algorithm 829: Software for generation of classes of test functions with known local and global minima for global optimization". In: *ACM Transactions on Mathematical Software (TOMS)* 29.4 (2003), pages 469–480.

[25] A. Gimbutas and A. Žilinskas. "An algorithm of simplicial Lipschitz optimization with the bi-criteria selection of simplices for the bi-section". In: *Journal of Global Optimization* 71.1 (2018), pages 115–127.

[26] Albertas Gimbutas. "Globalios optimizacijos algoritmas, naudojantis lokalų Lipšico konstantos įvertį". In: *Jaunųjų mokslininkų darbai* 1.45 (2016), pages 47–53.

[27] E. Gourdin, P. Hansen, and B. Jaumard. "Global optimization of multivariate Lipschitz functions: survey and computational comparison". In: *Les Cahiers du GERAD* (1994).

[28] R. Grbić, E. K. Nyarko, and R. Scitovski. "A modification of the DIRECT method for Lipschitz global optimization for a symmetric function". In: *Journal of Global Optimization* 57.4 (2013), pages 1193–1212.

[29] P. Hansen and B. Jaumard. "Lipshitz optimization". In: *Handbook of Global Optimization*. Edited by R. Horst and P. Pardalos. Dodrecht: Kluwer Academic Publisher, 1995, pages 407–493.

[30] P. Hansen, B. Jaumard, and S. H. Lu. "Global optimization of univariate Lipschitz functions: II. New algorithms and computational comparison". In: *Mathematical programming* 55.1 (1992), pages 273–292.

[31] P. Hansen, B. Jaumard, and S. H. Lu. "On the number of iterations of Piyavskii's global optimization algorithm". In: *Mathematics of Operations Research* 16.2 (1991), pages 334–350.

[32] J. He, L. T. Watson, N. Ramakrishnan, C. A. Shaffer, A. Verstak, J. Jiang, K. Bae, and W. H. Tranter. "Dynamic data structures for a direct search algorithm". In: *Computational Optimization and Applications* 23.1 (2002), pages 5–25.

[33] R. Horst. "A general class of branch-and-bound methods in global optimization with some new approaches for concave minimization". In: *Journal of Optimization Theory and Applications* 51.2 (1986), pages 271–291.

[34] R. Horst. "Bisecton by global optimization revisited". In: *Journal of optimization theory and applications* 144.3 (2010), pages 501–510.

[35] R. Horst. "On generalized bisection of n-simplices". In: *Mathematics of Computation of the American Mathematical Society* 66.218 (1997), pages 691–698.

[36] R. Horst and H. Tuy. *Global Optimization: Deterministic Approaches*. Springer Science & Business Media, 2013.

[37] V. V. Ivanov. "On optimal algorithms for the minimization of functions of certain classes". In: *Kibernetika* 4 (1972), pages 81–94.

[38] V. V. Ivanov. "Optimal Algorithms of Minimization in the Class of Functions with the Lipschitz Condition." In: *Information Processing* 71 (1972), pages 1324–1327.

[39] B. Jaumard, H. Ribault, and T. Herrmann. "An on-line cone intersection algorithm for global optimization of multivariate Lipschitz functions". In: *Les Cahiers du GERAD* (1995).

[40] D. R. Jones, C. D. Perttunen, and B. E. Stuckman. "Lipschitzian optimization without the Lipschitz constant". In: *Journal of Optimization Theory and Applications* 79.1 (1993), pages 157–181.

[41] D. Karaboga and B. Akay. "A comparative study of artificial bee colony algorithm". In: *Applied mathematics and computation* 214.1 (2009), pages 108–132.

[42] D. Karaboga and B. Basturk. "A powerful and efficient algorithm for numerical function optimization: artificial bee colony (ABC) algorithm". In: *Journal of global optimization* 39.3 (2007), pages 459–471.

[43] Carl T Kelley. *Iterative methods for optimization*. Volume 18. Siam, 1999.

[44] D. E. Kvasov and M. S. Mukhametzhanov. "Metaheuristic vs. deterministic global optimization algorithms: the univariate case". In: *Applied Mathematics and Computation* 318 (2018), pages 245–259.

[45] D. E. Kvasov, C. Pizzuti, and Ya. D. Sergeyev. "Local tuning and partition strategies for diagonal GO methods". In: *Numerische Mathematik* 94.1 (2003), pages 93–106.

[46] D. E. Kvasov and Ya. D. Sergeyev. "A univariate global search working with a set of Lipschitz constants for the first derivative". In: *Optimization Letters* 3.2 (2009), pages 303–318.

[47] D. E. Kvasov and Ya. D. Sergeyev. "Lipschitz gradients for global optimization in a one-point-based partitioning scheme". In: *Journal of Computational and Applied Mathematics* 236.16 (2012), pages 4042–4054.

[48] D. E. Kvasov and Ya. D. Sergeyev. "Univariate geometric Lipschitz global optimization algorithms". In: *Numerical Algebra, Control & Optimization* 2.1 (2012), pages 69–90.

[49] Q. Liu and W. Cheng. "A modified DIRECT algorithm with bilevel partition". In: *Journal of Global Optimization* 60.3 (2014), pages 483–499.

[50] Q. Liu and J. Zeng. "Global optimization by multilevel partition". In: *Journal of Global Optimization* 61.1 (2015), pages 47–69.

[51] G. Liuzzi, S. Lucidi, and V. Piccialli. "A DIRECT-based approach exploiting local minimizations for the solution of large-scale global optimization problems". In: *Computational Optimization and Applications* 45.2 (2010), pages 353–375.

[52] G. Liuzzi, S. Lucidi, and V. Piccialli. "A partition-based global optimization algorithm". In: *Journal of Global Optimization* 48.1 (2010), pages 113–128.

[53] G. Liuzzi, S. Lucidi, and V. Piccialli. "Exploiting derivative-free local searches in DIRECT-type algorithms for global optimization". In: *Computational Optimization and Applications* 65.2 (2016), pages 449–475.

[54] M. Locatelli. "Simulated annealing algorithms for continuous global optimization". In: *Handbook of global optimization*. Springer, 2002, pages 179–229.

[55] C. Malherbe and N. Vayatis. "Global optimization of Lipschitz functions". In: *arXiv preprint arXiv:1703.02628* (2017).

[56] D. Q. Mayne and E. Polak. "Outer approximation algorithm for nondifferentiable optimization problems". In: *Journal of Optimization Theory and Applications* 42.1 (1984), pages 19–30.

[57] C. C. Meewella and D. Q. Mayne. "Efficient domain partitioning algorithms for global optimization of rational and Lipschitz continuous functions". In: *Journal of Optimization Theory and Applications* 61.2 (1989), pages 247–270.

[58] R. H. Mladineo. "An algorithm for finding the global maximum of a multimodal, multivariate function". In: *Mathematical Programming* 34.2 (1986), pages 188–200.

[59] J. Mockus. "On the Pareto optimality in the context of Lipschitzian optimization". In: *Informatica* 22.4 (2011), pages 521–536.

[60] A. Molinaro, C. Pizzuti, and Ya. D. Sergeyev. "Acceleration tools for diagonal information global optimization algorithms". In: *Computational Optimization and Applications* 18.1 (2001), pages 5–26.

[61] M. Nast. "Subdivision of simplices relative to a cutting plane and finite concave minimization". In: *Journal of Global Optimization* 9.1 (1996), pages 65–93.

[62] P. M. Pardalos and H. E. Romeijn. "Handbook of global optimization, vol. 2". In: *Nonconvex Optim. Appl., Kluwer, Dordrecht, The Netherlands* (2002).

[63] R. Paulavičius, Ya. D. Sergeyev, D. E. Kvasov, and J. Žilinskas. "Globally-biased DISIMPL algorithm for expensive global optimization". In: *Journal of Global Optimization* 59.2-3 (2014), pages 545–567.

[64] R. Paulavičius and J. Žilinskas. "Analysis of different norms and corresponding Lipschitz constants for global optimization". In: *Technological and Economic Development of Economy* 12.4 (2006), pages 301–306.

[65] R. Paulavičius and J. Žilinskas. *Simplicial global optimization*. Springer, 2014.

[66] R. Paulavičius and J. Žilinskas. "Simplicial Lipschitz optimization without the Lipschitz constant". In: *Journal of Global Optimization* 59.1 (2014), pages 23–40.

[67] S. Pijavskij. "An algorithm for finding the global extremum of function". In: *Optimal Decisions* 2 (1967), pages 13–24.

[68] J. D. Pintér. "Branch-and bound algorithms for solving global optimization problems with Lipschitzian structure". In: *Optimization* 19.1 (1988), pages 101–110.

[69] J. D. Pintér. *Global optimization in action: continuous and Lipschitz optimization: algorithms, implementations and applications*. Volume 6. Nonconvex Optimization and Its Applications. Kluwer Academic Publishers, 1996.

[70] K. Price, R. M. Storn, and J. A. Lampinen. *Differential evolution: a practical approach to global optimization*. Springer Science & Business Media, 2006.

[71] F. Schoen. "On a sequential search strategy in global optimization problems". In: *Calcolo* 19.3 (1982), pages 321–334.

[72] D. Serafino, G. Liuzzi, V. Piccialli, F. Riccio, and G. Toraldo. "A modified DIviding RECTangles algorithm for a problem in astrophysics". In: *Journal of optimization theory and applications* 151.1 (2011), pages 175–190.

[73] Ya. D. Sergeyev. "An information global optimization algorithm with local tuning". In: *SIAM Journal on Optimization* 5.4 (1995), pages 858–870.

[74] Ya. D. Sergeyev. "Efficient strategy for adaptive partition of N-dimensional intervals in the framework of diagonal algorithms". In: *Journal of Optimization Theory and Applications* 107.1 (2000), pages 145–168.

[75] Ya. D. Sergeyev and D. E. Kvasov. *Deterministic Global Optimization: An Introduction to the Diagonal Approach*. Springer, 2017.

[76] Ya. D. Sergeyev and D. E. Kvasov. *Diagonalnyje metody globalnoj optimizacii*. In Russian. Moscow: Fizmatlit, 2008.

[77] Ya. D. Sergeyev and D. E. Kvasov. "Global search based on efficient diagonal partitions and a set of Lipschitz constants". In: *SIAM Journal on Optimization* 16.3 (2006), pages 910–937. DOI: http://dx.doi.org/10.1137/040621132.

[78] Ya. D. Sergeyev and D. E. Kvasov. "Global search based on efficient diagonal partitions and a set of Lipschitz constants". In: *SIAM Journal on Optimization* 16.3 (2006), pages 910–937.

[79] Ya. D. Sergeyev, M. S. Mukhametzhanov, D. E. Kvasov, and D. Lera. "Derivative-free local tuning and local improvement techniques embedded in the univariate global optimization". In: *Journal of Optimization Theory and Applications* 171.1 (2016), pages 186–208.

[80] Ya. D. Sergeyev, R. G. Strongin, and D. Lera. *Introduction to global optimization exploiting space-filling curves*. Springer Science & Business Media, 2013.

[81] B. O. Shubert. "A sequential method seeking the global maximum of a function". In: *SIAM Journal on Numerical Analysis* 9.3 (1972), pages 379–388.

[82] R. G. Strongin. "Algorithms for multi-extremal mathematical programming problems employing the set of joint space-filling curves". In: *Journal of Global Optimization* 2.4 (1992), pages 357–378.

[83] R. G. Strongin and Ya. D. Sergeyev. *Global optimization with non-convex constraints: Sequential and parallel algorithms*. Kluwer Academic Publishers, Dordrecht, 2000.

[84] A. G. Sukharev. "Optimal strategies of the search for an extremum". In: *USSR Computational Mathematics and Mathematical Physics* 11.4 (1971), pages 119–137.

[85] L. N. Timonov. "Algorithm for search of a global extremum". In: *Engineering Cybernetics* 15.3 (1977), pages 38–44.

[86] G. R. Wood. "Multidimensional bisection applied to global optimisation". In: *Computers & Mathematics with Applications* 21.6-7 (1991), pages 161–172.

[87] X. S. Yang. *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010.

[88] X. S. Yang and X. He. "Firefly algorithm: recent advances and applications". In: *International Journal of Swarm Intelligence* 1.1 (2013), pages 36–50.

[89] A. Žilinskas. "A one-step worst-case optimal algorithm for bi-objective univariate optimization". In: *Optimization Letters* 8.7 (2014), pages 1945–1960.

[90] A. Žilinskas and G. Gimbutienė. "On one-step worst-case optimal trisection in univariate bi-objective Lipschitz optimization". In: *Communications in Nonlinear Science and Numerical Simulation* 35 (2016), pages 123–136.

[91] A. Žilinskas and J. Žilinskas. "Adaptation of a one-step worst-case optimal univariate algorithm of bi-objective Lipschitz optimization to multidimensional problems". In: *Communications in Nonlinear Science and Numerical Simulation* 21.1-3 (2015), pages 89–98.

[92] J. Žilinskas. "Branch and bound with simplicial partitions for global optimization". In: *Mathematical Modelling and Analysis* 13.1 (2008), pages 145–159.

[93] E. Zitzler, K. Deb, and L. Thiele. "Comparison of multiobjective evolutionary algorithms: empirical results". In: *Evolutionary computation* 8.2 (2000), pages 173–195.

[94] E. Zitzler, J. Knowles, and L. Thiele. "Quality assessment of Pareto set approximations". In: *Multiobjective Optimization*. Springer, 2008, pages 373–404.

[95] E. Zitzler, M. Laumanns, and L. Thiele. "SPEA2: Improving the strength Pareto evolutionary algorithm". In: *TIK-report* 103 (2001).

Publications by the Author

Publications in Peer Reviewed Periodicals

1. Gimbutas, A. (2016). Globalios optimizacijos algoritmas, naudojantis lokalų Lipšico konstantos įvertį. *Jaunųjų mokslininkų darbai*, Vol. 1, No. 45, pp. 47-53. doi:10.21277/jmd.v1i45.44

2. Gimbutas, A., and Žilinskas, A. (2018). An algorithm of simplicial Lipschitz optimization with the bi-criteria selection of simplices for the bisection. *Journal of Global Optimization*, 71(1), pp. 115-127. doi:10.1007/s10898-017-0550-9

Peer Reviewed Conference Publications

3. Gimbutas, A. and Žilinskas, A. (2016). On global optimization using an estimate of Lipschitz constant and simplicial partition. In: *Numerical Computations: Theory and Algorithms*. Calabria: AIP Publishing, Vol. 1776, No. 1, p.060012. doi:10.1063/1.4965346

4. Gimbutas, A. and Žilinskas, A. (2018). Generalization of Lipschitzian global optimization algorithms to the multi-objective case. In: *International Workshop on Optimization and Learning: Challenges and Applications*. Alicante, pp.36-37.

Other Presentations in Conferences

1. Gimbutas, A. (2014). One-step worst-case optimal bivariate algorithm for bi-objective optimization. *Data Analysis Methods for Software Systems*. Druskininkai.

2. Gimbutas, A. (2015). Daugiadimensinis globalios optimizacijos algoritmas naudojantis adaptyviąją Lipšico konstantą. In: *Computer Days*. Panevėžys.

3. Gimbutas, A. (2015). DAKIS - algoritmų įvertinimo ir palyginimo įrankis. In: *Operacijų tyrimas ir taikymai*. Panevėžys.

4. Gimbutas, A. (2015). Multicriteria Lipschitz Optimization Algorithm Using Local Lipschitz Constant Estimate. *Data Analysis Methods for Software Systems*. Druskininkai, p. 20.

5. Gimbutas, A. (2016). Daugiakriterė globali optimizacija naudojant adaptyvią Lipšico konstantą. In: *Operacijų tyrimas ir taikymai*. Kaunas.

6. Gimbutas, A. and Žilinskas (2016). Remarks on a Multi-Criteria Simplicial Optimization with an Estimate of Lipschitz constant. *Data Analysis Methods for Software Systems*. Druskininkai, p. 22.

7. Gimbutas, A. (2017). Daugiaobjektinis Lipšico simpleksinis optimizavimas su Lipšico konstantos įvertinimu. In: *Computer Days*. Kaunas, p. 25.

Appendix A

Automation of Experiment Execution and Result Aggregation

A utility tool was designed and implemented to automate the execution of experiments and aggregation of the numerical results. This tool was extensively used during preparation of this work. The tool was really very helpful and saved a lot of time.

The main use case of this tool is:

1. A user implements global/multi-objective optimization algorithm and stores its source code in a version control system's repository, which is reachable over the internet.
2. The user goes to the GUI of the tool (which is a website) and creates an experiment by specifying a) a link to the source code repository, b) parameter sets with which the algorithm has to be executed and c) how results have to be aggregated.
3. The user starts the execution of the experiment and waits till it is completed. A user can reduce the waiting time by increasing a number of parallelly running threads of the experiment.
4. The user sees already aggregated results of the experiment on the experiment's web page. In addition, the user can see the results of each task in a separate web page.

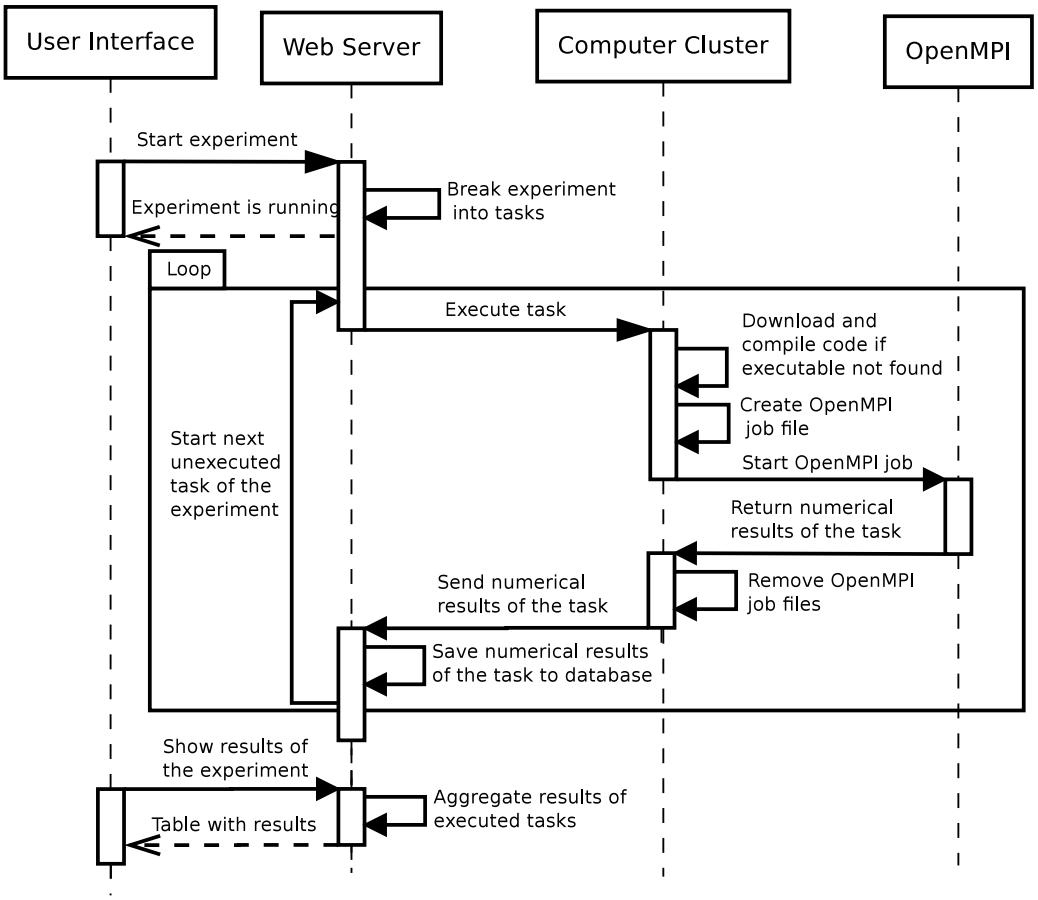


Figure A.1: Interaction diagram showing steps done by the tool during experiment's execution

In order to use the tool, the user has to create a profile in the website of the tool and specify a way to connect (via Secure Shell) to his profile of a computer cluster.

The consecutive steps of the experiment's execution are shown in Figure A.1. After the user starts the experiment, individual tasks are formulated, which have to be executed in order to obtain full results of the experiment. The task consists of the experiment's ID, link to source code repository and parameters, which have to be passed for the executable of the source code. Each task is sent to the computer cluster and executed using Open MPI interface. After execution, the results of the task are sent back to the web server and saved in the database. The user can see the aggregated results of all the tasks of the experiment by refreshing the web page of the experiment.

Appendix B

Libre Source Code

The source code of the proposed algorithm Libre is provided under Affero GPL v3 (or greater) licence is provided below and can be accessed online https://github.com/librealgorithm/libre.

Libre.h

```
#ifndef LIBRE_H
#define LIBRE_H
#include <math.h>
#include "utils.h"
#include <iostream>
#include <stdio.h>
#include <fstream>
#include <sstream>
using namespace std;

class Libre {
    Libre(const Libre& other) {};
    Libre& operator=(const Libre& other) {};
public:
    Libre(int max_calls=15000, double alpha=0.4) {
        _iteration = 0;
        Simplex::glob_L = numeric_limits<double>::max();    // Reset glob_L value
        Simplex::alpha = alpha;    // Reset glob_L value
        _max_calls = max_calls;
    };
    vector<Simplex*> _partition;
    Function* _func;
    int _iteration;
    string _status;
    int _max_calls;
    vector<Simplex*> partition_unit_cube_into_simplices_combinatoricly(int n) {
        // Partitions n-unit-cube into simplices using combinatoric vertex triangulation algorithm
        vector<Simplex*> partition;
        int number_of_simpleces = 1;
        for (int i = 1; i <= n; i++) {
            number_of_simpleces *= i;
        };
        int teta[n];
        for (int i=0; i < n; i++){
            teta[i] = i;
        };
        do {
```

```
        int triangle[n+1][n];
        for (int k = 0; k < n; k++) {
            triangle[0][k] = 0;
        };
        for (int vertex=0; vertex < n; vertex++) {
            for (int j = 0; j < n + 1; j++) {
                triangle[vertex + 1][j] = triangle[vertex][j];
            };
            triangle[vertex + 1][teta[vertex]] = 1;
        }
        Simplex* simpl = new Simplex();
        for (int i=0; i < n + 1; i++){
            Point* tmp_point = new Point(triangle[i], n);
            Point* point = _func->get(tmp_point);
            if (tmp_point != point) {
                delete tmp_point;
            };
            simpl->add_vertex(point);
        };
        simpl->init_parameters(_func);
        partition.push_back(simpl);
    } while (next_permutation(teta, teta+n));
    return partition;
};

vector<Simplex*> convex_hull(vector<Simplex*> simplices) {
    int m = simplices.size() - 1;
    if (m <= 1) { return simplices; };
    int START = 0;
    int v = START;
    int w = m;
    bool flag = false;
    bool leftturn = false;
    int a, b, c;
    double det_val;
    while ((nextv(v, m) != START) or (flag == false)) {
        if (nextv(v, m) == w) {
            flag = true;
        }
        a = v;
        b = nextv(v, m);
        c = nextv(nextv(v, m), m);
        double* matrix[3];
        double line1[3] = {simplices[a]->_diameter, simplices[a]->_min_lb, 1.};
        double line2[3] = {simplices[b]->_diameter, simplices[b]->_min_lb, 1.};
        double line3[3] = {simplices[c]->_diameter, simplices[c]->_min_lb, 1.};
        matrix[0] = line1;
        matrix[1] = line2;
        matrix[2] = line3;
        det_val = Determinant(matrix, 3);
        if (det_val >= 0){
            leftturn = 1;
        } else {
            leftturn = 0;
        };
        if (leftturn) {
            v = nextv(v, m);
        } else {
            simplices.erase(simplices.begin() + nextv(v, m));
            m -= 1;
            w -= 1;
            v = predv(v, m);
        };
    };
    return simplices;
};
int nextv(int v, int m) {
    if (v == m) {
        return 0;
    };
    return v + 1;
};
```

```
int predv(int v, int m) {
    if (v == 0) {
        return m;
    };
    return v - 1;
};
vector<Simplex*> select_simplices_to_divide() {
    vector<Simplex*> selected_simplices;
    // Sort simplices by their diameter
    vector<Simplex*> sorted_partition = _partition;
    sort(sorted_partition.begin(), sorted_partition.end(), Simplex::ascending_diameter);
    double f_min = _func->_f_min;
    // Find simplices with best lb_min values and unique diameters
    Simplex* min_lb_min_simplex = sorted_partition[0];  // Initial value
    vector<double> diameters;
    vector<Simplex*> best_for_size;
    bool unique_diameter;
    bool found_with_same_size;
    for (int i=0; i < sorted_partition.size(); i++) {
        if (sorted_partition[i]->_min_lb < min_lb_min_simplex->_min_lb) {
            min_lb_min_simplex = sorted_partition[i];
        };
        // Saves unique diameters
        unique_diameter = true;
        for (int j=0; j < diameters.size(); j++) {
            if (diameters[j] == sorted_partition[i]->_diameter) {
                unique_diameter = false; break;
            };
        };
        if (unique_diameter) {
            diameters.push_back(sorted_partition[i]->_diameter);
        };
        // If this simplex is better then previous with same size swap them
        found_with_same_size = false;
        for (int j=0; j < best_for_size.size(); j++) {
            if (best_for_size[j]->_diameter == sorted_partition[i]->_diameter){
                found_with_same_size = true;
                if (best_for_size[j]->_min_lb > sorted_partition[i]->_min_lb) {
                    best_for_size.erase(best_for_size.begin()+j);
                    best_for_size.push_back(sorted_partition[i]);
                };
            };
        };
        if (!found_with_same_size) {
            best_for_size.push_back(sorted_partition[i]);
        };
    };
    // Find strict pareto optimal solutions using convex-hull strategy
    vector<Simplex*> selected;
    if (min_lb_min_simplex == best_for_size[best_for_size.size()-1]) {
        selected.push_back(min_lb_min_simplex);
    } else {
        if ((best_for_size.size() > 2) && (min_lb_min_simplex != best_for_size[best_for_size.size()-1])) {
            vector<Simplex*> simplices_below_line;
            double a1 = min_lb_min_simplex->_diameter;  // Should be like this based on Direct Matlab implementation
            double b1 = min_lb_min_simplex->_min_lb;
            double a2 = best_for_size[best_for_size.size()-1]->_diameter;
            double b2 = best_for_size[best_for_size.size()-1]->_min_lb;
            double slope = (b2 - b1)/(a2 - a1);
            double bias = b1 - slope * a1;
            for (int i=0; i < best_for_size.size(); i++) {
                if (best_for_size[i]->_diameter >= a1) {  // Dont take into consideration smallel diameter simplices
                    if (best_for_size[i]->_min_lb < slope*best_for_size[i]->_diameter + bias +1e-12) {
                        simplices_below_line.push_back(best_for_size[i]);
                    };
                };
            };
            selected = convex_hull(simplices_below_line);
        } else {
            selected = best_for_size;
        };
    };
```

```
    for (int i=0; i < selected.size(); i++) {
        selected[i]->_should_be_divided = true;
    };
    // Remove simplices which were not selected and should not be divided
    selected.erase(remove_if(selected.begin(), selected.end(), Simplex::wont_be_divided), selected.end());
    // Select all simplices which have best min_lb for its size
    for (int i=0; i < sorted_partition.size(); i++) {
        for (int j=0; j < selected.size(); j++) {
            if ((sorted_partition[i]->_diameter == selected[j]->_diameter) &&
                (sorted_partition[i]->_min_lb == selected[j]->_min_lb)) {
                selected_simplices.push_back(sorted_partition[i]);
            };
        };
    };
    return selected_simplices;
};
vector<Simplex*> divide_simplex(Simplex* simplex) {
    vector<Simplex*> divided_simplices;
    // Find longest edge middle point
    int n = _func->_D;
    double c[n];
    for (int i=0; i < n; i++) {
        c[i] = (simplex->_le_v1->_X[i] + simplex->_le_v2->_X[i]) / 2.;
    };
    Point* tmp_point = new Point(c, n);
    Point* middle_point = _func->get(tmp_point);
    if (tmp_point != middle_point) {
        delete tmp_point;
    };
    // Construct two new simplices using this middle point.
    Simplex* left_simplex = new Simplex();
    Simplex* right_simplex = new Simplex();
    for (int i=0; i < simplex->size(); i++){
        if (simplex->_verts[i] != simplex->_le_v1){
            right_simplex->add_vertex(simplex->_verts[i]);
        } else {
            right_simplex->add_vertex(middle_point);
        };
        if (simplex->_verts[i] != simplex->_le_v2) {
            left_simplex->add_vertex(simplex->_verts[i]);
        } else {
            left_simplex->add_vertex(middle_point);
        };
    };
    left_simplex->init_parameters(_func);
    right_simplex->init_parameters(_func);
    simplex->_is_in_partition = false;
    divided_simplices.push_back(left_simplex);
    divided_simplices.push_back(right_simplex);
    return divided_simplices;
};
vector<Simplex*> divide_simplices(vector<Simplex*> simplices) {
    vector<Simplex*> new_simplices;
    for (int i=0; i < simplices.size(); i++) {
        vector<Simplex*> divided_simplices = divide_simplex(simplices[i]);
        for (int j=0; j < divided_simplices.size(); j++) {
            new_simplices.push_back(divided_simplices[j]);
        };
    };
    return new_simplices;
};
void minimize(Function* func){
    _func = func;
    _partition = partition_unit_cube_into_simplices_combinatoricly(_func->_D);
    sort(_partition.begin(), _partition.end(), Simplex::ascending_diameter);
    Simplex::update_min_lb_values(_partition, _func);
    while (!_func->is_accurate_enough()) {
        // Select simplices to divide
        vector<Simplex*> simplices_to_divide;
        if (_iteration == 0) {
            simplices_to_divide = _partition;
        } else {
```

```
            simplices_to_divide = select_simplices_to_divide();
        };
        // Divide seletected simplices
        vector<Simplex*> new_simplices = divide_simplices(simplices_to_divide);
        // Remove partitioned simplices
        _partition.erase(remove_if(_partition.begin(), _partition.end(), Simplex::not_in_partition), _partition.end());
        for (int i=0; i < simplices_to_divide.size(); i++) {
            delete simplices_to_divide[i];
        };
        simplices_to_divide.clear();
        // Add new simplices to _partition
        for (int i=0; i < new_simplices.size(); i++) {
            _partition.push_back(new_simplices[i]);
        };
        sort(_partition.begin(), _partition.end(), Simplex::ascending_diameter);
        Simplex::update_min_lb_values(_partition, _func);
        _iteration += 1;
    };
    if (_func->_evaluations <= _max_calls) {
        _status = "D";
    } else {
        _status = "S";
    };
};
virtual ~Libre(){
    for (int i=0; i < _partition.size(); i++) {
        delete _partition[i];
    };
    _partition.clear();
};
};
#endif
```

FuncUC.h

```
#ifndef FUNCTIONS_H
#define FUNCTIONS_H
#include <vector>
#include <iostream>
#include <algorithm>
#include <limits>
#include <cassert>
using namespace std;

class Simplex;
class Point {
    Point(const Point& other){}
    Point& operator=(const Point& other){};
public:
    Point(int D){
        _D = D;
        _X = (double*) malloc((D)*sizeof(double));
    };
    Point(int *c, int D){
        _D = D;
        _X = (double*) malloc((D)*sizeof(double));
        for (int i=0; i<D; i++){
            _X[i] = double(c[i]);
        };
    };
    Point(double *c, int D){
        _D = D;
        _X = (double*) malloc((D)*sizeof(double));
        for (int i=0; i<D; i++){
            _X[i] = c[i];
        };
    };
    int _D;          // Dimension of variable space
    double* _X;      // Coordinates in normalised [0,1]^n space
    double _value;   // Objective function value
    vector<Simplex*> _simplices;  // Simplices, which have this point as vertex
    void add_value(double value) {
```

```
        _value = value;
    };
    int size(){
        return _D;
    };
    static bool ascending_value(Point* p1, Point* p2) {
        return p1->_value < p2->_value;
    };
    virtual ~Point(){
        free(_X);
    };
};
// Binary balancing tree (or simply linked list) for storing points
// It returns cached point if a point with the same coordinates is added
class PointTree;
class PointTreeNode {
    PointTreeNode(const PointTreeNode& other){}
    PointTreeNode& operator=(const PointTreeNode& other){}
public:
    PointTreeNode(double value=numeric_limits<double>::max()){
        _height = 1;
        _value = value;
        _parent = 0;
        _left = 0;
        _right = 0;
        _subtree = 0;
        _point = 0;
    };
    int _height;
    double _value;
    PointTreeNode* _parent;
    PointTreeNode* _left;
    PointTreeNode* _right;
    PointTree* _subtree;     // Next dimension head
    Point* _point;           // Only last dimension node will have _point != 0;
    virtual ~PointTreeNode();
};
class PointTree { // Head of the tree
    PointTree(const PointTree& other){}
    PointTree& operator=(const PointTree& other){}
public:
    PointTree(){
         _tree_root = 0;
         _dim = 1;
    };
    PointTree(int dim){
         _tree_root = 0;
         _dim = dim;
    };
    PointTreeNode* _tree_root;
    int _dim;

    void update_height(PointTreeNode* node) {
        int lh = 0;
        int rh = 0;
        if (node->_left != 0) { lh = node->_left->_height; };
        if (node->_right != 0) { rh = node->_right->_height; };
        if (lh > rh) {
            node->_height = lh + 1;
        } else {
            node->_height = rh + 1;
        };
        // Also update all ancestors heights
        if (node->_parent != 0) {
            update_height(node->_parent);
        };
    };

    void left_right_rebalance(PointTreeNode* node) {
        PointTreeNode* diatteched_node;
        // node left right  <-  node left right left
        diatteched_node = node->_left->_right;
```

```
    node->_left->_right = node->_left->_right->_left;
    if (node->_left->_right != 0) { node->_left->_right->_parent = node->_left; };
    // Diatteched left = node->_left
    diatteched_node->_left = node->_left;
    node->_left->_parent = diatteched_node;
    // node left  <-  node left right
    node->_left = diatteched_node;
    diatteched_node->_parent = node;
    // Update heights
    update_height(node);
    update_height(diatteched_node);
    update_height(diatteched_node->_left);
};
void left_left_rebalance(PointTreeNode* node) {
    PointTreeNode* diatteched;
    diatteched = node->_left;
    node->_left = node->_left->_right;
    if (node->_left != 0) { node->_left->_parent = node; };
    diatteched->_parent = node->_parent;
    if (node->_parent != 0) {
        if (node->_parent->_left == node) {
            node->_parent->_left = diatteched;
        } else {
            node->_parent->_right = diatteched;
        };
    } else {
        _tree_root = diatteched;
    };
    diatteched->_right = node;
    node->_parent = diatteched;
    // Update heights
    update_height(node);
    update_height(diatteched);
};
void right_left_rebalance(PointTreeNode* node) {
    PointTreeNode* diatteched_node;
    // node left right  <-  node left right left
    diatteched_node = node->_right->_left;
    node->_right->_left = node->_right->_left->_right;
    if (node->_right->_left != 0) { node->_right->_left->_parent = node->_right; };
    // Diatteched left = node->_left
    diatteched_node->_right = node->_right;
    node->_right->_parent = diatteched_node;
    // node left  <-  node left right
    node->_right = diatteched_node;
    diatteched_node->_parent = node;
    // Update heights
    update_height(node);
    update_height(diatteched_node);
    update_height(diatteched_node->_right);
};
void right_right_rebalance(PointTreeNode* node) {
    PointTreeNode* diatteched;
    diatteched = node->_right;
    node->_right = node->_right->_left;
    if (node->_right != 0) { node->_right->_parent = node; };
    diatteched->_parent = node->_parent;
    if (node->_parent != 0) {
        if (node->_parent->_left == node) {
            node->_parent->_left = diatteched;
        } else {
            node->_parent->_right = diatteched;
        };
    } else {
        _tree_root = diatteched;
    };
    diatteched->_left = node;
    node->_parent = diatteched;
    // Update heights
    update_height(node);
    update_height(diatteched);
};
```

```
void check_if_balanced(PointTreeNode* node) {
    int lh = 0;
    int rh = 0;
    int llh = 0;
    int lrh = 0;
    int rlh = 0;
    int rrh = 0;
    if (node->_left != 0) {
        lh = node->_left->_height;
        if (node->_left->_left != 0) { llh = node->_left->_left->_height; };
        if (node->_left->_right != 0) { lrh = node->_left->_right->_height; };
    };
    if (node->_right != 0) {
        rh = node->_right->_height;
        if (node->_right->_left != 0) { rlh = node->_right->_left->_height; };
        if (node->_right->_right != 0) { rrh = node->_right->_right->_height; };
    };
    if (abs(rh - lh) > 1) {
        // Not balanced, so rebalance
        if (rh > lh) {
            if (rrh > rlh) {
                right_right_rebalance(node);
            } else {
                right_left_rebalance(node);
                right_right_rebalance(node);
            };
        };
        if (rh < lh) {
            if (llh > lrh) {
                left_left_rebalance(node);
            } else {
                left_right_rebalance(node);
                left_left_rebalance(node);
            };
        };
    };
    if (node->_parent != 0) {
        check_if_balanced(node->_parent);
    };
};
Point* process_next_dimension(PointTreeNode* node, Point* point) {
    // Creates next dimension tree if needed and adds point to it its last dimension
    if (point->_D == _dim) {        // Don't need next dimension
        if (node->_point == 0) {    // Save or return the point
            node->_point = point;
            return 0;
        } else {
            return node->_point;
        };
    };
    // Its not last dimension
    if (node->_subtree == 0) {  // Create subtree if it doesn't already exist
        node->_subtree = new PointTree(_dim + 1);
    };
    Point* found_point = node->_subtree->add(point);  // Get or insert point to the subtree
    if (found_point != 0) {  // We got point so return it
        return found_point;
    } else {
        return 0;  // We inserted point
    };
};
Point* add(Point* point){
    // Get same point or insert given (if inserted returns 0)
    PointTreeNode* node = _tree_root;
    double value = point->_X[_dim -1];
    if (_tree_root == 0) {  // Create first tree node
        _tree_root = new PointTreeNode(value);
        node = _tree_root;
        process_next_dimension(node, point);
    } else {
        while (true) {  // Walk through tree
            if (value > node->_value) {
```

```
                    if (node->_right == 0) {
                        node->_right = new PointTreeNode(value);
                        node->_right->_parent = node;
                        update_height(node->_right);
                        process_next_dimension(node->_right, point);
                        check_if_balanced(node->_right);
                        return 0;
                    };
                    node = node->_right;
                } else if (value < node->_value) {
                    if (node->_left == 0) {
                        node->_left = new PointTreeNode(value);
                        node->_left->_parent = node;
                        update_height(node->_left);
                        process_next_dimension(node->_left, point);
                        check_if_balanced(node->_left);
                        return 0;
                    };
                    node = node->_left;
                } else {
                    // Node value matches given point value, move to next dimension.
                    return process_next_dimension(node, point);
                };
            };
        };
    };
    virtual ~PointTree(){
        delete _tree_root;
    };
};
PointTreeNode::~PointTreeNode() {
    if (_left != 0) { delete _left; };
    if (_right != 0) { delete _right; };
    if (_point != 0) { delete _point; };
    if (_subtree != 0) { delete _subtree; };
};
class Function {        // Abstract function class
    Function(const Function& other){};
    Function& operator=(const Function& other){};
public:
    Function(){
        _f_min = numeric_limits<int>::max();
        _points = new PointTree();
        _evaluations = 0;
    };
    int _evaluations;
    int _D;                     // Dimension
    double _f_min;              // Best known function value
    PointTree* _points;         // Binary balancing tree to store points where objective function was evaluated
    vector<Point*> _new_points; // Points for which stopping condition was not checked yet
    void add_value(Point* p) {
        double val = value(p);
        _evaluations += 1;
        p->add_value(val);
        if (_f_min > val) {
            _f_min = val;
        };
    };
    Point* get(double *c, int D){
        // Returns a point with objective function value (the point may be from cache)
        Point* p = new Point(c, D);
        Point* cached_point = _points->add(p);
        if (cached_point) {   // Value at this point is already known
            delete p;
            return cached_point;
        } else {              // Value at this point is unknown, evaluate it
            add_value(p);
            _new_points.push_back(p);
            return p;
        };
    };
    Point* get(Point* p) {
```

```
        // Returns a point with objective function value (the point may be from cache)
        Point* cached_point = _points->add(p);
        if (cached_point) {   // Value at this point is already known
            return cached_point;
        } else {              // Value at this point is unknown, evaluate it
            add_value(p);
            _new_points.push_back(p);
            return p;
        };
    };
    virtual bool is_accurate_enough() = 0;
    virtual double value(Point* point) = 0;
    virtual ~Function(){
        delete _points;
    };
};

class FuncUC: public Function {      // Function which is define over a unit-cube
    FuncUC(const FuncUC& other){};
    FuncUC& operator=(const FuncUC& other){};
public:
    FuncUC(int D, double (*get_value_uc)(vector<double>), bool (*should_stop_uc)(vector<double>)){
        _D = D;
        get_value = get_value_uc;
        should_stop = should_stop_uc;
    };
    double (*get_value) (vector<double>);   // Objective function evaluation method provided as an argument
    bool (*should_stop) (vector<double>);   // Stopping criterion method provided as an argument
    double value(Point* point) {
        // Converts a point object to a vector and evaluates objective value at it
        vector<double> point_vector;
        for (int i=0; i < point->_D; i++) {
            point_vector.push_back(point->_X[i]);
        };
        return get_value(point_vector);
    };
    bool is_accurate_enough() {
        // Checks stopping criterion for all new points
        int nr_of_new_points = _new_points.size();
        for (int j=0; j < nr_of_new_points; j++) {
            // Pop one of the new points
            Point* p = _new_points.back();
            _new_points.pop_back();
            // Convert that point object to a vector
            vector<double> point_vector;
            for (int i=0; i < p->_D; i++) {
                point_vector.push_back(p->_X[i]);
            };
            // Check stopping condition
            if (should_stop(point_vector) == true) {
                return true;
            };
        };
        return false;
    };
};
#endif
```

utils.h

```
#ifndef UTILS_H
#define UTILS_H
#include <fstream>
#include <sstream>
using namespace std;

double l2norm(Point* p1, Point* p2) {
    // Finds Euclidean distance between two points
    double squared_sum = 0;
    for (int i=0; i < p1->size(); i++){
        squared_sum += pow(p1->_X[i] - p2->_X[i], 2);
    };
```

```
    return sqrt(squared_sum);
};
class Simplex {
    Simplex(const Simplex& other){}
    Simplex& operator=(const Simplex& other){}
public:
    Simplex() {
        _D = 0;
        _le_v1 = 0;
        _le_v2 = 0;
        _diameter = 0;
        _min_lb = numeric_limits<double>::max();
        _is_in_partition = true;
        _should_be_divided = false;
        _should_lb_mins_be_updated = true;
    };
    int _D;             // Dimension of variable space
    Point* _le_v1;      // First vertex of the longest edge
    Point* _le_v2;      // Second vertex of the longest edge
    double _diameter;   // Longest edge length
    double _min_L;      // Minimum L for this simplex
    double _min_lb;     // Minimum of the lower bound over simplex found using vertex with the lowest value
    static double alpha;              // Coeficient of search globality
    static double glob_L;             // Globally known biggest min L
    static bool glob_L_was_updated;
    vector<Point*> _verts;    // Vertices of this simplex (points with coordinates and values)
    bool _is_in_partition;    // Is this simplex in the current partition
    bool _should_be_divided;  // Should this simplex be divided in next iteration
    bool _should_lb_mins_be_updated;   // Should the minimums of Lipschitz lower bound be updated
    Point* _min_vert;   // Vertex with lowest function value
    void init_parameters(Function* func) {   // Called when all verts have been added
        _D = _verts.size() - 1;
        // Sorts vertexes ascending by their function value
        sort(_verts.begin(), _verts.end(), Point::ascending_value);
        // Find longest edge length (simplex diameter) and its vertices
        double edge_length;  // Temporary variable
        for (int a=0; a < _verts.size(); a++) {
            for (int b=0; b < _verts.size(); b++){
                if (b > a) {
                    edge_length = l2norm(_verts[a], _verts[b]);
                    if (edge_length > _diameter) {
                        _diameter = edge_length;
                        _le_v1 = _verts[a];
                        _le_v2 = _verts[b];
                    };
                };
            };
        };
        // Find minimum L for ths simplex
        _min_L = find_simplex_min_L();
        // Initialize or update global L if needed
        if (Simplex::glob_L == numeric_limits<double>::max()) {
            Simplex::glob_L = Simplex::alpha * _min_L;
        } else {
            if (Simplex::glob_L < Simplex::alpha * _min_L) {
                Simplex::glob_L = Simplex::alpha * _min_L;
                Simplex::glob_L_was_updated = true;
            };
        };
        // Find vertex with minimum function value
        _min_vert = _verts[0];
        for (int i=0; i < _verts.size(); i++) {
            if (_verts[i]->_value < _min_vert->_value) {
                _min_vert = _verts[i];
            };
        };
    };
    double find_min_vert_lb_min(Simplex* simpl, double L) {
        // Finds minimum of lower bound, which is constructed from the vertex with lowest function value
        return _min_vert->_value - L * simpl->_diameter;
    };
    double find_simplex_min_L() {
```

```
        // Finds minimum L for this simplex by finding min L for each simplex edge
        double dist;
        double f_diff;
        double edge_L;
        double max_edge_L = -numeric_limits<double>::max();
        for (int i=0; i < _verts.size(); i++) {
            for (int j=i+1; j < _verts.size(); j++) {
                f_diff = fabs(_verts[i]->_value - _verts[j]->_value);
                dist = l2norm(_verts[i], _verts[j]);
                edge_L = f_diff / dist;     // Note: maybe dist (division by zero) protection is needed?
                if (edge_L > max_edge_L) {  //       Practically this case does not occur.
                    max_edge_L = edge_L;
                };
            };
        };
        return max_edge_L;
    };
    void add_vertex(Point* vertex){
        _verts.push_back(vertex);
        vertex->_simplices.push_back(this);
    };
    int size() {
        return _verts.size();
    };
    static void update_min_lb_values(vector<Simplex*> simpls, Function* func);
    static bool wont_be_divided(Simplex* s) {
        return !s->_should_be_divided;
    };
    static bool not_in_partition(Simplex* s) {
        return !s->_is_in_partition;
    };
    static double ascending_diameter(Simplex* s1, Simplex* s2) {
        return s1->_diameter < s2->_diameter;
    };
    virtual ~Simplex(){
        for (int i=0; i < _verts.size(); i++) {
            _verts[i]->_simplices.erase(remove(_verts[i]->_simplices.begin(), _verts[i]->_simplices.end(), this),
                                        _verts[i]->_simplices.end());
        };
        _verts.clear();
    };
};
bool Simplex::glob_L_was_updated = false;
double Simplex::glob_L = numeric_limits<double>::max();
double Simplex::alpha = numeric_limits<double>::max();
void Simplex::update_min_lb_values(vector<Simplex*> simpls, Function* func) {
    for (int sid=0; sid < simpls.size(); sid++) {
        if (simpls[sid]->_should_lb_mins_be_updated or Simplex::glob_L_was_updated) {
            simpls[sid]->_min_lb = simpls[sid]->find_min_vert_lb_min(simpls[sid], Simplex::glob_L);
            simpls[sid]->_should_lb_mins_be_updated = false;
        };
    };
    Simplex::glob_L_was_updated = false;
};
double Determinant(double **a, int n) { // Based on http://paulbourke.net/miscellaneous/determinant/
    int i, j, j1, j2;
    double det = 0;
    double **m = NULL;

    if (n < 1) { /* Error */ cout << "Determinant␣cannot␣be␣calculated␣for␣empty␣matrix" << endl;
    } else if (n == 1) { /* Shouldn't get used */
        det = a[0][0];
    } else if (n == 2) {
        det = a[0][0] * a[1][1] - a[1][0] * a[0][1];
    } else {
        det = 0;
        for (j1=0;j1<n;j1++) {
            m = (double**) malloc((n-1)*sizeof(double *));
            for (i=0;i<n-1;i++)
                m[i] = (double*) malloc((n-1)*sizeof(double));
            for (i=1; i<n; i++) {
                j2 = 0;
```

```
                for (j=0; j<n; j++) {
                    if (j == j1) continue;
                    m[i-1][j2] = a[i][j];
                    j2++;
                }
            }
            det += pow(-1.0,1.0+j1+1.0) * a[0][j1] * Determinant(m,n-1);
            for (i=0;i<n-1;i++) free(m[i]);
            free(m);
        }
    }
    return(det);
};
#endif
```

main.cpp

```
#include <iostream>
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include "FuncUC.h"
#include "Libre.h"
using namespace std;

int d;                              // Dimension of the optimization problem
vector<double> lb;                  // Lower bound of feasible region
vector<double> ub;                  // Upper bound of feasible region
vector< vector<double> > points;    // Points where objective value was evaluated
double get_value(vector<double> point) {  // Evaluates objective function at given point in [lb, ub]^d feasible region
    points.push_back(point);              // And memorizes the point where the evaluation was made
    return func(&point[0]);  // Make funtion evaluation
};
bool should_stop(vector<double> point) {  // Checks if stopping condition is satisfied by a given point in [lb, ub]^d
    for (int i=0; i < d; i++) {
        if (true) { // Check stopping condition
            return false;
        };
    };
    return true;
};
// Transforms point's coordinates from [0, 1]^d unit-cube to [lb, ub]^d feasible region
vector<double> transform_from_uc(vector<double> point_uc, vector<double> _lb, vector<double> _ub) {
    vector<double> point;
    for (int i=0; i < _lb.size(); i++){
        point.push_back(point_uc[i] * (_ub[i] - _lb[i]) + _lb[i]);
    };
    return point;
};
double get_value_uc(vector<double> point_uc) {  // Evaluates objective function at a given point in [0, 1]^d feasible region
    return get_value(transform_from_uc(point_uc, lb, ub));
};
bool should_stop_uc(vector<double> point_uc) {  // Checks if stopping condition is satisfied by a given point in [0, 1]^d
    return should_stop(transform_from_uc(point_uc, lb, ub));
};
int main(int argc, char* argv[]) {  // Minimizes 100 functions from one GKLS class; prints intermediate and summarised results
    int max_calls = 1000000;
    double glob_L = numeric_limits<double>::max();
    double alpha = 0.4;
    Function* func_uc = new FuncUC(d, get_value_uc, should_stop_uc);
    // Minimize the function using Libre algorithm (it has alpha parameter set to 0.4)
    Asimpl* alg;
    alg = new Asimpl(max_calls, max_duration, alpha);
    alg->minimize(func_uc);
    cout << "calls=" << points.size() << "␣f_min=" << func_uc->_f_min << endl;
    delete func_uc;  // Clear allocated memory
    delete alg;
    points.clear();
    GKLS_free();
    return 0;
};
```

ALBERTAS GIMBUTAS

NONCONVEX OPTIMIZATION ALGORITHM WITH A NEW BI-CRITERIA SELECTION OF POTENTIAL SIMPLICES USING AN ESTIMATE OF LIPSCHITZ CONSTANT

Doctoral dissertation

Physical sciences (P000)

Informatics (09P)

Editor Nijolė Požėraitytė