university of groningen / faculty of science and engineering / mathematics and applied mathematics

A Hybrid Method for Numerical Optimization

Bachelor's Project Mathematics

January 2025

Student: J.D. van der Zeijden

First supervisor: dr. ir. R. Luppes

Second assessor: dr. A.E. Sterk

Abstract

Numerical optimization is concerned with minimizing or maximizing non-differentiable, real functions. Two very commonly used derivative-free minimization methods are the so-called Nelder-Mead method (NM) and Particle Swarm Optimization (PSO). This thesis examines a hybrid of these methods (NM-PSO) which aims to combine both of their strengths. To assess their performance, these methods are used to minimize the run-time of two different linear system solvers (SOR and preconditioned GMRES). These solvers use parameters that need to be chosen beforehand and a good choice can cause the algorithms to require significantly less computing time. The results of these tests suggest that NM-PSO is indeed just as accurate as PSO, while being considerably cheaper in terms of computational cost.

Contents

1 Introduction

Both within and outside the field of mathematics, lots of algorithms are used for all sorts of purposes. These often involve a lot of waiting for the process to finish and therefore we are always looking for ways to this speed up. These algorithms are typically configured by some parameters and the choice of these parameter values can sometimes have a great impacts on the run-time. We would thus like to be able to find the best parameter choices for any given algorithm and that is where the mathematical field of numerical optimization comes in.

Numerical optimization is concerned with minimizing or maximizing some function of the form:

$$f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R} \qquad \text{for some } n \in \mathbb{N}.$$

Minimization and maximization are in fact very closely related since minimizing some function f is equivalent to maximizing $-f$. Most literature therefore only focusses on one of the two and in this thesis we stick to minimization. The overall goal of numerical optimization can thus be described as finding some vector $x \in \Omega$ such that the corresponding function value $f(x)$ is as small as possible. In real world applications the function f is typically taken as the time it takes the run an algorithm with a specific choice of parameters.

In general, it is very unlikely that the function to be optimized is differentiable and therefore we need a method that does not rely on derivatives. Over the years many derivative-free minimization methods have been proposed, implemented and continuously improved. The works by Conn, Scheinberg and Vicente [1], Rios. and Sahinidis [2], and Larson, Menickelly and Wild [3] provide a good overview of these methods. There is currently no single method which performs the best in every situation [2] and there will likely never be such a method. Therefore some methods may perform very well in certain cases but very badly in others.

Previous bachelors projects by Petersen [4], Maquelin [5], Van der Meulen [6] and Ludwig [7] have taken a look at the performance of several of the most commonly used methods and at possible applications for numerically solving linear systems. This current project aims to build on their work by testing another promising minimization method (called NM-PSO) and comparing its performance to the previously studied methods on similar test cases.

The NM-PSO method was proposed in 2004 by Fan, Liang and Zahara [8] and it is a hybridization of the very popular Nelder-Mead (NM) and Particle Swarm Optimization (PSO) methods that were proposed by Nelder and Mead [9], and Kennedy and Eberhart [10] respectively. The hybrid method hopes to combine the strengths of both NM and PSO while diminishing their weaknesses and this thesis seeks to find out whether that is indeed the case.

After a brief introduction to derivative-free minimization methods in Section 2, the NM, PSO and NM-PSO method are described in Sections 3, 4 and 5. The performance of these three methods is compared in Section 6 where they are tested on a selection of test functions and two algorithms for solving linear systems called SOR and preconditioned GMRES. All the methods were implemented and tested in MATLAB and the code for this can be found in the Appendix.

2 Derivative-free minimization

As stated in the introduction, we are interested in minimizing functions of the form $f : \Omega \subseteq \mathbb{R}^n \to \mathbb{R}$. The function f that is to be minimized is usually called the **objective function** and the global minimizer is usually denoted by x^*. The only requirement we set for the objective function is that it must be well-defined.

Most importantly, this means that we do not assume the objective function to be differentiable, i.e. we cannot make use of derivatives. This excludes many of the commonly used minimization methods, such as Gradient Descend and the Conjugate Gradient method, since they fundamentally rely on the fact that $\bigtriangledown f = 0$ at a minimum [11]. The only available information to find an x^* such that $f(x^*) = \min\{f(x) \mid x \in U\}$ is thus the objective function f itself.

In a very, very general sense, every derivative-free minimization method does the same thing: it picks a number of points in the domain ω, evaluates the function at these points, and then it decides which point(s) to pick next. This is repeated until a certain stopping criterion is met. Figure 1 shows a diagram representation of the process.

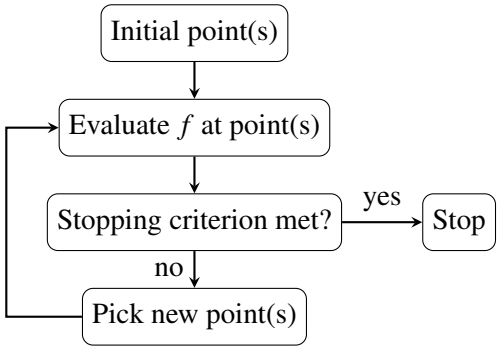


Figure 1: General derivative-free minimization method.

All the different global minimization methods that have been developed can generally be categorized by two classifications: a method is either **deterministic** or **stochastic** and it is either **direct** or **model-based** [2]. Deterministic methods use a deterministic algorithm to decide which points to try next, while stochastic methods use some form of randomness while picking the next points. Direct methods use the actual objective function when evaluating the points, while model-based methods construct and use an approximation of the objective function.

In general, we know more about the behavior and workings of the deterministic methods compared to the stochastic ones. Much research has been done into the conditions needed to assure convergence and recently these methods have enjoyed renewed interest because of their potential for parallelization [3]. Meanwhile, due to their inherent randomness, less is known about why and how the stochastic methods actually work. However, since they are relatively easy to implement, there does exist a lot of literature on their applications [2].

2.1 What makes a 'good' method?

Before we start comparing derivative-free minimization methods, we first need to define the properties by which we will judge them. A 'good' minimization method should satisfy the following three criteria:

- it always returns a 'decent' answer; (**robustness**)
- it gets as close as possible to the actual global minimum; (**accuracy**)
- it finds an answer as quickly as possible. (**computational cost**)

Throughout this thesis all the methods are implemented in such a way that they will always terminate at some point and such that they return the best answer they found during their search. We will consider this answer to be 'decent' enough, since it is at least better than all the other points that were tried. In this way, robustness is assured for all methods and therefore we will not consider it any further.

Our main goal is to find the global minimum of an objective function. Even if the global minimum is not known a priori, we still want to be relatively sure that the method has found it. Accuracy is thus going to be our primary concern and our methods will face two obstacles in this regard. The methods should not end up in a local minimum instead of a global one and they need to find the coordinates of the minimizer x^* as accurately as possible. The latter can usually be improved by decreasing the tolerance on the stopping criterion, but this is not necessarily always the case, as experienced by Maquelin [5].

The computational cost of finding a global minimum will be our secondary consideration when assessing minimization methods. The most obvious and relevant measurement of the computational cost is the time it takes until the method is done computing. In practice, the most time-consuming part of a method is usually the

evaluations of the objective functions [1]. This does however mean that a method might appear to be really fast when the objective function is relatively cheap to evaluate, while it can actually be really slow for more expensive objective functions. Almost all of the applications for minimization have very time consuming objective functions [1] and hence we will mainly focus on how quickly a method works for such functions. When testing a method on a cheap objective function we will therefore measure the computational cost in terms of the number of objective function evaluations instead of computation time.

2.2 Stopping criterion

There are several different ways to define stopping criteria for derivative-free minimization methods. We can for example stop the method, when:

- the difference between the best point of the current and that of the previous iteration is small enough; (**decrease-based**)
- the difference between the coordinates of the current points is small enough; (**coordinate-based**)
- the difference between the values of the current points is small enough; (**value-based**)
- a certain number of iterations is reached; (**iteration-based**)
- a certain number of objective function evaluations is reached. (**evaluation-based**)

To be able to fairly compare different global minimization methods in this thesis, we want to choose a stopping criterion that is applicable to all methods.

Decreased-based criteria are generally not well-suited for this purpose since certain methods, especially stochastic ones, exhibit behavior that could easily cause the stopping criterion to be met, even though the guess is nowhere near a minimum. It might, for example, be the case that the guess improves only a very little during the first 100 iterations, but massively improves in the 101st iteration. A decrease-based criterion would have been satisfied long before the 101st iteration and thus the method would have been stopped too early.

A coordinate-based or value-based criterion appears to be a more suitable option. When all the considered points or values are very close together, the method has usually arrived at a minimum and thus this would be a good moment to stop the method. There are however still two problems with two types of criteria:

- 1: They can still be falsely triggered:
    - coordinate-based: when the points accidentally all end up in the same place;
    - value-based: when the points accidentally all end up having the same objective function value;
- 2: They can cause the method to run indefinitely:
    - coordinate-based: when the objective function has multiple global minima;
    - value-based: when the objective function is very sensitive to changes in the coordinates of a point;
    - both: when the points or values never get close enough together to reach the desired tolerance;

By constructing a stopping criterion with both a coordinate-based and a value-based component we can minimize the risk of the first problem. We will only stop the method only when both the coordinates and the values of the points are close together, which lowers the chances of a false trigger.

To deal with the second problem we add either an iteration-based or evaluation-based component to our stopping criterion. By design, each method usually has a certain maximum number of objective function evaluations that might be performed during an iteration. The number of evaluations and iterations are thus closely related and thus it does not really matter which one we use. Since it is a little easier to implement, we will use an iteration-based criterion. The precise criteria we use throughout this thesis are the following:

- Coordinate-based criterion: $\max_{i\in[2,k]} ||x^{(1)} - x^{(i)}||_\infty < \text{tol}_x$
- Value-based criterion: $|f(x^{(1)}) - f(x^{(k)})| < \text{tol}_{fx}$
- Iteration-based criterion: $\#iterations > 1000 * n$ (or $100 * n$)

Here k denotes the number of points, n denotes the dimension of the domain and the x's are ordered from the lowest function value to the highest, such that: $f(x^{(1)}) \leq f(x^{(2)}) \leq ... \leq f(x^{(k)})$.

2.3 Test functions

To assess the performance of derivative-free minimization methods, many different test objective functions have been designed. The characteristics of these test functions are such that it should be very hard to find a global minimum. This is for example done by having many local minima, by having a steep wall around the global minimum, or by having the global minimum in a very shallow valley. Because the global minima of these test functions are known, we can use these functions to test the accuracy of a minimization method. Three of the test functions used in this thesis are visualized in Figure 2. The full list of test functions, which were obtained from [12], can be found in the Appendix along with their MATLAB implementations.

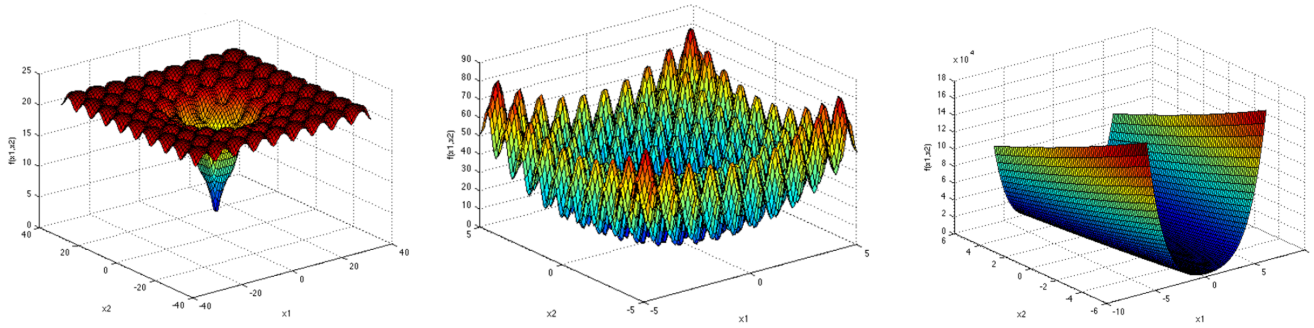


Figure 2: The Ackley, Rastrigin and Rosenbrock functions in two-dimensional form [12]. The function definitions can be found in A.1.1, A.1.15 and A.1.16 respectively.

3 Nelder-Mead method (NM)

One of the earliest derivative-free minimization methods was proposed by Nelder and Mead in 1965 [9]. This Nelder-Mead method (sometimes also called the Downhill Simplex method) is a direct, deterministic minimization method. It is very widely used and is even the underlying method of MATLAB's `fminsearch` function [3].

3.1 The algorithm

The NM method works by constructing a simplex, which is a shape of $n+1$ points in an n-dimensional space, and iteratively applying transformations to these points. The goal of these transformations is to 'move' the simplex in such a way that it ends up at a minimum of the objective value. At each iteration the method tries to replace the worst point (the one with the highest function value) with a better point by applying three types of transformations: *reflection*, *expansion* or *contraction*. In the, generally quite rare, case that none of these transformations result in a better point, a *shrink* is performed on all points except the best one. We will now discuss the method (based on [11], [13]) and the four transformations while using illustrations of a two-dimensional simplex (which is a triangle) as examples.

Initialization

We start out with some point in our domain $D \subseteq \mathbb{R}^n$, which we denote as $x^{(n+1)}$. Based on this point construct an initial simplex by defining n more points, which is usually done as follows:

$x^{(k)} = x^{(n+1)} + h_k e_k \quad$ for $k = 1, ..., n \quad$ where e_k is the k-th standard basis vector and $h_k \in \mathbb{R}$ is some scalar.

The most common choice for scalars is: $h_k := 1 \ \ \forall k = 1, ..., n$, but we will follow the example of MATLAB's `fminsearch` function and use the scalars that were allegedly suggested by L. Pfeffer [14].

$$h_k := \begin{cases} 0.05 * x_k^{(n+1)} & \text{if } x_k^{(n+1)} \neq 0, \text{ i.e. the } k\text{-th element of } x^{(n+1)} \text{ is non-zero} \\ 0.00025 & \text{otherwise} \end{cases}$$

After we have constructed the initial simplex we evaluate the objective function f at each of the $n+1$ points and sort the points based on their function values. This means that we renumber the points in such a way that $f(x^{(1)}) \leq f(x^{(2)}) \leq ... \leq f(x^{(n+1)})$.

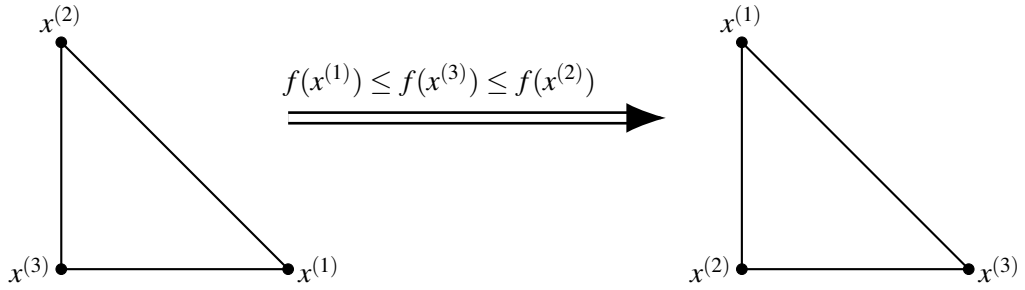


Figure 3: Renumbered initial simplex generated from the initial point $x^{(3)}$.

Reflection

The very first transformation we try, is reflecting the $x^{(n+1)}$ with respect to the so-called centroid which is defined as $x_c^{(n+1)} := \frac{1}{n} \sum_{i=1}^{n} x^{(i)}$. Reflection is performed by $x^{(r)} = (1+\alpha)x_c^{(n+1)} - \alpha\, x^{(n+1)}$, where $\alpha > 0$ is the reflection-coefficient. Depending on the objective function value of the reflection point $x^{(r)}$ we decide what to do next:

Case 1: $f(x^{(1)}) \leq f(x^{(r)}) < f(x^{(n)})$ $\Rightarrow$ reflection is successful, replace $x^{(n+1)}$ with $x^{(r)}$.

Case 2: $f(x^{(r)}) < f(x^{(1)})$ $\Rightarrow$ reflection is 'too' successful, perform expansion.

Case 3: $f(x^{(r)}) \geq f(x^{(n)})$ $\Rightarrow$ reflection is unsuccessful, perform contraction.

Expansion

When the reflection is 'too' successful, i.e. the reflection is better than all other points, we try to find an even better point by expanding the reflection point. Expansion is performed by $x^{(e)} = (1-\beta)x_c^{(n+1)} + \beta\, x^{(r)}$, where $\beta > 1$ is the expansion-coefficient. Depending on the objective function value of the expansion point $x^{(e)}$ we decide what the do next:

Case 1: $f(x^{(e)}) < f(x^{(r)})$ $\Rightarrow$ expansion is successful, replace $x^{(n+1)}$ with $x^{(e)}$.

Case 2: $f(x^{(e)}) \geq f(x^{(r)})$ $\Rightarrow$ expansion is unsuccessful, replace $x^{(n+1)}$ with $x^{(r)}$.

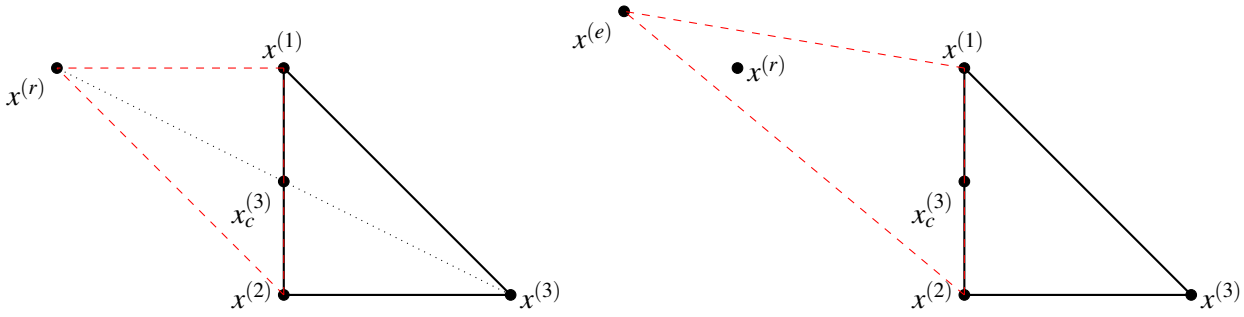


Figure 4: Reflection (left) and expansion (right).

Contraction

When the reflection is unsuccessful, i.e. the reflection is not better than the second-worst point, we try to find a better point by applying a contraction. Before we do this, we first check whether $f(x^{(r)}) \leq f(x^{(n+1)})$ and if this is indeed the case we replace $x^{(n+1)}$ by $x^{(r)}$. Contraction is performed by $x^{(c)} = (1-\gamma)x_c^{(n+1)} + \gamma\, x^{(n+1)}$, where $0 < \gamma < 1$ is the contraction-coefficient. Depending on the objective function value of the contraction point $x^{(c)}$ we decide what the do next:

Case 1: $f(x^{(c)}) \leq f(x^{(n+1)})$ $\Rightarrow$ contraction is successful, replace $x^{(n+1)}$ with $x^{(c)}$.

Case 2: $f(x^{(c)}) > f(x^{(n+1)})$ $\Rightarrow$ contraction is unsuccessful, perform shrink.

Shrink

When the contraction is unsuccessful there is a good chance that the minimum lies somewhere within the simplex and thus we will apply a shrink. The shrink is performed by $x_{\text{new}}^{(k)} = x^1 + \sigma(x^{(k)} - x^{(1)})$ for $k = 2, \ldots, n+1$, where $0 < \sigma < 1$ is the shrinkage-coefficient. For $k = 2, \ldots, n+1$ we replace $x^{(k)}$ with $x_{\text{new}}^{(k)}$.

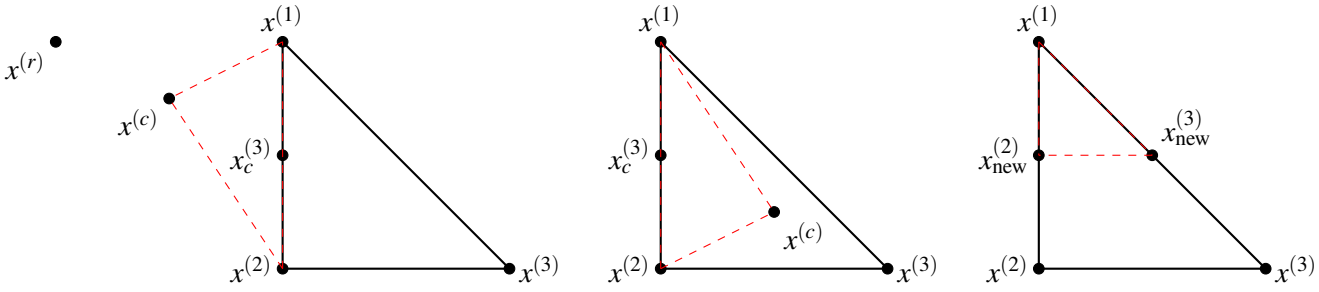


Figure 5: The two possible contractions (left and middle) and the shrink (right).

At the end of each iteration, the points are renumbered based on their objective function value, the stopping criterion is checked, and then a new iteration is started. Figure 6 contains a diagram of the full NM method and the MATLAB implementation can be found in the Appendix (A.2.1).

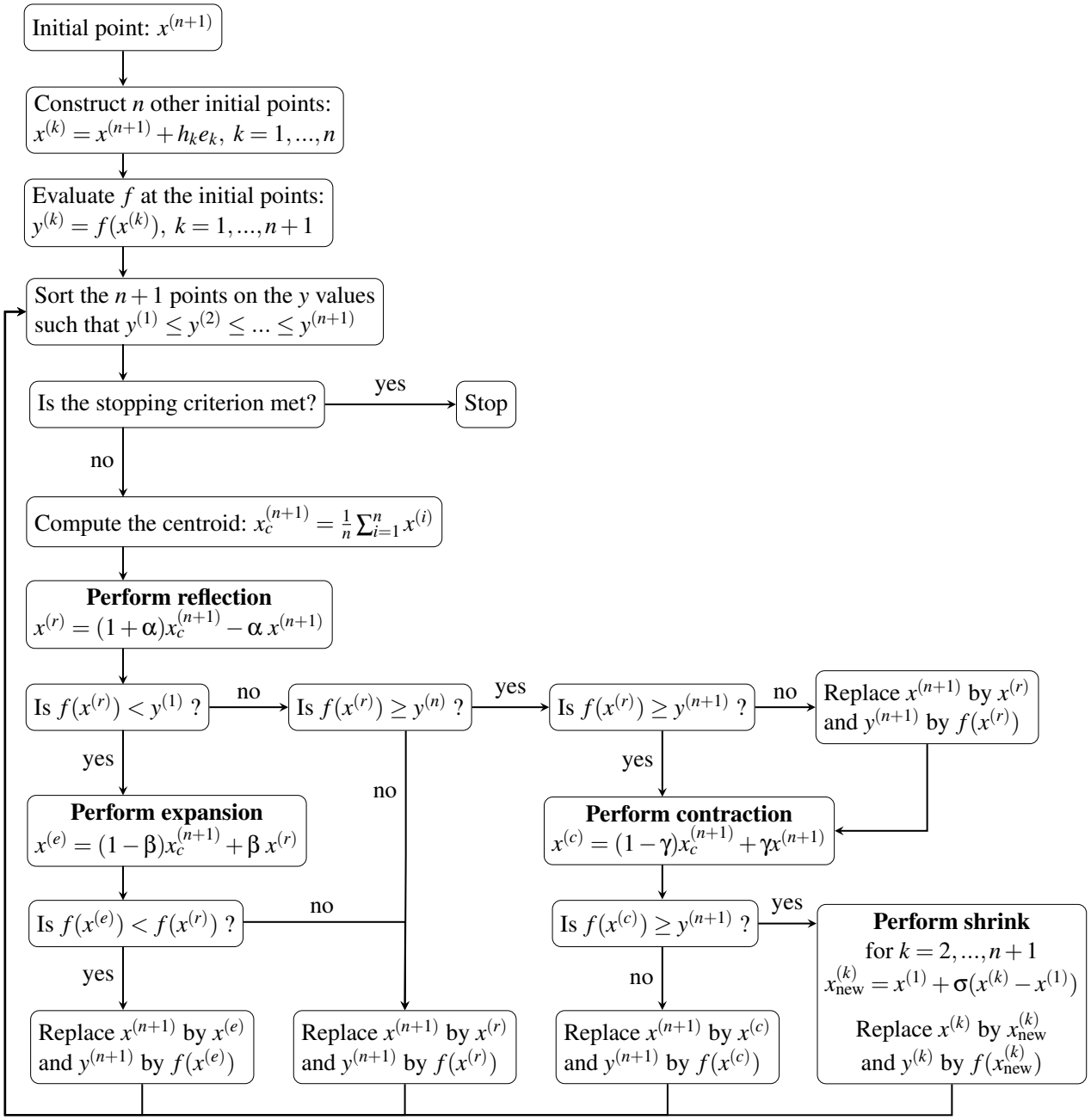


Figure 6: The NM algorithm (diagram inspired by [9].)

3.2 Parameter selection

Nelder and Mead [9] suggested using the parameter values: $\alpha = 1,\ \beta = 2,\ \gamma = 0.5$ and $\sigma = 0.5$. These values are still commonly used and hence we stick to them as well throughout this thesis.

3.3 Performance

The tests performed in the previous Bachelor's theses [4]–[6] all had similar results on the performance of the NM method. It turns out to be very good at finding a minimum and it does so using relatively few objective function evaluations, but it only finds the global minimum when the initial point is not too far away. Other literature, such as [2], also supports these findings. Therefore we can expect the NM method to be computationally cheap but also to be inaccurate for difficult objective functions, because of its tendency to find local minima instead of a global minimum.

4 Particle Swarm Optimization (PSO)

Another very popular derivative-free minimization method was proposed by Kennedy and Eberhart in 1995 [10] and the typically used, slightly altered version was proposed by Shi and Eberhart in 1998 [15]. The method is named Particle Swarm Optimization and it is a direct, stochastic minimization method. Over the years it has enjoyed a lot of interest and this has led to many suggested improvements and applications to various types of problems [16].

4.1 The algorithm

The PSO method works by constructing a group of points(/particles), and letting it move around based on rules which stem from simulating social behavior. These cause the points to behave as though they are part of a swarm, like a flock of birds or a school of fish. At each iteration the points decided where to move based on their best previous position and the position of the best point in their neighborhood. In this case 'best' means: with the lowest objective function value. The expected behavior is that after a while all the points end up in the same place, which should be a minimum of the objective value. We will now discuss the method (based on [15]–[17]) for a swarm of p points.

Initialization

We start out by generating our initial swarm of p points. Several suggestions have been made on how to do this, but the most common way is to generate the points randomly [18]. For our initial swarm we thus randomly pick p points within our domain $D \subseteq \mathbb{R}^n$. Each of these points $x^{(k)}$ (for $k = 1, ..., p$) is assigned some initial velocity $v^{(k)}$ as follows: $v^{(k)} = 0.1x^{(k)}$. Next we evaluate the objective function f at each point and sort the points based on their function values such that: $f(x^{(1)}) \leq f(x^{(2)}) \leq ... \leq f(x^{(p)})$.

Velocity update

The first thing we do for each iteration is updating the velocities of the points. For each $k = 1, ..., p$, $v^{(k)}$ is replaced with $v^{(k)}_{\text{new}}$, which is constructed as follows:

$$v^{(k)}_{\text{new}} = w * v^{(k)} + c_1 R(p^{(k)} - x^{(k)}) + c_2 R(n^{(k)} - x^{(k)}).$$

Here R is a diagonal matrix where each diagonal element is a uniform random number in $[0, 1]$, $p^{(k)}$ is the best previous position of the point $x^{(k)}$ and $n^{(k)}$ is the best point in the neighborhood of $x^{(k)}$. The parameter $w \geq 0$ is the inertia weight, $c_1 \geq 0$ is the cognitive learning factor and $c_2 \geq 0$ is the social learning factor and they can be chosen to be constant or to vary for each iteration. These three parameters largely determine the behavior of the swarm: a larger w makes the points move further apart and c_1, c_2 decide how much to move in the direction of a point's previous best position and the neighborhood's best position.

Position update

Now we update the positions of the points based on these new velocities. For each $k = 1, ..., p$, $x^{(k)}$ is replaced with $x^{(k)}_{\text{new}}$, which is constructed as follows:

$$x^{(k)}_{\text{new}} = x^{(k)} + v^{(k)}.$$

Next the objective function is evaluated at each of these new positions and the $p^{(k)}$'s and $n^{(k)}$'s are updated accordingly:

$$\text{If } f(x^{(k)}) < f(p^{(k)}), \text{ then } p^{(k)} \text{ is replaced by } x^{(k)}.$$

$$n^{(k)} = x^{(m)} \text{ where } f(x^m) = \max_{x \in S_k}(f(x)) \text{ with } S_k \text{ the neighborhood of } x^{(k)}.$$

At the end of each iteration, the points are renumbered based on their objective function value, the stopping criterion is checked, and a new iteration is started. Figure 7 contains a diagram of the full PSO method and the MATLAB implementation can be found in the Appendix (A.2.2).

4.2 Parameter selection

A lot of research has been done on finding increasingly well-performing parameter choices for the PSO method. The works by Shami et al. [19] and Wang et al. [18] provide good overviews of the numerous suggestions that have been made. The following are some of the most commonly used parameter choices, where *it* is the current iteration, it_{max} the maximum number of iterations and R is a uniformly random number in $[0,1]$.

Swarm size p

The swarm size is typically set to be some value in $[20,50]$ as described in [19]. The larger the swarm size, the more accurate the PSO method becomes, but this also increases the computational cost. Since we do not know in advance how hard it will be to minimize each objective function we use $p = 35$. However, when this fails to accurately find the global minimum we could still increase it.

Inertia weight w

- $0.5 + \frac{R}{2}$ as proposed in [20].
- $0.4 + 0.5\frac{it_{max}-it}{it_{max}}$ as proposed in [21].
- $0.4 + 0.55 * \exp(-\frac{8*it}{it_{max}})$ as proposed in [22].

Learning factors $(c_1,\ c_2)$

- $(1.49445, 1.49445)$ as described in [18].
- $(2.8, 1.3)$ as proposed in [23].
- $\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$ as proposed in [24].

Neighborhood

- Star topology as described in [19]:
  every point's neighborhood is the entire swarm.
- Von Neumann topology as described in [19]:
  the points are ordered in a rectangle and every point's neighborhood are its initial four neighbours, with wrapping around the edges.

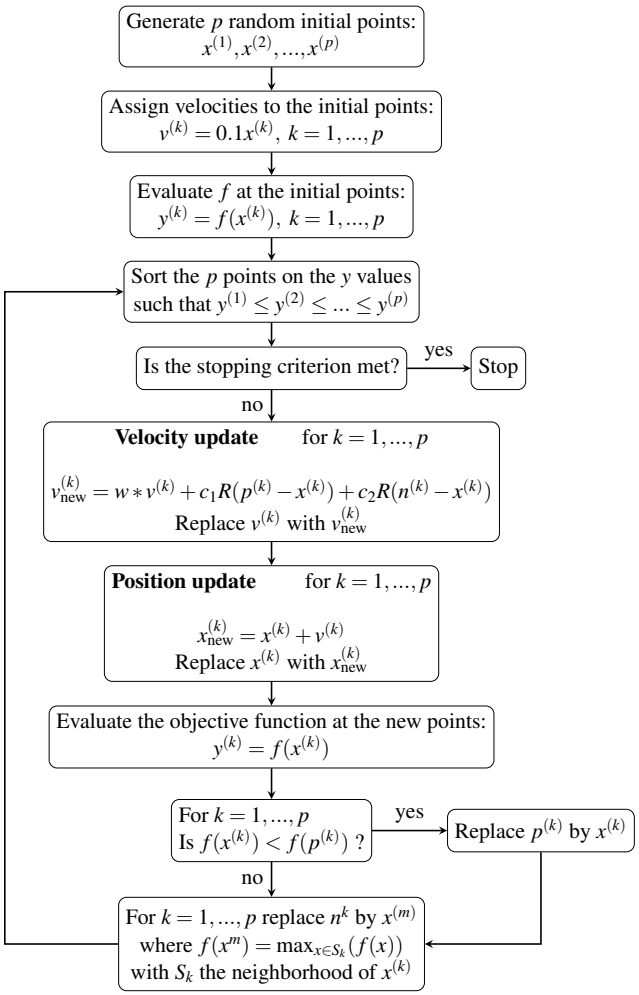


Figure 7: The PSO algorithm.

The Von Neumann topology is generally considered to perform better than the star topology [16] and therefore we choose to use only this neighborhood.

To determine which of these parameter choices we should use, every possible combination of these choices were tested on the same set of 40 test functions that will be used in Section 6.1 (see Table 4). For each configuration, every test function was solved 100 times to a tolerance of 1e-4 and afterward the success rate was determined by checking whether the minimum value that was found was within a margin of 0.0001 of the known global minimum. Table 1 contains the average success rate over all 40 functions of each configuration as well as the average number of objective function evaluations per run.

The stopping criterion was initially evaluated based on all p points, but evaluating the stopping criterion with only the $n+1$ best points was also tested. Lowering the number of points that are used for the stopping criterion increases the risk of terminating the method too early (hence decreasing accuracy) but it also causes the method to need less objective function evaluations (hence decreasing the computational cost). For the maximum number of iterations (it_{max}) both $1000 * n$ and $100 * n$ were tested.

$it_{max} = 1000 * n$		p		$n+1$	
learning factors $(c_1,\ c_2)$	**inertia weight** w	**Success**	**Evaluations**	**Success**	**Evaluations**
(1.49445, 1.49445)	$0.5 + \frac{R}{2}$	85 %	78461	85 %	63654
(1.49445, 1.49445)	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	90 %	88949	90 %	74394
(1.49445, 1.49445)	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	90 %	69955	90 %	56634
(2.8, 1.3)	$0.5 + \frac{R}{2}$	64 %	141820	64 %	123066
(2.8, 1.3)	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	87 %	117178	87 %	103229
(2.8, 1.3)	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	88 %	94181	88 %	76880
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.5 + \frac{R}{2}$	86 %	81051	86 %	63411
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	90 %	90175	90 %	76331
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	91 %	75046	91 %	57052

$it_{max} = 100 * n$		p		$n+1$	
learning factors $(c_1,\ c_2)$	**inertia weight** w	**Success**	**Evaluations**	**Success**	**Evaluations**
(1.49445, 1.49445)	$0.5 + \frac{R}{2}$	65 %	12960	66 %	11730
(1.49445, 1.49445)	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	79 %	12178	79 %	11193
(1.49445, 1.49445)	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	81 %	10141	80 %	8837
(2.8, 1.3)	$0.5 + \frac{R}{2}$	33 %	14245	34 %	14238
(2.8, 1.3)	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	65 %	14142	65 %	13472
(2.8, 1.3)	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	74 %	13290	75 %	11708
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.5 + \frac{R}{2}$	66 %	12710	67 %	11284
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.4 + 0.5\frac{it_{max}-it}{it_{max}}$	77 %	12122	77 %	11225
$\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$	$0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$	81 %	10133	81 %	8516

Table 1: Performance of different PSO parameter choices on the test functions (as indicated in Table 4).

Based on these results we make the following observations:

1. Using only the best $n+1$ points for the stopping criterion instead of all p points has no effect on the accuracy of the method. It does however considerably decrease the number of objective function evaluations that are needed.
2. Lowering it_{max} from $1000 * n$ to $100 * n$ significantly decreases the accuracy, but it also drastically decreases the number of objective evaluations.

3. $w = 0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$ always outperforms the other inertia weight options, both in terms of accuracy and computational cost.
4. $(c_1, c_2) = (2.8, 1.3)$ always performs worse than the other learning coefficients options, both in terms of accuracy and computational cost.

Unfortunately, but not unexpectedly, the most accurate configuration is not also the least computationally expensive. This means that we are going to have to make some compromise. Because the test functions were designed to be very hard for the minimization method, we should not necessarily go for the most accurate configuration. Instead, we require that the method has at least an 80% success rate and out of all the remaining options we choose the one with the fewest function evaluations. This choice leaves us with two, similarly well-performing, possible configuration:

- $(c_1, c_2) = (1.49445, 1.49445)$, $w = 0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$, $it_{max} = 100 * n$, and using only the $n+1$ best points for the stopping criterion.

- $\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$, $w = 0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$, $it_{max} = 100 * n$, and using only the $n+1$ best points for the stopping criterion.

We choose to use the second configuration throughout the rest of this thesis, because it performed ever so slightly better, both in terms of accuracy and computational cost.

4.3 Performance

Maquelin [5] found the PSO method to be very computationally expensive but also very accurate. These findings are also supported by the tests performed in [2]. It appears that the greater number of points and their relatively random movements through space, cause the PSO method be very good at finding a global minimum, but at the cost of many objective function evaluations.

5 Hybrid method (NM-PSO)

As we have seen in the previous chapters, the NM method is computationally cheap but not very good at finding the global minimum, while the PSO method is very good at finding the global minimum but computationally expensive. In 2004, Fan, Liang and Zahara [8] proposed to combine the NM and PSO methods into a new, hybrid method. This NM-PSO method, obviously, hopes to preserve the positive qualities of NM and PSO, while reducing the negative ones.

5.1 The algorithm

The NM-PSO method works by constructing a group of $p > n+1$ points in the domain $D \subseteq \mathbb{R}^n$. At each iteration the best $n+1$ points are updated using m iterations of the NM method. The other points are updated using one iteration of the PSO method. We will now discuss the method (based on [8]) with m NM iterations per NM-PSO iteration for a group of $p > n+1$ points.

Initialization

We start out by generating our initial p points. Just as in the case of PSO this is done by randomly generating p points in the domain Ω. Each of the points $x^{(k)}$, $k = 1,...,p$ is assigned some initial velocity as follows: $v^{(k)} = 0.1x^{(k)}$. Next we evaluate the objective function f at each point and sort the points based on their function values such that $f(x^{(1)}) \leq f(x^{(2)}) \leq ... \leq f(x^{(p)})$.

NM iterations

First we perform m iterations of the NM on the $n+1$ best points, which are $x^{(1)},...,x^{(n+1)}$. The original paper [8] always sets $m = 1$ and appears to disregard shrinks, as it states that only the $(n+1)$-th best particle will be updated by the NM iteration.

PSO iteration

Next we apply one PSO iteration to the remaining $p-(n+1)$ particles. This means that their velocities are updated first and after that their positions and function values.

At the end of each NM-PSO iteration, the best previous positions $(p^{(k)})$ of all p points are updated and the new best points in each neighborhood $(n^{(k)})$ are selected. Then the points are renumbered based on their objective function values, the stopping criterion is checked, and a new iteration is started. Figure 8 contains a diagram of the full NM-PSO method and the MATLAB implementation can be found in the Appendix (A.2.3).

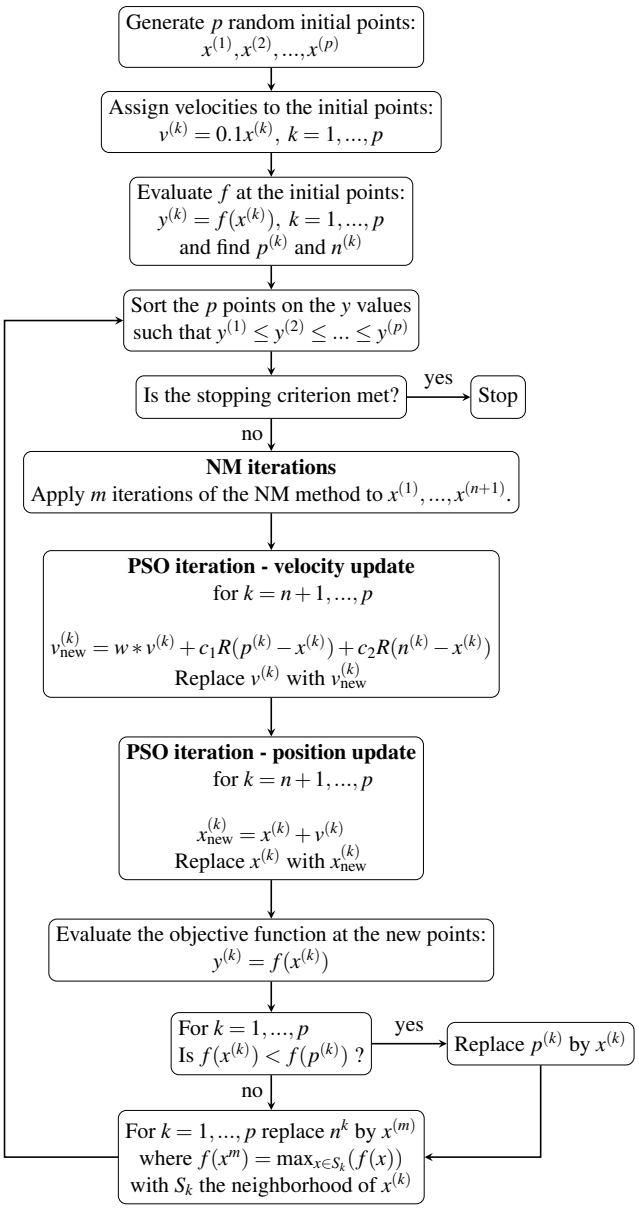


Figure 8: The NM-PSO algorithm.

5.2 Parameter selection

There are a lot of parameter choices we can make when configuring the (M-)NM-PSO method. For the NM and most of the PSO parameters we choose to use the same values as we decided to use for the NM and PSO methods, i.e. $\alpha = 1$, $\beta = 2$, $\gamma = 0.5$, $\sigma = 0.5$, $\begin{cases} c_1 = -2\frac{it}{it_{max}} + 2.5 \\ c_2 = 2\frac{it}{it_{max}} + 0.5 \end{cases}$ and $w = 0.4 + 0.55\exp(-\frac{8*it}{it_{max}})$.

This still leaves us with the following parameters still left to choose:

- the swarm size p,
- how often we want to apply the NM iteration per NM-PSO iterations (m),
- whether we allow all worst n points to shrink during the NM iteration or only the worst point, and
- whether we use all p points to evaluate the stopping criterion or only the $n+1$ best points.

To determine which parameter choices we should use, every possible combination of these choices were tested on the same set of 40 test functions that will be used in Section 6.1 (see Table 4). For each configuration, every test function was solved 100 times to a tolerance of 1e-4 and afterward the success rate was determined by checking whether the minimum value that was found was within a margin of 0.0001 of the known global minimum. For now, the swarm size is set at 35. Table 2 contains the average success rate over all 40 functions of each configuration as well as the average number of objective function evaluations per run.

In 2007, Fan and Zahara [25] proposed a different NM-PSO method, with the following changes:

- The swarm size is set to $3*n+1$, m is set to 1, and only the $n+1$ best points are used for the stopping criterion.
- The neighborhoods change every iteration such that the $2n$ worst points are divided into n neighborhoods of 2 points based on the current function value of the points.
- The past best position of each point $p^{(k)}$ is replaced with the current global best point.
- Each iteration a mutation heuristic is applied to the global best point. This means that five random new points near the global best point are tested and if a better point is found, it replaces the global best point.
- Each iteration a mutation heuristic is applied to the global best point. This means that five random new points near the global best point are tested and if a better point is found, it replaces the global best point.
- If the expansion during the NM-iterations is successful, a second reflection is attempted.

The performance of this altered NM-PSO method is also included in Table 2.

$it_{max} = 1000*n$		p		$n+1$	
m	**shrink**	**Success**	**Evaluations**	**Success**	**Evaluations**
1	yes	94 %	71908	77 %	21227
2	yes	94 %	79682	74 %	17264
3	yes	95 %	89527	72 %	15428
1	no	94 %	70971	83 %	23278
2	no	94 %	78953	74 %	19863
3	no	95 %	88175	72 %	16460
altered NM-PSO		-	-	52 %	40140

$it_{max} = 100*n$		p		$n+1$	
m	**shrink**	**Success**	**Evaluations**	**Success**	**Evaluations**
1	yes	88 %	9306	86 %	5889
2	yes	87 %	10101	79 %	5380
3	yes	88 %	10957	76 %	5015
1	no	89 %	9288	88 %	5973
2	no	89 %	10220	85 %	5511
3	no	87 %	11054	81 %	5240
altered NM-PSO		-	-	41 %	6402

Table 2: Performance of different NM-PSO parameter choices on the test functions.

Based on these results we make the following observations:

1. When using all p points for the stopping criterion:
   - increasing m hardly affects the accuracy while it significantly increases the computational cost,
   - using $it_{max} = 100 * n$ instead of $1000 * n$, slightly decreases the accuracy but also greatly decreases the computational cost, and
   - shrinking only the worst point hardly has any effect on the accuracy, but it does yield a slight improvement in terms of computational cost.
2. When using only the $n+1$ best points for the stopping criterion:
   - increasing m causes a decrease in accuracy while decreasing the computational cost,
   - using $it_{max} = 100 * n$ instead of $1000 * n$, (strangely) increases the accuracy but also greatly decreases the computational cost, and
   - shrinking only the worst point increases the accuracy, but also slightly increases the computational cost.
3. The altered NM-PSO method performs considerably worse on these test functions than the other suggested configurations.

Just a with the PSO method, we have to decide on some balance between accuracy and computational cost. In line with our previous decision, we require that the method has at least an 80% success rate. This means that we choose one of the configurations with $it_{max} = 100 * n$ while using only the $n+1$ best particles for the stopping criterion, because these methods require the least function evaluations, while also still reaching the threshold of 80% accuracy.

The obvious choice would then appear to be the configuration with $m = 3$ and no shrinks, but we can actually still lower the swarm size p (see Table 3).

$it_{max} = 100 * n$		$p = 20$		$p = 15$	
m	**shrink**	**Success**	**Evaluations**	**Success**	**Evaluations**
1	yes	80 %	3202	77 %	2205
2	yes	75 %	2996	72 %	2159
3	yes	72 %	2991	70 %	2173
1	no	85 %	3282	82 %	2318
2	no	79 %	3137	75 %	2267
3	no	75 %	3166	72 %	2314

Table 3: Performance of different swarm sizes p on the test functions.

Based on the results in Table 3 we thus settle on the configuration with $m = 1$, no shrinks and $p = 15$.

5.3 Performance

The tests performed in both papers ([8] and [25]) suggest that the NM-PSO method indeed performs as expected. In general, it is just as accurate as the PSO method while requiring significantly fewer objective function evaluations. This was however tested under different conditions (like the stopping criteria) and therefore the results may differ in our case.

6 Comparison

6.1 Test functions

To test the performance of the NM, PSO and NM-PSO methods (with the parameters selected in Sections 3.2, 4.2 and 5.2 respectively) we use them to solve a number of test functions. 20 different functions are used and some of them can be used with different dimensions. This results in 40 different test cases to which we apply the methods. The required tolerance for the stopping criterion is set to 1e-04, the maximum number of iterations is set to $100 * dim$ and each test case was given its corresponding boundaries. The methods are given random starting points within these boundaries and to minimize the effect of this randomness, which may cause one method to be more 'lucky' than the other, each test function is solved 100 times by each method. The results are then compared to the known global minimum and if the solution given by a method is within 0.0001 of the actual global minimum, the attempt is considered a success. The success rates are displayed in Table 4 along with the average number of objective function evaluations that each run of the method needed.

Results

		NM		PSO		NM-PSO	
Dim	**Function**	**Success**	**Evaluations**	**Success**	**Evaluations**	**Success**	**Evaluations**
2	Ackley	100 %	49	100 %	2751	100 %	681
2	Beale	0 %	381	96 %	7070	100 %	830
2	Bochachevksy	100 %	48	100 %	2934	97 %	810
2	Booth	0 %	395	100 %	7070	100 %	575
2	Branin	7 %	379	100 %	7070	100 %	735
2	De Jong 5	0 %	380	86 %	6606	82 %	1520
2	Dixon-Price	1 %	389	100 %	7070	100 %	627
2	Drop wave	100 %	41	87 %	5235	76 %	992
2	Easom	0 %	409	99 %	3861	86 %	1220
2	Griewank	100%	57	35 %	6988	59 %	1608
2	Levy	6%	398	100 %	2582	100 %	574
2	Matyas	100 %	42	100 %	2901	100 %	585
2	Perm	3%	339	93 %	7070	100 %	607
2	Rastrigin	100%	41	100 %	3330	90 %	833
2	Rosenbrock	0%	360	100 %	6908	100 %	1066
2	Schaffer	100 %	51	100 %	4187	100 %	1230
2	Sphere	100 %	41	100 %	2358	100 %	531
2	Three-hump camel	100 %	41	100 %	2394	100 %	530
2	Zakharov	100 %	42	100%	2437	100 %	547
4	Ackley	100 %	62	100 %	4394	100 %	1566
4	Colville	0 %	726	99 %	8911	58 %	3973
4	Dixon-Price	0 %	730	8 %	14070	80 %	3001
4	Griewank	100 %	70	36 %	13361	39 %	4956
4	Levy	0%	670	100 %	4240	100 %	1444
4	Perm	0 %	620	2 %	14070	56 %	3904
4	Powell	100 %	52	100%	13650	100 %	3624
4	Rastrigin	100 %	53	100 %	6752	87 %	2885
4	Rosenbrock	0 %	696	7 %	13910	22 %	4268
4	Sphere	100 %	55	100 %	3956	100 %	1299
4	Zakharov	100 %	56	100 %	4459	100 %	1405
8	Ackley	100 %	84	100 %	6995	100 %	2138
8	Dixon-Price	0 %	1379	0 %	28070	4 %	5752
8	Griewank	100 %	119	100 %	9891	58 %	6286
8	Levy	0 %	1340	100 %	6839	100 %	2053
8	Perm	0 %	1391	0 %	28070	6 %	7012
8	Powell	100 %	83	100 %	12482	91 %	5774
8	Rastrigin	100 %	69	91 %	12023	66 %	6139
8	Rosenbrock	0 %	1356	5 %	27060	19 %	5550
8	Sphere	100 %	79	100 %	6470	100 %	1865
8	Zakharov	100 %	81	100 %	10473	100 %	2343
	Average:	**55 %**	**341**	**81 %**	**8524**	**82 %**	**2333**

Table 4: Performance of the three methods on various test functions.

As can be seen by the averages, the methods indeed perform as we expected based on the tests performed by others.

The NM method is by far the fastest, but it is also the least accurate. It struggles especially with functions that have very flat valleys (such as Beale and Rosenbrock). This might perhaps be overcome partially by setting a stricter tolerance for the stopping criterion, which will make the NM method perform more iterations. Interestingly, the NM method did not appear to struggle too much with functions that have many local minima (such as Rastrigin and Griewank). This might be because the search spaces were restricted and this caused the starting points for NM to be close enough to the global minimum.

PSO and NM-PSO are similarly accurate, but NM-PSO requires considerably fewer function evaluations for each test function. Given the ranges of the maximum number of evaluations per method per dimension (see Table 5), it is clear that NM and NM-PSO rarely need the maximum number of iterations, while the PSO method regularly reaches it. This suggests that NM characteristics indeed help the NM-PSO method to reach a minimum faster. Where the PSO might by terminated by the maximum number of iterations before reaching a minimum, the NM-PSO may not always suffer the same fate. This might explain some of the instances where the NM-PSO method is more accurate than the PSO method, such as the 2-dimensional Beale and Perm functions.

	NM	**PSO**	**NM-PSO**
Dim = n *(best case)*	$(n+1)+(it_{max}+1)$	$35+(it_{max}+1)(35)$	$15+(it_{max}+1)(15-n)$
Dim = n *(worst case)*	$(n+1)+(it_{max}+1)(2+n)$	$35+(it_{max}+1)(35)$	$15+(it_{max}+1)(17-n)$
Dim = 2 $it_{max}=200$	204 - 807	7070	2628 - 3030
Dim = 4 $it_{max}=400$	406 - 2411	14070	4426 - 5228
Dim = 8 $it_{max}=800$	2610 - 8019	28070	5622 - 7224

Table 5: The maximum number of objective function evaluations for NM, PSO and NM-PSO (with the parameters selected in Sections 3.2, 4.2 and 5.2 respectively.)

6.2 SOR

Successive over-relaxation (SOR) [11] is a method that is used to numerically solve matrix equations of the form $Ax = b$ where $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ and x is the unknown vector. SOR starts out with an initial guess $x^{(1)}$ and at each iteration it produces a new guess $x^{(k)}$ as follows:

$$x_i^{(k)} = \omega \bar{x}_i^{(k)} + (1-\omega)x_i^{(k-1)} \qquad \text{for } i = 1,...,n.$$

Here $\omega > 0$ is a relaxation parameter which determines how much of the previous guess, $x^{(k-1)}$, is kept and how much of the so-called Gauss-Seidel iterate, $\bar{x}^{(k)}$, is added. The Gauss-Seidel iterate is defined as:

$$\bar{x}_i^{(k)} := \frac{(b_i - \sum_{j<i} a_{ij}x_j^{(k)} - \sum_{j>i} a_{ij}x_j^{(k-1)})}{a_{ii}}$$

Provided that A is symmetric and positive definite, the SOR method is guaranteed to converge for any $\omega \in (0,2)$, but the choice of ω greatly impacts the speed at which this happens [26]. To find the best choice for ω for a given problem $Ax = b$, we thus want to minimize the function:

$$f(\omega) = \text{time}(SOR(\omega, A, b, tol))$$

Here 'time()' returns the time that it took to complete the process of using SOR with ω to solve the system $Ax = b$ up to a certain tolerance tol. Clearly, there is no way to take derivatives of f and thus we have to resort to derivative-free minimization methods to find an optimal ω.

SOR can also be implemented with two relaxation variables:

$$x_i^{(k)} = \begin{cases} \omega_1 \bar{x}_i^{(k)} + (1-\omega_1)x_i^{(k-1)} & \text{for all odd } i \in [1,n] \\ \omega_2 \bar{x}_i^{(k)} + (1-\omega_2)x_i^{(k-1)} & \text{for all even } i \in [1,n] \end{cases}$$

or even more generally for some positive integer $m \leq n$:

$$x_i^{(k)} = \begin{cases} \omega_1 \bar{x}_i^{(k)} + (1-\omega_1)x_i^{(k-1)} & \text{for all } i \in [1,n] \text{ such that } i = 1 \mod m \\ \omega_2 \bar{x}_i^{(k)} + (1-\omega_2)x_i^{(k-1)} & \text{for all } i \in [1,n] \text{ such that } i = 2 \mod m \\ ... & \\ \omega_m \bar{x}_i^{(k)} + (1-\omega_m)x_i^{(k-1)} & \text{for all } i \in [1,n] \text{ such that } i = 0 \mod m \end{cases}$$

The corresponding functions that we would like to minimize are:

$$f(\omega_1, \omega_2) = \text{time}(SOR(\omega_1, \omega_2, A, b, tol))$$

$$f(\omega_1, \omega_2, ..., \omega_m) = \text{time}(SOR(\omega_1, \omega_2, ..., \omega_m, A, b, tol))$$

The use of more relaxation parameters makes it more difficulty to find the optimal ω's, but these optimal values generally cause SOR to be faster compared to when fewer relaxation parameters are used.

To compare the performance of our derivative-free minimization methods we will now use them to optimize SOR for a linear system of the form $Ax = b$ with either 1, 2 or 5 relaxation parameters. We stop the SOR method if the norm of the residue, $||Ax^{(k)} - b||$, is smaller that 0.001 or if the number of iterations exceeds 10^5.

ODE

The linear system we consider is the same as the one used in [5] and [6], which is derived from the following ordinary differential equation (ODE):

$$y''(x) + \alpha y(x) = x \qquad y(0) = y(1) = 1 \qquad \text{with } \alpha = 10^{-4} \text{ and } x \in [0,1]$$

We discretize the interval $[0,1]$ with a step size of h, which means that there will be $\frac{1}{h} + 1$ points with distances of size h between them. These points are denoted by x_i for $i = 1, ..., \frac{1}{h} + 1$ and the approximate solution at each

x_i is denoted by y_i. This means that $y_1 = y(x_1) = y(0) = 1$, $y_{\frac{1}{h}+1} = y(x_{\frac{1}{h}+1}) = y(1) = 1$ and the ODE for all $i = 2, ..., \frac{1}{h}$ is given by:

$$y''(x_i) + \alpha y_i = x_i \qquad \text{with } \alpha = 10^{-4}$$

Using the finite difference method we have the following approximation of the second derivative:

$$y''(x_i) = \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1})$$

The discretized ODE for all $i = 2, ..., \frac{1}{h}$ then becomes:

$$\frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) + \alpha y_i = x_i \qquad \text{with } \alpha = 10^{-4}$$

or equivalently:

$$\frac{1}{h^2}y_{i+1} + (\alpha - \frac{2}{h^2})y_i + \frac{1}{h^2}y_{i-1} = x_i \qquad \text{with } \alpha = 10^{-4}$$

This entire ODE can thus be written in matrix form as follows:

$$\begin{bmatrix} 1 & 0 & & & & & \\ \frac{1}{h^2} & \alpha - \frac{2}{h^2} & \frac{1}{h^2} & & & 0 & \\ & \frac{1}{h^2} & \alpha - \frac{2}{h^2} & \frac{1}{h^2} & & & \\ & & \ddots & \ddots & \ddots & & \\ & & & \frac{1}{h^2} & \alpha - \frac{2}{h^2} & \frac{1}{h^2} & \\ & 0 & & & \frac{1}{h^2} & \alpha - \frac{2}{h^2} & \frac{1}{h^2} \\ & & & & & 0 & 1 \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_{\frac{1}{h}-1} \\ y_{\frac{1}{h}} \\ y_{\frac{1}{h}+1} \end{bmatrix} = \begin{bmatrix} 1 \\ x_2 \\ x_3 \\ \vdots \\ x_{\frac{1}{h}-1} \\ x_{\frac{1}{h}} \\ 1 \end{bmatrix}$$

For any given h, all the x_i's are known and thus we have a linear system $Ax = b$ which we want to solve. The smaller we choose h, the harder it will be for SOR to solve this system. This in turn means that we can expect the choice of the relaxation parameters to have increasingly more impact on the speed of SOR as we decrease h and thereby increase the dimension of the problem.

Results

Table 6 shows the results of running each of the three minimization methods for different combinations of the number of relaxation parameters ω and step size h. Each method is run three times with the maximum number of iterations set to $100 * dim$, the tolerance is set to 0.001 and the search domain for each ω is $(0,5)$. The best solutions that each method finds are displayed in the table. In general, these results confirm our previous findings, but there are also some interesting major deviations.

For 2 ω's, the NM method always failed to optimize SOR. This is probably because with an $\omega > 2$ SOR sometimes fails to converge. It is possible that the initial points for NM all caused an SOR failure which made the NM fail.

The NM method is typically the fastest, but on three occasions it is not. One possible explanation for this is that the randomly generated starting points of the other methods were simply very close to a global minimum, which made it possible to finish considerably faster than usual. Another possible explanation is that the NM method came across some very bad parameter choices that it had to evaluate. Where the fastest parameter choices take only a few milliseconds to run SOR, a bad parameter choice may need, say, a full 2 seconds. In this way the NM method may require less objective function evaluations than the other methods, but still take more time to complete.

The NM-PSO method generally found similar solutions to the PSO method, while requiring less time. However, as the linear systems became harder to solve (i.e. h became smaller) and SOR became harder to optimize (i.e. more ω's), this behavior changed. PSO suddenly needed way less time and at the same time it also found solutions which were worse than the ones found by NM-PSO. One possible explanation might be that the PSO method quickly came across local minima and got stuck there.

	NM			**PSO**			**NM-PSO**		
	ω	**Time** (SOR)	**Time** (NM)	ω	**Time** (SOR)	**Time** (PSO)	ω	**Time** (SOR)	**Time** (NM-PSO)
1 ω, $h = 0.04$	1.77	0.0002	1.6	1.77	0.0002	3.3	1.77	0.0002	0.4
1 ω, $h = 0.02$	1.87	0.0013	5.2	1.87	0.0013	17.2	1.87	0.0013	9.0
1 ω, $h = 0.01$	1.93	0.0095	18.5	1.93	0.0095	223.5	1.93	0.0095	43.5
2 ω's, $h = 0.04$	$\begin{pmatrix} - \\ - \end{pmatrix}$	failed	2.9	$\begin{pmatrix} 1.79 \\ 1.76 \end{pmatrix}$	0.0002	29.9	$\begin{pmatrix} 1.79 \\ 1.76 \end{pmatrix}$	0.0002	5.4
2 ω's, $h = 0.02$	$\begin{pmatrix} - \\ - \end{pmatrix}$	failed	8.9	$\begin{pmatrix} 1.90 \\ 1.90 \end{pmatrix}$	0.0018	31.5	$\begin{pmatrix} 1.87 \\ 1.87 \end{pmatrix}$	0.0014	36.5
2 ω's, $h = 0.01$	$\begin{pmatrix} - \\ - \end{pmatrix}$	failed	34.7	$\begin{pmatrix} 1.93 \\ 1.93 \end{pmatrix}$	0.0099	797.4	$\begin{pmatrix} 1.93 \\ 1.93 \end{pmatrix}$	0.0100	52.8
5 ω's, $h = 0.04$	$\begin{pmatrix} 1.90 \\ 2.19 \\ 2.30 \\ 1.08 \\ 0.84 \end{pmatrix}$	0.0010	9.3	$\begin{pmatrix} 1.72 \\ 1.74 \\ 1.73 \\ 1.94 \\ 1.75 \end{pmatrix}$	0.0002	24.7	$\begin{pmatrix} 1.72 \\ 1.85 \\ 2.04 \\ 1.66 \\ 1.66 \end{pmatrix}$	0.0002	79.3
5 ω's, $h = 0.02$	$\begin{pmatrix} 1.87 \\ 2.09 \\ 2.13 \\ 1.39 \\ 1.85 \end{pmatrix}$	0.0025	66.6	$\begin{pmatrix} 1.49 \\ 1.47 \\ 2.15 \\ 2.14 \\ 1.54 \end{pmatrix}$	0.0049	42.7	$\begin{pmatrix} 1.83 \\ 2.04 \\ 1.85 \\ 1.97 \\ 1.73 \end{pmatrix}$	0.0014	116.7
5 ω's, $h = 0.01$	$\begin{pmatrix} 1.99 \\ 1.69 \\ 0.30 \\ 2.05 \\ 1.76 \end{pmatrix}$	0.4685	872.1	$\begin{pmatrix} 1.40 \\ 2.31 \\ 1.08 \\ 2.09 \\ 1.33 \end{pmatrix}$	0.1232	73.9	$\begin{pmatrix} 1.91 \\ 2.02 \\ 1.97 \\ 1.91 \\ 1.87 \end{pmatrix}$	0.0103	1586.2

Table 6: SOR applied to the ODE.

6.3 Preconditioned GMRES

The Generalized Minimal Residual method (GMRES) [26] is a method that is used to numerically solve matrix equations of the form $Ax = b$ where $A \in \mathbb{R}^{n\times n}$, $b \in \mathbb{R}^n$ and x is the unknown vector. GMRES is a so-called Krylov method, which means that it utilizes the Krylov subspaces K_n.

$$K_n := \text{span}\{r^{(0)}, A^1 r^{(0)}, A^2 r^{(0)}, ..., A^{n-1} r^{(0)}\}, \qquad \text{where } r^{(0)} := Ax^{(0)} - b$$

A basis $(v^{(1)}, v^{(2)}, ..., v^{(n)})$ of the Krylov subspaces is formed from an intial point $x^{(0)}$ as follows:

$$v^{(1)} = \frac{r^{(0)}}{||^{(0)}||_2}, \text{ and}$$

$$v^{(i)} = \frac{w^{(i-1)}}{||w^{(i-1)}||} \text{ for } i = 2, ..., n, \qquad \text{where } w^{(i)} = Av^{(i)} - \sum_{k=1}^{i} (w^{(i)}, v^{(i)}) v^{(k)}.$$

At every iteration k, GMRES generates a new estimation of the form:

$$x^{(k)} = x^{(0)} + y_1 v^{(1)} + ... + y_n v^{(n)}.$$

Here the coefficients y_i are chosen in such a way that the norm of the residual, $||b - Ax^{(n)}||$, is a small as possible. The precise algorithm that determines these y_i's is beyond the scope of this thesis, but it can be found in [11] and [26].

As we can see, all the previous basis vectors $v^{(i)}$ are needed for every new iteration. At most, the GMRES method requires as many iterations as the dimension of matrix A and at this final iteration it is guaranteed to converge [11]. This means that we need to store a lot of $v^{(i)}$'s throughout the GMRES run for large matrices A.

To prevent the GMRES method from stalling because of a lack of storage space, the method can be restarted after a certain number of iterations r. At such a restart the current estimation $x^{(r)}$ is used as the initial point $x^{(0)}$ for a new instance of the GMRES method. Because of this restarting the method is no longer guaranteed to converge, but it does require a lot less storage and it can improve the computation time [26]. If r is chosen too small then GMRES may fail to converge, but if it is chosen too large the computational cost and storage requirements may be unnecessarily large. There are no general rules regarding which r is best in which scenario, but we can try to use our minimization methods to find the optimal r in our test case.

For the tests we make use of MATLAB's `gmres` function, which quite an efficient implementation of the GMRES algorithm.

SSOR-preconditioner

An equivalent way of solving $Ax = b$ is to solve the linear system $PAx = Pb$ where P is some matrix called a preconditioner. If P is chosen properly, then this preconditioned linear system might be a lot easier for GMRES to solve that the original system, but the solution x will clearly be the same.

To improve the performance of the GMRES method, we combine it with the so-called SSOR-preconditioner [27], which is defined as:

$$M_{SSOR} := \frac{\omega}{2-\omega}(\frac{1}{\omega}D + L)D^{-1}(\frac{1}{\omega}D + R)$$

Here D is a diagonal matrix consisting of the diagonal entries of A, L a strictly lower triangular matrix consisting of the strictly lower triangular entries of A, R a strictly upper triangular matrix consisting of the strictly upper triangular entries of A and $\omega > 0$ some scalar. We can use our optimization methods to find the optimal ω for which GMRES has the lowest run-time.

When we use preconditioned GMRES, this SSOR-preconditioner is used to solve the system $M_{SSOR}^{-1}Ax = M_{SSOR}^{-1}b$ (so $P = M_{SSOR}^{-1}$). A result of using a preconditioner is that at some point in the GMRES iteration k

we need to solve $M_{SSOR}r^{(k)} = r^{(0)}$ to find $r^{(k)}$. This requires a very computationally expensive inverse operation, but it can be overcome by splitting the SSOR-preconditioner in two triangular factors M_1, M_2 (such that $M_{SSOR} = M_1 * M_2$).

$$M_1 := \frac{\omega}{2-\omega}(\frac{1}{\omega}D + L), \qquad \text{which is a lower triangular matrix.}$$

$$M_2 := D^{-1}(\frac{1}{\omega}D + R), \qquad \text{which is an upper triangular matrix.}$$

In this way, solving $M_{SSOR}r^{(k)} = r^{(0)}$ turns into first solving $M_1 z = r^{(0)}$ for z and then solving $M_2 r^{(k)} = z$. This splitting allows us to use the triangular properties of M_1 and M_2 for a considerably cheaper computation.
To see why an equation of the form $M_1 y = c$ can be solved so cheaply we take a look at its matrix form:

$$\begin{bmatrix} m_{11} & & & & \\ m_{21} & m_{22} & & & \\ m_{31} & m_{32} & m_{33} & & \\ \vdots & \vdots & \vdots & \ddots & \\ m_{n1} & m_{n2} & m_{n3} & \cdots & m_{nn} \end{bmatrix} \begin{bmatrix} y_1 \\ y_2 \\ y_3 \\ \vdots \\ y_n \end{bmatrix} = \begin{bmatrix} c_1 \\ c_2 \\ c_3 \\ \vdots \\ c_n \end{bmatrix}$$

y_1 can be readily computed since $m_{11} * y_1 = c_1$, but this in turn means that y_2 can also be computed since $m_{21} * y_1 + m_{22} * y_2 = c_2$. This continues all the way down and eventually we have computed y without needing to invert the matrix M_1. This works similarly for the upper triangular matrix M_2, but in that case we start at the bottom and work our way upwards.

The MATLAB function `gmres` allows use to input M_1 and M_2 instead of M_{SSOR} and it will use this fast computation instead of the computationally expensive inverse operation.

PDE

The linear system we consider is similar to the one used in [6], which is derived from the following partial differential equation (PDE):

$$u_{xx} + u_{yy} = x + y \qquad u(0,j) = u(1,j) = u(i,0) = u(i,1) = 0 \qquad \text{with } (x,y) \in [0,1] \times [0,1]$$

We discretize the space $[0,1] \times [0,1]$ with a step size of h in both the x and y direction. This means that there will be $(\frac{1}{h}+1)^2$ points with distances of size h between them. These points are denoted by $p_{(i,j)}$ for $i, j = 1, ..., \frac{1}{h}+1$ with x_i and y_j the corresponding x and y coordinates, and the approximate solution of at each $p_{(i,j)}$ is denoted by $u_{(i,j)}$. This means that $u_{(1,j)} = u(x_1, y_j) = u(0, ...) = 0$ and similarly $u_{(\frac{1}{h}+1,y)} = u_{(x,1)} = u_{(x,\frac{1}{h}+1)} = 0$. Also the PDE for all $(i,j) \in [2, \frac{1}{h}] \times [2, \frac{1}{h}]$ is now given by:

$$u_{xx}(i,j) + u_{yy}(i,j) = x_i + y_j$$

Using the finite difference method we have the following approximation of the second order partial derivatives:

$$u_{xx}(i,j) \approx \frac{1}{h^2}(u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)})$$

$$u_{yy}(i,j) \approx \frac{1}{h^2}(u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)})$$

The discretized PDE for all $(i,j) \in [2, \frac{1}{h}] \times [2, \frac{1}{h}]$ then becomes:

$$\frac{1}{h^2}(u_{(i+1,j)} - 2u_{(i,j)} + u_{(i-1,j)}) + \frac{1}{h^2}(u_{(i,j+1)} - 2u_{(i,j)} + u_{(i,j-1)}) = x_i + y_i$$

or equivalently:

$$\frac{u_{(i-1,j)}}{h^2} + \frac{u_{(i,j-1)}}{h^2} + \frac{-4u_{(i,j)}}{h^2} + \frac{u_{(i,j+1)}}{h^2} + \frac{u_{(i+1,j)}}{h^2} = x_i + y_i$$

This entire PDE can thus be written in matrix form, where the empty spaces are all zeros, as follows:

$$
\begin{bmatrix}
1 & 0 & (\frac{1}{h}\rightarrow) & 0 \\
0 & 1 & 0 & & 0 \\
(\downarrow\frac{1}{h}) & \ddots & \ddots & \ddots & & \ddots \\
0 & & 0 & 1 & 0 & & 0 \\
 & \frac{1}{h^2} & & \frac{1}{h^2} & \frac{-4}{h^2} & \frac{1}{h^2} & & \frac{1}{h^2} \\
 & & \ddots & & \ddots & \ddots & \ddots & & \ddots \\
 & & & \frac{1}{h^2} & & \frac{1}{h^2} & \frac{-4}{h^2} & \frac{1}{h^2} & & \frac{1}{h^2} \\
 & & & & 0 & & 0 & 1 & 0 & & 0 \\
 & & & & & 0 & & 0 & 1 & 0 & & 0 \\
 & & & & & & \frac{1}{h^2} & & \frac{1}{h^2} & \frac{-4}{h^2} & \frac{1}{h^2} & & \frac{1}{h^2} \\
 & & & & & & & \ddots & & \ddots & \ddots & \ddots & & \ddots \\
 & & & & & & & & \ddots & & \ddots & \ddots & \ddots & & \ddots \\
 & & & & & & & & & \frac{1}{h^2} & & \frac{1}{h^2} & \frac{-4}{h^2} & \frac{1}{h^2} & & \frac{1}{h^2} \\
 & & & & & & & & & & 0 & & 0 & 1 & 0 & & 0 \\
 & & & & & & & & & & & \ddots & & \ddots & \ddots & \ddots \\
 & & & & & & & & & & & & 0 & & 0 & 1 & 0 \\
 & & & & & & & & & & & & & 0 & & 0 & 1
\end{bmatrix}
\begin{bmatrix}
u_{(1,1)} \\ u_{(1,2)} \\ \vdots \\ u_{(2,1)} \\ u_{(2,2)} \\ \vdots \\ u_{(2,\frac{1}{h}))} \\ u_{(2,\frac{1}{h}+1))} \\ u_{(3,1)} \\ u_{(3,2)} \\ \vdots \\ \vdots \\ u_{(\frac{1}{h},\frac{1}{h})} \\ u_{(\frac{1}{h},\frac{1}{h}+1)} \\ \vdots \\ u_{(\frac{1}{h}+1,\frac{1}{h})} \\ u_{(\frac{1}{h}+1,\frac{1}{h}+1)}
\end{bmatrix}
=
\begin{bmatrix}
0 \\ 0 \\ \vdots \\ 0 \\ x_2+y_2 \\ \vdots \\ x_2+y_{\frac{1}{h}} \\ 0 \\ 0 \\ x_3+y_2 \\ \vdots \\ \vdots \\ x_{\frac{1}{h}}+y_{\frac{1}{h}} \\ 0 \\ \vdots \\ 0 \\ 0
\end{bmatrix}
$$

For any given h, all the x_i's and y_i's are known and thus we have a linear system $Ax = b$ which we can solve using preconditioned GMRES.

The choice for the ω in preconditioned GMRES is typically not optimized since any choice within $(0,2)$ tends to give good results [28]. A common practice appears to be to just set ω equal to 1 [29]. It will thus be interesting to see, how much influence the choice of ω has in our test case and whether an optimal ω results in a significant speed-up.

Results

Table 7 shows the results of running each of the three minimization methods with different step size h, and with and without restarting GMRES. Each method is run three times with the maximum number of iterations set to $100 * dim$, the tolerance is set to 0.001 and the search domain for ω is $(0,2)$. The best solutions that each method found are displayed in the table. Apart from one instance, these results agree with our previous findings.
The NM-PSO always finds the same solution as the PSO method while generally requiring considerably less time to do so. The NM method is however still the fastest method, but contrary to our previous tests, it always finds similarly good solutions to that of PSO and NM-PSO. This is probably due to the optimization problem being relatively easy to solve, i.e. there are no flat valleys, no high walls, and not many local minima.

We also observe that the run-time of GMRES with the default ω can be shortened by about 50% when using the parameters that the minimization methods find.

	default	NM			PSO			NM-PSO		
h	**Time** (GMRES)	r, ω	**Time** (GMRES)	**Time** (NM)	r, ω	**Time** (GMRES)	**Time** (PSO)	r, ω	**Time** (GMRES)	**Time** (NM-PSO)
0.0625	0.0020	-, 1.99	0.0038	0.3	-, 1.74	0.0032	10.5	-, 1.73	0.0030	14.6
0.04	0.0153	-, 1.74	0.0100	0.9	-, 1.17	0.0100	68.4	-, 1.06	0.0100	48.8
0.02	0.2068	-, 0.001	0.0357	276.9	-, 1.97	0.0357	482.4	-, 1.39	0.0357	2359.7
0.1*	0.0017	86, 1.41	0.0005	0.9	12, 1.77	0.0005	17.7	15, 1.76	0.0005	7.9
0.0625*	0.0020	217, 1.51	0.0011	3.2	43, 1.76	0.0010	3639.0	15, 1.58	0.0010	88.5
0.05*	0.0043	375, 1.73	0.0028	5.1	24, 1.73	0.0027	9743.3	19, 1.74	0.0027	2844.7

Table 7: Preconditioned GMRES applied to the PDE. For rows with a * the restart iteration r was also optimized.

We have two options to make this optimization problem even harder. The first thing we could do, is making h even smaller. This however causes us to run into memory issues, since matrix A is already a $(\frac{1}{0.02}+1)^2 = 2601$ dimensional square matrix for $h = 0.02$. Instead, we decide to make the matrix A less regular (and a-symmetric) by disturbing its values. For each of the 5 diagonals, we randomly select 10% of the elements and replace them by a random value in the range $[-\frac{10}{h^2}, \frac{10}{h^2}]$. This should make it harder for GMRES to solve the linear system and in turn we expect the minimization methods to find the minimization problem harder to solve.

	default	NM			PSO			NM-PSO		
h	**Time** (GMRES)	r, ω	**Time** (GMRES)	**Time** (NM)	r, ω	**Time** (GMRES)	**Time** (PSO)	r, ω	**Time** (GMRES)	**Time** (NM-PSO)
0.0625	0.0021	-, 0.45	0.0011	0.2	-, 1.09	0.0009	5.4	-, 1.99	0.0009	5.8
0.04	0.0093	-, 1.81	0.0063	0.3	-, 1.59	0.0061	12.5	-, 1.17	0.0061	11.5
0.02	0.0813	-, 1.26	0.0720	33.4	-, 0.46	0.0714	808.1	-, 1.27	0.0713	887.3
0.1*	0.0017	69, 1.95	0.0004	4.6	26, 1.95	0.0004	2162.0	5, 1.96	0.0004	860.7
0.0625*	0.0021	85, 0.55	0.0007	4.5	1, 1.99	0.0007	2.9	1, 1.99	0.0007	0.1
0.05*	0.0045	31, 1.86	0.0021	18.2	374, 1.99	0.0022	213.3	1, 1.99	0.0020	422.0

Table 8: Preconditioned GMRES applied to the disturbed PDE. For rows with a * the restart iteration r was also optimized.

Based on the results in Table 8 is clear that disturbing the variables hardly affects run-time of GMRES. Both the default and optimal parameter values result in GMRES run-times that are very similar to that of the undisturbed linear system.

Just as with the undisturbed linear system, the minimization methods find solutions which have a considerably shorter GMRES run-time. The NM method again consistently finds the same global minimum as the PSO and NM-PSO methods, while generally requiring less computation time.

For the un-restarted GMRES the PSO and NM-PSO methods need about the same amount of time to find a minimum. This is not the case for the restarted GMRES, where in some instances the NM-PSO method is the fastest and in another instance this is the PSO method. For $h = 0.0625$ they are even both faster than the NM method. The reason for this is not clear, but perhaps the suggested values for r and ω provide a clue. Both PSO and NM-PSO regularly return $\omega = 1.99$ and/or $r = 1$. These values are both on the boundary of their respective search domains. The particular characteristics of this optimization problem might thus be such that the minimum is on (or beyond) the boundary. The way in which boundary violations are handled (see A.2.5), might cause a lot of the points to be moved into the same place, which causes a false trigger of the stopping criterion. Broadening the search space could be a solution, but further research would be needed to find out what is exactly going on.

7 Conclusion

After having analyzed the configuration and performance of the NM, PSO and NM-PSO methods we can conclude that the NM-PSO indeed appears to combine the strengths of both the NM and the PSO method. In most of the test cases it was similarly accurate as the PSO method, but it requires considerably fewer objective function evaluations/run-time. This was especially apparent in the results of the test functions and to a certain extent also in the tests of the linear system solvers SOR and preconditioned GMRES. We therefore conclude that for general applications the NM-PSO should probably be preferred over PSO.

During the tests on SOR and preconditioned GMRES we saw that with a good choice of parameters indeed a significant speed-up in the computation times of these linear system solvers can be achieved. Given their widespread use, it is useful to look into ways of combining these solvers with derivative-free minimization methods. One way of doing this could be to construct a simultaneous implementation. All the run-times during a series of, for example, preconditioned GMRES runs could be fed into a minimization method, which constantly produces a new parameter choice for the next preconditioned GMRES run. This would however only be feasible if the optimal parameters are actually somewhat similar for each of the GMRES runs.

Another observation regarding the linear system solvers is that the NM method finds solutions that are just as good as PSO and NM-PSO. The only difference is that NM finds these a lot faster and therefore we conclude that further research on these types of problems should probably focus on NM rather than on PSO or NM-PSO. If the NM-PSO method is however further tested on such linear system solvers, it would be wise to reconsider the parameter choices based on easier test functions than the ones used in this thesis.

References

[1] A. Conn, K. Scheinberg, and L. Vicente, *Introduction to derivative-free optimization*. SIAM, 2009.

[2] L. Rios and N. Sahinidis, "Derivative-free optimization: A review of algorithms and comparison of software implementations," *Journal of Global Optimization*, vol. 56, no. 3, pp. 1247–1293, 2013.

[3] J. Larson, M. Menickelly, and S. Wild, "Derivative-free optimization methods," *Acta Numerica*, vol. 28, pp. 287–404, 2019.

[4] S. Petersen, "Methods of optimization for numerical algorithms," Bachelor thesis, University of Groningen, 2017.

[5] E. Maquelin, "Optimizing parameters of iterative methods," Bachelor thesis, University of Groningen, 2018.

[6] J. van der Meulen, "Optimizing sor using derivative-free optimization," Bachelor thesis, University of Groningen, 2019.

[7] T. Ludwig, "Optimizing parameters in relaxed krylov subspace methods using derivative-free optimization," Bachelor thesis, University of Groningen, 2024.

[8] S. Fan, Y. Liang, and E. Zahara, "Hybrid simplex search and particle swarm optimization for the global optimization of multimodal functions," *Engineering optimization*, vol. 36, no. 4, pp. 401–418, 2004.

[9] J. Nelder and R. Mead, "A simplex method for function minimization," *The computer journal*, vol. 7, no. 4, pp. 308–313, 1965.

[10] J. Kennedy and R. Eberhart, "Particle swarm optimization," in *Proceedings of ICNN'95-international conference on neural networks*, ieee, vol. 4, 1995, pp. 1942–1948.

[11] A. Quarteroni, R. Sacco, and F. Saleri, *Numerical mathematics*. Springer Science & Business Media, 2010, vol. 37.

[12] S. Surjanovic and D. Bingham, *Virtual library of simulation experiments: Test functions and datasets*, Retrieved January 25, 2025, from `http://www.sfu.ca/~ssurjano`.

[13] J. Lagarias, J. Reeds, M. Wright, and P. Wright, "Convergence properties of the nelder–mead simplex method in low dimensions," *SIAM Journal on optimization*, vol. 9, no. 1, pp. 112–147, 1998.

[14] E. Fan, "Global optimization of lennard-jones atomic clusters," Ph.D. dissertation, McMaster University, 2002.

[15] Y. Shi and R. Eberhart, "A modified particle swarm optimizer," in *1998 IEEE international conference on evolutionary computation proceedings. IEEE world congress on computational intelligence (Cat. No. 98TH8360)*, IEEE, 1998, pp. 69–73.

[16] K. Kameyama, "Particle swarm optimization-a survey," *IEICE transactions on information and systems*, vol. 92, no. 7, pp. 1354–1361, 2009.

[17] A. Olsson, *Particle swarm optimization: theory, techniques and applications*. Nova Science Publishers, Inc., 2010.

[18] D. Wang, D. Tan, and L. Liu, "Particle swarm optimization algorithm: An overview," *Soft computing*, vol. 22, no. 2, pp. 387–408, 2018.

[19] T. Shami, A. El-Saleh, M. Alswaitti, Q. Al-Tashi, M. Summakieh, and S. Mirjalili, "Particle swarm optimization: A comprehensive survey," *Ieee Access*, vol. 10, pp. 10 031–10 061, 2022.

[20] M. Li, H. Chen, X. Shi, S. Liu, M. Zhang, and S. Lu, "A multi-information fusion "triple variables with iteration" inertia weight pso algorithm and its application," *Applied Soft Computing*, vol. 84, p. 105 677, 2019.

[21] Y. Shi and R. C. Eberhart, "Empirical study of particle swarm optimization," in *Proceedings of the 1999 congress on evolutionary computation-CEC99 (Cat. No. 99TH8406)*, IEEE, vol. 3, 1999, pp. 1945–1950.

[22] J. Lu, H. Hu, and Y. Bai, "Generalized radial basis function neural network based on an improved dynamic particle swarm optimization and adaboost algorithm," *Neurocomputing*, vol. 152, pp. 305–315, 2015.

[23] A. Carlise and G. Dozier, "An off-the-shelf pso," in *Proceedings of the Particle Swarm Optimization Workshop, Indianapolis, IN, USA*, 2001.

[24] A. Ratnaweera, S. K. Halgamuge, and H. C. Watson, "Self-organizing hierarchical particle swarm optimizer with time-varying acceleration coefficients," *IEEE Transactions on evolutionary computation*, vol. 8, no. 3, pp. 240–255, 2004.

[25] S. Fan and E. Zahara, "A hybrid simplex search and particle swarm optimization for unconstrained optimization," *European Journal of Operational Research*, vol. 181, no. 2, pp. 527–548, 2007.

[26] R. Barrett, M. Berry, T. Chan, *et al.*, *Templates for the solution of linear systems: building blocks for iterative methods*. SIAM, 1994.

[27] D. Young and T. Mai, "Iterative algorithms and software for solving large sparse linear systems," *Communications in Applied Numerical Methods*, vol. 4, no. 3, pp. 435–456, 1988.

[28] Y. Saad, *Iterative methods for sparse linear systems*. SIAM, 2003.

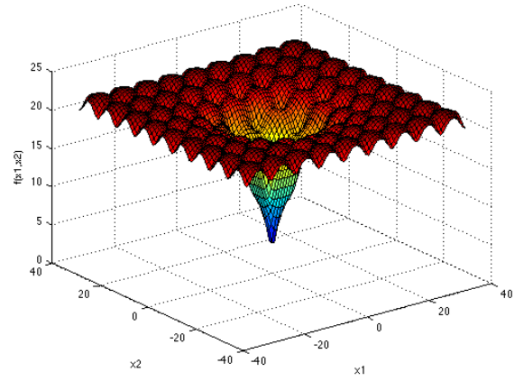
[29] R. Chen, X. Ping, D. Wang, and E. Yung, "Ssor preconditioned gmres for the fem analysis of waveguide discontinuities with anisotropic dielectric," *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, vol. 17, no. 2, pp. 105–118, 2004.

A Appendix

A.1 Test functions

All test functions and all 2-dimensional illustrations were copied from [12].

A.1.1 Ackley Function

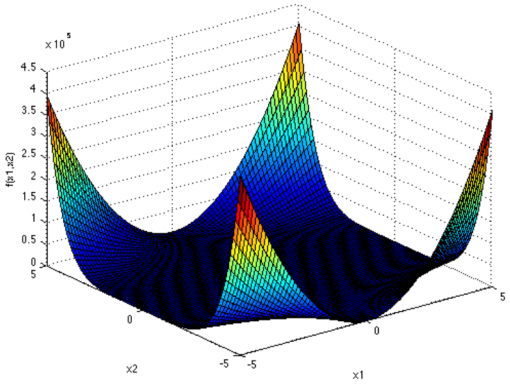


Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-32.768, 32.768]$ for all $i = 1, 2, ..., d$
Global minimum: $f(x^*) = 0$

$$f(x) = -a \exp\left(-b\sqrt{\frac{1}{d}\sum_{i=1}^{d} x_i^2}\right) - \exp\left(\frac{1}{d}\sum_{i=1}^{d}\cos(cx_i)\right),$$

where $a = 20$, $b = 0.2$ and $c = 2\pi$

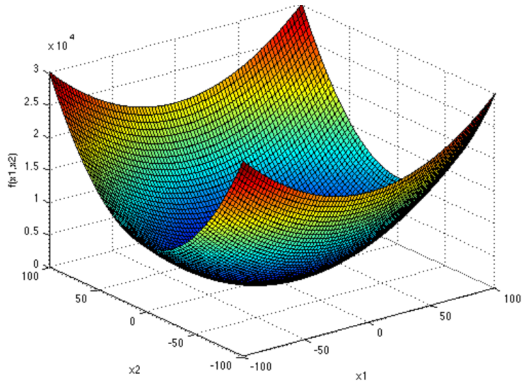
A.1.2 Beale Function



Dimension: 2
Bounds: $x_i \in [-4.5, 4.5]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$

$$f(x) = (1.5 - x_1 + x_1x_2)^2 + (2.25 - x_1 + x_2x_2^2)^2 + (2.625 - x_1 + x_1x_2^3)^2$$

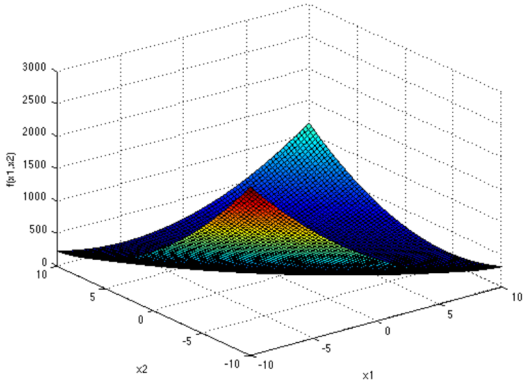
A.1.3 Bohachevsky Function



Dimension: 2
Bounds: $x_i \in [-100, 100]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$

$$f(x) = x_1^2 + 2x_2^2 - 0.3\cos(3\pi x_1) - 0.4\cos(4\pi x_2) + 0.7$$

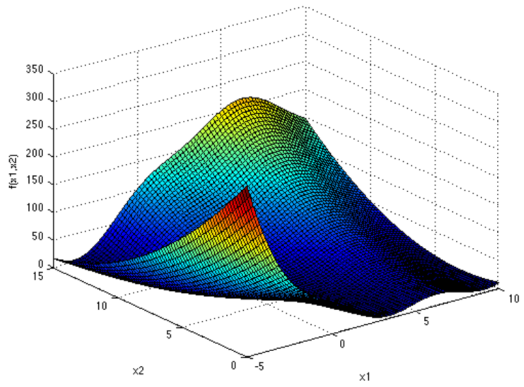
A.1.4 Booth Function



Dimension: 2
Bounds: $x_i \in [-10, 10]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$

$$f(x) = (x_1 + 2x_2 - 7)^2 + (2x_1 + x_2 - 5)^2$$

A.1.5 Branin Function



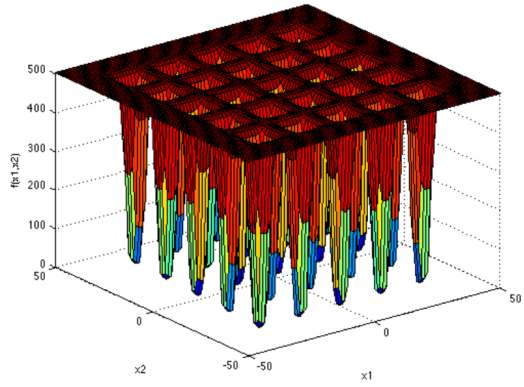
Dimension: 2
Bounds: $x_1 \in [-5, 10]$ and $x_2 \in [0, 15]$
Global minimum: $f(x^*) = 0.397887$

$f(x) = a(x_2 - bx_1^2 + cx_1 - r)^2 + s(1-t)\cos(x_1) + s$,
where $a = 1$, $b = \frac{5.1}{4\pi^2}$, $c = \frac{5}{\pi}$, $r = 6$, $s = 10$ and $y = \frac{1}{8\pi}$

A.1.6 Colville Function

Dimension: 4
Bounds: $x_i \in [-10, 10]$ for $i = 1, 2, 3, 4$
Global minimum: $f(x^*) = 0$
$f(x) = 100(x_1^2 - x_2)^2 + (x_1 - 1)^2 + (x_3 - 1)^2 + 90(x_3^2 - x_4)^2 + 10.1((x_2 - 1)^2 + (x_4 - 1)^2) + 19.8(x_2 - 1)(x_4 - 1)$

A.1.7 De Jong Function n.5

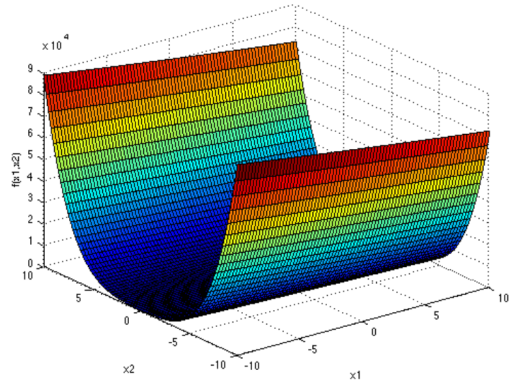


Dimension: 2
Bounds: $x_i \in [-65.536, 65.536]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0.998004$

$f(x) = (0.002 + \sum_{i=1}^{25} \frac{1}{i + (x_1 - a_{1i})^6 + (x_2 - a_{2i})^6})^{-1}$,

where $a = \begin{pmatrix} [-32 & -16 & 0 & 16 & 32] \times 5 \\ [-32] \times 5 & [-16] \times 5 & [0] \times 5 & [16] \times 5 & [32] \times 5 \end{pmatrix}$

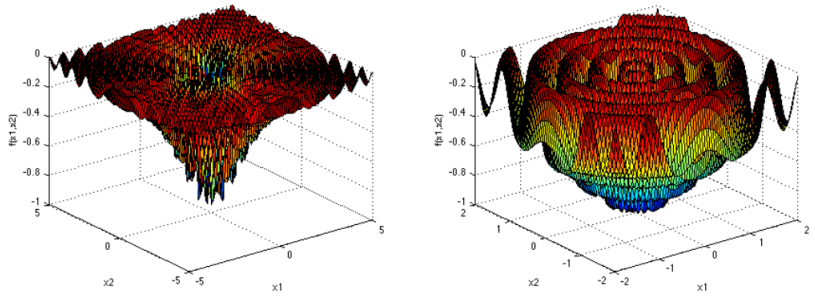
A.1.8 Dixon-Price Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-10, 10]$ for all $i = 1, ..., d$
Global minimum: $f(x^*) = 0$

$f(x) = (x_1 - 1)^2 + \sum_{i=2}^{d} i(2x_i^2 - x_{i-1})^2$

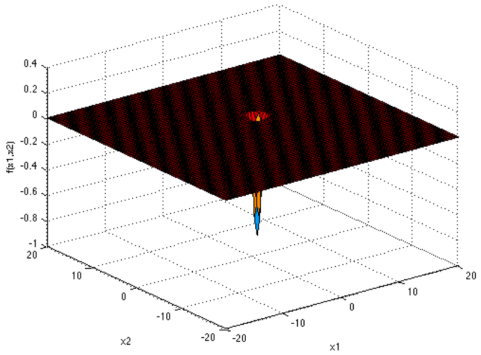
A.1.9 Drop-Wave Function



Dimension: 2
Bounds: $x_i \in [-5.12, 5.12]$ for all $i = 1, ..., d$
Global minimum: $f(x^*) = -1$

$$f(x) = -\frac{1+\cos(12\sqrt{x_1^2+x_2^2})}{0.5(x_1^2+x_2^2)+2}$$

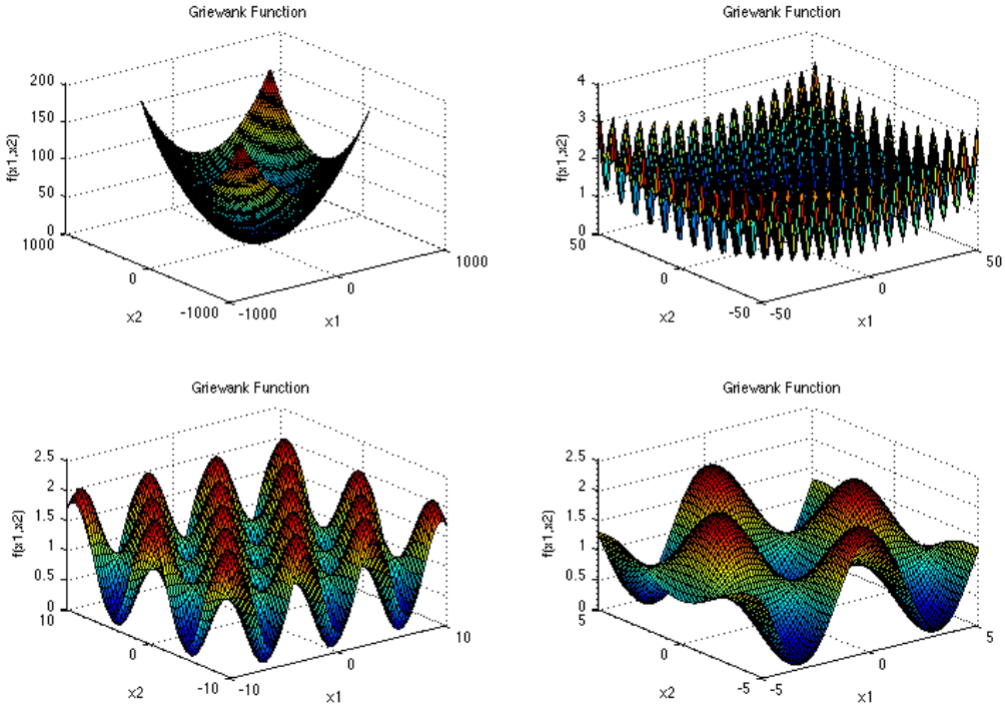
A.1.10 Easom Function



Dimension: 2
Bounds: $x_i \in [-100, 100]$ for $i = 1, 2$
Global minimum: $f(x^*) = -1$

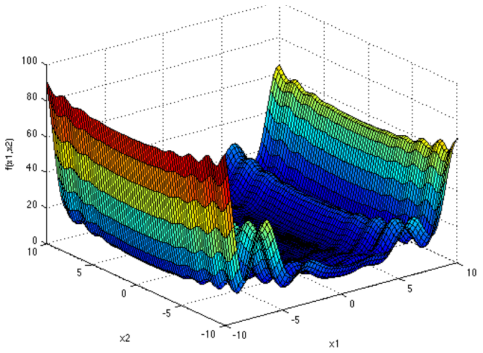
$$f(x) = -\cos(x_1)\cos(x_2)\exp(-(x_1-\pi)^2-(x_2-\pi)^2)$$

A.1.11 Griewank Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-600, 600]$ for all $i = 1, ..., d$
Global minimum: $f(x^*) = 0$
$f(x) = \sum_{i=1}^{d} \frac{x_i^2}{4000} - \prod_{i=1}^{d} \cos(\frac{x_1}{\sqrt{i}}) + 1$

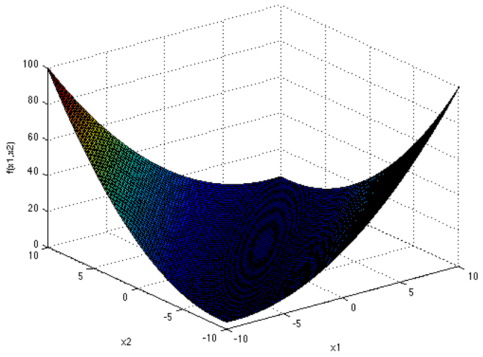
A.1.12 Levy Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-10, 10]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = \sin^2(\pi w_1) + \sum_{i=1}^{d-1}(w_i - 1)^2[1 + 10\sin^2(\pi w_i + 1)] + (w_d - 1)^2[1 + \sin^2(2\pi w_d)]$, where $w_i = 1 + \frac{x_i - 1}{4}$

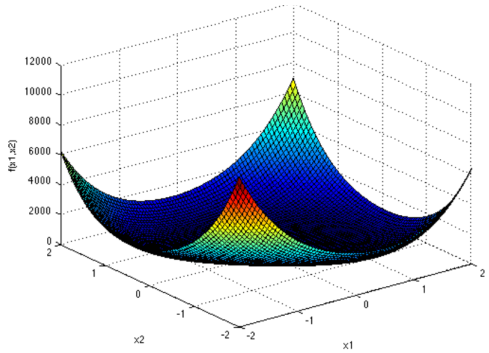
A.1.13 Matyas Function



Dimension: 2
Bounds: $x_i \in [-10, 10]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$

$f(x) = 0.26(x_1^2 + x_2^2) - 0.48x_1x_2$

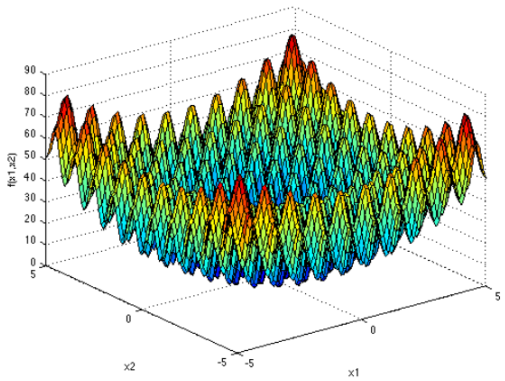
A.1.14 Perm Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-d, d]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = \sum_{i=1}^{d}(\sum_{j=1}^{d}(j + 10)(x_j^i - \frac{1}{j^i}))^2$

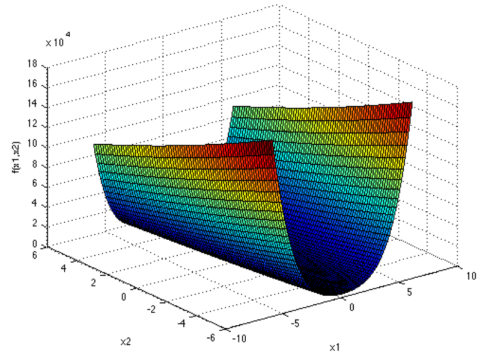
A.1.15 Rastrigin Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-5.12, 5.12]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = 10d + \sum_{i=1}^{d}[x_i^2 - 10\cos(2\pi x_i)]$

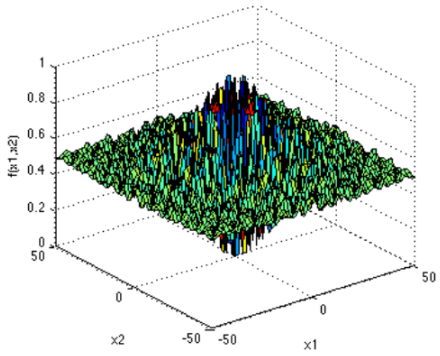
A.1.16 Rosenbrock Function

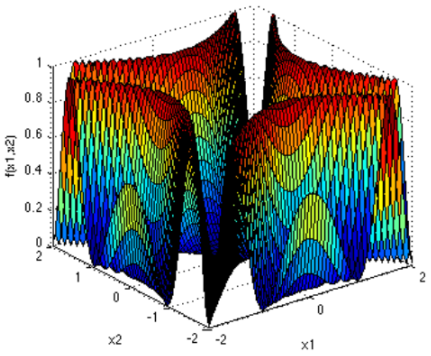


Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-5, 10]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = \sum_{i=1}^{d-1}[100(x_{i+1} - x - i^2)^2 + (x_i - 1)^2]$

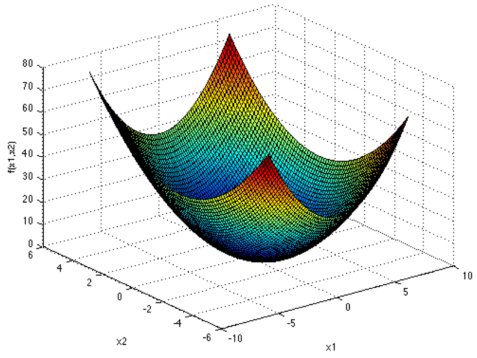
A.1.17 Schaffer Function n.2





Dimension: 2
Bounds: $x_i \in [-100, 100]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$
$f(x) = 0.5 + \frac{\sin^2(x_1^2 - x_2^2) - 0.5}{[1 + 0.001(x_1^2 + x_2^2)]^2}$

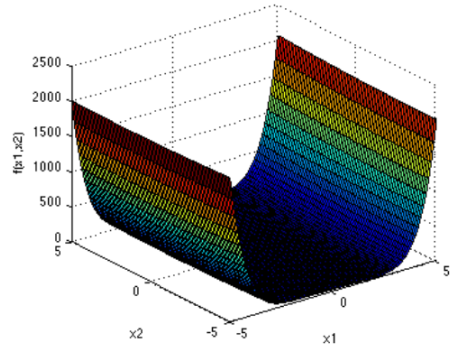
A.1.18 Sphere Function

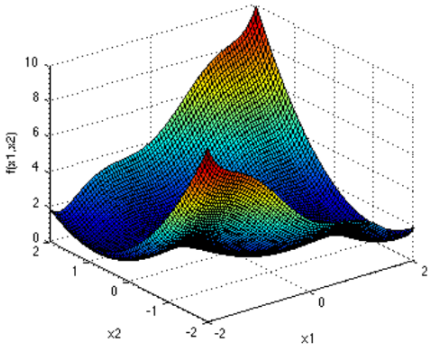


Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-5.12, 5.12]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = \sum_{i=1}^{d} x_i^2$

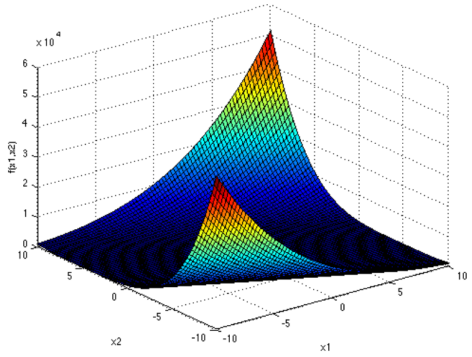
A.1.19 Three-hump Camel Function





Dimension: 2
Bounds: $x_i \in [-5, 5]$ for $i = 1, 2$
Global minimum: $f(x^*) = 0$
$f(x) = 2x_1^2 - 1.05x_1^4 + \frac{x_1^6}{6} + x_1 x_2 + x_2^2$

A.1.20 Zakharov Function



Dimension: any $d \in \mathbb{N}$
Bounds: $x_i \in [-5, 10]$ for $i = 1, .., d$
Global minimum: $f(x^*) = 0$

$f(x) = \sum_{i=1}^{d} x_i^2 + (\sum_{i=1}^{d} 0.5 i x_i)^2 + (\sum_{i=1}^{d} 0.5 i x_i)^4$

A.2 MATLAB code

A.2.1 Nelder-Mead method (NM)

```
function [bestSol, bestVal, totalEvals, solsPerRun, valsPerRun, ...
          evalsPerRun, time] = method_NM(func, LB, UB, dim, maxRuns,...
                                         maxIts, tol)
    start = tic;
    n = dim;
    alpha = 1;      %reflection
    beta = 0.5;     %expansion
    gamma = 2;      %contraction
    sigma = 0.5;    %shrink

    solsPerRun = zeros(n,maxRuns);
    valsPerRun = zeros(1,maxRuns);
    evalsPerRun = zeros(1,maxRuns);
    for run = 1:maxRuns
        x = zeros(n,n+1);       %points
        y = zeros(1,n+1);       %values
        for i = 1:n         %first point
            x(i,n+1) = LB(i) + rand() * (UB(i)-LB(i));
        end
        y(n+1) = func(x(:,n+1));
        for j = 1:n         %other n points
            x(:,j) = x(:,1);
            if x(j,j) ~= 0
                x(j,j) = (1 + 0.05)*x(j,j);
                x(j,j) = func_BoundaryViolations(x(j,j),LB,UB,j);
            else
                x(j,j) = 0.00025;
            end
            y(j) = func(x(:,j));
        end
        [y,idx] = sort(y); %sort so x(:,1) has the lowest function value
        x = x(:,idx);
        it = 0;
        evals = n+1;
        stopCriteriumMet = func_StopCriterium(it,maxIts,x,y,tol);
        while stopCriteriumMet == false
            it = it + 1;
            xbar = sum(x(:,1:n), 2)/n; %perform reflection
            xr = (1 + alpha)*xbar - alpha*x(:,n+1);
            xr = func_BoundaryViolations(xr, LB, UB);
            fxr = func(xr);
            evals = evals + 1;
```

```
if fxr < y(1)   %perform expansion
    xe = (1 - gamma)*xbar + gamma*x(:,n+1);
    xe = func_BoundaryViolations(xe, LB, UB);
    fxe = func(xe);
    evals = evals + 1;
    if fxe < fxr        %keep expension
        x(:,n+1) = xe;
        y(n+1) = fxe;
    else                %keep reflection
        x(:,n+1) = xr;
        y(n+1) = fxr;
    end
else %fxr >= y(1)
    if fxr < y(n)       %keep reflection
        x(:,n+1) = xr;
        y(n+1) = fxr;
    else % fxr >= y(n)  %perform contraction
        if fxr < y(n)
            x(:,n+1) = xr;
            y(n+1) = fxr;
        end
        xc = (1-beta)*xbar + beta*x(:,n+1);
        xc = func_BoundaryViolations(xc, LB, UB);
        fxc = func(xc);
        evals = evals + 1;
        if fxc < y(n+1) %keep contraction
            x(:,n+1) = xc;
            y(n+1) = fxc;
        else            %perform shrink
            for j = 2:n+1
                x(:,j) = x(:,1) + sigma*(x(:,j) - x(:,1));
                x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
                y(j) = func(x(:,j));
            end
            evals = evals + n;
        end
    end
end
[y,idx] = sort(y);
```

A.2.2 Particle Swarm Optimization method (PSO)

```
function [bestSol, bestVal, totalEvals, solsPerRun, valsPerRun, ...
          evalsPerRun, time] = method_PSO(func, LB, UB, dim, maxRuns, ...
                                          maxIts, tol)
    start = tic;
    n = dim;
    p = 35;            %number of points
    N = func_VonNeumann(p);
    solsPerRun = zeros(n,maxRuns);
    valsPerRun = zeros(1,maxRuns);
    evalsPerRun = zeros(1,maxRuns);
    for run = 1:maxRuns
        x = zeros(n,p);    %points
        v = zeros(n,p);    %velocities
        y = zeros(1,p);    %values
        for j = 1:p
            for i = 1:n
                x(i,j) = LB(i) + rand()*(UB(i) - LB(i));
            end
            x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
            v(:,j) = 0.1*x(:,j);
            y(j) = func(x(:,j));
        end
        [y,idx] = sort(y);
        x = x(:,idx);
        v = v(:,idx);
        N = func_ReIndexN(N,idx);
        N_bestY = zeros(1,p); %best value in each neigborhood
        N_bestX = zeros(n,p); %best point in each neigborhood
        [N_bestY,N_bestX] = func_updateNbests(N,N_bestY,N_bestX,y,x);
        valsPerRun(run) = y(1);
        solsPerRun(:,run) = x(:,1);
        it = 0;
        evals = p;
        stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
            y(1:n+1),tol);
        x_past = x;     %best past position of each point
        y_past = y;     %best past value of each point
        while stopCriteriumMet == false
            it = it + 1;
            w = 0.5 + 0.55*exp(-8*it/maxIts);
            for j = 1:p              %update velocities
                for i = 1:n
```

```
                dir1 = (2.5-2*(it/maxIts))*rand()*(x_past(i,j)-x(i,j));
                dir2 =(0.5+2*(it/maxIts))*rand()*(N_bestX(i,j)-x(i,j));
                v(i,j) = w * v(i,j) + dir1 + dir2;
            end
        end
        for j = 1:p
            x(:,j) = x(:,j) + v(:,j);   %update position
            x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
            y(j) = func(x(:,j));        %update value
            if y(j) < y_past(j)         %update best past value
                x_past(:,j) = x(:,j);
                y_past(j) = y(j);
            end
        end
        evals = evals + p;
        [y,idx] = sort(y);          %sort the points
        x = x(:,idx);
        v = v(:,idx);
        N = func_ReIndexN(N, idx);
        [N_bestY,N_bestX] = func_updateNbests(N,N_bestY,N_bestX,y,x);
        x_past = x_past(:,idx);
        y_past = y_past(:,idx);
        if y(1) < valsPerRun(run)   %check for a new best point
            solsPerRun(:,run) = x(:,1);
            valsPerRun(run) = y(1);
        end
        stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
            y(1:n+1),tol);
    end  % while (iteration)
    evalsPerRun(run) = evals;
end  % for (run)
time = toc(start);
[bestVal, bestRun] = min(valsPerRun(1:maxRuns));
bestSol = solsPerRun(:,bestRun);
totalEvals = sum(evalsPerRun);
end
```

A.2.3 Hybrid method (NM-PSO)

```
function [bestSol, bestVal, totalEvals, solsPerRun, valsPerRun, ...
          evalsPerRun, time] = method_NMPSO(func, LB, UB, dim, ...
                                            maxRuns, maxIts, tol)
    start = tic;
    m = 1;
    n = dim;
    p = 15;            %number of points
    N = func_VonNeumann(p);
    solsPerRun = zeros(n,maxRuns);
    valsPerRun = zeros(1,maxRuns);
    evalsPerRun = zeros(1,maxRuns);
    for run = 1:maxRuns
        x = zeros(n,p);    %points
        v = zeros(n,p);    %velocities
        y = zeros(1,p);    %values
        for j = 1:p
            for i = 1:n
                x(i,j) = LB(i) + rand()*(UB(i) - LB(i));
            end
            x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
            v(:,j) = 0.1*x(:,j);
            y(j) = func(x(:,j));
        end
        [y,idx] = sort(y);
        x = x(:,idx);
        v = v(:,idx);
        N = func_ReIndexN(N,idx);
        N_bestY = zeros(1,p); %best value in each neigborhood
        N_bestX = zeros(n,p); %best point in each neigborhood
        [N_bestY,N_bestX] = func_updateNbests(N,N_bestY,N_bestX,y,x);
        valsPerRun(run) = y(1);
        solsPerRun(:,run) = x(:,1);
        it = 0;
        evals = p;
        stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
            y(1:n+1),tol);
        x_past = x;     %best past position of each point
        y_past = y;     %best past value of each point
        while stopCriteriumMet == false
            it = it + 1;
            for i = 1:m             %NM iterations
                [x,y,evals] = func_NM(x,y,n,evals,func,LB,UB);
                for j = 2:n+1
```

```
                    if y(j) < y_past(j)  %update best past value
                        x_past(:,j) = x(:,j);
                        y_past(j) = y(j);
                    end
                end
                [x,v,y,N,x_past,y_past] = func_Sort(x,v,y,N,x_past,y_past);
            end
            [N_bestY,N_bestX] = func_updateNbests(N,N_bestY,N_bestX,y,x);
            w = 0.5 + 0.55*exp(-8*it/maxIts);              %PSO iteration
            for j = n+2:p                %update velocities
                for i = 1:n
                    dir1 = (2.5-2*(it/maxIts))*rand()*(x_past(i,j)-x(i,j));
                    dir2 =(0.5+2*(it/maxIts))*rand()*(N_bestX(i,j)-x(i,j));
                    v(i,j) = w * v(i,j) + dir1 + dir2;
                end
            end
            for j = n+2:p       %j=1:p in case of M-NM-PSO
                x(:,j) = x(:,j) + v(:,j);   %update position
                x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
                y(j) = func(x(:,j));        %update value
                evals = evals + 1;
                if y(j) < y_past(j)         %update best past value
                    x_past(:,j) = x(:,j);
                    y_past(j) = y(j);
                end
            end
            [x,v,y,N,x_past,y_past] = func_Sort(x,v,y,N,x_past,y_past);
            if y(1) < valsPerRun(run)   %check for a new best point
                solsPerRun(:,run) = x(:,1);
                valsPerRun(run) = y(1);
            end
            stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
                y(1:n+1),tol);
        end  % while (iteration)
        evalsPerRun(run) = evals;
    end  % for (run)
    time = toc(start);
    [bestVal, bestRun] = min(valsPerRun(1:maxRuns));
    bestSol = solsPerRun(:,bestRun);
    totalEvals = sum(evalsPerRun);
end
```

A.2.4 Altered hybrid method (altered NM-PSO)

```
function [bestSol, bestVal, totalEvals, solsPerRun, valsPerRun, ...
          evalsPerRun, time] = method_altNMPSO(func, LB, UB, dim, ...
                                                maxRuns, maxIts, tol)
    start = tic;
    c1 = 0.6;           %acceleration factor
    c2 = 1.6;           %acceleration factor
    lambda = 0.85;      %mutation coefficient
    n = dim;            %dimension
    p = 3*n + 1;        %swarm size

    evalsPerRun = zeros(maxRuns,1);
    solsPerRun = zeros(maxRuns,n);
    valsPerRun = zeros(maxRuns,1);

    for run = 1:maxRuns
        x = zeros(n,p);
        v = zeros(n,p);
        y = zeros(1,p);
        sigma = 1;  %reset the mutation heuristic variable

        % generate a population of 3n+1 particles
        for i = 1:n                 % 1st particle
            x(i,1) = LB(i) + rand() * (UB(i)-LB(i));
        end
        x(:,1) = func_BoundaryViolations(x(:,1),LB,UB);
        v(:,1) = 0.1*x(:,1);
        y(1) = func(x(1,:));
        for j = 2 : p         % other 3n particles
            x(:,j) = x(:,1);
            if j <= n+1                     % first n particles
                x(j-1,j) = x(j-1,j) + 1;
            elseif j > n+1 && j <=  2*n+1   % next n particles
                x(j-n-1,j) = x(j-n-1,j) + rand()*(UB(j-n-1)-LB(j-n-1));
            else                            % last n particles
                x(j-2*n-1,j) = 2*x(j-2*n-1,j) - x(j-2*n-1, j-n);
            end
            x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
            v(:,j) = 0.1*x(:,j);
            y(j) = func(x(:,j));
        end
        [y,idx] = sort(y);
        x = x(:,idx);
        v = v(:,idx);
        valsPerRun(run) = y(1);
        solsPerRun(run,:) = x(:,1);
```

```
it = 0;
evals = p;
stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
                                      y(1:n+1),tol);
while stopCriteriumMet == false
    it = it + 1;
    [x(:,1), y(1), sigma] = func_MutationHeuristic(x(:,1), y(1),...
                                     func, lambda, sigma, LB, UB);
    evals = evals + 5;
    % === perform NM ==== for the n+1 best particles
    [x, y, evals] = func_NM2(x, y, dim, evals, func, LB, UB);
    % === perform PSO ==== for 2n worst particles
     w = 0.5 + (rand()/2);
    [~, globalBestIndex] = min([y(1),y(n+1)]);
    for j = n+2 : p
        %find the best neighborhood(/local) particle
        if mod(i,2) == mod(p,2)
            localBestIndex = i-1;
        else
            localBestIndex = i;
        end
        for i = 1:n
            dir_1 = c1*rand()*( x(i,localBestIndex) - x(i,j) );
            dir_2 = c2*rand()*( x(i,globalBestIndex) - x(i,j) );
            v(i,j)= w * v(i,j) + dir_1 + dir_2;
        end
    end
    for j = n+2 : p
        for i = 1:n
            x(i,j) = x(i,j) + v(i,j);
        end
        x(:,j) = func_BoundaryViolations(x(:,j),LB, UB);
        y(j) = func(x(:,j));
    end
    evals = evals + p - (n+1);
    [x,v,y] = func_Sort2(x,v,y);
    % check whether we found a new best particle of the run
    if y(1) < valsPerRun(run)
        solsPerRun(run,:) = x(:,1);
        valsPerRun(run) = y(1);
    end
```

```
            % update stopping criterium
            stopCriteriumMet = func_StopCriterium(it,maxIts,x(:,1:n+1),...
                                                  y(1:n+1),tol);
        end %while (iteration)
        evalsPerRun(run) = evals;
    end % for (run)
    time = toc(start);
    [bestVal, bestRun] = min(valsPerRun(1:maxRuns));
    bestSol = solsPerRun(bestRun,:);
    totalEvals = sum(evalsPerRun);
end
```

A.2.5 func_BoundaryViolations

```
function x = func_BoundaryViolations(x, LB, UB, idx)
    if nargin > 3   %check single coordinate
        if x < LB(idx)
            x = LB(idx)+1e-3;
        elseif x > UB(idx)
            x = UB(idx)-1e-3;
        end
    else            %check complete point
        for j = 1:numel(x)
            x(j) = func_BoundaryViolations(x(j), LB, UB, j);
        end
    end
end
```

A.2.6 func_DiscretizeODE

```
function [A,b] = func_DiscretizeODE(h)
    alpha = 1E-4;
    n = 1/h+1;
    A = zeros(n,n) ;
    b = zeros(n,1) ;
    for i = 2:n-1
        A(i,i-1) = 1/(h^2);
        A(i,i) = alpha - 2/(h^2);
        A(i,i+1) = 1/(h^2);
        b(i) = 0 + (i-1)*h;
    end
    A(1,1) = 1;
    A(1,2) = 0;
    A(n,n) = 1;
    A(n,n-1) = 0;
    b(1) = 1;
    b(n) = 1;
end
```

A.2.7 func_DiscretizePDE

```
function [A,b] = func_DiscretizePDE(h)
    n = 1/(h)+1;
    A_i = [1:n^2 2:n^2 1:n^2-1 1:n^2-n n+1:n^2];
    A_j = [1:n^2 1:n^2-1 2:n^2 n+1:n^2 1:n^2-n];
    vals_diag = zeros(1,n^2);
    vals_L1 = zeros(1,n^2-1);
    vals_R1 = zeros(1,n^2-1);
    vals_L2 = zeros(1,n^2-n);
    vals_R2 = zeros(1,n^2-n);
    b = zeros(n^2, 1);
    for i = 1 : n^2         %diag and b
        if mod(i,1000000) == 0
            disp(i)
        end

        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_diag(i) = 1;
            b(i) = 0;
        else
            vals_diag(i) = -4/(h^2);
            xi = floor((i-1)/n);
            yi = mod(i-1,n);
            b(i) = xi*h + yi*h;
        end
    end
    for i = 2 : n^2         %L1
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_L1(i-1) = 0;
        else
            vals_L1(i-1) = 1/(h^2);
        end
    end
    for i = 1 : n^2-1       %R1
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_R1(i) = 0;
        else
            vals_R1(i) = 1/(h^2);
        end
    end
    for i = 1 : n^2 - n     %L2
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_L2(i) = 0;
        else
            vals_L2(i) = 1/(h^2);
        end
    end
    for i = n+1 : n^2       %R2
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_R2(i-n) = 0;
        else
            vals_R2(i-n) = 1/(h^2);
```

```
        end
    end
    A_vals = [vals_diag vals_L1 vals_R1 vals_L2 vals_R2];
    A = sparse(A_i, A_j, A_vals);
end
```

A.2.8 func_DiscretizePDEandDisturb

```
function [A,b] = func_DiscretizePDEandDisturb(h)
    rng("default") %fix randomness for repeatability
    rate = 0.1;


    n = 1/(h)+1;
    A_i = [1:n^2 2:n^2 1:n^2-1 1:n^2-n n+1:n^2];
    A_j = [1:n^2 1:n^2-1 2:n^2 n+1:n^2 1:n^2-n];
    vals_diag = zeros(1,n^2);
    vals_L1 = zeros(1,n^2-1);
    vals_R1 = zeros(1,n^2-1);
    vals_L2 = zeros(1,n^2-n);
    vals_R2 = zeros(1,n^2-n);
    b = zeros(n^2, 1);
    for i = 1 : n^2         %diag and b
        if mod(i,1000000) == 0
            disp(i)
        end

        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_diag(i) = 1;
            b(i) = 0;
        else
            vals_diag(i) = -4/(h^2);
            xi = floor((i-1)/n);
            yi = mod(i-1,n);
            b(i) = xi*h + yi*h;
        end
    end
    for i = 2 : n^2         %L1
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_L1(i-1) = 0;
        else
            vals_L1(i-1) = 1/(h^2);
        end
    end
    for i = 1 : n^2-1       %R1
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_R1(i) = 0;
        else
            vals_R1(i) = 1/(h^2);
        end
    end
    for i = 1 : n^2 - n     %L2
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_L2(i) = 0;
        else
            vals_L2(i) = 1/(h^2);
        end
    end
```

```
    for i = n+1 : n^2       %R2
        if mod(i,n) == 0 || mod(i,n) == 1 || i < n || i > n^2-n
            vals_R2(i-n) = 0;
        else
            vals_R2(i-n) = 1/(h^2);
        end
    end
    A_vals = [vals_diag vals_L1 vals_R1 vals_L2 vals_R2];
    A = sparse(A_i, A_j, A_vals);

    d_D = randperm(n^2, round(rate*(n^2)));
    d_L1 = randperm(n^2-1, round(rate*(n^2-1)));
    d_L2 = randperm(n^2-n, round(rate*(n^2-n)));
    d_R1 = randperm(n^2-1, round(rate*(n^2-1)));
    d_R2 = randperm(n^2-n, round(rate*(n^2-n)));


    for i = d_D
        A(i,i) = 10/(h^2) - 20/(h^2) * rand();
    end
    for i = d_L1
        A(i+1,i) = 10/(h^2) - 20/(h^2) * rand();
    end
    for i = d_L2
        A(i+n,i) = 10/(h^2) - 20/(h^2) * rand();
    end
    for i = d_R1
        A(i,i+1) = 10/(h^2) - 20/(h^2) * rand();
    end
    for i = d_R2
        A(i,i+n) = 10/(h^2) - 20/(h^2) * rand();
    end

end
```

A.2.9 func_MutationHeuristic

```
function [x, y, sigma] = func_MutationHeuristic(x, y, func, lambda, sigma, LB, UB)
    n = numel(x);
    newX = zeros(n,5);
    newY = zeros(1,5);
    for j = 1:5
        newX(:,j) = x + normrnd(0,sigma,n,1);
        newX(:,j) = func_BoundaryViolations(newX(:,j),LB,UB);
        newY(j) = func(newX(:,j));
    end
    succesfulMutations = 0;
    for j = 1:5
       if newY(j) < y
           succesfulMutations = succesfulMutations + 1;
       end
    end
    if succesfulMutations/5 > 0.4
       sigma = (1/lambda)*sigma;
    elseif succesfulMutations/5 < 0.4
       sigma = lambda*sigma;
    end
    [bestNewVal,bestNewGlobalValindex] = min(newY);
    if bestNewVal < y
       x = newX(:,bestNewGlobalValindex);
       y = bestNewVal;
    end
end
```

A.2.10 func_NM

```
function [x, y, evals] = func_NM(x, y, dim, evals, func, LB, UB)

    n = dim;
    alpha = 1;    % reflection
    beta = 0.5;   % expansion
    gamma = 2;    % contraction
    sigma = 0.5;  % shrink

    %perform reflection
    xbar = sum(x(:,1:n), 2)/n;
    xr = (1 + alpha)*xbar - alpha*x(:,n+1);
    xr = func_BoundaryViolations(xr, LB, UB);
    fxr = func(xr); evals = evals + 1;
    if fxr < y(1)   %perform expansion
        xe = (1 - gamma)*xbar + gamma*x(:,n+1);
        xe = func_BoundaryViolations(xe, LB, UB);
        fxe = func(xe); evals = evals + 1;
        if fxe < fxr        %keep expension
            x(:,n+1) = xe;
            y(n+1) = fxe;
        else                %keep reflection
            x(:,n+1) = xr;
            y(n+1) = fxr;
        end
    else %fxr >= y(1)
        if fxr < y(n)       %keep reflection
            x(:,n+1) = xr;
            y(n+1) = fxr;
        else % fxr >= y(n)  %perform contraction
            if fxr < y(n)
                x(:,n+1) = xr;
                y(n+1) = fxr;
            end
            xc = (1-beta)*xbar + beta*x(:,n+1);
            xc = func_BoundaryViolations(xc, LB, UB);
            fxc = func(xc); evals = evals + 1;
            if fxc < y(n+1) %keep contraction
                x(:,n+1) = xc;
                y(n+1) = fxc;
            else            %perform shrink
                for j = n+1:n+1       % sensible should be j=2:n+1
                    x(:,j) = x(:,1) + sigma*(x(:,j) - x(:,1));
                    x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
                    y(j) = func(x(:,j));
                end
                evals = evals + n;
            end
        end
    end
end
```

A.2.11 func_NM2

```
function [x, y, evals] = func_NM2(x, y, dim, evals, func, LB, UB)

    n = dim;
    alpha = 1;    % reflection
    beta = 0.5;   % expansion
    gamma = 2;    % contraction
    sigma = 0.5;  % shrink
    thetha = 2;  % 2nd expansion

    %perform reflection
    xbar = sum(x(:,1:n), 2)/n;
    xr = (1 + alpha)*xbar - alpha*x(:,n+1);
    xr = func_BoundaryViolations(xr, LB, UB);
    fxr = func(xr); evals = evals + 1;
    if fxr < y(1)   %perform expansion
        xe = (1 - gamma)*xbar + gamma*x(:,n+1);
        xe = func_BoundaryViolations(xe, LB, UB);
        fxe = func(xe); evals = evals + 1;
        if fxe < fxr        %try second expension
            xee = (1 - thetha)*xbar + thetha*xe;
            xee = func_BoundaryViolations(xee, LB, UB);
            fxee = func(xee); evals = evals + 1;
            if fxee < fxe   %keep 2nd expension
                x(:,n+1) = xee;
                y(n+1) = fxee;
            else            %keep expension
                x(:,n+1) = xe;
                y(n+1) = fxe;
            end
        else                %keep reflection
            x(:,n+1) = xr;
            y(n+1) = fxr;
        end
    else %fxr >= y(1)
        if fxr < y(n)       %keep reflection
            x(:,n+1) = xr;
            y(n+1) = fxr;
        else % fxr >= y(n)  %perform contraction
            if fxr < y(n)
                x(:,n+1) = xr;
                y(n+1) = fxr;
            end
            xc = (1-beta)*xbar + beta*x(:,n+1);
            xc = func_BoundaryViolations(xc, LB, UB);
            fxc = func(xc); evals = evals + 1;
            if fxc < y(n+1) %keep contraction
                x(:,n+1) = xc;
                y(n+1) = fxc;
            else            %perform shrink
                for j = n+1:n+1       % sensible should be j=2:n+1
                    x(:,j) = x(:,1) + sigma*(x(:,j) - x(:,1));
                    x(:,j) = func_BoundaryViolations(x(:,j),LB,UB);
                    y(j) = func(x(:,j));
                end
                evals = evals + n;
            end
        end
    end
end
```

A.2.12 func_PreconditionerSSOR1

```
function [M1,M2] = func_PreconditionerSSOR1(A, omega)
    D = diag(diag(A));
    L = tril(A,-1);
    R = triu(A,1);
    M1 = (1 / (2 - omega)) * ((D/omega + L));
    M2 = eye(size(A))+ diag(omega./diag(D))*R;
end
```

A.2.13 func_ReIndexNM

```
function [NewN] = func_ReIndexN(N, idx)
    NewN = zeros(4, numel(idx));
    for j = 1:numel(idx)
        for i = 1:4
            NewN(i,j) = idx(N(i,j));
        end
    end
    NewN = NewN(:,idx);
end
```

A.2.14 func_Sort

```
function [x,v,y,N,x_past,y_past] = func_Sort(x,v,y,N,x_past,y_past)
    [y,idx] = sort(y);          %sort the points
    x = x(:,idx);
    v = v(:,idx);
    N = func_ReIndexN(N, idx);
    x_past = x_past(:,idx);
    y_past = y_past(:,idx);
end
```

A.2.15 func_Sort2

```
function [x,v,y] = func_Sort2(x,v,y)
    [y,idx] = sort(y);          %sort the points
    x = x(:,idx);
    v = v(:,idx);
end
```

A.2.16 func_StopCriterium

```
function [stopCriteriumMet] = func_StopCriterium(it, maxIts, x, fx, tol)
    d = numel(fx);
    testX = max(abs(x(:,2:d)-x(:,1)),[],"all");
    testFX = abs(fx(1)-fx(d));
    if it > maxIts || (testX < tol(1) && testFX < tol(2))
        stopCriteriumMet = true;
    else
        stopCriteriumMet = false;
    end
end
```

A.2.17 func_updateNbests

```
function [N_bestY,N_bestX] = func_updateNbests(N, N_bestY, N_bestX, y, x)
    for j = 1:numel(y)
        N_bestY(j) = y(j);
        N_bestX(:,j) = x(:,j);
        for i = 1:4
            if y(N(i,j)) < N_bestY(j)
                N_bestY(j) = y(N(i,j));
                N_bestX(:,j) = x(N(i,j));
            end
        end
    end
end
```

A.2.18 func_VonNeumann

```
function [N] = func_VonNeumann(p)
    N = zeros(4,p);
    for j = 1:p
        if mod(j,p/5) == 1          %left neighbor
            N(1,j) = j+(p/5-1);
        else
            N(1,j) = j-1;
        end
        if mod(j,p/5) == 0          %right neighbor
            N(2,j) = j-(p/5-1);
        else
            N(2,j) = j+1;
        end
        if j <= p/5                 %upper neighbor
            N(3,j) = j+(p-p/5);
        else
            N(3,j) = j-p/5;
        end
        if j >= p-(p/5-1)           %lower neighbor
            N(4,j) = j-(p-p/5);
        else
            N(4,j) = j+p/5;
        end
    end
end
```

A.2.19 GMRES1

```
function [time, xsol] = GMRES1(A,b,w)
    gmresStart = tic;
    [M1,M2] = func_PreconditionerSSOR1(A,w);
    [xsol,flag] = gmres(A,b,[],1e-3,10^4,M1,M2);
    if flag ~= 0
        time = flag*10^7;
    else
        time = toc(gmresStart);
    end
end
```

A.2.20 GMRES2

```
function [time, xsol] = GMRES2(A,b,w)
    gmresStart = tic;
    [M1,M2] = func_PreconditionerSSOR1(A,w(2));
    [xsol,flag] = gmres(A,b,round(w(1)),1e-3,10^4,M1,M2);
    if flag ~= 0
        time = flag*10^7;
    else
        time = toc(gmresStart);
    end
end
```

A.2.21 SOR1

```
function [time, xsol] = SOR1(A,b,n,w,maxIts,tol)
    startSOR = tic;
    xsol = zeros(n,1);  %initial solution
    resid = 1E3;
    it = 0;
    while norm(resid) > tol
        it = it+1;
        if it > maxIts || norm(resid) > 10^30
            time = 1000*it;
            return
        end
        xsolold = xsol;          %solution of previous iteration
        for i = 1:n
            sum = 0;
            for j = 1:i-1
                sum = sum + A(i,j)*xsol(j);
            end
            for j = i+1:n
                sum = sum + A(i,j)*xsol(j);
            end
            xbar_i = (b(i)-sum)/A(i,i);
            xsol(i) = w*xbar_i + (1-w)*xsolold(i);
        end
        for i = 1:n
            if isnan(xsol(i)) ~= 0 || isinf(xsol(i)) ~= 0
                time = 2000*it;
                return
            end
        end
        resid = A*xsol-b;
    end
    time = toc(startSOR);
end
```

A.2.22 SOR2

```
function [time, xsol] = SOR2(A,b,n,w,maxIts,tol)
    tic;
    xsol = zeros(n,1);  %initial solution
    resid = 1E3;
    it = 0;
    while norm(resid) > tol
        it = it+1;
        if it > maxIts || norm(resid) > 10^30
            time = 1000*it;
            return
        end
        xsolold = xsol;          %solution of previous iteration
        for i = 1:n
            sum = 0;
            for j = 1:i-1
                sum = sum + A(i,j)*xsol(j);
            end
            for j = i+1:n
                sum = sum + A(i,j)*xsol(j);
            end
            xbar_i = (b(i)-sum)/A(i,i);
            if mod(i,2) == 1
                xsol(i) = w(1)*xbar_i + (1-w(1))*xsolold(i);
            else
                xsol(i) = w(2)*xbar_i + (1-w(2))*xsolold(i);
            end
        end
        for i = 1:n
            if isnan(xsol(i)) ~= 0 || isinf(xsol(i)) ~= 0
                time = 2000*it;
                return
            end
        end
        resid = A*xsol-b;
    end
    time = toc;
end
```

A.2.23 SOR5

```
function [time, xsol] = SOR5(A,b,n,w,maxIts,tol)
    tic;
    xsol = zeros(n,1);  %initial solution
    resid = 1E3;
    it = 0;
    while norm(resid) > tol
        it = it+1;
        if it > maxIts || norm(resid) > 10^30
            time = 1000*it;
            return
        end
        xsolold = xsol;          %solution of previous iteration
        for i = 1:n
            sum = 0;
            for j = 1:i-1
                sum = sum + A(i,j)*xsol(j);
            end
            for j = i+1:n
                sum = sum + A(i,j)*xsol(j);
            end
            xbar_i = (b(i)-sum)/A(i,i);
            if mod(i,5) == 1
                xsol(i) = w(1)*xbar_i + (1-w(1))*xsolold(i);
            elseif mod(i,5) == 2
                xsol(i) = w(2)*xbar_i + (1-w(2))*xsolold(i);
            elseif mod(i,5) == 3
                xsol(i) = w(3)*xbar_i + (1-w(3))*xsolold(i);
            elseif mod(i,5) == 4
                xsol(i) = w(4)*xbar_i + (1-w(4))*xsolold(i);
            else
                xsol(i) = w(5)*xbar_i + (1-w(5))*xsolold(i);
            end
        end
        for i = 1:n
            if isnan(xsol(i)) ~= 0 || isinf(xsol(i)) ~= 0
                time = 2000*it;
                return
            end
        end
        resid = A*xsol-b;
    end
    time = toc;
end
```

A.2.24 run accuracy test

```
method = @method_NMPSO; %choose the method
maxRuns = 100;
tol = [1e-4,1e-4];

successes = 0;
evals = 0;
actualEvals = 0;
successesPerFunc = zeros(40,1);
evalsPerFunc = zeros(40,1);
actualEvalsPerFunc = zeros(40,1);
i=1;

%Ackley 2D
func = @Ackley; knownBestVal = 0; dim = 2; maxIts = dim*1e2;
LB = [-32.768,-32.768];
UB = [32.768,32.768];
[successes, evals, actualEvals] = func_TestAccuracy(successes, evals, ...
    actualEvals, method, func, knownBestVal, LB, UB, dim, maxRuns, ...
    maxIts, tol);
successesPerFunc(i) = successes-sum(successesPerFunc);
evalsPerFunc(i) = evals-sum(evalsPerFunc);
actualEvalsPerFunc(i) = actualEvals-sum(actualEvalsPerFunc);
i=i+1;

...
    ...

percentage = successes/4000;
averageActualEvals = actualEvals/4000;
```

A.2.25 run SOR/GMRES test

```
[A,b] = func_DiscretizePDE(0.05);

w = zeros(9,1);
GMREStime = zeros(9,1);
methodTime = zeros(9,1);

h = 0.05;
func = @(x) GMRES1(A,b,x);
LB = [0]; UB = [2]; dim = 1;
maxRuns = 3; maxIts = 100*dim;
tol = [1e-3,1e-3];
i = 1;
[w(i,:), GMREStime(i), ~,~, ~,~,methodTime(i)] = method_NM(func, LB, UB,...
    dim, maxRuns, maxIts, tol);
i = 2;
...
    ...
```

A.2.26 Ackley Function

```
function y = Ackley(x)
    dim = numel(x);
    sum1 = 0;
    sum2 = 0;
    for i = 1:dim
        sum1 = sum1 + x(i)^2;
        sum2 = sum2 + cos(2*pi*x(i));
    end
    y1 = -20 * exp(-0.2*sqrt(sum1/dim));
    y2 = -exp(sum2/dim) + 20 + exp(1);
    y = y1 + y2;
end
```

A.2.27 Beale Function

```
function y = Beale(x)
    y1 = (1.5 - x(1) + x(1)*x(2))^2;
    y2 = (2.25 - x(1) + x(1)*x(2)^2)^2;
    y3 = (2.625 - x(1) + x(1)*x(2)^3)^2;
    y = y1 + y2 + y3;
end
```

A.2.28 Bohachevsky Function

```
function y = Bohachevsky1(x)
    y = x(1)^2 + 2*x(2)^2 - 0.3*cos(3*pi*x(1)) - 0.4*cos(4*pi*x(2)) + 0.7;
end
```

A.2.29 Booth Function

```
function y = Booth(x)
    y = (x(1)+2*x(2)-7)^2 + (2*x(1)+x(2)-5)^2;
end
```

A.2.30 Branin Function

```
function y = Branin(x)
    y1 = (x(2) - 5.1/(4*pi^2)*x(1)^2 + 5/pi*x(1) - 6)^2;
    y2 = 10*(1 - 1/(8*pi))*cos(x(1));
    y = y1 + y2 + 10;
end
```

A.2.31 Colville Function

```
function y = Colville(x)
    y1 = 100 * (x(2)-x(1)^2)^2;
    y2 = (1-x(1))^2;
    y3 = (1-x(3))^2;
    y4 = 90 * (x(4)-x(3)^2)^2;
    y5 = 10.1 * ((x(2)-1)^2 + (x(4)-1)^2);
    y6 = 19.8 * (x(2)-1)*(x(4)-1);
    y = y1 + y2 + y3 + y4 + y5 + y6;

end
```

A.2.32 De Jong Function n.5

```
function y = DeJong5(x)
    sum = 0;
    A = zeros(2,25);
    a = [-32,-16,0,26,32];
    A(1,:) = repmat(a,1,5);
    ar = repmat(a,5,1);
    ar = ar(:)';
    A(2,:) = ar;
    for i = 1:25
        sum = sum + 1 / (i + (x(1) - A(1,i))^6 + (x(2) - A(2,i))^6);
    end
    y = 1 / (0.002 + sum);
end
```

A.2.33 Dixon-Price Function

```
function y = DixonPrice(x)
    dim = numel(x);
    y = (x(1)-1)^2;
    for i = 2:dim
        y = y + i * (2*x(i)^2-x(i-1))^2;
    end
end
```

A.2.34 Drop-Wave Function

```
function y = DropWave(x)
   y = -(1 + cos(12*sqrt(x(1)^2+x(2)^2))) / (0.5 * (x(1)^2 + x(2)^2) + 2);
end
```

A.2.35 Easom Function

```
function y = Easom(x)
    y = -cos(x(1)) * cos(x(2)) * exp(-(x(1)-pi)^2 - (x(2)-pi)^2);
end
```

A.2.36 Griewank Function

```
function y = Griewank(x)
    dim = numel(x);
    sum = 0;
    prod = 1;
    for i = 1:dim
        sum = sum + x(i)^2/4000;
        prod = prod * cos(x(i)/sqrt(i));
    end
    y = sum - prod + 1;
end
```

A.2.37 Levy Function

```
function y = Levy(x)
    dim = numel(x);
    w = zeros(1, dim);
    for i = 1:dim
        w(i) = 1 + (x(i)-1)/4;
    end
    y1 = (sin(pi*w(1)))^2;
    y2 = 0;
    for i = 1:(dim-1)
        y2 = y2 + (w(i)-1)^2 * (1 + 10*(sin(pi*w(i)+1))^2);
    end
    y3 = (w(dim)-1)^2 * (1 + (sin(2*pi*w(dim)))^2);
    y = y1 + y2 + y3;
end
```

A.2.38 Matyas Function

```
function y = Matyas(x)
    y = 0.26*(x(1)^2+x(2)^2) - 0.48*x(1)*x(2);
end
```

A.2.39 Perm Function

```
function y = Perm(x)
    dim = numel(x);
    y = 0;
    for i = 1:dim
        sum = 0;
        for j = 1:dim
            sum = sum + (j+10)*(x(j)^i-1/j^i);
        end
        y = y + sum^2;
    end
end
```

A.2.40 Rastrigin Function

```
function y = Rastrigin(x)
    dim = numel(x);
    y = 10*dim;
    for i = 1:dim
        y = y + (x(i)^2 - 10*cos(2*pi*x(i)));
    end
end
```

A.2.41 Rosenbrock Function

```
function y = Rosenbrock(x)
    dim = numel(x);
    y = 0;
    for i = 1:(dim-1)
        y = y + 100 * (x(i+1)-x(i)^2)^2 + (x(i)-1)^2;
    end
end
```

A.2.42 Schaffer Function n.2

```
function y = Schaffer(x)
    y1 = ((sin(x(1)^2-x(2)^2))^2 - 0.5);
    y2 = (1 + 0.001*(x(1)^2+x(2)^2))^2;
    y = 0.5 + y1/y2;
end
```

A.2.43 Sphere Function

```
function y = Sphere(x)
    dim = numel(x);
    y=0;
    for i = 1:dim
        y = y + x(i)^2;
    end
end
```

A.2.44 Three-hump Camel Function

```
function y = ThreeHumpCamel(x)
    y = 2*x(1)^2 - 1.05*x(1)^4 + x(1)^6/6 + x(1)*x(2) + x(2)^2;
end
```

A.2.45 Zakharov Function

```
function y = Zakharov(x)
    dim = numel(x);
    sum1 = 0;
    sum2 = 0;
    for i = 1:dim
        sum1 = sum1 + x(i)^2;
        sum2 = sum2 + 0.5*i*x(i);
    end
    y = sum1 + sum2^2 + sum2^4;
end
```