UNIVERSITA' DEGLI STUDI DI PADOVA

Dipartimento di Matematica "Tullio Levi-Civita"
Laurea Magistrale in Matematica

# A Derivative-Free Approach for Large Scale Problems

*Laureando:*
Michele Roselli

*Relatore:*
Prof. Francesco Rinaldi

Anno Accademico 2017-2018

# Abstract

It is a common problem in optimization to deal with objective functions whose gradient cannot be calculated. This is the case of Black-Box functions, where the only available tool is a program (so called Black-Box) which outputs the value of the function for a given input, without telling any information about the function itself.

There are actually plenty of well performing algorithms which are made to solve this kind of problem, making use of clever tecniques to find a descent direction without using the gradient. However, most of them requires a number of function evaluations (and often a cpu time as well) which increases dramatically along with the dimension of the problem. One function evaluation is usually time consuming in real world problems and this is the reason why the performances of those algorithms are evaluated by taking into account the number of function evaluations needed to get a certain reduction in the objective function.

The purpose of this work is to provide practical results about the performance of a new optimization algorithm. This algorithm solves large scale problems with a small budget of function evaluations. In the first chapter we just report the state of the art regarding derivative-free optimization, in the second one we present and explain our algorithm and provide its performance and data profiles in the third. The conclusions are drawn in the last chapter.

# Contents

# Chapter 1

# Introduction

The purpose of this chapter is to explain the reasons why derivative free algorithms are widely used in practice. We will present the state of the art and the theoretical foundations of our work.

## 1.1 The Lack of the Gradient

Derivatives are the most fundamental and reliable elements when an optimization problem needs to be solved. Infact, nearly every mathematical characterization of a (generally local) minimum requires the first order derivatives to be zero. However, there is a variety of situations where those derivatives are not available or computationally hard to obtain and that is why efficient methods for derivative free optimization have always been needed. Optimization without derivatives is considered one of the most important, open, and challenging areas in computational science and engineering, and one with enormous practical potential. are some of the reasons why derivative-free optimization is currently an area of great demand.

The increasing complexity in mathematical modeling, higher sophistication of scientific computing, and an abundance of legacy codes will unavoidably lead to an increase of problem dimensions (i.e., the number of variables used to model our real application). With the growth and development of derivative-based nonlinear optimization methods it became evident that large-scale problems can be solved efficiently, but only if there is accurate derivative information at hand. Not surprisingly, as we already suggested, there are considerable disadvantages in not having derivative information, so one cannot expect the performance of derivative-free methods to be comparable with those of derivative-based methods. In particular, the scale of the problems that can currently be efficiently solved by derivative-free methods is still relatively small and does not exceed a few hundred variables even in easy cases. [1].

As will be explained in the following pages, the algorithm we developed is meant to raise this practical limit by an order of magnitude. Our intent is indeed to provide a tool which can significantly reduce objective functions with thousands of variables.

## 1.2 DFO Methods

Many techniques have been developed in derivative free optimization. In this section we will just present the most common methods currently applied in practice.

Before explaining of such techniques, we briefly recall that we are interested in solving (in practice, we are interested in finding a reasonable approximation of the solution of) the following:

$$\min f(x),$$

$$\text{subject to } x \in X,$$

where $f$ is a function whose derivatives are not available. The common assumption on the set of feasible points $X$ is that it is convex. However, since Section 1.3 we will make this assumption more strict. We will ask $X$ to be the convex combination of a finite number of vertices:

$$X = \text{conv}\{v_1, \ldots, v_n\},$$

that is,

$$x = \sum_{i=1}^{n} \lambda_i v_i,$$

$$\lambda_i \geq 0 \text{ and } \textstyle\sum_{i=1}^{n} \lambda_i = 1.$$

### 1.2.1 Directional Direct Search Methods

A simple idea to solve an optimization problem without using derivatives consist of sampling the function in the neighborhood of the current best guess and move to a point where the function decreases. This is the strategy followed by the Compass algorithm. This method makes use of the basis $D_{\oplus}$:

$$D_{\oplus} = [I - I] = [e_1 \ldots e_n - e_1 \ldots - e_n].$$

However, any positive basis (that is, a positive independent set which generate $\mathbb{R}^n$) can be used instead.

**Inizialization:** Choose $x_0$ and $\alpha_0 > 0$;
**for** $k = 0, 1, 2, \ldots$ **do**
1. **Poll Step:** Order the poll set $P_k = \{x_k + \alpha_k d : d \in D_\oplus\}$. Start evaluating $f$ at the poll points following the order determined. If a poll point $x_k + \alpha_k d_k$ is found such that $f(x_k + \alpha_k d_k) < f(x_k)$, then stop polling, set $x_{k+1} = x_k + \alpha_k d_k$, and declare the iteration and the poll step successful. Otherwise, declare the iteration (and the poll step) unsuccessful and set $x_{k+1} = x_k$.;
2. **Parameter Update:** If the iteration was successful, set $\alpha_{k+1} = \alpha_k$ (or $\alpha_{k+1} = 2\alpha_k$). Otherwise, set $\alpha_{k+1} = \alpha_k/2$.;
3. **Optimal Conditions:** If $\alpha_{k+1} < \alpha_{limit}$ the algorithm stops.

**end**

**Algorithm 1:** Compass Algorithm [1]

This algorithm is used for unconstrained optimization problems. However, if we start from a feasible point $x_0$ and perform a feasibility check every time we evaluate a poll point, we can also perform constrained optimization, as long as the feasible region is convex.

The poll set ordering here is extremely important, especially when the problem dimension is large. Indeed, start evaluating from the most promising directions could save a huge number of function evaluations.

### 1.2.2 Simplicial Direct Search Methods

This subsection will briefly report one of the most used DFO algorithm, which we also used as a comparison to evaluate the performance of our algorithm, the Nelder-Mead algorithm. This is a direct search method, in the sense that it evaluates the objective function at a finite number of points per iteration and decides which action to take next solely based on those function values and without any explicit or implicit derivative approximation. The substantial difference between the Nelder-Mead and the directional direct search methods, is that this time we keep in memory a simplex of $n+1$ points and replace the worst one at each iteration.

**Inizialization:** Choose an initial simplex of vertices $Y_0 = \{y_0^0, y_0^1, \ldots, y_0^n\}$. Evaluate $f$ at the points in $Y_0$. Choose constants: $0 < \gamma^s < 1$, $-1 < \delta^{ic} < 0 < \delta^{oc} < \delta^r < \delta^e$;

**for** $k = 0, 1, 2, \ldots$ **do**

**0.** Set $Y = Y_k$;

**1. Order:** Order the $n+1$ vertices of $Y = \{y^0, y^1, \ldots, y^n\}$ so that: $f^0 = f(y^0) \leq f^1 = f(y^1) \leq \cdots \leq f^n = f(y^n)$.

**2. Reflect:** Reflect the worst vertex $y^n$ over the centroid $y^c = \sum_{i=0}^{n-1} y^i/n$ of the remaining $n$ vertices: $y^r = y^c + \delta^r(y^c - y^n)$. Evaluate $f^r = f(y^r)$. If f $f^0 \leq f^r < f^{n-1}$, then replace $y^n$ by the reflected point $y^r$ and terminate the iteration: $Y_{k+1} = \{y^0, y^1, \ldots, y^{n-1}, y^r\}$.

**3. Expand:** If $f^r < f^0$, then calculate the expansion point $y^e = y^c + \delta^e(y^c - y^n)$ and evaluate $f^e = f(y^e)$. If f $f^e \leq f^r$, replace $y^n$ by the expansion point $y^e$ and terminate the iteration: $Y_{k+1} = y^0, y^1, \ldots, y^{n-1}, y^e$. Otherwise, replace $y^n$ by the reflected point $y^r$ and terminate the iteration: $Y_{k+1} = y^0, y^1, \ldots, y^{n-1}, y^r$.

**4. Contract:** Contract: If $f^r \geq f^{n-1}$, then a contraction is performed between the best of $y^r$ and $y^n$.

**4.1 Outside Contraction:** If $f^r < f^n$ then compute $y^{oc} = y^c + \delta^{oc}(y^c - y^n)$ and evaluate $f^{oc} = f(y^{oc})$. If $f^{oc} \leq f^r$, then replace $y^n$ by the outside contraction point $y_k^{oc}$ and terminate the iteration: $Y_{k+1} = \{y^0, y^1, \ldots, y^{n-1}, y^{oc}\}$. Otherwise, perform a shrink.

**4.2 Inside Contraction:** Else if $f^r \geq f^n$ then compute $y^{ic} = y^c + \delta^{ic}(y^c - y^n)$ and evaluate $f^{ic} = f(y^{ic})$. If $f^{ic} < f^n$, then replace $y^n$ by the inside contraction point $y_k^{ic}$ and terminate the iteration: $Y_{k+1} = \{y^0, y^1, \ldots, y^{n-1}, y^{ic}\}$. Otherwise, perform a shrink.

**5. Shrink:** Evaluate $f$ at the $n$ points $y^0 + \gamma^s(y^i - y^0), i = 1, \ldots, n$, and replace $y^1, \ldots, y^n$ by these points, terminating the iteration: $Y_{k+1} = \{y^0 + \gamma^s(y^i - y^0), i = 0, ..., n\}$.

**6. Optimal Conditions:** A common stopping criterion consists of terminating the run when the diameter of the simplex becomes smaller than a chosen tolerance $\Delta_{tol} > 0$.

**end**

**Algorithm 2:** Nelder-Mead Algorithm [1]

The strenght of the Nelder-Mead algorithm lies in the number of function evaluations per iteration which does not usually grow with $n$. In fact, the shrink step rarely occurs in practice and therefore the function evaluations are at most 2.

### 1.2.3 Line-Search Methods

The algorithm presented in the following pages is a modified version of the Implicit Filtering method by Kelley et al [2], taken from [1]. This algorithm is the first we present which makes an estimation of the gradient (the so called *simplex gradient*) to fix

a direction and carry out the function reduction using the line search.

We start giving a few definitions and the most common example of simplex gradient. We consider as before a sample set of $n+1$ points which are assumed to be the vertices of a simplex: $Y_k = \{y_k^0, y_k^1, \ldots, y_k^n\}$. This set is assumed to be poised, that is:

$$det \begin{bmatrix} 1 & y_{k,1}^0 & \ldots & y_{k,n}^0 \\ 1 & y_{k,1}^1 & \ldots & y_{k,n}^1 \\ \vdots & \vdots & \vdots & \vdots \\ 1 & y_{k,1}^n & \ldots & y_{k,n}^n \end{bmatrix} \neq 0. \tag{1.1}$$

The simplex gradient in $x_k = y_k^0$ is then given by:

$$\nabla_s f(x_k) = L_k^{-1} \delta f(Y_K),$$

where

$$L_k = \left[ y_k^1 - y_k^0 \ldots y_k^n - y_k^0 \right]^T$$

and

$$\delta f(Y_K) = \begin{bmatrix} f(y_k^1) - f(y_k^0) \\ \vdots \\ f(y_k^n) - f(y_k^0) \end{bmatrix}.$$

We also define:

$$\Delta_k = \max_{1 \leq i \leq n} \|y_k^i - y_k^0\|.$$

Every time line search is performed, we strenghten the strict decrease condition, asking a sufficient decrease instead:

$$f(x_k - \alpha \nabla_s f(x_k)) \leq -\eta \alpha \|\nabla_s f(x_k)\|^2. \tag{1.2}$$

Finally we are ready to present the algorithm:

**Inizialization:** Choose an initial point $x_0$ and an initial poised sample set $\{y_0^0(=x_0), y_0^1, \ldots, y_0^n\}$. Choose $\beta, \eta \in (0,1)$. Select $j_{max} \in \mathbb{N}$.

**for** $k = 0, 1, 2, \ldots$ **do**

**1. Simplex Gradient Calculation:** Compute a simplex gradient $\nabla_s f(x_k)$ such that $\Delta_k \leq \|\nabla_s f(x_k)\|$. Set $j_{current} = j_{max}$ and $\mu = 1$.

**2. Line Search:** For $j = 0, 1, 2, \ldots, j_{current}$

(a) Set $\alpha = \beta^j$. Evaluate $f$ at $x_k - \alpha \nabla_s f(x_k)$.

(b) If the sufficient decrease condition (1.1) is satisfied for $\alpha$, then stop this step with $\alpha_k = \alpha$ (and go to Step 4).

**3. Line Search Failure:** If the line search failed, then divide $\mu$ by two, recompute a simplex gradient $\nabla_s f(x_k)$ such that $\Delta_k \leq \|\nabla_s f(x_k)\|$, increase $j_{current}$ by one, and repeat the line search (go back to Step 2).

**4. New Point:** Set:

$$x_{k+1} = \underset{x \in \chi_k}{\arg\min} \{f(x_k - \alpha_k \nabla_s f(x_k)), f(x)\},$$

where $\chi_k$ is the set of points where $f$ has possibly been evaluated during the course of Steps 1 and 3. Set $y_{k+1}^0 = x_{k+1}$. Update $y_{k+1}^1, \ldots, y_{k+1}^n$ from $y_k^0, y_k^1, \ldots, y_k^n$ by dropping one of these points.

**5. Optimal Conditions:** A common stopping criterion consists of terminating the run when $\Delta_k$ becomes smaller than a chosen tollerance $\Delta_{tol} > 0$.

**end**

**Algorithm 3:** Modified Implicit Filtering [1]

The computation of a simplex gradient with the property $\Delta_k \leq \|\nabla_s f(x_k)\|$ can be easily be done by a greedy algorithm presented in [1].

### 1.2.4 Model Based Trust-Region Methods

This broad class of optimization method differs radically from the others seen before. This time we are not working directly on the objective function, but we use it to create a model which is easier to deal with. This model hopefully gives a good approximation of the objective function itself in a neighborhood (the trust region) of the current guess. Given the fact that these kind of algorithms requires a lot of theoretical explanation, we will just report as an example a heavily simplified pseudo code of a first order trust-region method.

**Inizialization:** Choose $x_0$ and $\Delta > 0$ (the trust region radius);
**for** $k = 0, 1, 2, \ldots$ **do**

**1. Model Computation:** Compute $m_k()$ (the model used to approximate $f$, usually quadratic);

**2. Step Computation:** Compute $s_k$ as:

$$s_k = \min_{\|s\|<\Delta} m_k(x_k + s)$$

(we are solving the local model subproblem);

**3. Descent Ratio Computation:**

$$\rho_k = \frac{f(x_k) - f(x_k + s)}{m_k(x_k) - m_k(x_k + s)}$$

($\rho_k$ indicates how well the reduction in the model transfers itself on the real problem);

**4. Trust-Region Radius Update:** the updates are usually made based on two fixed parameters $0 < \eta_0 < \eta_1$ in the following way:
(4.1) If $\rho_k > \eta_1$: $x_{k+1} \leftarrow x_k + s_k$ and increase $\Delta$.
(4.2) Elseif $\rho_k > \eta_0$ and $m_k()$ satisfy certain structural conditions: $x_{k+1} \leftarrow x_k + s_k$ and leave $\Delta$ unchanged.
(4.3) Else: $x_{k+1} \leftarrow x_k$ and decrease $\Delta$.

**5. Optimal Conditions:** The algorithm usually terminates whenever the trust-region radius $\Delta$ becomes too small.

**end**

**Algorithm 4:** Simple Trust Region [3]

The reason why model based approaches are not much considered in this thesis is because the number of points needed to decently fit a model typically raise as $n^2$. With the standard computational budget being in the order of 100 simplex gradients ($100(n+1)$) it is clear that approach large scale problems with model based methods is not possible.

## 1.3 Simplicial Decomposition

Simplicial decomposition is a common method used in derivative based optimization. The reason why we are introducing such a technique is simple: we took simplicial decomposition from that framework and made it the key element of our algorithm.

As we have seen in the previous section, operating on a simplex can be extremely useful. This time, instead of keeping a fixed number of vertices as a sample set, we dinamically increase the vertices and solve an optimization subproblem into the simplex at each iteration. The detailed presentation of the algorithm is left for the next chapter, while the end of the current one is meant to explain how the simplcial decomposition works.

Without further delay, let's introduce the elements of the aglorithm:

$$P = \{v_1, \ldots, v_n\},\ v_i \in \mathbb{R}^m$$

$$X = \text{conv}(v_1, \ldots, v_n)$$

$$\hat{I}, \text{ a subset of } \{1, \ldots, n\}.$$

The elements of $\hat{I}$ point to the elements of $P$ which make up the current simplex. For instance, if $\hat{I} = \{1, 4, 18\}$, then the simplex vertices are $v_1$, $v_4$ and $v_{18}$.

With that said, the simplicial decomposition interation is splitted in two fundamental phases:

**Simplex Optimization:**

Solve the master problem on the current simplex:

$$x_k^* = \arg\min f(x)$$

$$x = \sum_{i \in \hat{I}} \lambda_i v_i$$

$$\lambda_i \geq 0 \text{ and } \sum \lambda_i = 1.$$

**Simplex Update:**

Solve a linearized subproblem on the whole $X$:

$$y = \arg\min c^T x$$

$$x \in X$$

Because of the linearity of the objective function, $y = v_i, \exists i$. Then put $\hat{I} = \hat{I} \cup \{i\}$. If $i$ is already in $\hat{P}$, then $x_k^*$ is optimal and the algorithm terminates.

The linearization of the problem is usually created in a Frank-Wolfe fashion: $c = \nabla f(x_k^*)$. This, of course, cannot be done in our framework, where the gradient is not available.

The simplicial decomposition technique starts working well when $n$, the dimension of the problem (that is, the number of vertices of the feasible region $X$), becomes very large. That's because the real optimization is made on the simplex whose dimension grows slowly and most importantly with no dependence on $n$. This is the key point we will exploit to make our algorithm work in the context of large scale problems.

# Chapter 2

# A Simplicial Decomposition DFO Algorithm

In the following sections we will present and carefully detail our algorithm DF-SD, which perform simplicial decomposition over a convex combination of vertices without using derivatives.

## 2.1 Algorithm Overview

Firstly, we recall the problem we are insterested in:

$$\min f(x)$$

$$\text{subject to } x \in X = \text{conv}\{v_1, \ldots, v_n\},$$

$$v_i \in \mathbb{R}^n.$$

Before giving a brief pseudo code of the algorithm, we introduce the elements we will use. Let $\hat{I}$ be the subset of $\{1, \ldots, n\}$ containing the indices of the vertices $v_i$ which compose the current simplex $S$ and:

$$A = [v_{i_1}, \ldots, v_{i_l}],$$

the matrix whose columns are the vertices of $S$, that is, $\hat{I} = \{i_1, \ldots, i_l\}$. Let also:

$$V = [v_1, \ldots, v_n],$$

the matrix of all the vertices of $X$.

To be clear, we also recall that we consider 3 dimensions during the execution of the algorithm:

- $n$, the number of vertices of the feasible region $X$.
- $m$, the dimension of the space where $x$ and $v_1, \ldots, v_n$ live.
- $l$, the dimension of the current simplex $S$, that is, the number of its vertices. As already pointed out, $l$ will change at each iteration of the algorithm.

While the *dimension of the function evaluations* is $m$, because $f : \mathbb{R}^m \to \mathbb{R}$, the dimension of the problem is actually $n$. Infact the problem we are going to solve can be rewritten as the following:

$$\begin{aligned} &\min \hat{f}(y) = f(Vy) \\ &s.t. \;\; e^T y = 1, \\ &y^i \geq 0. \end{aligned} \tag{2.1}$$

We will now present the basic scheme of the algorithm, which will be detailed in the next sections.

**Inizialization:** Choose the starting $\hat{I}$ (and therefore $S$) and $y_0$, the convex combination of the vertices in $S$. Then $l = |\hat{I}|$ and $A$ is the $m$ by $l$ matrix whose columns are the vertices of $S$;
**for** $k = 0, 1, 2, \ldots$ **do**
1. **Simplex Optimization:** Approximately solve:

$$\begin{aligned} y_k^* &= \arg\min f(Ay) \\ s.t. \;\; & e^T y = 1, \\ & y^i \geq 0. \end{aligned}$$

(This is how the objective function is reduced);
2. **Simplex Update:** Find an approximation $\hat{g}$ of the gradient of $f$ in $x_k = Ay_k^*$ and solve:

$$v_{new} = \arg\min \hat{g}^T v.$$

Then update the simplex: $S \leftarrow S \cup \{v_{new}\}$ and so update $\hat{I}$. Add $v_{new}$ as a new column of $A$ and put $y_k^* \leftarrow [(y_k^*)^1 \ldots (y_k^*)^l \;\; 0]^T$;
3. **Optimal Conditions:** If all the possible vertices are in $A$ already, the algorithm stops.

**end**

**Algorithm 5:** DF-SD Algorithm

## 2.2 Simplex Optimization Step

This is the step of the algorithm where the objective function is reduced. Before explaining how this is done, we will point out that in the whole we are operating in the reduced space of the $y \in \mathbb{R}^l$. The function we are evaluating is $\hat{f}_A = f \circ A : \mathbb{R}^l \to \mathbb{R}$. Every point $y \in \mathbb{R}^l$ subject to the simplex constraints $e^T y = 1$, $y^i \geq 0$ links to a (not necessarily in an injective way) point $x \in X \subset \mathbb{R}^m$. Consequently, while moving in $\mathbb{R}^l$ reducing the function $\hat{f}_A$, we are also reducing $f$ in $\mathbb{R}^n$.

### 2.2.1 Direct Search

For the optimization in the reduced space, we used direct search. Given the fact that the affine dimension of the simplex is $l-1$, we actually need just $l$ direction to conduct optimization. However, so we decided to get a better set of directions. The simplex is contained in the hyperplane $e^T x = 1$ so we did not use the carthesian (or gradient) directions $e_1, \ldots, e_l$. Instead, we chose a set of direction contained in the hyperplane. Such directions are the canonic ones, in the sense that if we are currently in the point $y$, then:

$$D_c = \{e_1 - y, \ldots, e_l - y, -(e_1 - y), \ldots, -(e_l - y)\}. \tag{2.2}$$

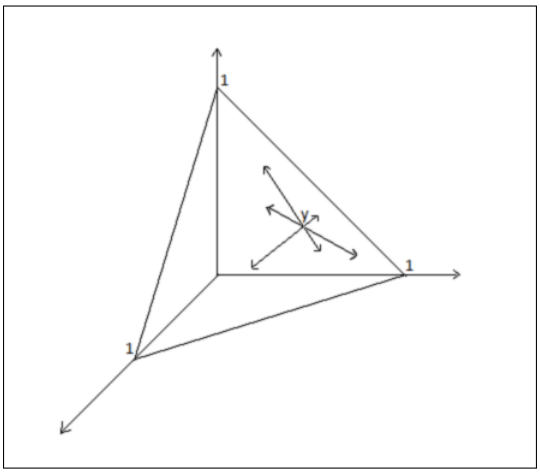


Figure 2.1: Canonic directions for l = 3.

The reason why we used both positive and negative directions even if the sole positive ones form a positive basis is that, heuristically, if direction $d$ is not a descent direction, there is a good probability that $-d$ will be one. We remind that this addition of directions does not represent a problem because $l$ is always a small number.

After setting the initial step of the search $\alpha_{start}$ and the initial point $y_0$ of the iteration, the algorithm evaluates the function in the point $y_j = y_0 + \alpha d_j$, $d_j \in D_c$. If $\hat{f}_A(y_j)$ satisfies the sufficient decrease condition and the point $y_j$ does not break the simplex boundaries, then an expansion step is performed. Otherwise, the opposite direction is tested, and subsequently all the directions in $D_c$ in the same way. If no direction can provide a feasible point with a sufficient decrease, the step $\alpha$ is reduced and the directions trial is repeated. When $\alpha$ falls below a certain threshold $\alpha_{min}$, the simplex optimization terminates.

### 2.2.2 Expansion Step

When the algorithm successfully find a new point $y_j = y_0 + \alpha d_j$, there is the possibility that a larger step (that is, $\alpha$) will grant a larger decrease. Consequently, chosen $\delta > 1$,

the algorithm checks if $y_j = y_0 + \alpha\delta d_j$ still satisfy sufficient decrease. If another reduction is attained, the procedure ($\alpha \leftarrow \alpha\delta$) continues until the sufficient decrease condition is violated.

When the expansion step ends, the direct search starts again from the new best guess. The step $\alpha$ for the direct search is restored as the last one before the expansion step. Therefore, after each cycle of simplicial decomposition the search step $\alpha$ either remains unchanged or decreases. As we just pointed out a couple of paragraphs above, the simplex optimization terminates whenerver $\alpha$ falls below the threshold $\alpha_{min}$.

### 2.2.3 Pseudo Code

Before presenting the pseudo code of the simplex optimization algorithm, we recall the parameters needed for this step.

- $\alpha_{start}$, the starting search step,
- $\alpha_{min}$, the threshold under which the optimization terminates,
- $\epsilon$, the multiplicative decrement of alpha performed whenever no descent direction is found in the direct search,
- $\delta$, the multiplicative increment of alpha performed during the expansion step,
- $\gamma$, the threshold for the sufficient decrease check.

**Inizialization:** $\alpha = \alpha_{start}$, $y \leftarrow y_0$;
**while** *true* **do**
  *reduction* $\leftarrow$ *false*;
  **for** $d_j \in D_c$ **do**
    **while** $\hat{f}_A(y + \alpha d_j) \leq \hat{f}_A(y) - \gamma\alpha^2$ **do**
      *reduction* $\leftarrow$ *true*;
      $\alpha \leftarrow \alpha\delta$;
    **end**
    **if** *reduction* **then**
      $\alpha \leftarrow \alpha/\delta$;
      $y \leftarrow y + \alpha d_i$;
      *break*;
    **end**
  **end**
  **if** *reduction* **then**
    restore $\alpha$ as before the first expansion;
  **else**
    $\alpha \leftarrow \alpha\epsilon$;
    **if** $\alpha < \alpha_{min}$ **then**
      *break*;
    **end**
  **end**
**end**

**Algorithm 6:** Simplex Optimization Step

## 2.3 Simplex Update Step

The crucial part of our algorithm lies in the simplex update. In the standard simplex decomposition framework, this operation is performed solving a linearization of the problem based on the gradient. DF-SD will try to estimate the gradient using the information gathered during the simplex optimization phase. This estimation is far from optimal, but it does not require any computational effort. In the following section, we will also present a modified version of the algorithm, which estimate the full simplex gradient instead. This will spend a lot of the budget function evaluations, but in smooth contexts will provide an optimal update for the simplex.

### 2.3.1 Gradient Estimation

The procedure used by the algorithm to find a good approximation of the gradient is quite simple. It requires just basic linear algebra and the concept of first order Taylor approximation. We briefly recall that, given $f : \mathbb{R}^n \to \mathbb{R}$, the first order approximation

of $f$ in a neighborhood of $x_0$ is:

$$f(x) = f(x_0) + \nabla f(x_0)^T (x - x_0). \tag{2.3}$$

From this equation we can see that $g = \nabla f(x_0)$ can be obtained as the solution of the underdetermined linear system:

$$(x - x_0)^T g = f(x) - f(x_0). \tag{2.4}$$

If we take multiple points $x_1, \ldots, x_j$ in a neighborhood of $x_0$, we can add rows to the system, so the (2.4) becomes:

$$S^T g = df, \tag{2.5}$$

where:

$$S = \begin{bmatrix} x_1 - x_0 & \ldots & x_j - x_0 \end{bmatrix}$$

and:

$$df = \begin{bmatrix} f(x_1) - f(x_0) \\ \vdots \\ f(x_j) - f(x_0) \end{bmatrix}.$$

The next step we need is to obtain the gradient $g = \nabla f(x_0) \in \mathbb{R}^n$ through the informations we have in the reduced space $\mathbb{R}^l$. If we consider the system (2.5) in the reduced space, we have:

$$\hat{S}^T g_y = d\hat{f}, \tag{2.6}$$

where, obviously:

$$\hat{S} = \begin{bmatrix} y_1 - y_0 & \ldots & y_j - y_0 \end{bmatrix},$$

$$d\hat{f}_A = \begin{bmatrix} \hat{f}_A(y_1) - \hat{f}_A(y_0) \\ \vdots \\ \hat{f}_A(y_j) - \hat{f}_A(y_0) \end{bmatrix}$$

and:

$$g_y = \nabla_y \hat{f}_A(y_0). \tag{2.7}$$

We can now expand (2.7) for the final equivalence we need:

$$g_y = \nabla_y \hat{f}_A(y_0) = \nabla_y f(Ay_0) = A^T \nabla_x f(x_0) = A^T g_x. \tag{2.8}$$

Combining (2.6) and (2.8), we obtain the system the algorithm will use to estimate $g_x$, that is:

$$\hat{S}^T A^T g_x = d\hat{f}_A. \tag{2.9}$$

The sampling points which compose the matrix $\hat{S}$ are the last $2l$ points where the algorithm evaluated the function in the last cycle of the simplex optimization. Hoping into a better estimation, we also add another sampling point, using the direction which is orthogonal to the $\mathbb{R}^l$-simplex: $d_{2l+1} = \sqrt{l}[1 \ldots 1]^T$.

Once we have an estimation of the gradient $g_x$, we can finally add a new vertex to the simplex. This is done solving the linear program:

$$v_{new} = \arg\min g_x^T v, \tag{2.10}$$

$$\text{subject to } v \in \text{conv}\{v_{i_1}, \ldots, v_{i_{n-l}}\},$$

where $v_{i_1}, \ldots, v_{i_{n-l}}$ are the columns of $V_p$.

What the algorithm does is taking the submatrix $V_p$ of $V$ composed by the vertices which are not in the current simplex and performing a simple matrix multiplication to see which vector of $V_p$ could provvide the best reduction:

$$r = V_p^T g_x.$$

So:

$$v_{new} = (V_p)_{i^*}, \tag{2.11}$$

where $i^*$ is the column index that satisfy:

$$i^* = \underset{i \in \{1,\ldots,n-l\}}{\arg\min} \; r^i.$$

There is the possibility that the every entry of $r$ is positive, that is, there is no descent vertex. However, because of the unreliability of $g_x$, the algorithm does not stop until the budget computations are over.

### 2.3.2 Pseudo Code

We recall that, when the simplex optimization step is over, we already have the evaluation of $\hat{f}_A$ in the points $y_j = y_0 + \alpha d_j, d_j \in D_c$ for $j = 1, \ldots, 2l$, where:

- $\alpha$ is the smallest acceptable (that is, greater or equal $\alpha_{min}$) step,
- $y_0$ is the actual best guess, because the last directions trial from $y_0$ has not made any decrease.

With that said, the pseudo code follows.

1. **Create Sample:** Set $\alpha \leftarrow \alpha/\epsilon$ and put:
   - $y_{2l+1} = y_0 + \alpha d_{2l+1}$;
   - $\hat{S} = \left[y_1 - y_0 \ \dots \ y_{2l+1} - y_0\right]$;
   - $d\hat{f}_A = \begin{bmatrix} \hat{f}_A(y_1) - \hat{f}_A(y_0) \\ \vdots \\ \hat{f}_A(y_{2l+1}) - \hat{f}_A(y_0) \end{bmatrix}$.
2. **Least Squares Estimation:** Solve:
$$\hat{S}^T A^T g_x = d\hat{f}_A.$$
3. **Simplex Update:** Solve:
$$v_{new} = (V_p)_{i^*}$$
and put:
   - $A \leftarrow [A \ v_{new}]$,
   - $y_0 \leftarrow [y_0^1 \dots y_0^l \ 0]$.

**Algorithm 7:** Simplex Update Step

## 2.4 Additional Notes

Before moving onto the Chapter related to the analysis of the numerical results, we will spend a couple of words on the convergence of the algorithm, as well as on the *full gradient* variant.

### 2.4.1 Algorithm Convergence

First of all, we will assume that $f$, the function to be minimized, is smooth (once differentiable at least). Given the fact that $X$ is a compact set, $f$ admits a minimum.

We refer to an algorithm iteration, or major cycle, as the sum of the simplex optimization step plus the simplex update step. A simplex optimization cycle, or minor cycle, will be a single *while* cycle of Algorithm 6. The following proposition will prove that the algorithm converges to stationary points.

**Proposition 2.1.** *Given a smooth function $f$ with $\nabla f$ Lipschitz continuous, DF-SD terminates into a stationary point after a finite number of iterations.*

*Proof.* The directions set $D_c$ is a positive spanning set in $\mathbb{R}^l$, the simplex optimization setp is therefore a directional direct search on the simplex, which converges to a stationary point (of the simplex) thanks to the convergence results in [1].

After at most $n$ major cycles, the matrix $A$ contains all vertices of $X$, that is, $A = V$ (up to a permutation of the columns). Consequently, the direct search is done on the entire $X$ and the algorithm terminates returning a stationary point for the master problem. $\square$

### 2.4.2 Full Gradient Variant

We already mentioned before that a possible alternative of estimating the gradient through the reduced space is to approximate a sort of full gradient instead, that is, the variation of the function along the directions $v_j - x_0$, $v_j \in V_p$ and $x_0 = Ay_0$. This will require a huge number $(n-l)$ of function evaluations, which is usually slightly less than 1 percent of the total budget, but in a very smooth context will provide an optimal new vertex. To avoid wasting too many function evaluations, the full gradient estimation is turned off whenever the reduction of the function between the end of a simplex optimization step and the next one (that is, a full major cycle lenght) is less than a fixed percentage. We call $\pi_{reduction}$ such percentage and $h$ the step to compute this gradient, then the pseudo code of the simplex update step becomes:

**0. Exit Check:** If $\frac{\hat{f}_A(y_0^{old}) - \hat{f}_A(y_0)}{\hat{f}_A(y_0^{old})} < \pi_{reduction}$ then switch to the standard simplex update.

**1. Create Gradient:** Put:

$$g_p = \left[ \frac{f(x_0 + h(v_{i_1} - x_0)) - f(x_0)}{h} \quad \cdots \quad \frac{f(x_0 + h(v_{i_{n-l}} - x_0)) - f(x_0)}{h} \right]^T$$

**2. Simplex Update:** Solve:

$$i^* = \underset{i \in \{1,\ldots,n-l\}}{\arg\min}\ g_p^i,$$

$$v_{new} = (V_p)_{i^*}$$

and put:

- $A \leftarrow [A\ v_{new}]$,
- $y_0 \leftarrow [y_0^1 \ldots y_0^l\ 0]$.

**Algorithm 8:** Full Gradient Simplex Update Step

# Chapter 3

# Algorithm Performance

Here we will present the performance and data profiles of our algorithm compared to the current meta. In the following sections we will carefully explain how the performance test were made and analyze the results.

## 3.1 Performance and Data Profiles

In this section we will describe the tools used to evaluate the algorithm performance, that is, the performance and data profiles for DFO solvers, proposed by More' and Wild [4]. Performance profiles, introduced the first time by Dolan and More' [5], have proved to be important for benchmarking optimization solvers. Dolan and More' define a benchmark in terms of a set $\mathcal{P}$ of benchmark problems, a set $\mathcal{S}$ of optimization solvers, and a convergence test $\mathcal{T}$.

The convergence test, which is used to understand how fast (in terms of function evaluations) the algorithm reaches the desired target, is the following [4]:

$$f(x_0) - f(x) \leq (1 - \tau)(f(x_0) - f_L), \tag{3.1}$$

where $f_L$ is the smallest value of $f$ obtained by any solver within a given number $\mu_f$ of function evaluations and $\tau$ is the precision parameter which is usually $10^{-k}$, $k \in \{1, 3, 5, 7\}$. A smaller value of $\tau$ is requesting a target which is closer to $f_L$.

### 3.1.1 Performance Profiles

Performance profiles are defined in terms of a performance measure $t_{p,s} > 0$ obtained for each $p \in \mathcal{P}$ and $s \in \mathcal{S}$. In the context of Black-Box programming, $t_{p,s}$ is usually defined as the number of function evaluations needed by solver $s$ to reach the convergence test (3.1) on problem $p$. Consequently, larger values of $t$ indicate worst performance. We also put $t_{p,s} = +\infty$ whenever the convergence test fails within the budget number of function evaluations. Fixed a problem $p \in \mathcal{P}$, we can define the performance ratio $r_{p,s}$ of a solver $s$ as:

$$r_{p,s} = \frac{t_{p,s}}{\min\{t_{p,s} : s \in \mathcal{S}\}},$$

that is, the ratio between the time taken by solver $s$ and the best time taken by any solver to reach (3.1). Therefore, $r_{p,s} = 1$ if and only if $s$ is (one of) the best solver(s) for problem

$p$ and $r_{p,s} = +\infty$ if and only if $s$ failed to converge on problem $p$. The performance profile of a solver $s \in \mathcal{S}$ is defined as the fraction of problems where the performance ratio is at most $\alpha$, that is:

$$\rho_s(\alpha) = \frac{1}{|\mathcal{P}|} size\{p \in \mathcal{P} : r_{p,s} \leq \alpha\}, \tag{3.2}$$

where $|\mathcal{P}|$ denotes the cardinality of $\mathcal{P}$. The plot of $\rho_s(\alpha)$ will thus represent an increasing function that ranges between the percentage of $\mathcal{P}$ which $s$ is best solving at and the percentage of $\mathcal{P}$ which $s$ can solve.

### 3.1.2 Data Profiles

Performance profiles provide an accurate view of the relative performance of solvers within a given number $\mu_f$ of function evaluations. Performance profiles do not, however, provide sufficient information for a user with an expensive optimization problem [...]. Users with expensive optimization problems are often interested in the performance of solvers as a function of the number of functions evaluations. In other words, these users are interested in *the percentage of problems that can be solved (for a given tolerance $\tau$) with $\kappa$ function evaluations* [4].

We can provide this information with data profiles simply putting:

$$d_s(\alpha) = \frac{1}{|\mathcal{P}|} size\{p \in \mathcal{P} : t_{p,s} \leq \alpha\}.$$

This definition of $d_s$ is independent of the number of variables in the problem $p \in \mathcal{P}$. This is not realistic because, in our experience, the number of function evaluations needed to satisfy a given convergence test is likely to grow as the number of variables increases [4]. The data profile is then defined switching the number of function evaluations with the number of simplex gradients, that is:

$$d_s(\alpha) = \frac{1}{|\mathcal{P}|} size\{p \in \mathcal{P} : \frac{t_{p,s}}{n+1} \leq \alpha\}. \tag{3.3}$$

Data profiles will thus indicate how many problems $s$ can solve in $\alpha$ simplex gradients time or less.

## 3.2 Test Problems Definition

Now that the method of performance evaluation is clear, we will describe the problems set $\mathcal{P}$. We conducted two separate tests. The first one requires the solvers to minimize quadratic convex functions, while the latter takes smooth but non convex functions. The feasible region is $X = conv(v_1, \ldots, v_n)$, $v_i \in \mathbb{R}^m$ for both, where every entry of $v_i$ is randomly generated as:

$$v_i^j \sim \mathcal{N}(0, 100),$$

that is, a normally distributed random variable with $\mu = 0$ and $\sigma^2 = 100$.

### 3.2.1 Convex Problems

The set of convex problems is fairly simple. We start generating a positive definite matrix $Q$ with condition number $\kappa$ using the following algorithm:

- Generate $P \in \mathbb{R}^{m \times m}$, with $p_{i,j} \sim Unif(0,1)$,
- Symmetrize $P$: $P \leftarrow P + P^T$,
- Change $P$'s eigenvalues: $P \leftarrow P + (\frac{\lambda_n - \lambda_1}{\kappa - 1} - \lambda_1)\mathbb{I}$, where $\lambda_1 \leq \cdots \leq \lambda_n$ are the eigenvalues of $P$,
- Take $Q$ as a rotation of $P$: $Q \leftarrow R^T P R$, where $R$ is a random orthogonal matrix.

**Algorithm 9:** Q Generation

The quadratic function to be minimized is then:

$$f(x) = \frac{1}{2} x^T Q x + c^T x, \tag{3.4}$$

where $c_i \sim \mathcal{N}(0, 100m^2)$.

We also make sure that the minimum of $f$ is outside of $X$ (with a tolerance of $10^{-10}$), repeating the function generation if this is not the case. With that said, for every fixed $m$, we generate several instances of this problem by varying $\kappa = 1.1^i$, $i \in \{1, \ldots, 11\}$.

### 3.2.2 Non-Convex Problems

The set of non-convex problems is made of 11 scalable functions which (for conducting reasonably large tests in a fair amount of time) can be computed with a linear number of operations. We list just the function names, while the Matlab code of the functions is provided in the Appendix which also include the starting point (that we switch for one vertex of the ranom generated feasible region) for each function.

- Arwhead Function,
- Brown Almost Linear Function,
- Broyden Function,
- Boundary Value Problem Function,
- Chained Rosenbrock Function,
- Extended ENGVL1 Funcion,
- Extended Freudenstein and Roth Function,
- Oren Function,

- Penality Function 1,
- Tridiagonal Function (Nash version),
- Trigonometric Function (Nocedal version).

## 3.3 Solver Definition

Before going into the details of the test, we briefly report the elements of $\mathcal{S}$, the solvers which compete in the performance evaluation along with the setup we used for every one of them.

### 3.3.1 DF-SD

The first algorithm is, of course, DF-SD. As we already presented the algorithm itself, we just report the setup:

- $\alpha_{start} = 0.1$,
- $\alpha_{min} = 10^{-4}$,
- $\epsilon = 0.1$,
- $\delta = 1.5$,
- $\gamma = 0.5$.

### 3.3.2 DF-SD Full Gradient Variant

The setup for the variant is identical to the previous. The only things we have to add are:

- $h = 10^{-4}$, the step to compute the pseudo gradient in $\mathbb{R}^{n-l}$,
- $\pi_g = 0.05$, the minimum percentage of reduction needed not to switch on the standard approximation.

### 3.3.3 SDPEN

SDPEN is a sequential penality derivative free algorithm, that is, an algorithm which solves the original nonlinear constrained optimization problem by a sequence of approximate minimizations of a merit function where penalization of constraint violation is progressively increased. The merit function is minimized by a derivative-free aproach based on line search [6]. The Matlab code was provided by the Derivative-Free Software Library [7].

We passed the SDPEN algorithm the function $\hat{f}(y) = f(Ay)$ subject to the constraint $|e^T y - 1| \leq 0$. The bound constraints and the additional parameters were put at:

- $lb = [0 \dots 0]^T$,

- $ub = [1 \dots 1]^T$,
- $\epsilon = 10^{-3}$, the constraint violation initial weight,
- $\alpha_{stop} = 10^{-6}$, the minimum tolerance before the algorithm stops.

If the returned solution provided a constraint violation which is larger than $10^{-3}$, we consider the optimization failed.

### 3.3.4 Particle Swarm

The Particle Swarm is a metaheuristic algorithm that tries to solve a problem by having a population of candidate solutions and moving these particles (the so called swarm) around in the search-space. This algorithm is proven to be a good alternative to standard (non heuristic methods) when the dimension of the problem are very large so we decided to use it as another competitor.

The code is provided by Matlab in the Global Optimization Toolbox. Given the fact that this method does not support non-bound constraints, we implemented a merit function given by:

$$g(y) = \hat{f}(y) + f(y_0)\frac{1}{\epsilon}|e^T y - 1|. \tag{3.5}$$

We considered a solution to be acceptable if $|e^T y - 1| < 10^{-3}$. If the solution was not acceptable, we put $\epsilon \leftarrow 0.1\epsilon$ and repeat the optimization until either an acceptable point is returned or $\epsilon < 10^{-10}$.

### 3.3.5 Nelder Mead

The last algorithm we used in our comparison is the Nelder Mead implementation provided by the Matlab Matrix Computation Toolbox. Unfortunatly, neither generic or bound constraints support is provided, so we repeated (3.5) and the acceptability check, with the addition of the bound constraint:

$$g(y) = -\hat{f}(y) - f(y_0)\frac{1}{\epsilon}(|e^T y - 1| + \sum \max(-y^i, 0)), \tag{3.6}$$

where the negative sign is needed since the algorithm *nmsmax* solves maximization problems.

## 3.4 Additional Setup Informations

The test was conducted on Matlab R2017b installed on Windows 10.0.17134. The machine mounted an Intel Pentium G3258 at 3.20 GHz and 8.0 GB of RAM. The total computation was run on one single core and took about one day.

For convex problems, we tested the algorithms on the 11 problems presented above with the following parameters:

- $n \in \{20, 40, 60, 100, 200, 500, 1000, 2000\}$,
- $m = \beta n$, $\beta \in \{0.2, 0.5\}$,

for a total of 176 problems. We divided those problems in three groups based on the problem dimension:

- Small: $n \in \{20, 40, 60\}$,
- Medium: $n \in \{100, 200, 500\}$,
- Large: $n \in \{1000, 2000\}$.

Given the fact that the performance of the solver on a single problem are very different, we chose to use $\tau = 0.1$ for the profiles.

During the test on non-convex problems, we encountered the following problem: the local minimums of the unconstrained functions heuristically are with high probability in the interior of the feasible reagion. This means that comparing a constraint based algorithm with a penality one is unfair because the latter is basically doing unconstrained optimization. We can also see this by the fact that SDPEN cannot solve a single convex problem without breaking the constraints when $n \geq 200$, while on the non-convex ones it does not break any. With that said, we provide only results with $20 \leq n \leq 100$, with different values of $\tau$: $\tau \in \{10^{-1}, 10^{-3}, 10^{-5}\}$.

## 3.5 Results

We report the plots in the following pages, along with a brief commentary on the results.

(this page is intentionally left blank for ease of reading)

### 3.5.1 Convex Problems

Without any doubt, both version of DF-SD outperform the other benchmark algorithms. The only exception is on the small dimension problems, where all the methods are at least comparable. SDPEN is still a better option whenever the user is interested in reaching a good reduction of the objective function with a tiny budget, as it solves more than a half of the problems faster than the other algorithms and it does not gain any more solving power as the simplex gradients increases.

When we move to medium size problems, the differences become substantial. SDPEN has a little advantage again on the very low budget side, but it was unable to solve more than 80 percent of the problems, compared to DF-SD with full gradient which was capable of succesfully complete more than 95 percent of the convergence tests and did best on roughly 80 percent.

Finally, on the large scale problems, none of the other algorithms was able to reduce the objective function up to the convergence test in any problem (*nmsmax* was excluded here due to the large amount of computation time needed to solve problems of this dimension). As we can see from the plots, the full gradient variant really shines compared to its counterpart whenever the problem has an easily approximable gradient. That's because even if it tooks nearly one simplex gradient just to add a new vertex to the simplex, the new direction often provides a huge decrease in the objective function.

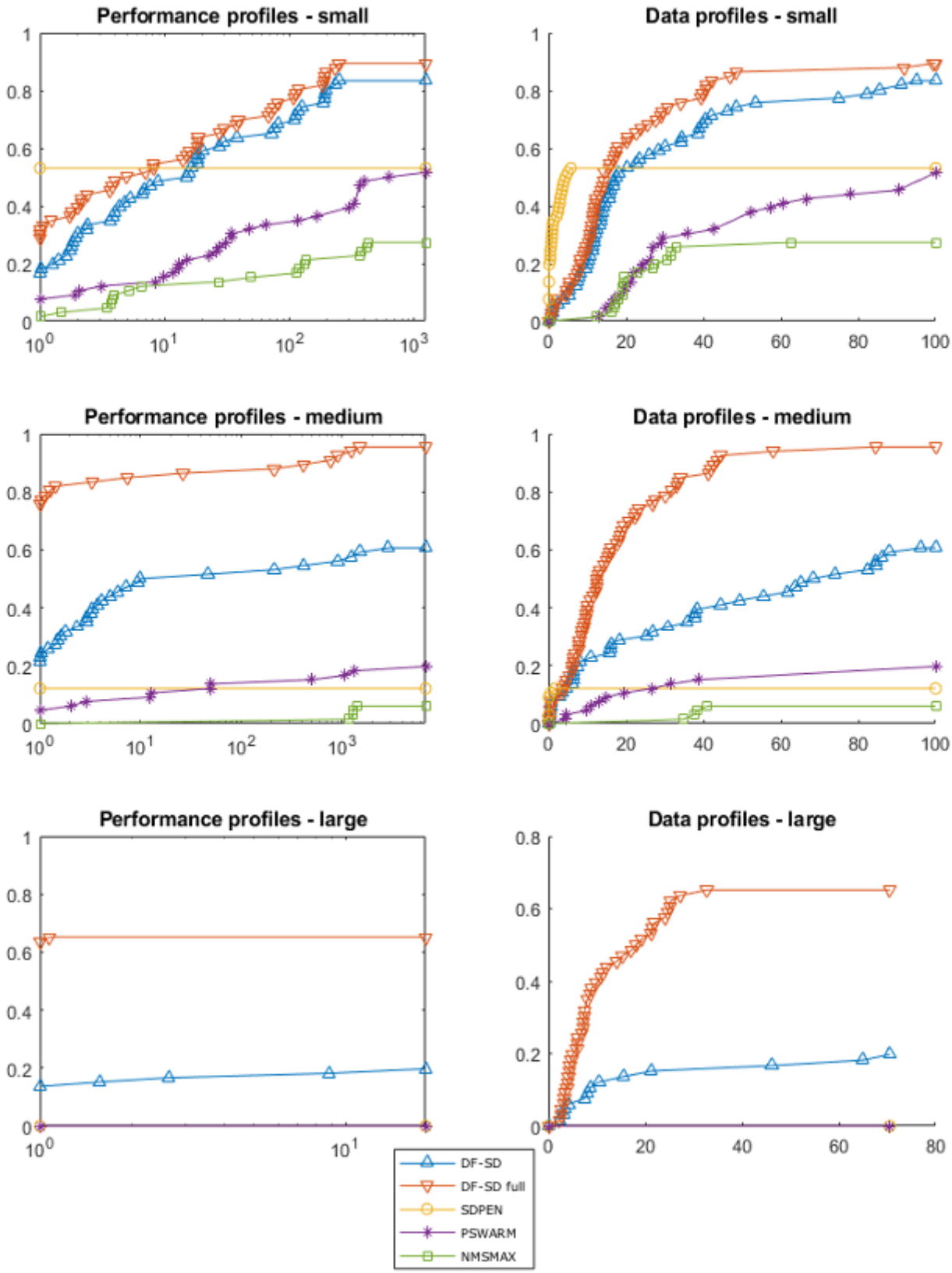


Figure 3.1: Performance and Data Profiles for Convex Problems.

### 3.5.2 Non-Convex Problems

Even if DF-SD was able to reach the convergence on a larger number of problems, SDPEN seems actually preferable considering the really small number of iterations it needs to achive an acceptable objective function reduction. The particle swarm method and *nmsmax* also provided better performances compared to the convex case. We already explain the problem of local minimums in section 3.4 and the plots agree with that.

As $\tau$ decreases, the precision required for the convergence test $\mathcal{T}$ ranges from 90 percent to 99.999 percent. However, the solvers ranking does not change much, being DF-SD full gradient the best solver in terms of total problem completed and SDPEN the best solver in terms of budget management.

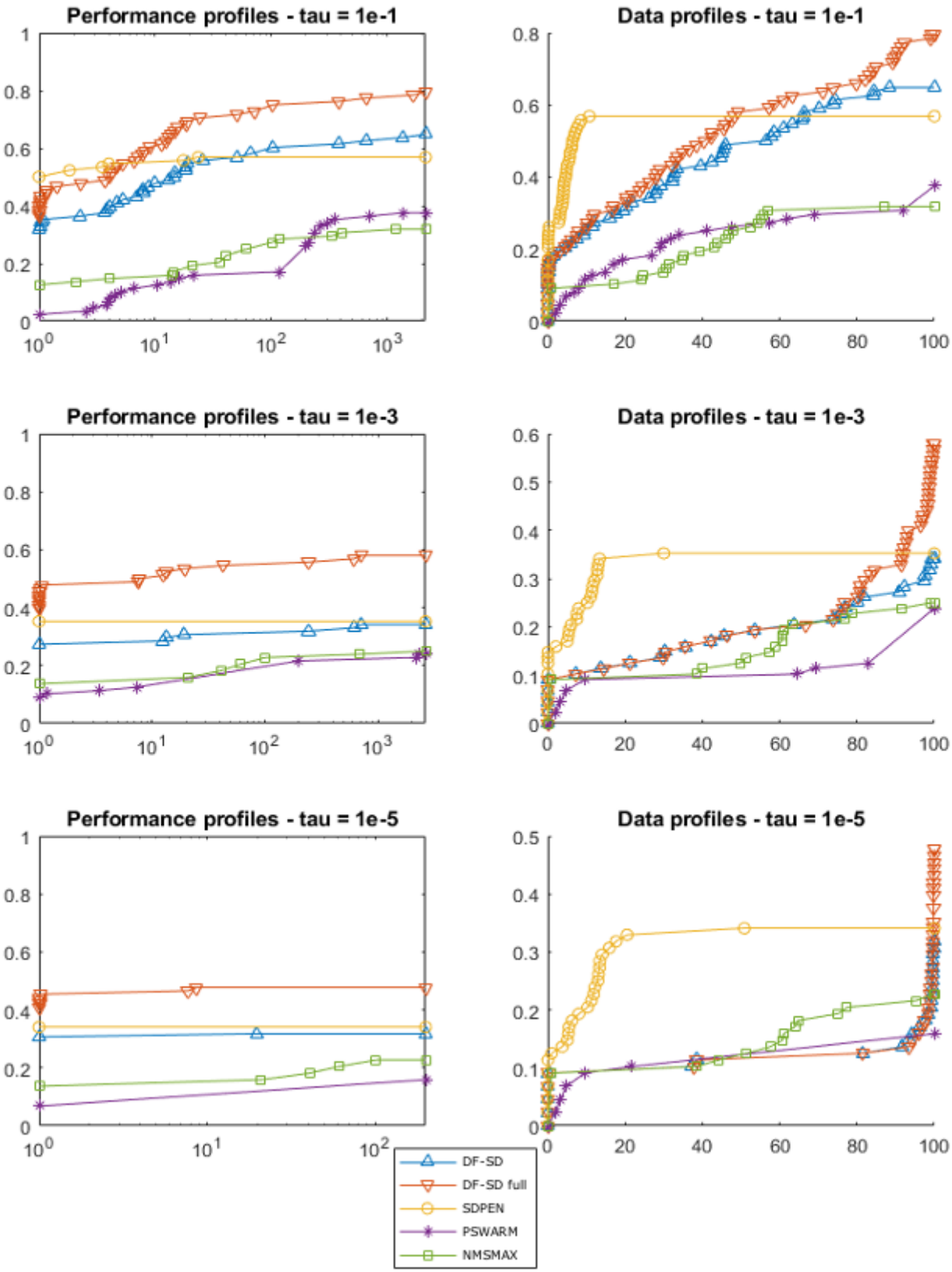


Figure 3.2: Performance and Data Profiles for Convex Problems.

## 3.6 Conclusions and Future Work

The conclusions follow clearly from the previous section. Whenever a derivative-free optimization problem defined on a polytope whose vertices are known needs to be solved, DF-SD is a good choice, especially if the problem dimension is large. When dealing with functions with multiple local minima, classic approaches still have good behavior. Hence, some clever strategies for exploring the feasible set should be developed in this case.

Future tests for this algorithm will therefore include non-convex problems with few minimums and defined on more complex polyhedra. A larger set of benchmark solvers could further be included.

Another important line of research might be working on a strategy that cuts out the worst vertices of the simplex. This add-on could really speed up the computation and extend the number of algorithm iterations (that is, major cycles) since the dimension of the simplex would stay restrained. However, a stronger convergence analysis is needed in this case. With that said, we believe the algorithm represents a good alternative to the current state-of-the-art methods.

# Appendices

# Appendix A

# Matlab Code

## A.1 DF-SD

```
1
2  function [f_min,x_min,f_step,f_tot_comput] = dfsd(f_ob,...
3    vertex_array,y0,f_budget,verbose)
4  %DFSD, DFO in a convex combination
5  %    Perform derivative free optimization into the convex hull of
6  %    a set of given points using simplicial decomposition.
7  %    f_min, x_min: best value and point for the algorithm
8  %    step_decr: best value after each computation of f
9  %    f_tot_comput: total computations of f
10 %    f_ob: objective function passed as anonymous function
11 %    vertex_array: a mxn matrix representing the n points which
12 %    give the convex hull
13 %    y0: initial point given as a combination of the points of
14 %    vertex_array
15 %    f_budget: maximum number of computation of f allowed
16 %    verbose: bool, print output
17
18
19 % PARAMETERS
20
21 alpha_start = .1; % starting value for the step in the df research
22 alpha_min = 10^(-4); % step for the conclusion of simplex opt
23 decr = 0.1; % multiplicative decrement of alpha
24 delta = 1.5; % multiplicative increment of alpha in the expansion
25 gamma = 0.5; % threshold for sufficient decrease
26
27
28 % LOG ITEMS
29
30 f_step = []; % value of f(y) each computation
31 f_comput = 0; % computations for the current simplex opt
32 f_tot_comput = 0; % total computations
33 inner_cycles = 0; % cycles of simplex minimization
```

```
34 avg_exp = [0,0]; % avg. expansions during an expansion step
35
36
37 % START
38
39 n = size(vertex_array,2); % problem dimension (n of points)
40 possible_vert = vertex_array(:,y0==0); % points we can add
41 A = vertex_array(:,y0~=0); % points of the simplex
42 y = y0(y0~=0); % starting point as a combination of A
43 f = f_ob;
44 check = 0;
45
46
47 while ~check % MAIN CYCLE
48
49     l = size(y,1);
50     while l < 3 % dimension check
51         % updates A and y
52         A = [A, possible_vert(:,1)]; %#ok<AGROW>
53         possible_vert = possible_vert(:,2:end);
54         y = [y; 0]; %#ok<AGROW>
55         l = size(y,1);
56     end
57
58     alpha = alpha_start;
59     E = eye(l); % to get easy canonic base vectors
60     f_around = zeros(2*l,1); % f values around the point
61
62     %
63     % DF OPT ON THE SIMPLEX
64     while alpha > alpha_min && ~check
65     %
66
67         reduction = 0; % of the objective function
68         f_y = f(A*y);
69         f_comput = f_comput + 1; % LOG
70         f_step = [f_step; f_y]; %#ok<AGROW> % LOG
71         % checks if we went out of budget
72         if f_tot_comput+f_comput > f_budget-1
73             check = 1;
74             if verbose
75                 disp(' ');
76                 disp('OUT OF BUDGET');
77                 disp(' ');
78             end
79             break
80         end
81         %
82
```

```
83          alpha_not_exp = alpha; % reminder of the current step
84
85          %
86          for i=1:l % find the descent direction and move y
87          %
88
89              % tries the positive direction
90              f_around(2*i-1) = f(A*(y+alpha*(E(:,i)-y)));
91              f_comput = f_comput + 1; % LOG
92              f_step = [f_step; f_y]; %#ok<AGROW> % LOG
93              % checks if we went out of budget
94              if f_tot_comput+f_comput > f_budget-1
95                  check = 1;
96                  if verbose
97                      disp(' ');
98                      disp('OUT OF BUDGET');
99                      disp(' ');
100                 end
101                 break
102             end
103             %
104
105             while f_around(2*i-1) <= f_y - gamma*alpha^2 &&...
106                 alpha <= 1
107                 reduction = 1;
108                 alpha = alpha*delta; % expansion try
109                 f_around(2*i-1) = f(A*(y+alpha*(E(:,i)-y)));
110                 f_comput = f_comput + 1; % LOG
111                 f_step = [f_step; f_y]; %#ok<AGROW> % LOG
112                 % checks if we went out of budget
113                 if f_tot_comput+f_comput > f_budget-1
114                     check = 1;
115                     if verbose
116                         disp(' ');
117                         disp('OUT OF BUDGET');
118                         disp(' ');
119                     end
120                     break
121                 end
122                 %
123             end
124
125             if reduction % update the y and exit the for loop
126                 alpha = alpha/delta; % last alpha is out so go back
127                                                     % 1 step
128                 avg_exp(1) = (avg_exp(1)*avg_exp(2)+... % LOG
129                     log(alpha/alpha_not_exp)/log(delta))/...
130                           (avg_exp(2)+1);
131                 avg_exp(2) = avg_exp(2)+1;
```

```
132
133                 y = y+alpha*(E(:,i)-y);
134                 break
135             end
136
137             % tries the negative direction
138             f_around(2*i) = f(A*(y-alpha*(E(:,i)-y)));
139             f_comput = f_comput + 1; % LOG
140             f_step = [f_step; f_y]; %#ok<AGROW> % LOG
141             % checks if we went out of budget
142             if f_tot_comput+f_comput > f_budget-1
143                 check = 1;
144                 if verbose
145                     disp(' ');
146                     disp('OUT OF BUDGET');
147                     disp(' ');
148                 end
149                 break
150             end
151             %
152
153             while f_around(2*i) <= f_y - gamma*alpha^2 &&...
154                 alpha <= y(i)/(1-y(i))
155                 reduction = 1;
156                 alpha = alpha*delta; % expansion try
157                 f_around(2*i) = f(A*(y-alpha*(E(:,i)-y)));
158                 f_comput = f_comput + 1; % LOG
159                 f_step = [f_step; f_y]; %#ok<AGROW> % LOG
160                 % checks if we went out of budget
161                 if f_tot_comput+f_comput > f_budget-1
162                     check = 1;
163                     if verbose
164                         disp(' ');
165                         disp('OUT OF BUDGET');
166                         disp(' ');
167                     end
168                     break
169                 end
170                 %
171             end
172
173             if reduction % update the y and exit the for loop
174                 alpha = alpha/delta; % last alpha is out so go back
175                                                     % 1 step
176                 avg_exp(1) = (avg_exp(1)*avg_exp(2)+... % LOG
177                     log(alpha/alpha_not_exp)/log(delta))/...
178                           (avg_exp(2)+1);
179                 avg_exp(2) = avg_exp(2)+1;
180
```

```
181                 y = y-alpha*(E(:,i)-y);
182                 break
183             end
184         %
185         end % for
186         %
187
188         if reduction % back to the original alpha
189             alpha = alpha_not_exp;
190         else % decrease the step
191             alpha = alpha*decr;
192         end
193
194         inner_cycles = inner_cycles + 1; % LOG
195
196     %
197     end % while
198     %
199
200     % set alpha to the last used in the simplex
201     if alpha <= alpha_min
202         alpha = alpha/decr;
203     end
204
205
206
207     if ~check
208         f_tot_comput = f_tot_comput + f_comput;
209         if verbose
210             disp('f(y):');
211             disp(f_y);
212             disp(' ');
213             disp('number of inner cycles:');
214             disp(inner_cycles);
215             disp(' ');
216             disp('computations of f:');
217             disp(f_comput);
218             disp(' ');
219         end
220         inner_cycles = 0;
221         f_comput = 0;
222
223         % gradient estimation
224
225         f_around = [f_around; f(A*(y+alpha*ones(l,1)*l^0.5))];
226                         %#ok<AGROW> (sqrt(l) rescales)
227         f_tot_comput = f_tot_comput + 1; % LOG
228         f_step = [f_step; f_y]; %#ok<AGROW> % LOG
229
```

```
230         df = f_around-f_y*ones(2*l+1,1); % variation of f
231
232         S = zeros(l,2*l+1); % points variation (y_around(i) - y)
233
234         for i=1:l
235             S(:,2*i-1) = alpha*(E(:,i)-y); % = (y + alpha*d_i) - y
236             S(:,2*i) = -alpha*(E(:,i)-y); % = (y - alpha*d_i) - y
237         end
238         S(:,2*l+1) = alpha*ones(l,1)*l^0.5; % orth. direction
239
240         % finds the approximation of the gradient in the n-dim space
241         g_x = lsqlin(S'*A',df); % least square beacause system has
242                                 % rank l and there is 2l+1 rows
243
244         % computes the new vertex (if all entry are positive exits)
245         gradient_values = possible_vert'*g_x;
246
247         % checks if budget is over during the gradient computation
248         if check
249             break
250         end
251
252         % computes the new vertex
253         [~,lower_index] = sort(gradient_values,'ascend');
254         v_new = possible_vert(:,lower_index(1));
255
256         % updates the possible_vert
257         possible_vert(:,lower_index(1)) = [];
258
259         % checks if we used all points
260         if ~size(possible_vert,2)
261             disp(' ');
262             disp('ALL POINTS USED');
263             disp(' ');
264             break
265         end
266
267         % updates A and y
268         A = [A, v_new]; %#ok<AGROW>
269         y = [y; 0]; %#ok<AGROW>
270     end
271
272 end % while
273
274 f_min = f_y;
275 x_min = A*y;
276
277 end
```

## A.2 Non-Convex Problems

```
function [ARWHEAD,ARWHEAD_V,BLIN,BLIN_V,BROYDEN,BROYDEN_V,BVP, ...
 BVP_V,ChROS,ChROS_V,EXTENG,EXTENG_V,EXTFRE,EXTFRE_V,OREN,OREN_V, ..
 PEN1,PEN1_V,TRID,TRID_V,TRIG,TRIG_V] = problems_generator(n,m)

    % n: dimension (number of samples)
    % m: dimension of the space
    % V: matrix of vertices (each column is a vertex)

    rng(0)
    V = 1e1*randn(m,n-1);

    % Starting points

    ARWHEAD_V = [ones(m,1), V];

    BLIN_V = [.5*ones(m,1), V];

    BROYDEN_V = [[0;-ones(m-2,1);0], V];

    BVP_start = zeros(m,1);
    for i_0 = 2:m-1
        st = i_0/(n-1);
        BVP_start(i_0) = st*(st-1);
    end
    BVP_V = [BVP_start, V];

    ChROS_V = [-ones(m,1), V];

    EXTENG_V = [2*ones(m,1), V];

    EXTFRE_V = [-2*ones(m,1), V];

    OREN_V = [ones(m,1), V];

    PEN1_V = [(1:m)', V];

    TRID_V = [ones(m,1), V];

    TRIG_V = [(1/m)*ones(m,1), V];


    % ARWHEAD

    function f = ARWHEAD_f(x)
        temp_ARWHEAD = 0;
        for i = 1:m-1
            temp_ARWHEAD = temp_ARWHEAD + (-4.0*x(i)+3.0) + ...
```

```
48              (x(i)^2+x(m)^2)^2;
49          end
50          f = temp_ARWHEAD;
51      end
52      ARWHEAD = @ARWHEAD_f;
53
54
55      % BLIN
56
57      function f = BLIN_f(x)
58          s = sum(x)-m-1;
59          b = x(m);
60          temp_BLIN = 0;
61          for i = 1:m-1
62              temp_BLIN = temp_BLIN + (s+x(i))^2;
63              b = b*x(i);
64          end
65          f = temp_BLIN + (b-1)^2;
66      end
67      BLIN = @BLIN_f;
68
69
70      % BROYDEN
71
72      function f = BROYDEN_f(x)
73          temp_BROYDEN = 0;
74          for i = 2:m-1
75              temp_BROYDEN = temp_BROYDEN + ...
76              ((3-2*x(i))*x(i)-x(i-1)-2*x(i+1)+1)^2;
77          end
78          f = temp_BROYDEN;
79      end
80      BROYDEN = @BROYDEN_f;
81
82
83      % BVP
84
85      function f = BVP_f(x)
86          temp_BVP = 0;
87          for i = 2:m-1
88              s = (1/(m-1))^2*(x(i)+i/(m-1)+1)^3;
89              temp_BVP = temp_BVP + (2*x(i)-x(i-1)-x(i+1)+s/2)^2;
90          end
91          f = temp_BVP;
92      end
93      BVP = @BVP_f;
94
95
96      % ChROS
```

```
97
98      function f = ChROS_f(x)
99          temp_ChROS = 0;
100         for i = 2:m
101             temp_ChROS = temp_ChROS + 4*(x(i-1)-x(i))^2+(1-x(i))^2;
102         end
103         f = temp_ChROS;
104     end
105     ChROS = @ChROS_f;
106
107
108     % EXTENG
109
110     function f = EXTENG_f(x)
111         temp_NoAF = 0;
112         for i = 1:m-1
113             s = x(i)^2+x(i+1)^2;
114             temp_NoAF = temp_NoAF + s^2 -4*x(i)+3;
115         end
116         f = temp_NoAF;
117     end
118     EXTENG = @EXTENG_f;
119
120
121     % EXTFRE
122
123     function f = EXTFRE_f(x)
124         temp_EXTFRE = 0;
125         for i = 1:m-1
126             s = (5-x(i+1))*x(i+1)-2;
127             t = (x(i+1)+1)*x(i+1)-14;
128             temp_EXTFRE = temp_EXTFRE + (x(i)+x(i+1)*s-13)^2+...
129             (x(i)+x(i+1)*t-29)^2;
130         end
131         f = temp_EXTFRE;
132     end
133     EXTFRE = @EXTFRE_f;
134
135
136     % OREN
137
138     function f = OREN_f(x)
139         temp_OREN = 0;
140         for i = 1:m
141             temp_OREN = temp_OREN + i*x(i)^2;
142         end
143         f = temp_OREN^2;
144     end
145     OREN = @OREN_f;
```

```
146
147
148       % PEN1
149
150       function f = PEN1_f(x)
151           temp_PEN1 = 0;
152           s = 0;
153           for i = 1:m
154               temp_PEN1 = temp_PEN1 + 1e-5*(x(i)-1)^2;
155               s = s + x(i)^2;
156           end
157           f = temp_PEN1 + (s-0.25)^2;
158       end
159       PEN1 = @PEN1_f;
160
161
162       % TRID
163
164       function f = TRID_f(x)
165           temp_TRID = (x(1)-1)^2;
166           for i = 2:m
167               temp_TRID = temp_TRID + 4*i*(x(i)-x(i-1))^2;
168           end
169           f = temp_TRID;
170       end
171       TRID = @TRID_f;
172
173
174       % TRIG
175
176       function f = TRIG_f(x)
177           s = 0;
178           temp_TRIG = 0;
179           for i = 1:m
180               s = s + cos(x(i));
181           end
182           for i = 1:m
183               temp_TRIG = temp_TRIG + (m+i-sin(x(i))-s-i*cos(x(i)))^2;
184           end
185           f = temp_TRIG;
186       end
187       TRIG = @TRIG_f;
188
189   end
```

# Bibliography


[1] A. R. Conn, K. Scheinberg, L. N. Vicente, *Introduction to Derivative Free Optimization*, MPS-SIAM Series on Optimization, 2009.

[2] C. T. Kelley, *Implicit Filtering*, SIAM, 2011.

[3] A. R. Conn, *Trust Region Methods and Derivative Free Optimization*, Presentation at IBM T.J. Watson Research Center, 2014.

[4] J. J. More', S. M. Wild, *Benchmarking Derivative-Free Optimization Algorithms*, SIAM Journal of Optimization, 2009.

[5] E. D. Dolan, J. J. More', *Benchmarking Optimization Software with Performance Profiles*, Math. Program., 91 (2002).

[6] G. Liuzzi, S. Lucidi, M. Sciandrone, *Sequential Penalty Derivative-Free Methods for Nonlinear Constrained Optimization* SIAM Journal of Optimization 2010.

[7] *Derivative-Free Software Library*: www.iasi.cnr.it/∼liuzzi/DFL