university of groningen

faculty of science and engineering

mathematics and applied mathematics

Optimizing Parameters of Iterative Methods

Bachelor's Project Mathematics

July 2018

Student: E.I. Maquelin

First supervisor: Dr.ir. R. Luppes

Second assessor: Dr. A.E. Sterk

Image designed by Vexels

Abstract

Numerical optimization methods provide a way of computing the optimum of a function, even when the function is not differentiable. There are many numerical methods and it is important to choose the right one for your situation. Not only functions, but also iterative methods can be optimized. If an algorithm depends on some parameters, then the optimal parameter values, giving the minimum number of iterations required for convergence, can be found by applying a minimization method. This study discusses the idea behind, and convergence behaviour of, the Downhill Simplex Method, the Powell Methods and Particle Swarm Optimization. The methods are applied to the Rosenbrock and the Rastrigin function, two well known test functions for numerical optimization, and hereafter to some numerical algorithms depending on various parameters. The convergence behaviour of a numerical optimization method can depend highly on the given starting point. In this study, we see that especially Particle Swarm Optimization works well for the test cases where function evaluations are computationally cheap. However, for other optimization problems another method might be preferred, as the quality of the convergence behaviour of the numerical optimization method ultimately depends on the problem being optimized.

Contents

1 Introduction

Finding the extrema of a function is a common problem in mathematics. For well-defined functions this is easy to do, as one just determines the gradient and finds its zero. But if the functions are defined in such a way that it is not possible to determine the (partial) derivative(s) analytically, then the extrema cannot be computed this way; instead we can make use of numerical optimization methods. Of course, it is preferred to use a method that is fast (in terms of the number of iterations required for convergence) yet still reliable.

Now consider the situation where the function to be minimized is in fact an iterative method, taking a certain number of iterations to solve a mathematical problem. The number of iterations required for the algorithm to convergence depends on some parameters. As the dependence on the variables is in such a manner that it is not possible to take the partial derivatives, one has to apply an optimization method that numerically finds the minimum for the number of iterations.

The Bachelor Thesis "Methods of Optimization for Numerical Algorithms" by S.J. Petersen [5] addresses this problem. It focuses on the Downhill Simplex Method and Powell's Methods of optimization, both applied to functions and algorithms depending on one or two variables. But what happens when the algorithm depends on more than two variables?

This paper discusses the problem of minimizing the number of iterations of an algorithm that depends on e.g. four variables. We will investigate which of the methods of numerical minimization are robust and fast. The methods considered are the Downhill Simplex Method and Powell's Method, based on the Golden Section Search and Parabolic Interpolation. Also, Particle Swarm Optimization, a method inspired by natural concepts such as bird flocking, will be discussed. For each method the workings are briefly explained, then they are applied to some test functions with interesting features, and hereafter to an algorithm depending on four variables where the convergence behaviour of the methods is studied. Having found a method that converges to the minimum number of iterations of our algorithm depending on four variables, a few variations on the algorithm and a whole new problem are considered to see if the method also can find the minimum in these cases. We will conclude with the pros and cons of the methods and determine which is generally the preferred one.

MATLAB will be used to implement the numerical optimization methods. The relevant codes are given in Appendix A.

2 Preliminaries

As mentioned in the introduction, the numerical methods will first be applied to two test functions, namely the Rosenbrock function and the Rastrigin function, which are given below. The iterative method that will be optimized in this study depends on four variables, hence the test functions will mostly be used in their 4-dimensional form. The MATLAB implementations are given in Appendices A.1 and A.2.

2.1 Rosenbrock Function

The Rosenbrock function is often used to test the efficiency and robustness of numerical minimization methods. The variation around the global minimum is rather low, as is seen in Figure 1a. This makes convergence to the global minimum difficult, which is why this function is a good test case [7]. The Rosenbrock function is the 2-dimensional function $f(x, y) = (a - x)^2 + b(y - x^2)^2$. Usually $a = 1$ and $b = 100$, which will be assumed throughout the rest of this paper. The n-dimensional generalisation of the Rosenbrock function for even n is the following:

$$f(\mathbf{x}) = \sum_{i=1}^{n/2} 100(x_{2i-1}^2 - x_{2i})^2 + (x_{2i-1} - 1)^2$$

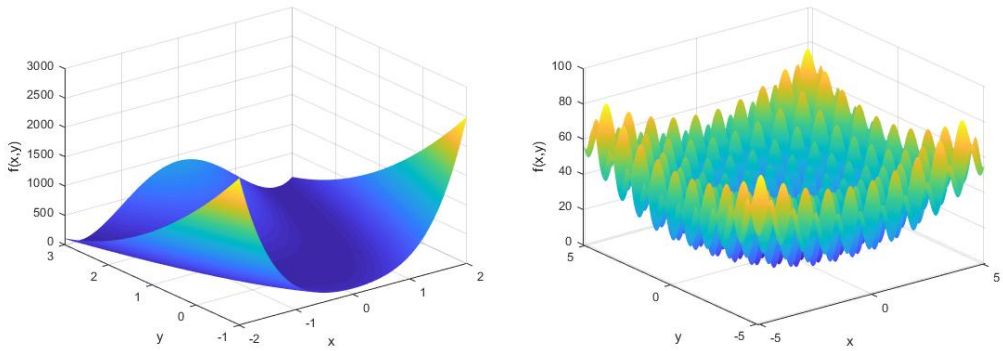
The function attains its global minimum at $\mathbf{x} = (1, 1, ..., 1)$, where $f(\mathbf{x}) = 0$, and it has a local minimum near $x = (-1, 1, ..., 1)$ [12].

2.2 Rastrigin Function

The n-dimensional Rastrigin function is defined as follows [3]:

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

The function has a global minimum at $\mathbf{x} = (0, 0, ..., 0)$, where $f(\mathbf{x}) = 0$, and it has many local minima, as can be seen in Figure 1b. Due to the large number of local minima and the steep gradients toward these minima, it is hard for optimization methods to find the global minimum and therefore a great test case [4].



(a) (b)

Figure 1: (a) The 2-dimensional Rosenbrock function. The global minimum lies in a long, narrow, parabolic shaped flat valley. (b) The 2-dimensional Rastrigin function. There are many local minima and a single global minimum.
MATLAB-codes in Appendices A.3 and A.4

2.3 Iterative Method

After the test functions in closed form, we will study how the methods behave when applying them to an iterative method that depends on a number of variables. The goal is to find the optimal values of these variables that give the minimum number of iterations for the method. The iterative method considered will be a modified version of the Newton Method; therefore, Newton's Method is explained briefly below.

2.3.1 Newton's Method

Newton's Method is used to find a root of a function $f(x)$. Note that solving $f(x) = a$ for some number a, is the equivalent to finding the root of $f(x) - a$. A starting guess x_0 is given and the tangent line to the curve is computed: $y(x) = f(x_0) + f'(x_0)(x - x_0)$. The intersection point x_1 of this line with the x-axis is the next approximation of the root. Repeating these steps gives the algorithm $x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$, provided that $f'(x_k) \neq 0$ [8].
This algorithm can be generalized into the Multivariate Newton Method, where $\mathbf{f(x)}$ is an n-dimensional multivariate equation system. An initial vector $\mathbf{x}_0$ is given and the next points are computed by solving $J_{\mathbf{f}}(\mathbf{x}_k)\boldsymbol{\delta}\mathbf{x}_k = -\mathbf{f}(\mathbf{x}_k)$ and setting $\mathbf{x}_{k+1} = \mathbf{x}_k + \boldsymbol{\delta}\mathbf{x}_k$; in other words, the next point is $\mathbf{x}_{k+1} = \mathbf{x}_k - J^{-1}(\mathbf{x}_k)\mathbf{f}(\mathbf{x}_k)$, for $k = 0, 1, 2, ...$ and where $J(\mathbf{x})$ is the Jacobian matrix (the analogue of the derivative of the one-variable case).
It is important to mention that Newton's Method in general does not converge for all possible initial guesses, but only for those that are sufficiently close to the root, obtained by for instance computing a few iterations of the bisection method [7].

2.3.2 *MyNewtonMethod*

Despite the fact that Newton's Method looks rather simple, it generally is computationally demanding when the dimension n is large, as one has to evaluate many partial derivatives. Therefore, we insert four parameters in the 2-dimensional Multivariate Newton for a specific problem and try to find the values that minimize the number of iterations required. The problem we will consider is finding the solution $(0.1, 2)$ of

$$f(x,y) = \begin{bmatrix} (10x-1)^2 + (y-2)^2 \\ (10x-1)^2(y-2)^2 - cos(5\pi xy) \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

The initial values are $x = 50$ and $y = 120$. The parameters $relax1$ and $relax2$ are used as relaxations and the Jacobian $J(x,y)$ is modified through the parameters α and β:

$$J(x,y) = \begin{bmatrix} 20(10x-1) & 2(y-2) \\ 2\alpha(10x-1)(y-2)^2 & 2(10x-1)^2(y-2) \\ +5\pi y \cdot sin(5\pi xy) & +\beta\pi x \cdot sin(5\pi xy) \end{bmatrix}$$

$$\delta = J^{-1}(x,y) \cdot f(x,y)^T$$

$$x = x - relax1 \cdot \delta(1)$$

$$y = y - relax2 \cdot \delta(2)$$

Note that in our implementation of this algorithm, called *myNewtonMethod*, in MATLAB (Appendix A.5) we use $J \backslash f^T$ instead of $inv(J) * f^T$, as the backslash calculation is quicker and has less residual error. It finds the solution using Gaussian elimination, without explicitly computing the inverse [11]. Another thing to mention is that the output of the function is the number of iterations plus a small error term, to prevent problems in the optimization methods that arise when the difference between two function values is exactly zero; for readability, the results in this study will be shown without this error term.

For $relax1 = 1$, $relax2 = 1$, $\alpha = 10$ and $\beta = 5$ we get the pure Newton Method, hence $(1, 1, 10, 5)$ would be a natural starting point for a numerical optimization method trying to find the minimum number of iterations. The number of iterations required for the pure Newton Method is 51, but this can be reduced using the Downhill Simplex Method as will be shown in the following section.

3 Downhill Simplex Method

The first method we consider is the Downhill Simplex Method, also known as the Nelder-Mead (Simplex) Search, which was proposed by John Nelder and Roger Mead in 1965. The idea of the method is to find the minimum of a function by rolling a polyhedron downhill to its lowest possible value [8].
The method does not use (partial) derivatives, only function evaluations. Even though the method is not very efficient with respect to the number of evaluations it takes, it is often a good method to use if the objective is to get a solution quickly when the computational burden of the problem is small [6].
In two dimensions, a simplex consists of three points/vertices and the line segments connecting them, i.e. a triangle. In three dimensions, the simplex is a tetrahedron (not necessarily regular). More generally, in N dimensions, a simplex consists of $N + 1$ vertices and the hyperplane segments connecting them [1]. The simplexes considered for this method are nondegenerate, meaning that they have a finite N-dimensional volume.
The algorithm is given an initial guess; however, to define the initial simplex $N + 1$ points are required and these are obtained as follows: the N-dimensional initial guess, P_0, is given; define the rest of the points by $P_i = P_0 + \Delta e_i$ where $i = 1, ..., N + 1$, e_i is a unit vector and Δ a constant.
Given the starting point, the algorithm takes a series of steps to move downhill to the lowest function value until it reaches a (at least local) minimum. These steps consist of reflections, contractions and expansions and are shown in Figure 2, resulting in the alternative name "Amoeba" for the method [6].
Conveniently, MATLAB has an implementation of the Downhill Simplex Method called "fminsearch". This function outputs the minimum when given the name of the function $f(x)$ to be minimized (a scalar-valued function of a vector variable) and an initial guess. The termination criteria are to require that the decrease in the function value and the vector distance moved in the terminating step should be fractionally smaller than some given tolerances *ftol* and *tol* respectively, whose default values are 10^{-4} [1]. Both criteria can be fooled however, when a step of the algorithm fails to go anywhere while it might not have reached the minimum yet. Therefore, it is a good idea to restart the algorithm at a point where it claims to have found a minimum, using as initial point one of the vertices of the claimed minimum [6].
The convergence of the method to the global minimum is only guaranteed in special cases. Moreover, its rate of convergence depends highly on the initial simplex. Nevertheless, this algorithm is known to be quite efficient and robust for small dimensional problems [7]. Now let us find out if this is also the case for our test functions.

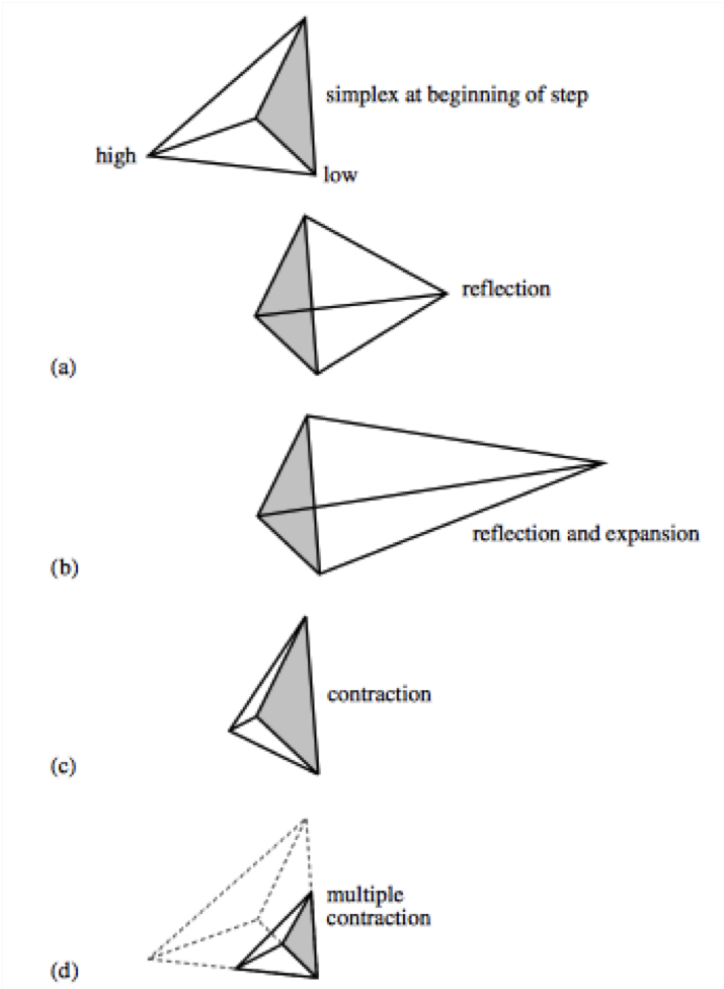


Figure 2: (a) reflection away from the point where the function value is high, i.e. the high point, (b) reflection and expansion away from the high point, (c) contraction along one dimension from the high point, (d) contraction along all dimensions toward the low point. An appropriate sequence of these steps always converges to a (at least local) minimum.
Source: Figure 10.5.1 [6]

3.1 Rosenbrock Function

The results of applying the Downhill Simplex Method to the Rosenbrock function depending on four variables are shown in Table 1. Recall that the Rosenbrock function attained its global minimum at $(1, 1, 1, 1)$, where the function value is 0.
When starting close to or at the global minimum, the method converges nicely; even when the initial point is $(-1, 1, 1, 1)$, near the local minimum. However, if the initial

point is a bit further from the global minimum, we do not get the correct answer. Instead, the method encounters a point where the gradient is relatively small, hence observes changes smaller than the given (default) tolerances, which causes the method to incorrectly claim that is has reached the minimum.
By decreasing the tolerances and allowing a higher number of function evaluations and iterations, the range of convergence can be increased. Take for example $(5, 5, 5, 5)$ as initial point, where the method did not converge to the minimum. Decreasing the tolerances from the default 10^{-4} to 10^{-7} and increasing the maximum number of function evaluations and iterations accordingly results in $(1.0000, 1.0000, 1.0000, 1.0000)$, where the Rosenbrock function takes the value $1.3115 \cdot 10^{-15}$; a very good approximation of the global minimum indeed.
The Downhill Simplex Method requires for the Rosenbrock function on average circa 1.7 function evaluations per iteration, based on the results in Table 1. As expected, decreasing the tolerances increases the number of function evaluations and iterations the method performs.

Initial Point	**Downhill Simplex Method**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(1.0000, 1.0000, 1.0000, 1.0000)	725	441
(1, 1, 1, 1)	(1, 1, 1, 1)	114	64
(-1, 1, 1, 1)	(1.0000, 1.0000, 1.0000, 1.0000)	569	339
(2, -2, 2, -2)	(1.0000, 1.0000, 1.0000, 1.0000)	593	348
(5, 5, 5, 5)	(2.8014, 7.8508, 2.4230, 5.8733)	169	91
(5, 5, 5, 5) with lowered tolerances	(1.0000, 1.0000, 1.0000, 1.0000)	1434	851
(1000, 1000, 1000, 1000)	$10^3 \cdot$ (0.0454, 2.0592, -0.0386, 1.4901)	248	134

Table 1: Estimates of the location of the global minimum $(1, 1, 1, 1)$ obtained by applying the Downhill Simplex Method to the Rosenbrock function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

3.2 Rastrigin Function

Table 2 shows the results of applying the Downhill Simplex Method to the Rastrigin function depending on four variables.
Starting the method at the global minimum $(0, 0, 0, 0)$ returns correctly the initial point. However, when the initial point is near the global minimum, but not the minimum itself, the method just returns a point very close to where it started: a local minimum.
The curious thing is that when beginning at a point far away, the result is not as bad as expected based on the behaviour observed above. For instance, starting at $(100, 100, 100, 100)$ yields a point closer to the minimum than starting at $(5, 5, 5, 5)$.

This can be explained by the fact that one of the possible steps of the Downhill Simplex Method is expansion, causing the simplex to become larger. If the algorithms start far away, it uses a lot of expansion steps, which causes the simplex to grow and allows the method to "skip over" some of the local minima close to the global minimum [5].
The Downhill Simplex Method performs on average 1.8 function evaluations per iteration for the Rastrigin function, according to Table 2. When starting further from the global minimum, the total number of function evaluations and iterations performed generally increases.

Initial Point	**Downhill Simplex Method**	**#FuncEval**	**#It**
(0, 0, 0, 0)	(0, 0, 0, 0)	18	9
(1, 1, 1, 1)	(0.9950, 0.9950, 0.9949, 0.9950)	102	58
(1.5, -1.5, 1.5, -1.5)	(1.9899, -1.9899, 1.9900, -0.9949)	167	91
(2, -2, 2, -2)	(1.9899, -1.9899, 1.9899, -1.9900)	113	64
(5, 5, 5, 5)	(4.9747, 4.9747, 4.9746, 4.9747)	128	73
(100, 100, 100, 100)	(0.0000, -1.9899, 0.9949, -0.9950)	320	182

Table 2: Estimates of the location of the global minimum $(0, 0, 0, 0)$ obtained by applying the Downhill Simplex Method to the Rastrigin function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

For the Rosenbrock function the results could be improved by lowering the tolerances, for the Rastrigin function this unfortunately does not give better convergence behaviour.

3.3 *MyNewtonMethod*

The location of the global minimum of *myNewtonMethod* is not known yet. Therefore, the function value of *myNewtonMethod*, i.e. the number of iterations it takes, at the location where the Downhill Simplex Method claims to have found the minimum is also given in Table 3. Note that the method reduces the number of iterations compared to the pure Newton Method which corresponds to the variables $(1, 1, 10, 5)$ and 51 iterations.
We see in the table that for most inputs the method just returns a point close to the initial point. Decreasing the tolerances also does not result in better convergence. This behaviour is similar as to what happened for the Rastrigin function; probably because there again are many local minima.

Initial Point	**Downhill Simplex Method**	**Func. Value**	**Time (sec)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(2.0234, 1.4949, -1.9028, -1.3438)	19	0.828	373	159
(1, 1, 1, 1)	(1.0542, 0.9650, 1.0057, 1.0378)*	37	0.436	801	224
(1, 2, 3, 4)	(0.9981, 2.0024, 3.0047, 4.0035)	22	0.178	343	137
(2, 2, -3, 5)	(2.0000, 2.0000, -3.0002, 5.0000)	6	0.037	284	113
(1, 1, 10, 5)	(0.9940, 1.0217, 10.3409, 5.0470)	38	0.102	148	56
(2, 2, 0, 5)	(2.0000, 2.0000, 0, 5.0236)	2	0.026	313	106

Table 3: Estimates of the minimum obtained by applying the Downhill Simplex Method to *myNewtonMethod* for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.
**Exiting: Maximum number of function evaluations has been exceeded*

A special situation is indicated by * in Table 3: the method encounters some difficulties when starting at $(1, 1, 1, 1)$ and gives the error message that it exceeds the maximum number of function evaluations. Increasing the maximum does not help, as the method makes futile iterations indefinitely, while already having reached a local minimum from which it cannot get away. To solve this *ftol* is set to 10^{-3} instead of the default 10^{-4}; then the method recognizes that it has reached a minimum and gives the following result: $(1.0542, 0.9650, 1.0057, 1.0378)$ where the function value is 38 iterations.
The Downhill Simplex Method requires for *myNewtonMethod* on average about 2.8 function evaluations per iteration. Each function evaluation in turn performs a certain number of iterations of the modified Newton Method, depending on the values given for the parameters of *myNewtonMethod*.
The results do not show convergence to a single point. Even though we now know that the global minimum is at most 2, we do not know its value exactly yet. Therefore, in the next section another numerical optimization method is introduced for comparison, called Powell's Method.

4 Powell's Methods

Powell's Methods [1, 6], in contrast to the Downhill Simplex Method, do make use of algorithms performing 1-dimensional minimization. We will consider two 1-dimensional optimization methods that do not use the (partial) derivative of the function: the Golden Section Search and Parabolic Interpolation [5]. The relevant MATLAB-codes can be found in Appendices A.6 to A.9.
First the methods are explained, then they are both applied to the test functions and hereafter to the algorithm, as was done for the Downhill Simplex Method.

4.1 Golden Section Search

The Golden Section Search uses the bracketing of the minimum. A minimum of the function $f(x)$ is bracketed by the points $a < b$ when there is a point c, such that $a < c < b$ with $f(c) < f(a)$ and $f(c) < f(b)$. The aim is to find a sequence of approximations to the minimum such that it is contained in the interval, the length of the interval is reduced at each iteration and the number of function evaluations is as low as possible.
At each step of the process there is a bracketing triplet (a, t_1, b) and a new point t_2 is generated. t_2 is a fraction $r \approx 0.38197$ into the larger of the subintervals $[a, t_1]$ or $[t_1, b]$. By comparing the function values at t_1 and t_2, a new bracketing triplet is selected: if $f(t_1) < f(t_2)$, then the new bracketing triplet is (a, t_1, t_2); if $f(t_1) > f(t_2)$, then the new bracketing triplet is (t_1, t_2, b).
The process repeats these steps, having at each step the width of the search interval reduced by a fraction $1 - r \approx 0.61803$. The value of r is chosen such that the possible new intervals are of the same size. $1 - r$ is the reciprocal of the golden section $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$, which is where the algorithm gets its name from [1].
The process of bracketing is continued until the distance between the two outer points of the bracketing triplet is smaller than some given tolerance. This method guarantees that each step brackets the minimum with an interval only 0.61803 times the size of the preceding interval. Moreover, the convergence is linear [6].

4.2 Parabolic Interpolation

The Golden Section Search is designed to handle the worst possible case of function minimization. But if the function is close to parabolic near its minimum, then the parabola fitted through any three points takes us in a single step (very close) to the minimum [6].
A parabola $P(x)$ interpolates a set of data points $(x_1, y_1), ..., (x_n, y_n)$ if it passes through those points, i.e. if $P(x_i) = y_i$ for $i = 1, ..., n$. Moreover, a unique interpolating polynomial always exists if the x-coordinates of the points are distinct [8].
Since the x-coordinate rather than the y-coordinate is needed, the procedure used is

technically called Inverse Parabolic Interpolation. The formula for the x-coordinate that is the minimum of a parabola through three points y_1, y_2 and y_3 is $x = x_2 - \frac{1}{2}\frac{(x_2-x_1)^2(y_2-y_3)-(x_2-x_3)^2(y_2-y_1)}{(x_2-x_1)(y_2-y_3)-(x_2-x_3)(y_2-y_1)}$. The formula fails when the denominator is zero, which happens only if the three points lie on the same straight line [6].
The Parabolic Interpolation in Powell's Method is performed by the so called Coggin's Method, which performs the line search procedure for optimization using Parabolic Interpolation.

4.3 Multidimenionsal Optimization

Powell's Methods perform multidimensional optimization by carrying out the 1-dimensional optimization in a number of different directions until a minimum is found [5]. A trivial way of doing this is the following: take the set of unit vectors as directions; find the minimum along the first direction, go from there along the second direction to its minimum, etc.; go through the set of directions as many times as necessary, until the function does not decrease anymore. This procedure is not too bad for many functions, for some however, it is very inefficient.
Consider for instance a function whose contour map is a long, narrow valley not parallel to any of the unit vectors. If you try to go down the valley along these basis vectors, you will end up taking a series of very small steps. In general, in N dimensions, if the second derivatives of a function are much larger in some directions than in others, then many cycles through all N basis vectors are needed to get anywhere. This example indicates that it would be wise to choose a different set of directions.
We would like a direction set that either has some very convenient directions that take us far along narrow valleys, or has some "non-interfering"/conjugate directions with the property that minimization along one direction is not being undone by subsequent minimization along another direction, so that endlessly cycling through set of directions is avoided.
Powell discovered a direction set method that produces N mutually conjugate directions. There was however a problem with his algorithm for choosing direction vectors. Namely, its procedure for replacing direction vectors tended to produce sets of directions that become linearly dependent. If this happens, the procedure finds a minimum of the function over a subspace instead of the whole space, hence giving the wrong answer. The problem was solved by discarding the direction that caused the largest decrease in the function value. It may sound contradictory, but it is done because dropping it decreases the chance of linear dependence. This last procedure will be used in the Powell Methods throughout the paper.
Powell's Method is almost surely faster than the Downhill Simplex Method in most applications, if it converges [6].

4.4 Rosenbrock Function

Golden Section Search

In Table 4 the results are shown for applying Powell's Method using the Golden Section Search to the Rosenbrock function. The method converges to the global minimum when starting nearby. Moreover, we can start further away than with the Downhill Simplex Method and still obtain the minimum, without having to decrease the tolerance.
When we start very far away, the results are nowhere near the minimum. But also here, decreasing the tolerance improves the results. Even when starting as far away as $(1000, 1000, 1000, 1000)$, we only have to decrease the tolerance from the default 10^{-4} to 10^{-8} to find the minimum, not yet having to increase the maximum number iterations. The number of function evaluation and iterations does increase when lowering the tolerance, but the iterations do not yet exceed the default maximum value in this case. The average number of function evaluation the method uses per iteration is 101.

Initial Point	**Powell's Method (Golden Section Search)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(1.0000, 1.0000, 1.0000, 1.0000)	1726	18
(1, 1, 1, 1)	(1.0000, 1.0000, 1.0000, 1.0000)	70	1
(-1, 1, 1, 1)	(1.0000, 1.0000, 1.0000, 1.0000)	1911	20
(2, -2, 2, -2)	(1.0000, 1.0000, 1.0000, 1.0000)	2755	28
(5, 5, 5, 5)	(1.0000, 1.0000, 1.0000, 1.0000)	1636	17
(10, 10, 10, 10)	(0.9803, 0.9585, -0.7732, 0.5970)	3240	31
(100, 100, 100, 100)	(2.0706, 4.2876, 0.3143, 0.0985)	4601	35
(1000, 1000, 1000, 1000)	(-31.6226, 999.9889, -31.6226, 999.9895)	224	2
(1000, 1000, 1000, 1000) with lowered tolerance	(1.0000, 1.0000, 1.0000, 1.0000)	28661	132

Table 4: Estimates of the location of the global minimum $(1, 1, 1, 1)$ obtained by applying Powell's Method based on the Golden Section Search to the Rosenbrock function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

Coggin's Method

The Powell Method based on Coggin's Method converges for a larger range than the Golden Section Search does for the same tolerance. If the initial point is at a great distance from the actual minimum, the method does not converge to the correct point, though its result is not as far from the minimum as with the Golden Section Search and Downhill Simplex Method. The convergence can even be improved by lowering the tolerance, though not as easily as before. Lowering the tolerances greatly increases the

number of function evaluations and iterations, hence it takes rather long for the method to give a solution. For $(100, 100, 100, 100)$ we can lower the tolerance to obtain a result within reasonable time, but for a point as far away as $(1000, 1000, 1000, 1000)$ it takes very long because of the large number of iterations required. The method performs for the Rosenbrock function on average circa 80 function evaluation per iteration.

Initial Point	**Powell's Method (Coggin's Method)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(1.0000, 1.0000, 1.0000, 1.0000)	1626	16
(1, 1, 1, 1)	(1.0000, 1.0000, 1.0000, 1.0000)	42	1
(-1, 1, 1, 1)	(1.0000, 1.0000, 0.9986, 0.9973)	1042	12
(2, -2, 2, -2)	(1.0000, 1.0000, 1.0000, 1.0000)	3629	36
(5, 5, 5, 5)	(1.0000, 1.0001, 1.0000, 1.0000)	3968	45
(10, 10, 10, 10)	(1.0000, 1.0000, 1.0000, 1.0001)	2194	27
(100, 100, 100, 100)	(0.2006, 0.0400, 2.1794, 4.7500)	2493	36
(100, 100, 100, 100) with lowered tolerance	(1.0000, 1.0000, 1.0000, 1.0000)	6679	65
(1000, 1000, 1000, 1000)	(-4.4770, 20.0446, 0.0260, -0.0036)	3579	52

Table 5: Estimates of the location of the global minimum $(1, 1, 1, 1)$ obtained by applying Powell's Method based on Coggin's Method to the Rosenbrock function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

We can conclude that both versions of the Powell Method work pretty good for the Rosenbrock function, and decreasing the tolerance usually results in a better estimate of the global minimum, though increasing the time.
An interesting observation when comparing the Downhill Simplex Method and the Powell Method, is that when starting at the global minimum, the former finds the exact minimum, while the latter gives a close approximation to it, but not the exact point. This is because the Downhill Simplex Method evaluates the function at the initial point, while the Powell Method works with bracketing intervals around the given point.

4.5 Rastrigin Function

Golden Section Search

For the Rastrigin function the convergence behaviour of Powell's Method based on the Golden Section Search is similar to the that of the Downhill Simplex Method, as can be seen in Table 6. However, there are two initial points that cause some problems,

indicated in the table by *.
When starting at the global minimum, Powell's Method gets into trouble and reaches the maximum number of stages (i.e. iterations performed by the Powell Method), even when increasing this number by a large amount. What happens is that it circles around the minimum, where the function values are very close to each other, but the distance between two consecutive estimates is slightly above the given tolerance. When decreasing the tolerance, we do get a better approximation of the location of the minimum, namely $10^{-10} \cdot (0.9241, 0.9241, 0.9241, 0.9241)$. However, it keeps oscillating, hence still exceeding the maximum number of stages because the decrease in the vector distance moved stays just above the given tolerance.
When starting as far away as the point $(100, 100, 100, 100)$, the method also exceeds the maximum number of stages. Increasing the maximum again does not help as the method oscillates between two values. Increasing the tolerance does work in this case as it prevents oscillations and allows the method to terminate, resulting in the point $(0.0011, -0.0000, -0.0000, -0.0000)$, close to the minimum. A higher tolerance results in less function evaluations and iterations. The average number of function evaluations the method uses per iteration is approximately 97, based on the results in Table 6.

Initial Point	**Powell Method (Golden Section Search)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	$10^{-4}\cdot$ (-0.2372, -0.2372, -0.2372, -0.2372)*	345001	5000
(1, 1, 1, 1)	(0.9949, 0.9950, 0.9950, 0.9950)	160	2
(1.5, -1.5, 1.5, -1.5)	(0.9949, -0.9950, 0.9950, -0.9950)	210	2
(2, -2, 2, -2)	(1.9899, -1.9899, 1.9899, -1.9899)	158	2
(5, 5, 5, 5)	(4.9747, 4.9747, 4.9747, 4.9747)	172	2
(50, 50, 50, 50)	(0.9950, 0.9949, 0.9950, 0.9950)	272	2
(100, 100, 100, 100)	$10^{-4}\cdot$ (0.1987, 0.1987, 0.1987, 0.1987)*	345125	5000
(100, 100, 100, 100) with increased tolerance	(0.0011, -0.0000, -0.0000, -0.0000)	231	3

Table 6: Estimates of the location of the global minimum $(0, 0, 0, 0)$ obtained by applying Powell's Method based on the Golden Section Search to the Rastrigin function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.
**warning reached max nr of stages*

Coggin's Method

The results of applying Powell's Method based on Coggin's Method to the Rastrigin function are shown in Table 7. The results are very similar to those in Table 6, but

without the problems that arose for the Golden Section Search. Also, the average number of function evaluations per iteration is lower, namely 47. The convergence of Coggin's Method appears to be the best so far, though still far from optimal. This makes sense as the Rastrigin function contains quadratic terms (and no higher order terms) and Coggin's Method performs Parabolic Interpolation [5].

Initial Point	**Powell's Method (Coggin's Method)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(0, 0, 0, 0)	30	1
(1, 1, 1, 1)	(0.9950, 0.9950, 0.9950, 0.9950)	88	2
(1.5, -1.5, 1.5, -1.5)	(0.9950, 0.9950, 0.9950, 0.9950)	111	2
(2, -2, 2, -2)	(1.9899, -1.9899, 1.9899, -1.9899)	91	2
(5, 5, 5, 5)	(4.9747, 4.9747, 4.9747, 4.9747)	99	2
(100, 100, 100, 100)	$10^{-7}\cdot$ (0.1298, 0.1298, 0.1767, 0.1298)	168	3

Table 7: Estimates of the location of the global minimum $(0, 0, 0, 0)$ obtained by applying Powell's Method based on Coggin's Method to the Rastrigin function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

4.5.1 Improvements

For both versions of Powell's Method applied to the Rastrigin function, we have that starting far away gives better results than when starting relatively close to the global minimum. This is because when starting far away, the step size of both the Golden Section Search and Coggin's Method is larger and hence it skips over some local minima. Also, for both versions, decreasing the tolerance does not result in better convergence behaviour.

So what goes wrong for the Rastrigin function? Let us first look at the 1-dimensional Rastrigin function and investigate what happens when applying the 1-dimensional optimizations.

The 1-dimensional Rastrigin function is $f(x) = 10 + x^2 - 10cos(2\pi x)$. Since the cosine function has a period of 2π, the $cos(2\pi x)$ part has a period of 1. Moreover, because of the x^2 term, we have that each local minimum is smaller than its predecessor when moving in the direction of the origin, see Figure 3.

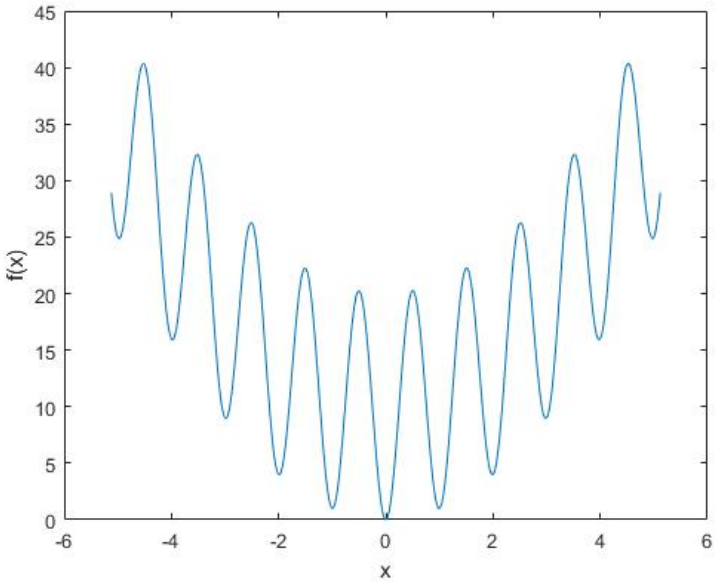


Figure 3: The 1-dimensional Rastrigin function: $f(x) = 10 + x^2 - 10cos(2\pi x)$. MATLAB-code in Appendix A.10

First, the Golden Section Search is modified such that it converges for the 1-dimensional Rastrigin function. The problem is that the method gets stuck in a local minimum instead of converging to the global minimum. This happens because the step size is so small that it does not get out of the neighbourhood of the local minimum near the initial point. A possible solution is the following: let x_1 and x_2 bracket the minimum; increase the size of the direction vector, hence increasing the size of the default step size, and recompute the bracketing interval until $|(x_2 - x_1)| \geqslant 2$ (code in Appendix A.11). This condition ensures that the bracketing interval contains multiple minima and then the method will converge to the smaller one.
However, this might not be enough, since even though it now does not converge to the first minimum it encounters, it could get stuck at the second one. This is solved by applying the algorithm again on the output, which gives convergence to a minimum smaller than the one it started at, providing that we did not start at the global minimum. Applying the method multiple times eventually results in the global minimum (code in Appendix A.12).

Initial Point	**Powell's Method (Adapted Golden Section Search)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	$10^{-7}\cdot$ (0.1848, 0.1848, 0.1848, 0.1848)	386	1
(1, 1, 1, 1)	$10^{-3}\cdot$ (-0.2548, -0.2548, -0.2548, -0.2548)	783	2
(1.5, -1.5, 1.5, -1.5)	$10^{-3}\cdot$ (-0.1678, 0.3257, -0.1678, 0.3257)	1129	2
(2, -2, 2, -2)	$10^{-3}\cdot$ (0.3294, -0.3294, 0.3294, -0.3294)	1464	3
(5, 5, 5, 5)	$10^{-3}\cdot$ (0.1982, 0.1982, 0.1982, 0.1982)	1635	2
(100, 100, 100, 100)	$10^{-3}\cdot$ (0.3758, 0.3758, 0.3758, 0.3758)	891	2

Table 8: Estimates of the location of the global minimum $(0, 0, 0, 0)$ obtained by applying Powell's Method based on the adapted Golden Section Search to the Rastrigin function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

Coggin's Method is modified in a similar way (code in Appendix A.13). It does not need the addition of applying the algorithm multiple times on its own output, it already converges after one try. Recall that the convergence of Coggin's Method was already better than the Golden Section Search before modifying the methods.

Initial Point	**Powell's Method (Adapted Coggin's Method)**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(0, 0, 0, 0)	42	1
(1, 1, 1, 1)	(0, 0, 0, 0)	87	2
(1.5, -1.5, 1.5, -1.5)	$10^{-15}\cdot$ (-0.5826, 0.4429, -0.8598, -0.3051)	203	4
(2, -2, 2, -2)	(0, 0, 0, 0)	87	2
(5, 5, 5, 5)	(0, 0, 0, 0)	87	2
(100, 100, 100, 100)	$10^{-9}\cdot$ (-0.8510, -0.8513, -0.8511, -0.8511)	184	3

Table 9: Estimates of the location of the global minimum $(0, 0, 0, 0)$ obtained by applying Powell's Method based on the adapted Coggin's Method to the Rastrigin function for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

The Powell Methods based on these adapted algorithms exhibit far better convergence behaviour than before, as is seen in Tables 8 and 9. The Powell Method based on the adapted Golden Section Search performs on average approximately 516 function evaluations per iteration. Note that the number of iterations remained almost equal, while the number of function evaluations increased compared to the Powell Method based on the original Golden Section Search. The Powell Method based on the adapted Coggin's Method still requires about 47 function evaluations per iteration, like it did before it was modified.

4.6 *MyNewtonMethod*

The results for applying both versions of Powell's Method to *myNewtonMethod* are shown in Tables 10 and 11. They show a high dependency on the initial guess and no convergence to a single point. As with the Rastrigin function we often just obtain a point close to the initial point. From the tables the average numbers of function evaluations per iteration can be computed and are 72 and 78 for Powell's Method based on the Golden Section Search and Coggin's Method, respectively.
Decreasing the tolerance does result in a slightly lower function-value, but still does not give the global minimum, which we know it to be at most 2 from applying the Downhill Simplex Method.

Initial Point	**Powell's Method (Golden Section Search)**	**Func. Value**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(3.0128, 0.9401, 0.5744, -0.5740)	18	10130	137
(1, 1, 1, 1)	(1.4872, 2.5488, 1.1310, 2.7563)	14	44260	618
(1, 2, 3, 4)	(1.1966, 1.3868, 1.9854, 5.5307)	24	71773	994
(2, 2, -3, 5)	(1.9996, 1.9995, -2.9634, 4.9878)	6	15254	215
(1, 1, 10, 5)	(1.8885, 2.0011, 8.2079, 7.7844)	13	69581	961
(2, 2, 0, 5)	(2.0182, 2.0327, 0.0825, 5.0964)	6	21505	306

Table 10: Estimates of the as yet unknown global minimum obtained by applying Powell's Method based on the Golden Section Search to *myNewtonMethod* for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

Initial Point	**Powell's Method (Coggin's Method)**	**Func. Value**	**#Func-Eval**	**#It**
(0, 0, 0, 0)	(2.7371, 1.0221, 0.5702, -0.1296)	17	330	4
(1, 1, 1, 1)	(2.0219, 2.0723, 0.2969, 5.2520)	6	1597	20
(1, 2, 3, 4)	(1.7498, 1.9817, 3.0079, 5.9358)	10	733	9
(2, 2, -3, 5)	(1.9998, 2.0000, -2.9340, 5.0000)	6	827	11
(1, 1, 10, 5)	(1.8772, 1.7879, 9.9765, 4.6461)	11	699	9
(2, 2, 0, 5)	(2.0000, 1.9999, -0.0008, 5.0002)	3	208	3

Table 11: Estimates of the as yet unknown global minimum obtained by applying Powell's Method based on Coggin's Method to *myNewtonMethod* for various initial points. #FuncEval and #It give the number of function evaluations and iterations performed by the method.

For the Rastrigin function we could modify the 1-dimensional optimization methods used in the Powell Methods such that they did converge to the global

minimum. The situation for *myNewtonMethod* unfortunately is not as simple as for the Rastrigin function. We now do not have equidistant minima, hence requiring the bracketing interval to have a certain length does not work. Due to the irregularity of *myNewtonMethod*, there does not seem to be an equivalent way of modifying the methods such that we get convergence.
In the following section we will try another way of applying the three methods discussed so far in order to find the global minimum.

5 Grid

Another way to search for the global minimum of *myNewtonMethod* is by using a grid (code in Appendix A.14). We begin simple, by just evaluating the function at all grid points and selecting the lowest function value. First, we apply a grid-wise search where each variable in *myNewtonMethod* goes from -100 to 100 with a step size of 2. This results in a minimum of three iterations found for $relax1 = 2$, $relax2 = 2$, $\alpha = 0$ and $\beta = 6$. Then a more focused grid-wise search is performed, with each variable going from -1 to 9 with step size 0.5. The lowest value found is two iterations, for $relax1 = 2$, $relax2 = 2$, $\alpha = 0$ and $\beta = 5$. Such a simple method already finds a very low function value, which equals the lowest value found by the Downhill Simplex Method, and is the lowest result obtained so far. Is this the global minimum or is it possible to reach the solution of the problem in only one iteration?

5.1 Global Minimum

Reaching the solution in only one iteration is indeed possible. This is shown by using the fact that the solution of the problem is $x = 0.1$, $y = 2$ and by fixing $\alpha = 0$ and $\beta = 5$. The values for α and β can be set to any value we like; we have chosen these values in order to see how far the global minimum is from the point found by the grid-search. Now we can compute the values for $relax1$ and $relax2$ such that we get the desired result of only one iteration.

$$
\begin{aligned}
\text{Set } \alpha &= 0,\ \beta = 5 \\
x &= -50 \\
y &= 120 \\
f(x,y) &= \begin{bmatrix} (10x-1)^2 + (y-2)^2 \\ (10x-1)^2(y-2)^2 - cos(5\pi xy) - 1 \end{bmatrix} \\
J(x,y) &= \begin{bmatrix} 20(10x-1) & 2(y-2) \\ 2\alpha(10x-1)(y-2)^2 & 2(10x-1)^2(y-2) \\ +5\pi y \cdot sin(5\pi xy) & +\beta\pi x \cdot sin(5\pi xy) \end{bmatrix} \\
\delta &= J^{-1}(x,y) \cdot f(x,y)^T \\
relax1 &= (x-0.1)/\delta(1) \approx 1.999999999936510 \\
relax2 &= (y-2)/\delta(2) \approx 2.000000001144512
\end{aligned}
\tag{2}
$$

Thus, the global minimum is one iteration and is obtained when $relax1 = 1.999999999936509$, $relax2 = 2.000000001144512$, $\alpha = 0$ and $\beta = 5$. Note that changing the value for α also gives other values for $relax1$ and $relax2$. β is completely arbitrary, as it is multiplied by a sine term which equals zero for the

initial x and y values. This means that there are infinitely many ways to reach the minimum of *myNewtonMethod* in this case. Recall that the pure Newton Method requires 51 iterations, hence inserting the parameters with the correct values in the method greatly reduces the number iterations.

When applying the three methods with as initial point the values found in Equation 2, only the Downhill Simplex Method finds the minimum of one iteration. The Powell Methods have trouble locating the global minimum when starting there, as they use bracketing intervals around the given point. The function value around the minimum is very sensitive to changes in the variables, which makes it hard for Powell's Methods to find the exact value of the global minimum within this interval. Especially the Powell Method based Golden Section Search struggles, as it takes too large steps to find the global minimum; after decreasing the default step size from 10^{-2} to 10^{-15} and decreasing the tolerance to 10^{-15} as well, we do manage to obtain the global minimum. When using Coggin's Method it is also possible to find the minimum when decreasing the tolerance and increasing the maximum number of iterations.

5.2 Applying Methods to a Grid

Knowing the global minimum of *myNewtonMethod*, we realize that we have not been able to find this minimum with any of our optimization methods yet. However, there is another way to use the grid than the simple manner suggested above. Instead of evaluating the function at the grid points, apply a numerical method to each point of the grid. The strength of a grid is that it tests many initial points, increasing the odds of finding the one that returns the global minimum. When using a grid where all variables go from -1 to 4 with step size 1, only the Downhill Simplex Method finds the global minimum, the Powell Methods do not.

To conclude, we have found the global minimum using a grid and the Downhill Simplex Method. The drawback of using a grid is that it takes rather long, circa one hour. When knowing where the minimum is in advance or which point to use as initial point, we can find the minimum very fast when applying the Downhill Simplex Method to it. However, in real life we often do not know where the minimum is located beforehand. Therefore, let us try a new optimization method: Particle Swarm Optimization, to see if we can find the global minimum faster and without the prior knowledge of its location.

6 Particle Swarm Optimization

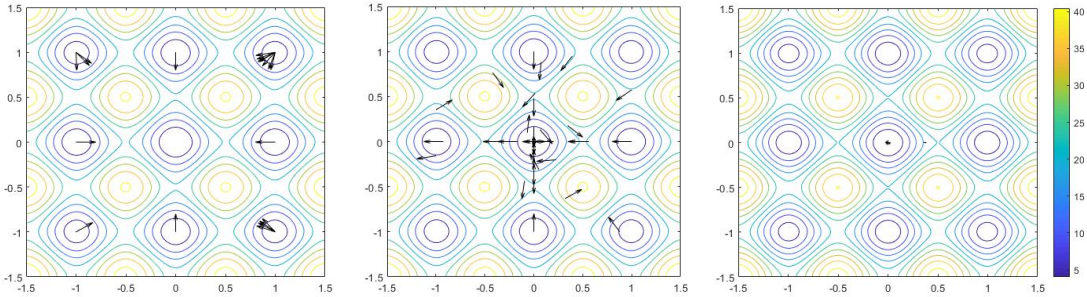
The numerical optimization method called Particle Swarm Optimization (PSO) [2, 4] was introduced by Kennedy and Eberhart in 1995, inspired by natural concepts such as bird flocking and fish schooling.
PSO minimizes an objective function f by iteratively trying to improve a candidate solution. It uses a set of candidate solutions, called particles, that move through the search area to find the global minimum of the function. Maximization can be performed by considering the function $-f$ instead. The movement of a particle is determined by some simple mathematical formulas over the particle's position and velocity. The trajectory of the particle depends on the current particle velocity as well as on the histories of the particle and its neighbours.
The method has become very popular due to its search efficiency, even for high dimensional objective functions with multiple local minima. Also, it does not make use of the (partial) derivatives of the objective function. The algorithm is simple and flexible while performing an efficient search for minima, even for tricky functions. This makes it sound like a very promising candidate for our optimization problem *myNewtonMethod*.
An important remark is that PSO does not guarantee an optimal solution is ever found, however, many improvements have been suggested of the years to attain convergence [2].

PSO finds the global minimum of real, scalar valued objective functions defined on a certain domain. The set consisting of all particles (the candidate solutions) is called the swarm, hence the name of the method. It takes advantage of the particles' ability to explore and optimize toward the minimum, by having the particles communicate their findings amongst each other [4]. Each particle in the swarm moves through the domain looking for the global minimum. The movement of a particle i is influenced by both its own best known position in the domain (denoted $\mathbf{p}_i$), as well as by the best known position of the particles in its neighbourhood (denoted $\mathbf{g}_i$). When we use the simple setup where the neighbourhood of a particle is the whole swarm, then the findings of a particle are shared with all other particles, hence $\mathbf{g} = \mathbf{g}_i$ is the best known position of the entire swarm. The particles then move through the search area according to the following updating rules for their coordinates and velocity:

$$\begin{aligned} \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1), \\ \mathbf{v}_i(t+1) &= \mathbf{v}_i(t) + \phi_1 R(\mathbf{p}_i - \mathbf{x}_i(t)) + \phi_2 R(\mathbf{g} - \mathbf{x}_i(t)). \end{aligned} \tag{3}$$

Here, t denotes the time and R a random diagonal matrix where each diagonal element is a uniform random number in $[0, 1]$ and is regenerated for each evaluation. The purpose of these random numbers is to imitate the unpredictable behaviour of nature swarms. The parameters $\phi_1 \geqslant 0$ and $\phi_2 \geqslant 0$ are to be determined in advance, the recommended values being $\phi_1 = \phi_2 = 2$.
The velocity $\mathbf{v}_i$ is updated in the direction of $\mathbf{p}_i$ weighted by ϕ_1, in the direction of $\mathbf{g}$

weighted by ϕ_2 and randomness is introduced by R. This causes the swarm to move to where good solutions were found in the past [2]. See Figure 4 for an illustration of Particle Swarm Optimization.



(a) Initial phase of the search, the particles are randomly placed in the search space and given an initial velocity

(b) Situation after a couple of iterations, the particles are moving toward the minimum

(c) Final phase of the search, the minimum is found at (0,0)

Figure 4: An example of PSO: PSO applied to the 2-dimensional Rastrigin function; the particles, indicated by arrows, search for the location of the global minimum (0,0) of the function.

6.1 Parameters

The choice for the size of the swarm is an educated guess, depending on the researchers past experience with PSO and the function being optimized. Moreover, increasing the sample size leads to a greater computation time. This choice is very important, as having enough particles initialized is crucial for the effectiveness of the method.

We also have to select which topology to use for the communication within the particle swarm: to use either circle or wheel topologies. In circle (aka local best) topology, the particles communicate only with their closest neighbours; while wheel (aka global best) topology allows all particles to communicate with one common particle in order to determine the movement direction. The communication between particles affects the swarm's movement and hence its convergence.
Eberhart and Kennedy discovered that for the Rastrigin function the wheel/global best topology performs better than circle topology, hence we will use this topology.

The search area of the swarm needs to be restricted somehow. This was initially done by limiting the velocity of the particles, while making sure not to restrict it too much as this could cause the swarm to get trapped in a local optimum.

Another way of doing this redefining the search area is by using an inertia weight, which improves the stability of the search method [4]. This improvement was suggested by Shi and Eberhart [2], who introduced an inertia weight, w, to control the amount by which the current velocity affects the velocity of the next step. Using the inertia weight, the updating rules in Equation 3 become

$$\begin{aligned}\mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1),\\ \mathbf{v}_i(t+1) &= w\mathbf{v}_i(t) + \phi_1 R(\mathbf{p}_i - \mathbf{x}_i(t)) + \phi_2 R(\mathbf{g} - \mathbf{x}_i(t)).\end{aligned}$$

The effect of w being large is that it makes it relatively hard to change the particle's direction, resulting in scattering the swarm over the search area. This is desirable in the initial phase of the search when we do not yet know in which part of the domain the global minimum resides. After the initial phase, the search should be restricted to smaller, promising regions for a finer search, requiring a smaller inertia weight. Therefore, it is common to start with $w = 1$ and to gradually reduce it (e.g. exponentially) to $w = 0$. As the value of the inertia weight decreases, the search area gets smaller [2].

PSO is in fact a particular case of Generalized Particle Swarm Optimization (GPSO), which has the updating equations:

$$\begin{aligned}\mathbf{x}_i(t+\Delta t) &= \mathbf{x}_i(t) + \Delta t\mathbf{v}_i(t+\Delta t),\\ \mathbf{v}_i(t+\Delta t) &= (1-\Delta t(1-w))\mathbf{v}_i(t) + \phi_1 R\Delta t(\mathbf{p}_i - \mathbf{x}_i(t))\\ &\quad + \phi_2 R\Delta t(\mathbf{g} - \mathbf{x}_i(t)).\end{aligned}$$

The Δt factor influences the stability of the method, determining whether the method focuses on the area around the global best solution, or whether it searches the whole space. It can be used to prevent the method from getting trapped in local minima. PSO is obtained when Δt is set to 1. PSO with an inertia weight already works very good for the problems we are trying to solve, as will be shown below; hence the improvements suggested by GPSO are not necessary and we will just use PSO, i.e. $\Delta t = 1$ [10].

6.2 Applying PSO

Over the years, many other improvements have been suggested for the PSO algorithm. Let us leave them be for now and find out how the basic algorithm performs. The first three rows in Table 12 show the results of applying PSO without an inertia weight to the test functions and our algorithm. The result for the Rosenbrock function is not the global minimum, while for the Rastrigin function we do get a rather good approximation. The method claims that the minimum number of iterations for *myNewtonMethod* is two, instead of one.
The results obtained by the basic PSO are not very good. Therefore, we also try the improved version which makes use of an inertia weight that decreases exponentially,

leaving the lower and upper bounds and swarm size unchanged. As is shown in the next three rows of the table, the convergence behaviour of this version of the method is much better for the test functions. Moreover, the method now does find the correct global minimum of *myNewtonMethod*. This improved PSO takes approximately 32 minutes to find the minimum of *myNewtonMethod*, which is almost twice as fast as applying the Downhill Simplex Method to the grid.

Problem	**LB**	**UB**	**Swarm Size**	**Output PSO**	**Func. Value**	**Time (sec)**
Rosenbrock (without w)	-5	10	500	(1.8419, 3.3817, 0.9009, 0.8078)	0.7321	1.0148
Rastrigin (without w)	-5.12	5.12	500	(0.0144, 0, 0, 0)	0.0413	0.9226
myNewton-Method (without w)	-1	10	1000	(2, 2, 0, 5)	2	3027.2
Rosenbrock (with w)	-5	10	500	(1.0000, 1.0000, 1.0000, 1.0000)	$1.9328 \cdot 10^{-14}$	4.7443
Rastrigin (with w)	-5.12	5.12	500	$10^{-8} \cdot$ (0.1533, -0.2509, 0.1539, -0.1434)	0	1.2542
MyNewton-Method (with w)	-1	10	1000	(2.0000, 2.0000, 0.0000, 8.0187)	1	1934.0

Table 12: Results of applying PSO to various objective functions. LB and UB are the lower respectively upper bounds for each variable, defining the search area. The Swarm Size is the number of particles initialized. Func. Value gives the function value at the point found by PSO.

Because the PSO method works better for our test functions and *myNewton-Method* with an inertia weight, this version of the method will be used from now on (MATLAB-code in Appendix A.15).

6.2.1 Swarm Size and Search Area

As was mentioned before, the convergence of PSO depends on the swarm size and the search area: the search area must contain the global minimum and the swarm size must be large enough such that this minimum is indeed found. For *myNewtonMethod* the computation time is measured for different swarm sizes in Table 13 and for a number of search areas, represented by lower and upper bounds, in Table 14.

Swarm Size	Time (sec)	Time/Swarm Size	Min. Found
500	881.686570	1.7634	No
1000	2190.047727	2.1900	Yes
1500	2449.725901	1.6332	Yes
2000	2763.140167	1.3816	Yes
2500	4514.675964	1.8059	Yes
3000	5311.872125	1.7706	Yes

Table 13: Times for applying PSO to *myNewtonMethod* for various swarm sizes; the lower bound for each variable of is -1 and the upper bound is 10.

The relation between the swarm size and the time is approximately linear, as can also be seen in Figure 5.

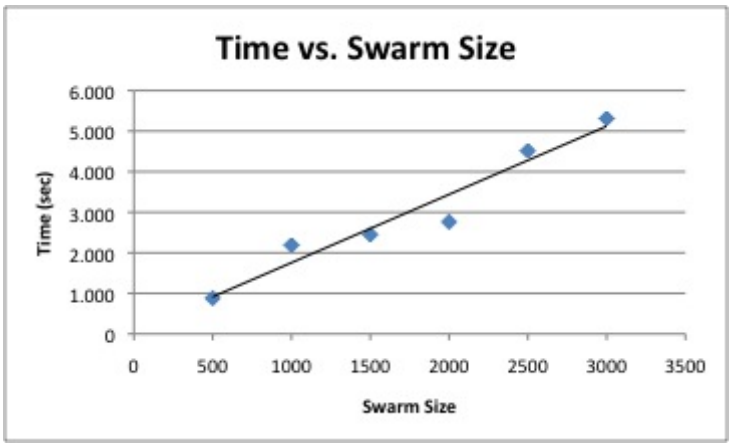


Figure 5: Graph showing an approximately linear relation between swarm size and time

In Table 14 we see that the size, i.e. the hypervolume, of the search area clearly influences the computation time, but there is not a clear relation visible; increasing the search area does not necessarily result in a longer time. The lack hereof is due to the multiple global minima: depending on the size of the search area, PSO will converge to different global minima, hence causing different computation times.
A special case in Table 14, indicated by *, is when the method does not find the global minimum. However, when the swarm size is increased a bit, to 1100, it does locate the minimum. Another, though less reliable, option is repeating the search with other random numbers (i.e. not using rng('default') in the MATLAB-code in A.15), which also can give the correct answer, indicating that it had bad luck with choosing the random numbers. This shows that it might help to repeat PSO with different random

numbers, though it is more sensible to use a larger swarm size when the minimum is not found at the first try.

LB	**UB**	**Time (sec)**	**Min. Found**
-1	2	343.944887	Yes
-1	3	1798.958278	Yes
-1	4	2787.301390	Yes
-1	5	1116.118964	No*
-1	6	1200.379516	Yes
-1	7	1709.458208	Yes
-1	8	1572.854437	Yes
-1	9	1633.012427	Yes
-1	10	1934.0	Yes

Table 14: Times for applying PSO to *myNewtonMethod* for various search areas; the swarm size is 1000; LB and UB are the lower respectively upper bounds for each variable of *myNewtonMethod*, defining the search area.

7 Variations of *MyNewtonMethod*

Now that we have found a method that converges for several test cases without having to use a time-consuming search grid, let us see if PSO still works for some variations of the test cases. We will change the initial guess for the Newton Method, add two more parameters, add some exponential terms and discuss the convergence behaviour. We conclude with expanding the problem to a 3-dimensional problem, being solved by the Newton Method (MATLAB-codes in Appendices A.16 to A.19).

The Downhill Simplex Method and Powell's Methods in general perform poorly for the variations of *myNewtonMethod*, not finding the global minimum when starting close to it nor when starting at the values that give the pure Newton Method. Therefore, we will focus on the PSO method.

7.1 *MyNewtonMethod_ 2*

The first variation considered entails changing the initial x and y values used in *myNewtonMethod*, such that $sin(5\pi xy) \neq 0$. This is done because for the present initial values, $x = -50$ and $y = 120$, β is arbitrary for the minimum of only 1 iteration, as it is multiplied by $sin(5\pi \cdot -50 \cdot 120) = 0$. Setting $x = -50.2$ and $y = 120.3$ prevents this from happening. PSO finds the minimum of one iteration without any problems; the result is shown in Table 15, where this variation is called *myNewtonMethod_ 2*.
MyNewtonMethod_ 2 is a little bit faster than *myNewtonMethod*, about one minute. The difference is time is due to the fact that MATLAB is not able to measure the computation times very precisely, and the global minimum lies at a different location.

7.2 *MyNewtonMethod_ 3*

For the next problem, called *myNewtonMethod_ 3*, two more parameters, γ and ϵ, are inserted in *myNewtonMethod_ 2* to modify the Jacobian in the following way

$$J(x,y) = \begin{bmatrix} 2\gamma(10x-1) & 2\epsilon(y-2) \\ \\ \begin{array}{c} 2\alpha(10x-1)(y-2)^2 \\ +5\pi y \cdot sin(5\pi xy) \end{array} & \begin{array}{c} 2(10x-1)^2(y-2) \\ +\beta\pi x \cdot sin(5\pi xy) \end{array} \end{bmatrix}$$

For $relax1 = 1$, $relax2 = 1$, $\alpha = 10$, $\beta = 5$, $\gamma = 10$, $\epsilon = 1$ we get the pure Newton Method.
Again, PSO finds the global minimum. Note in Table 15 that, relative to *myNewtonMethod_ 2*, there is a significant increase (circa 50%) in the time it takes for the method to locate the minimum. This is expected, as the problem now has two more parameters for which it has to find the correct values and hence a larger search area.

7.3 *MyNewtonMethod_ 4*

Now we turn the problem into a seemingly more difficult one by adding some exponential terms, making it more sensitive to changes in the x and y values. The problem is called *myNewtonMethod_ 4* in Table 15 and consists of finding the solution $(0.1, 2)$ of

$$f(x,y) = \begin{bmatrix} (10x-1)^2 + (y-2)^2 e^{xy} \\ (10x-1)^2(y-2)^2 - cos(5\pi xy)e^{xy} \end{bmatrix} = \begin{bmatrix} 0 \\ e^{xy} \end{bmatrix}$$

The initial values are $x = -50.2$ and $y = 120.3$. The parameters $relax1$ and $relax2$ are used as relaxations in Newton and the Jacobian is modified through the parameters α and β:

$$J(x,y) = \begin{bmatrix} 20(10x-1) + y(y-2)^2 e^{xy} & 2(y-2)e^{xy} + x(y-2)^2 e^{xy} \\ \\ \begin{array}{c} 2\alpha(10x-1)(y-2)^2 \\ +5\pi y \cdot sin(5\pi xy)e^{xy} \\ -y \cdot cos(5\pi xy)e^{xy} - ye^{xy} \end{array} & \begin{array}{c} 2(10x-1)^2(y-2) \\ +\beta\pi x \cdot sin(5\pi xy)e^{xy} \\ -x \cdot cos(5\pi xy)e^{xy} - xe^{xy} \end{array} \end{bmatrix}$$

$$\begin{aligned} \delta &= J^{-1}(x,y) \cdot f(x,y)^T \\ x &= x - relax1 \cdot \delta(1) \\ y &= y - relax2 \cdot \delta(2) \end{aligned}$$

It takes PSO a little bit longer (almost three minutes) than for *myNewtonMethod_ 2* to find the minimum, which is located somewhere else than the minimum of *myNewtonMethod_ 2*. It is interesting to note that for *myNewtonMethod_ 2*, we needed a swarm size of 1000, and PSO did not find the minimum for a swarm of 500 particles. Here however, we already obtain convergence when the swarm size is only 50, in less than three minutes; being able to decrease the swarm size lowers the computation time significantly. The added exponential terms cause the gradient to often have a larger absolute value, which makes it easier for PSO to find the minimum.

7.4 *MyNewtonMethod_ 5*

MyNewtonMethod_ 5 is obtained by combining *myNewtonMethod_ 3* and *myNewtonMethod_ 4*: it contains the exponential terms and six parameters. PSO finds the minimum and the result shown in Table 15. It does take a lot longer than for *myNewtonMethod_ 3*, to which we have added the exponential terms. However, as with *myNewtonMethod_ 4*, adding the exponential terms allows us to decrease the swarm size. PSO already works with a swarm size of 250, finding the minimum in about 19 minutes, which is five times as fast as for the swarm containing 1000 particles.

7.5 *MyNewtonMethod_3D*

MyNewtonMethod finds the solution of a 2-dimensional problem. Instead, now consider finding the solution $(0.1, 2, 1)$ of the following 3-dimensional problem:

$$f(x,y) = \begin{bmatrix} (10x-1)^2 + (y-2)^2 + (5z-5)^2 \\ (10x-1)^2(y-2)^2 + (5z-5)^2 - cos(5\pi xy) \\ e^z \end{bmatrix} = \begin{bmatrix} 0 \\ 1 \\ e^1 \end{bmatrix}$$

The initial values are $x = -50$, $y = 120$ and $z = 15$. The parameters $relax1$ and $relax2$ are used as relaxations in Newton's Method and the Jacobian is modified through the parameters α and β:

$$J(x,y,z) = \begin{bmatrix} 20(10x-1) & 2(y-2) & 10(5z-5) \\ \begin{matrix} 2\alpha(10x-1)(y-2)^2 \\ +5\pi y \cdot sin(5\pi xy) \end{matrix} & \begin{matrix} 2(10x-1)^2(y-2) \\ +\beta\pi x \cdot sin(5\pi xy) \end{matrix} & 10(5z-5) \\ 0 & 0 & e^z \end{bmatrix}$$

$$\begin{aligned} \delta &= J^{-1}(x,y) \cdot f(x,y)^T \\ x &= x - relax1 \cdot \delta(1) \\ y &= y - relax2 \cdot \delta(2) \\ z &= z - \delta(3) \end{aligned}$$

As before, for $relax1 = 1$, $relax2 = 1$, $\alpha = 10$ and $\beta = 5$ we get the pure Newton Method.
We could still compute $relax1$ and $relax2$ when fixing α and β to find the correct x and y values in only one iteration; however, the minimum number of iterations is not one anymore. This is because z also needs to converge, which does not happen in a single step; in fact, this takes at least 19 iterations. PSO finds this global minimum of 19 iterations rather fast, in almost three minutes with a swarm size of 50, as is shown in Table 15.

Problem	**LB**	**UB**	**Swarm Size**	**Output PSO**	**Func. Value**	**Time (sec)**
MyNewton-Method	-1	10	1000	(2.0000, 2.0000, 0.0000, 8.0187)	1	1934.0
MyNewton-Method_2	-1	10	1000	(2.0023, 1.9587, -0.2106, 4.7384)	1	1892.7
MyNewton-Method_3	-1	10	1000	(2.0004, 3.2925, 3.9266, 8.2038, 10.0000, 1.6518)	1	2834.2
MyNewton-Method_4	-1	10	1000	(2, 4, 5, 9)	1	2040.5
MyNewton-Method_5	-1	10	1000	(2, 4, 5, 3, 10, -1)	1	5770.1
MyNewton-Method_3D	-10	10	50	(2.0024, 1.9998, -2.2452, 4.9942)	19	200.81

Table 15: Results of applying PSO to variations of *myNewtonMethod*. LB and UB are the lower respectively upper bounds for each variable, defining the search area. The Swarm Size is the number of particles initialized. Func. Value gives the function value at the point found by PSO.

8 Successive Over-Relaxation

We have studied *myNewtonMethod* and some variations of it and found that PSO locates the minimum for all these problems. Now we consider a whole new problem and investigate if PSO is also able to find an optimum when solving an ordinary differential equation, exploring the range of applications of PSO.
The new problem concerns solving the ordinary differential equation $y''(x) + \alpha y(x) = x$, with $\alpha = 10^{-4}$, $y(0) = y(1) = 1$ and $x \in [0, 1]$. The problem is first discretized and then solved using successive over-relaxation (SOR) [1]. However, instead of just one, SOR is given three parameters.
The discretization of the ODE is done using the finite difference method (FDM) [1], where the derivatives are replaced by their discrete approximations. The interval $[0, 1]$ is divided into 100 subintervals and y_i denotes the approximate solution at point x_i. The approximation for the second derivative then is $y''(x_i) \approx \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1})$, with $h = 0.01$. After the discretization (Appendix A.20), we obtain a system of the form $Ax = b$, which will be solved using SOR (Appendix A.21).

8.1 SOR

SOR is a modification of the Gauss-Seidel Method. The Gauss-Seidel Method performs fixed-point iterations for a system of equations. First the equations are rewritten, solving equation i for the ith unknown. Then the resulting function is iterated, starting with an initial given guess. In this iteration process, the method uses the most recent values of the unknown in every iteration step [8].
SOR is obtained by adding a relaxation parameter $\omega \in (1, 2)$ to the Gauss-Seidel Method that can accelerate the convergence toward the solution of the ODE. It combines the approximated values of the previous and current iteration [1]. As mentioned, we will not use just one parameter ω, but three parameters ω_1, ω_2, ω_3. Each ω_i determines for the update the ratio between the previous and current solution; w_1 does this for y_1, w_2 for y_i if $i \geqslant 2$ is even and w_3 for y_i if $i \geqslant 2$ is odd.
The solution of the ODE obtained by using SOR is shown in Figure 6.

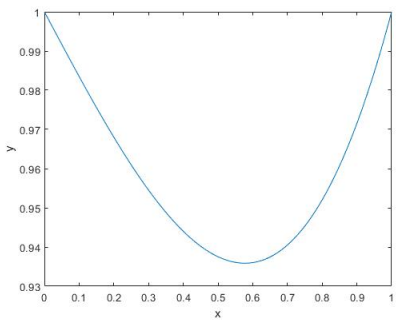


Figure 6: Plot of the solution obtained by SOR using the parameter values found by PSO that give 501 iterations.

8.2 One Parameter

Before looking at the problem with three parameters, we first discuss the normal situation with only one parameter. Note that for some special occasions the optimal parameter value can be determined in advance, for instance when the matrix A in Appendix A.21 is positive definite and tridiagonal ([1] p. 230). However, in general we have to use a search algorithm to find this optimal value. We conduct a simple grid search evaluating the function, which gives the number of iterations at each grid point. When the variables go from 1 to 2 with step size 10^{-4}, this results in $\omega = 1.9373$ with 582 iterations. Applying the Downhill Simplex Method or the Powell Methods starting at 2 or at 1 all yield the same result. Moreover, PSO finds this point as well. All together, it is plausible to assume that the minimum number of iterations is 582 if we use only one parameter ω.

8.3 Three Parameters

Now consider the situation with three parameters, which hopefully leads to a lower minimum number of iterations. The results of applying various methods to the problem are shown in Table 16. Conducting a simple grid-search or PSO both give a minimum of 501 iterations. The Downhill Simplex Method and Powell Method based on the Golden Section Search find this minimum when starting in the middle of the search space. Interestingly, they do not find the minimum when starting close to it. The Powell Method using Coggin's Method has problems finding the minimum and often leaves the search space. It does locate the minimum when starting very close to it, indicating that the Powell Method heavily relies on the choice of the initial values of the parameters. When applying the methods to a grid, they are all able to find the minimum.

In Table 16 we also see that the minimum of 501 iterations is not too sensitive to small changes in the parameters. PSO for instance finds this minimum at $\omega_1 = 1.0000$, $\omega_2 = 1.9258$ and $\omega_3 = 1.9531$, while the simple grid-search found the same minimum at $\omega_1 = 1.0200$, $\omega_2 = 1.9400$ and $\omega_3 = 1.9400$.

None of the methods considered in this study finds a number of iterations lower than 501, so this is probably the global minimum. The values found for the parameters ω_i differ from each other and the minimum is lowered by almost 14% compared to the normal SOR with one parameter, hence it is profitable to insert these extra parameters in SOR.

Initial Point	**Method**	**Output**	**Func. Value**
Grid 1:0.01:2	Function Evaluation	(1.0200, 1.9400, 1.9400)	501
(1.5, 1.5, 1.5)	Downhill Simplex Method	(1.0132, 1.9263, 1.9527)	501
(1, 2, 2)	Downhill Simplex Method	(1.0500, 2.0000, 2.0000)	divergence
(1.5, 1.5, 1.5)	Powell Method (Golden Section Search)	(1.0081, 1.9390, 1.9397)	501
(1, 2, 2)	Powell Method (Golden Section Search)	(1.0052, 1.8852, 1.9966)	503
(1.5, 1.5, 1.5)	Powell Method (Coggin's Method)	(1.0160, 2.4103, 3.0001)	divergence
(1, 2, 2)	Powell Method (Coggin's Method)	(0.2499, 1.8892, 1.9906	divergence
(1.1, 1.9, 1.9)	Powell Method (Coggin's Method)	(0.1612, 1.9589, 1.9063)	divergence
(1, 1.95, 1.95)	Powell Method (Coggin's Method)	(1.0288, 1.9291, 1.9503)	501
Grid 1:0.2:2	Downhill Simplex Method	(1.0069, 1.9393, 1.9395)	501
Grid 1:0.2:2	Powell Method (Golden Section Search)	(1.0005, 1.9395, 1.9394)	501
Grid 1:0.11:1:99	Powell Method (Coggin's Method)	(1.0078, 1.9219, 1.9577)	501
LB=1, UB=2	PSO, swarm size = 50	(1.0000, 1.9258, 1.9531)	501

Table 16: Results of applying various methods to SOR with three parameters, when 100 subintervals are used for the discretization. Func. Value gives the function value (i.e. the number of SOR iterations) at the point found by the method considered. LB and UB are the lower respectively upper bounds for each variable, defining the search area for PSO.

8.3.1 Increasing the Number of Subintervals

Now increase the number of subintervals for the discretization of the ODE from 100 to 1000, then the discrete approximation is closer to the real solution, but SOR requires more iterations and time to obtain the solution, even when increasing the allowed residual term from 10^{-9} to 10^{-3}. Moreover, the value and location of the minimum changes: the location of the minimum found by PSO for 100 subintervals would now give a result of 39614 iterations, which is a lot larger than the 501 iterations found for 100 subintervals and is not the global minimum, as we will soon see.
Due to the increase in time to evaluate SOR, we consider only the simple grid-search, Downhill Simplex Method, Powell Methods and the Downhill Simplex Method applied to a grid. The Powell Methods will not be applied to a grid, neither will we use PSO

since these cannot be computed within reasonable time. The Powell Methods often diverge and take longer to be evaluated than the Downhill Simplex Method, whose grid-search already took over 13 hours to complete. PSO uses many function evaluations, and because of the increased number of subintervals these take longer to be evaluated. Hence, the computation time for PSO will also be very large (more details regarding the costs per iteration can be found in the section 10).
In Table 17 we see that the lowest function value found is 3401 SOR iterations, which is over 6 times larger than the minimum number of iterations found in Table 16. So increasing the number of subintervals to obtain a more precise solution of the ODE results in a higher computation time and alters the applicability of the numerical optimization methods. This example emphasizes the importance of having as little function evaluations in a method as possible!

Initial Point	**Method**	**Output**	**Func. Value**
Grid 1:0.2:2	Function Evaluation	(1.0000, 1.8000, 2.0000)	86534
Grid with $\omega_1 \in$ 1:0.01:1.1 $\omega_2 \in$ 1.9:0.01:2 $\omega_3 \in$ 1.9:0.01:2	Function Evaluation	(1.0000, 1.9900, 2.0000)	4375
(1.5, 1.5, 1.5)	Downhill Simplex Method	(1.2256, 1.9935, 1.9932)	3455
(1, 2, 2)	Downhill Simplex Method	(1.0000, 2.1000, 2.0000)	divergence
(1.1, 1.9, 1.9)	Downhill Simplex Method	(1.1249, 1.9937, 1.9930)	3435
(1.1249, 1.9937, 1.9930)	Downhill Simplex Method	(1.0001, 1.9934, 1.9933)	3401
(1.5, 1.5, 1.5)	Powell Method (Golden Section Search)	(100.5832, 101.6030, 101.2346)	divergence
(1, 2, 2)	Powell Method (Golden Section Search)	(0.748, 172.2884, 172.2884)	divergence
(1.1, 1.9, 1.9)	Powell Method (Golden Section Search)	(1.3535, 1.9875, 1.9991)	3490
(1.5, 1.5, 1.5)	Powell Method (Coggin's Method)	(1.4889, 2.3565, 3.0001)	divergence
(1, 2, 2)	Powell Method (Coggin's Method)	(0.2500, 2.0000, 2.0000)	divergence
(1.1, 1.9, 1.9)	Powell Method (Coggin's Method)	(1.4990, 1.9982, 1.9678)	10772
Grid 1:0.25:2	Downhill Simplex Method	(1.0000, 1.9933, 1.9934)	3401

Table 17: Results of applying various methods to SOR with three parameters, when 1000 subintervals are used for the discretization. Func. Value gives the function value (i.e. the number of SOR iterations) at the point found by the method considered.

9 PSO Failure

It seems like PSO can minimize many problems. Can we come up with a function for which the methods fails?
Having tried many test functions [9] (Ackley function, Bukin6 function, Three-Hump Camel function, Easom function, Eggholder function, McCormick function, Schaffer function N.2, Schaffer function N.4, Styblinski-Tang function and the Sphere function, see Appendix A.22) we do not find one for which the method fails, when setting the parameters of PSO to the correct values for each function.
Consider the discontinuous function $y(x) = 1 - \delta_{x,j}$, where $\delta_{i,j}$ is the Kronecker delta function and $j = 1.2$. This function is constant 1 everywhere except for one specific point, here 1.2, where it is 0:

$$y(x) = \begin{cases} 1 \text{ if } x \neq 1.2 \\ 0 \text{ if } x = 1.2 \end{cases}$$

The minimum of this function (MATLAB-code in Appendix A.23) should be very hard to find, and indeed, when applying PSO to the function we do not get the correct solution, even after increasing the swarm size, the maximum number of iterations and runs and decreasing the search area.

The Downhill Simplex Method and Powell Methods perform poorly as well, failing to locate the minimum. The only (not so useful) exceptions are when the Downhill Simplex Method and the Powell Method based on Coggin's Method are given the minimum as initial value, then they do find the minimum.

10 Iteration Cost

Numerical optimization methods can be compared by considering the costs of their iterations. With the costs we primarily mean the amount of function evaluation per iteration, since this is the major time component of each iteration, especially when the optimization problem becomes larger or more difficult.

From the tables in the previous chapters we can compute approximations to the average number of function evaluations per iteration for each method, based on their results when applying them to the Rosenbrock function, the Rastrigin function and *myNewtonMethod*. The Downhill Simplex Method then executes approximately 2 function evaluations per iteration, Powell's Method based on the Golden Section Search 90 and based on Coggin's Method 68. This shows that the Downhill Simplex Method has by far the cheapest iterations, then comes the Powell Method based on Coggin's Method, quickly followed by the Golden Section Search. PSO performs about 500 function evaluations per iteration for the Rosenbrock function and the Rastrigin function. Thus the iterations of PSO are the most expensive, but this is also the method that performed best for the test cases.

Table 18 shows the number of function evaluations and iterations PSO uses when optimizing the variations of *myNewtonMethod*. For *myNewtonMethod* itself, PSO carries out circa 500 function evaluations per iteration. When changing the initial x and y values as was done in *myNewtonMethod_2*, this number increases to 1000, which makes sense as it then also has to find the correct value for β to obtain the minimum. When adding the exponential terms in *myNewtonMethod_4* and *myNewtonMethod_5*, the time per function evaluation (and hence per iteration) increases significantly as the function and especially its partial derivatives become more complex, hence take longer to be evaluated. *MyNewtonMethod_3D* has a lower number of function evaluation per iteration than the other variations of *myNewtonMethod*, caused by its smaller swarm size (see Table 15).

Problem	**#Func-Eval**	**#It**	**#FuncEval/It**	**Time/FuncEval**
MyNewton-Method	2370010	4730	501	$8.2 \cdot 10^{-4}$
MyNewton-Method_2	4743010	4743	1000	$4.0 \cdot 10^{-4}$
MyNewton-Method_3	5954010	5954	1000	$4.8 \cdot 10^{-4}$
MyNewton-Method_4	1020010	1020	1000	0.0020
MyNewton-Method_5	2980010	2980	1000	0.0019
MyNewton-Method_3D	169760	3395	50	0.0012

Table 18: PSO is applied to variations of *myNewtonMethod*, #FuncEval and #It give the number of function evaluations and iterations performed by the method. #FuncEval/It is the number of function evaluations per iteration and Time/FuncEval gives how long a single function evaluation approximately takes on average.

The importance of a low number of function evaluations was clearly visible when we increased the number of subintervals in the discretization of the ODE solved using SOR. Table 19 shows for the methods that converged the number of function evaluations and iterations for the original case with 100 subintervals. This indicates why we have not used PSO to optimize SOR when we have 1000 subintervals, as the number of function evaluations is very large, compared to other methods.

Initial Point	**Method**	**#Func-Eval**	**#It**	**#FuncEval/It**
(1.5, 1.5, 1.5)	Downhill Simplex Method	430	173	6
(1.5, 1.5, 1.5)	Powell Method (Golden Section Search)	3855	68	57
(1, 2, 2)	Powell Method (Golden Section Search)	2663	48	55
(1,1.95, 1.95)	Powell Method (Coggin's Method)	304	5	61
LB=1, UB=2	PSO, swarm size = 50	118910	2378	50

Table 19: Information about the methods that converged in Table 16, where various methods were applied to SOR with three parameters, using 100 subintervals for the discretization. #FuncEval and #It give the number of function evaluations and iterations performed by the method and #FuncEval/It is the number of function evaluations per iteration.

The function evaluations and iterations when using 1000 subintervals in the discretization are given in Table 20, for those initial points from Table 17 for which the methods converged to a point (though not necessarily the minimum). Here, a single function evaluation takes on average circa 220 times as long compared to the discretization with 100 subintervals. Together with the observation that PSO took 118910 function evaluations in about 5 minutes, it would take about one day to perform PSO for the 1000 subintervals, if the same number of function evaluations was used. However, we saw in Table 17 that the methods find it harder to locate the minimum than before, hence the number of function evaluation used by PSO will almost surely be higher than the number used for 100 subintervals. This will increase the computation time even more and therefore it is not possible to evaluate PSO for SOR using 1000 subintervals for the discretization within reasonable time. Even though PSO was a great method for locating the minimum, it would now take a very long time to execute it. Hence, in this case the Downhill Simplex Method is preferred over PSO because it performs fewer function evaluations.

Initial Point	**Method**	**#Func-Eval**	**#It**	**#FuncEval/It**
(1.5, 1.5, 1.5)	Downhill Simplex Method	195	100	2
(1.1, 1.9, 1.9)	Downhill Simplex Method	92	45	2
(1.1, 1.9, 1.9)	Powell Method (Golden Section Search)	552	8	69
(1.1, 1.9, 1.9)	Powell Method (Coggin's Method)	699	11	64

Table 20: Information about the methods that converged in Table 17, where various methods were applied to SOR with three parameters, using 1000 subintervals for the discretization. #FuncEval and #It give the number of function evaluations and iterations performed by the method and #FuncEval/It is the number of function evaluations per iteration.

11 Conclusion

Having applied various methods to a number of problems, it is time to decide which one is generally the best to use. First a summary is given of the performance of the methods discussed.

The Downhill Simplex Method performed well for the Rosenbrock function and came close to the minimum of *myNewtonMethod*; on the other hand, for the Rastrigin function it behaved poorly. The Powell Methods performed even better for the Rosenbrock function, and after some improvements converged to the minimum of the Rastrigin function as well. However, for *myNewtonMethod* they did not find the minimum. PSO with an inertia weight worked very well for all the problems discussed, with the exception of the one specifically chosen such that PSO failed; recall that the Downhill Simplex Method and Powell Methods could not find the minimum in this case either. Moreover, PSO is able to deal with functions having many local minima much better than the Downhill Simplex Method and Powell's Methods. Applying the Downhill Simplex Method or Powell Methods to a grid improved the results for *myNewtonMethod*, as multiple initial points were tested. This grid approach proved to be useful, though time-consuming. For SOR with three parameters and 100 subintervals, all methods located the minimum without using a grid. However, Powell's Method using Coggin's Method only found the minimum when starting very close to it.

When the location of the minimum is not yet known, PSO can search a large area and often still find the minimum, as long as the number of particles initialized is large enough. Increasing the swarm size increases the computation time, but more importantly, it also increases the chance of finding the minimum. Once the location of the minimum becomes clearer, the search area can be chosen more precise and the swarm size can be decreased. The downside is that PSO generally uses a lot of function evaluations and hence is very computationally demanding if the function to be optimized requires significant time to be evaluated, as we saw when increasing the number of subintervals used in the discretization from 100 to 1000.

The quality of the convergence behaviour of numerical optimization methods clearly depends on the problem to be optimized. For the Rosenbrock function, all the methods worked just fine, while for the Rastrigin function this was not always the case.

To conclude, PSO with an inertia weight is the preferred method here, based on the optimization problems discussed. Note however, that for other problems another method might be better, in terms of speed and precision. If one already has some idea where the minimum is located, the Downhill Simplex Method or Powell's Method might be faster, if they converge. If there is no prior knowledge about the minimum, PSO is a good method to find the minimum, as it can effectively search relatively large areas in case of computationally cheap function evaluations.

References

[1] Laurene V. Fausett. *Applied Numerical Analysis Using MATLAB*. Pearson, Prentice-Hall, 2 edition, 2008.

[2] Keisuke Kameyama. Particle Swarm Optimization - A Survey. *IEICE Transactions on Information and Systems*, Volume E92-D, Issue 7, 2009.

[3] José Mira and Juan V. Sánchez-Andrés. *Engineering Applications of Bio-Inspired Artificial Neural Networks*, pages 62–63. Springer Science & Business Media, 1 edition, 1999.

[4] A.E. Olsson. *Particle Swarm Optimization: Theory, Techniques and Applications*. Nova Science Publishers, Inc., 1 edition, 2011.

[5] S.J. Petersen. *Methods of Optimization for Numerical Algorithms*. July 2017. University of Groningen.

[6] W.H. Press, S.A. Teukolsky, W.T. Vetterling, and B.P. Flannery. *Numerical Recipes, The Art of Scientific Computing*. Cambridge University Press, 3 edition, 2007.

[7] Alfio Quarteroni, Fausto Saleri, and Paola Gervasio. *Scientific Computing with MATLAB and Octave*. Springer-Verlag, 4 edition, 2014.

[8] Timothy Sauer. *Numerical Analysis*. Pearson, 2 edition, 2012.

[9] S. Surjanovic and D. Bingham. Virtual Library of Simulation Experiments. https://www.sfu.ca/~ssurjano/optimization.html, August 2017.

[10] Y. Tan, Y. Shi, Y. Chai, and G. Wang. *Advances in Swarm Intelligence, Part I: Second International Conference, ICSI 2011, Chongqing, China, June 12-15, 2011, Proceedings*. Number pt. 1 in Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2011.

[11] Inc. The MathWorks. Matrix Inverse. https://nl.mathworks.com/help/matlab/ref/inv.html, 1994-2018.

[12] S. Tsutsui and P. Collet. *Massively Parallel Evolutionary Computation on GPGPUs*, page 74. Springer Science & Business Media, 6 edition, 2013.

A MATLAB Codes

A.1 Rosenbrock Function

```
% This script defines the 4-dimensional Rosenbrock function
   globally for easy access in other scripts.
function z = rosenbrock(x)
z = 100*(x(1)^2-x(2))^2 + (x(1)-1)^2 + 100*(x(3)^2-x(4))^2 +
    (x(3)-1)^2;
end
```

A.2 Rastrigin Function

```
% This script defines the 4-dimensional Rastrigin function
   globally for easy access in other scripts.
function y = rastrigin(x)
N = length(x);
A = 10;
y = A*N + x(1)^2 - A * cos(2*pi*x(1)) + x(2)^2 - A * cos(2*
   pi*x(2)) + x(3)^2 - A * cos(2*pi*x(3)) + x(4)^2 - A * cos
   (2*pi*x(4));
end
```

A.3 Plot Rosenbrock Function

```
% This script makes a plot of the 2-dimensional Rosenbrock
   function on the interval where x goes from -2 to 2 and y
   from -1 to 3.
clear all
rosenbrock2D =@(x,y) (1-x)^2+100*(y-x^2)^2;
x = -2:0.01:2;
y = -1:0.01:3;
for i = 1:length(y)
  for j = 1:length(x)
    z(i,j) = rosenbrock2D(x(j),y(i));
  end
end
surf(x,y,z,'EdgeColor','none','LineStyle','none','
   FaceLighting','gouraud')
xlabel('x')
ylabel('y')
zlabel('f(x,y)')
```

A.4 Plot Rastrigin Function

```
% This script makes a plot of the 2-dimensional Rastrigin
   function on the interval where x and  y go from -5.12 to
   5.12.
clear all
rastrigin2D =@(x,y) 20 + x^2 - 10*cos(2*pi*x) + y^2 - 10*cos
   (2*pi*y);
x = -5.12:0.01:5.12;
y = -5.12:0.01:5.12;
for i = 1:length(y)
  for j = 1:length(x)
    z(i,j) = rastrigin2D(x(j),y(i));
  end
end
surf(x,y,z,'EdgeColor','none','LineStyle','none','
   FaceLighting','gouraud')
xlabel('x')
ylabel('y')
zlabel('f(x,y)')
```

A.5 *MyNewtonMethod*

```
function [out] = myNewtonMethod(in)
% Newton algorithm to solve:
% F1= (10*x-1)^2 + (y-2)^2 = 0
% F2= (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) = 1
% with (multiple) solution (x,y) = (0.1, 2).
% Relaxation in Newton: relax1, relax2
% Jacobian modified through parameters: alf, bet
% Pure Newton for relax1=relax2=1, alf=10, bet=5
relax1 = in(1);
relax2 = in(2);
alf = in(3);
bet = in(4);
nmax = 1000;
n = 0;
x = -50;
y = 120;
% For MyNewtonMethod_2 use:
% x = -50.2;
% y = 120.3;
F(1) = (10*x-1)^2 + (y-2)^2;
```

```
F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) - 1;
error = norm(F);
while error > 1.0E-16 && n < nmax
  n = n + 1;
  J(1,1) = 2*10*(10*x-1);
  J(1,2) = 2*(y-2);
  J(2,1) = 2*alf*(10*x-1)*(y-2)^2 + 5*y*pi*sin(5*x*y*pi);
  J(2,2) = 2*(10*x-1)^2*(y-2) + bet*x*pi*sin(5*x*y*pi);
  % Check if all entries in the Jacobian are finite numbers:
  if isnan(J(1,1)) ~= 0 || isinf(J(1,1)) ~= 0|| isnan(J(1,2)
     ) ~= 0 || isinf(J(1,2)) ~= 0 || isnan(J(2,1)) ~= 0||
     isinf(J(2,1)) ~= 0 || isnan(J(2,2)) ~= 0 || isinf(J
     (2,2)) ~= 0
    n = nmax;
    break
  end
  rcond = cond(J);
  if rcond < 1.0E-16 || rcond > 10^10
    n = nmax;
    break
  end
  delta = J\F';
  x = x - relax1*delta(1);
  y = y - relax2*delta(2);
  F(1) = (10*x-1)^2 + (y-2)^2;
  F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) -1;
  error = norm(F);
end
out = n + error^0.1;
```

A.6 Powell's Method

```
function [xo,Ot,nS]=powell(S,x0,ip,method,Lb,Ub,problem,tol,
   mxit)
% Unconstrained optimization using Powell.
% S: objective function
% x0: initial point
% ip: (0): no plot (default), (>0) plot figure ip with pause
   , (<0) plot figure ip
% method: (0) Coggins (default), (1): Golden Section
% Lb, Ub: lower and upper bound vectors to plot (default =
   x0*(1+/-2))
% problem: (-1): minimum (default), (1): maximum
```

```
% tol: tolerance (default = 1e-4)
% mxit: maximum number of stages (default= 50*(1+4*~(ip>0)))
% xo: optimal point
% Ot: optimal value of S
% nS: number of objective function evaluations
% Copyright (c) 2001 by LASIM-DEQUI-UFRGS
% $Revision: 1.0 $  $Date: 2001/07/07 21:10:15 $
% Argimiro R. Secchi (arge@enq.ufrgs.br)
if nargin < 2
  error('powell requires 2 input arguments');
end
if nargin < 3 || isempty(ip)
  ip = 0;
end
if nargin < 4 || isempty(method)
  method = 0;
end
if nargin < 5 || isempty(Lb)
  Lb = -x0-~x0;
end
if nargin < 6 || isempty(Ub)
  Ub = 2*x0+~x0;
end
if nargin < 7 || isempty(problem)
  problem=-1;
end
if nargin < 8 || isempty(tol)
  tol=1e-4;
end
if nargin < 9 || isempty(mxit)
  mxit = 1000*(1+4*~(ip>0));
end

x0 = x0(:);
y0 = feval(S,x0)*problem;
n = size(x0,1);
D = eye(n);
ips = ip;
if ip && n == 2
  figure(abs(ip));
  [X1,X2] = meshgrid(Lb(1):(Ub(1)-Lb(1))/20:Ub(1),Lb(2):(Ub
     (2)-Lb(2))/20:Ub(2));
```

```
  [n1,n2] = size(X1);
  f = zeros(n1,n2);
  for i = 1:n1
    for j = 1:n2
      f(i,j) = feval(S,[X1(i,j);X2(i,j)]);
    end
  end
  mxf = max(max(f));
  mnf = min(min(f));
  df = mnf+(mxf-mnf)*(2.^(([0:10]/10).^2)-1);
  [v,h] = contour(X1,X2,f,df); hold on;
  clabel(v,h);
  h1 = plot(x0(1),x0(2),'ro');
  legend(h1,'start point');
  if ip > 0
    ips = ip + 1;
    disp('Pause: hit any key to continue');
    pause;
  else
    ips = ip - 1;
  end
end
xo = x0;
yo = y0;
it = 0;
nS = 1;
while it < mxit
% exploration
  delta = 0;
  for i = 1:n
    if method
    % to see the linesearch plot, remove the two 0* below
      [stepsize,x,Ot,nS1] = goldenSection(S,xo,D(:,i),0*ips,
         problem,tol,mxit);
      Ot = Ot*problem;
    else
      [stepsize,x,Ot,nS1] = coggins(S,xo,D(:,i),0*ips,
         problem,tol,mxit);
      Ot = Ot*problem;
    end
    nS = nS+nS1;
    di = Ot-yo;
```

```
  if di > delta
    delta = di;
    k = i;
  end
  if ip && n == 2
    plot([x(1) xo(1)],[x(2) xo(2)],'r');
    if ip > 0
      disp('Pause: hit any key to continue');
      pause;
    end
  end
  yo = Ot;
  xo = x;
end
% progression
it = it+1;
xo = 2*x-x0;
Ot = feval(S,xo)*problem;
nS = nS+1;
di = y0-Ot;
j = 0;
if di >= 0 || 2*(y0-2*yo+Ot)*((y0-yo-delta)/di)^2 >= delta
  if Ot >= yo
    yo = Ot;
  else
    xo = x;
    j = 1;
  end
else
  if k < n
    D(:,k:n-1) = D(:,k+1:n);
  end
  D(:,n) = (x-x0)/norm(x-x0);
  if method
  % to see the linesearch plot, remove the two 0* below
    [stepsize,xo,yo,nS1] = goldenSection(S,x,D(:,n),0*ips,
       problem,tol,mxit);
    yo = yo*problem;
  else
    [stepsize,xo,yo,nS1] = coggins(S,x,D(:,n),0*ips,
       problem,tol,mxit);
    yo = yo*problem;
```

```
    end
    nS = nS+nS1;
  end
  if ip && n == 2 && ~j
    plot([x(1) xo(1)],[x(2) xo(2)],'r');
    if ip > 0
      disp('Pause: hit any key to continue');
      pause;
    end
  end
  if norm(xo-x0) < tol*(0.1+norm(x0)) && abs(yo-y0) < tol
     *(0.1+abs(y0))
    break;
  end
  y0 = yo;
  x0 = xo;
end
Ot = yo*problem;
if it == mxit
  disp('Warning Powell: reached maximum number of stages!');
elseif ip && n == 2
  h2=plot(xo(1),xo(2),'r*');
  legend([h1,h2],'start point','optimum');
end
```

A.7 Bracketing Minimum

```
function [x1,x2,nS]=bracket(S,x0,d,problem,stepsize)
% Bracket the minimum (or maximum) of the objective function
    in the search direction.
% S: objective function
% x0: initial point
% d: search direction vector
% problem: (-1): minimum (default), (1): maximum
% stepsize: initial stepsize (default = 0.01*norm(d))
% [x1,x2]: unsorted lower and upper limits
% nS: number of objective function evaluations
% Copyright (c) 2001 by LASIM-DEQUI-UFRGS
% $Revision: 1.0 $  $Date: 2001/07/04 21:45:10 $
% Argimiro R. Secchi (arge@enq.ufrgs.br)
if nargin < 3
  error('bracket requires 3 input arguments');
end
```

```
if nargin < 4
  problem = -1;
end
if nargin < 5
  stepsize = 0.5*norm(d);
end
d = d(:);
x0 = x0(:);
j = 0;
nS = 1;
y0 = feval(S,x0)*problem;
while j < 2
  x = x0+stepsize*d;
  y = feval(S,x)*problem;
  nS = nS+1;
  if y0 >= y
    stepsize = -stepsize;
    j = j+1;
  else
    while y0 < y
      stepsize = 2*stepsize;
      y0 = y;
      x = x+stepsize*d;
      y = feval(S,x)*problem;
      nS = nS+1;
    end
    j = 1;
    break;
  end
end
x2 = x;
x1 = x0+stepsize*(j-1)*d;
```

A.8 Golden Section Search

```
function [stepsize,xo,Ot,nS]=goldenSection(S,x0,d,ip,problem
   ,tol,mxit,stp)
% Performs line search procedure for unconstrained
   optimization using golden section.
% S: objective function
% x0: initial point
% d: search direction vector
% ip: (0): no plot (default), (>0) plot figure ip with pause
```

```
   , (<0) plot figure ip
% problem: (-1): minimum (default), (1): maximum
% tol: tolerance (default = 1e-4)
% mxit: maximum number of iterations (default = 50*(1+4*~(ip
   >0)))
% stp: initial stepsize (default = 0.01*sqrt(d'*d))
% stepsize: optimal stepsize
% xo: optimal point in the search direction
% Ot: optimal value of S in the search direction
% nS: number of objective function evaluations
% Copyright (c) 2001 by LASIM-DEQUI-UFRGS
% $Revision: 1.0 $  $Date: 2001/07/04 22:30:45 $
% Argimiro R. Secchi (arge@enq.ufrgs.br)
if nargin < 3
  error('goldenSection requires 3 input arguments');
end
if nargin < 4 || isempty(ip)
  ip = 0;
end
if nargin < 5 || isempty(problem)
  problem = -1;
end
if nargin < 6 || isempty(tol)
  tol = 1e-4;
end
if nargin < 7 || isempty(mxit)
  mxit = 50*(1+4*~(ip>0));
end
d = d(:);
nd = d'*d;
if nargin < 8 || isempty(stp)
  stepsize = 0.01*sqrt(nd);
else
  stepsize = abs(stp);
end
x0 = x0(:);
[x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
z(1) = d'*(x1-x0)/nd;
z(2) = d'*(x2-x0)/nd;
fi = .618033985;
k = 0;
secao = fi*(z(2)-z(1));
```

```
p(1) = z(1)+secao;
x = x0+p(1)*d;
y(1) = feval(S,x)*problem;
p(2) = z(2)-secao;
x = x0+p(2)*d;
y(2) = feval(S,x)*problem;
nS = nS+2;
if ip
  figure(abs(ip)); clf;
  c = ['m','g'];
  B = sort([z(1),z(2)]);
  b1 = 0.05*(abs(B(1))+~B(1));
  b2 = 0.05*(abs(B(2))+~B(2));
  X1 = (B(1)-b1):(B(2)-B(1)+b1+b2)/20:(B(2)+b2);
  n1 = size(X1,2);
  for i = 1:n1,
    f(i) = feval(S,x0+X1(i)*d);
  end
  plot(X1,f,'b'); axis(axis); hold on;
  legend('S(x0+\alpha d)');
  xlabel('\alpha');
  plot([B(1),B(1)],[-1/eps 1/eps],'k');
  plot([B(2),B(2)],[-1/eps 1/eps],'k');
  plot(p,y*problem,'ro');
  if ip > 0
    disp('Pause: hit any key to continue');
    pause;
  end
end
it = 0;
while abs(secao/fi) > tol && it < mxit
  if y(2) < y(1)
    j = 2;
    k = 1;
  else
    j = 1;
    k = 2;
  end
  z(k) = p(j);
  p(j) = p(k);
  y(j) = y(k);
  secao = fi*(z(2)-z(1));
```

```
  p(k) = z(k)+(j-k)*secao;
  x = x0+p(k)*d;
  y(k) = feval(S,x)*problem;
  nS = nS+1;
  if ip
    plot([z(k),z(k)],[-1/eps 1/eps],c(k));
    plot(p(k),y(k)*problem,'ro');
    if ip > 0
      disp('Pause: hit any key to continue');
      pause;
    end
  end
  it = it+1;
end
stepsize = p(k);
xo = x;
Ot = y(k)*problem;
if it == mxit
  disp('Warning goldenSection: reached maximum number of
     iterations!');
elseif ip
  plot(stepsize,Ot,'r*');
end
```

A.9 Coggin's Method

```
function [stepsize,xo,Ot,nS]=coggins(S,x0,d,ip,problem,tol,
   mxit,stp)
% Performs line search procedure for unconstrained
   optimization using quadratic interpolation.
% S: objective function
% x0: initial point
% d: search direction vector
% ip: (0): no plot (default), (>0) plot figure ip with pause
   , (<0) plot figure ip
% problem: (-1): minimum (default), (1): maximum
% tol: tolerance (default = 1e-4)
% mxit: maximum number of iterations (default = 50*(1+4*~(ip
   >0)))
% stp: initial stepsize (default = 0.01*sqrt(d'*d))
% stepsize: optimal stepsize
% xo: optimal point in the search direction
% Ot: optimal value of S in the search direction
```

```
% nS: number of objective function evaluations
% Copyright (c) 2001 by LASIM-DEQUI-UFRGS
% $Revision: 1.0 $  $Date: 2001/07/04 21:20:15 $
% Argimiro R. Secchi (arge@enq.ufrgs.br)
if nargin < 3
  error('coggins requires 3 input arguments');
end
if nargin < 4 || isempty(ip)
  ip = 0;
end
if nargin < 5 || isempty(problem)
  problem = -1;
end
if nargin < 6 || isempty(tol)
  tol = 1e-4;
end
if nargin < 7 || isempty(mxit)
  mxit = 100*50*(1+4*~(ip>0));
end
d = d(:);
nd = d'*d;
if nargin < 8 || isempty(stp)
  stepsize = 0.5*sqrt(nd);
else
  stepsize = abs(stp);
end
x0 = x0(:);
[x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
z(1) = d'*(x1-x0)/nd;
y(1) = feval(S,x1)*problem;
z(3) = d'*(x2-x0)/nd;
y(3) = feval(S,x2)*problem;
z(2) = 0.5*(z(3)+z(1));
x = x0+z(2)*d;
y(2) = feval(S,x)*problem;
nS = nS+3;
if ip
  figure(abs(ip)); clf;
  B = sort([z(1),z(3)]);
  b1 = 0.05*(abs(B(1))+~B(1));
  b2 = 0.05*(abs(B(2))+~B(2));
  X1 = (B(1)-b1):(B(2)-B(1)+b1+b2)/20:(B(2)+b2);
```

```
  n1 = size(X1,2);
  for i = 1:n1
    f(i) = feval(S,x0+X1(i)*d);
  end
  plot(X1,f,'b',X1(1),f(1),'g'); axis(axis); hold on;
  legend('S(x0+\alpha d)','P_2(x0+\alpha d)');
  xlabel('\alpha');
  plot([B(1),B(1)],[-1/eps 1/eps],'k');
  plot([B(2),B(2)],[-1/eps 1/eps],'k');
  plot(z,y*problem,'ro');
  if ip > 0
    disp('Pause: hit any key to continue');
    pause;
  end
end
it = 0;
while it < mxit
  a1=z(2)-z(3); a2=z(3)-z(1); a3=z(1)-z(2);
  if y(1) == y(2) && y(2) == y(3)
    zo = z(2);
    x = x0+zo*d;
    ym = y(2);
  else
    zo = .5*(a1*(z(2)+z(3))*y(1)+a2*(z(3)+z(1))*y(2)+a3*(z
       (1)+z(2))*y(3))/(a1*y(1)+a2*y(2)+a3*y(3));
    x = x0+zo*d;
    ym = feval(S,x)*problem;
    nS = nS+1;
  end
  if ip
    P2 = -((X1-z(2)).*(X1-z(3))*y(1)/(a3*a2)+(X1-z(1)).*(X1-
       z(3))*y(2)/(a3*a1)+(X1-z(1)).*(X1-z(2))*y(3)/(a2*a1))
       *problem;
    plot(X1,P2,'g');
    if ip > 0
      disp('Pause: hit any key to continue');
      pause;
    end
    plot(zo,ym*problem,'ro');
  end
  for j=1:3
    if abs(z(j)-zo) < tol*(0.1+abs(zo))
```

```
      stepsize = zo;
      xo = x;
      Ot = ym*problem;
      if ip
        plot(stepsize,Ot,'r*');
      end
      return;
    end
  end
  if (z(3)-zo)*(zo-z(2)) > 0
    j = 1;
  else
    j = 3;
  end
  if ym > y(2)
    z(j) = z(2);
    y(j) = y(2);
    j = 2;
  end
  y(4-j) = ym;
  z(4-j) = zo;
  it = it+1;
end
if it == mxit
  disp('Warning Coggins: reached maximum number of
     iterations!');
end
stepsize = zo;
xo = x;
Ot = ym*problem;
```

A.10 1-dimensional Rastrigin Function

```
% This script plots the Rastrigin function on the standard
   interval where x and y go from -5.12 to 5.12.
clear all
close all
rastrigin1d =@(x) 10 + x^2 - 10 * cos(2*pi*x);
x = linspace(-5.12,5.12,1001);
for i = 1:length(x)
  y(i) = rastrigin1d(x(i));
end
plot(x,y)
```

```
xlabel('x')
ylabel('f(x)')
```

A.11 Golden Section Search Improved

```
function [stepsize,xo,Ot,nS]=goldenSectionImp(S,x0,d,ip,
   problem,tol,mxit,stp)
% Golden Section Search improved for the Rastrigin function.
% S: objective function
% x0: initial point
% d: search direction vector
% ip: (0): no plot (default), (>0) plot figure ip with pause
   , (<0) plot figure ip
% problem: (-1): minimum (default), (1): maximum
% tol: tolerance (default = 1e-4)
% mxit: maximum number of iterations (default = 50*(1+4*~(ip
   >0)))
% stp: initial stepsize (default = 0.01*sqrt(d'*d))
% stepsize: optimal stepsize
% xo: optimal point in the search direction
% Ot: optimal value of S in the search direction
% nS: number of objective function evaluations
if nargin < 3
  error('goldenSectionImp requires 3 input arguments');
end
if nargin < 4 || isempty(ip)
  ip = 0;
end
if nargin < 5 || isempty(problem)
  problem = -1;
end
if nargin < 6 || isempty(tol)
  tol = 1e-4;
end
if nargin < 7 || isempty(mxit)
  mxit = 50*(1+4*~(ip>0));
end
d = d(:);
nd = d'*d;
if nargin < 8 || isempty(stp)
  stepsize = 0.01*sqrt(nd);
else
  stepsize = abs(stp);
```

```
end
x0 = x0(:);
[x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
first = 1;
while abs(x2-x1)<2
  if first == 1
    stepsize = stepsize * 10;
    first = 0;
  end
  for i = 1:length(x0)
    if d(i) ~= 0
      d(i) = d(i)+1;
      if d(i) == 0
        d(i) = d(i)+0.1;
      end
    end
  end
  nd = d'*d;
  [x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
end
z(1) = d'*(x1-x0)/nd;
z(2) = d'*(x2-x0)/nd;
fi = .618033985;
% The rest of the code is the same as in the normal Golden
   Section Search given in Appendix A.8
```

A.12 Golden Section Search Repeated

```
% Repeat the Golden Section Search until the minimum is
   found
function[stepsize,xo,Ot,mS] = goldenSectionRep(S,x0,d,ip,
   problem,tol,mxit,stp)
T = zeros(1);
j = 0;
mS = 0;
[stepsize,xo,T(1),nS]= goldenSectionImp(S,x0,d);
mS = mS+nS;
[stepsize,xo,T(2),nS]= goldenSectionImp(S,xo,d);
mS = mS+nS;
i = 2;
while abs(T(1)-T(2)) > tol*10
  i = 1+j*1;
 [stepsize,xo,T(i),nS]=goldenSectionImp(S,xo,d);
```

```
  mS = mS+nS;
  j = 1-j;
end
Ot = T(2-i+1);
end
```

A.13 Coggin's Method Improved

```
function [stepsize,xo,Ot,nS]=cogginsImp(S,x0,d,ip,problem,
   tol,mxit,stp)
% Coggin's Method improved for the Rastrigin function
% S: objective function
% x0: initial point
% d: search direction vector
% ip: (0): no plot (default), (>0) plot figure ip with pause
   , (<0) plot figure ip
% problem: (-1): minimum (default), (1): maximum
% tol: tolerance (default = 1e-4)
% mxit: maximum number of iterations (default = 50*(1+4*~(ip
   >0)))
% stp: initial stepsize (default = 0.01*sqrt(d'*d))
% stepsize: optimal stepsize
% xo: optimal point in the search direction
% Ot: optimal value of S in the search direction
% nS: number of objective function evaluations
if nargin < 3
  error('coggins requires 3 input arguments');
end
if nargin < 4 || isempty(ip)
  ip = 0;
end
if nargin < 5 || isempty(problem)
  problem = -1;
end
if nargin < 6 || isempty(tol)
  tol = 1e-4;
end
if nargin < 7 || isempty(mxit)
  mxit = 100*50*(1+4*~(ip>0));
end
d = d(:);
nd = d'*d;
if nargin < 8 || isempty(stp)
```

```
  stepsize = 0.5*sqrt(nd);
else
  stepsize = abs(stp);
end
x0 = x0(:);
[x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
first = 1;
while abs(x2-x1)<2
  if first == 1
    stepsize = stepsize * 10;
    first = 0;
  end
  for i = 1:length(x0)
    if d(i) ~= 0
      d(i) = d(i)+1;
      if d(i) == 0
        d(i) = d(i)+0.1;
      end
    end
  end
  nd = d'*d;
  [x1,x2,nS] = bracket(S,x0,d,problem,stepsize);
end
z(1) = d'*(x1-x0)/nd;
% The rest of the code is the same as in the normal Coggin's
    Method given in Appendix A.9
```

A.14 Grid

```
% Evaluating a function or applying a method on a grid.
   Insert the stepSize, the start and end values of the
   required number of parameters w.
clear all
tic
% insert values:
stepSize = ...;
w1start=..; w1end=..;
w2start=..; w2end=..;
w3start=..; w3end=..;
w4start=..; w4end=..;
dw1 = (w1end-w1start)/stepSize;
dw2 = (w2end-w2start)/stepSize;
dw3 = (w3end-w3start)/stepSize;
```

```
dw4 = (w4end-w4start)/stepSize;
minnit=10^6;
for i = 0:dw1
for j = 0:dw2
for k = 0:dw3
for l = 0:dw4
  w1 = w1start + i*stepSize;
  w2 = w2start + j*stepSize;
  w3 = w3start + k*stepSize;
  w4 = w4start + l*stepSize;
  % choose function evaluation or a method
  nitval = ... % insert preferred function or gssor
  [xo,nitval] = ... % insert preferred method: fminsearch or
     powell
  if nitval < minnit
    minnit = nitval;
    % if chosen function evaluation or gssor:
    w1opt = w1;
    w2opt = w2;
    w3opt = w3;
    w4opt = w4;
    % if chosen optimization method:
    w1opt = xo(1);
    w2opt = xo(2);
    w3opt = xo(3);
    w4opt = xo(4);
  end
end
end
end
end
toc
minnit = floor(minnit)
[w1opt, w2opt, w3opt, w4opt]
```

A.15 PSO

```
% This script performs Particle Swarm Optimization (PSO).
% Insert the number of variables (m), population size (n)
   and replace all notions of 'function' by the name of the
   function to be optimized.
tic
clear all
```

```
close all
rng('default')
LB = ...;   % insert lower bounds of variables
UB = ...;   % insert upper bounds of variables
% pso parameters values
m = ...;    % insert number of variables
n = ...;    % insert population size (i.e. number of
   particles)
wmax = 1    % inertia weight
wmin = 0    % inertia weight
c1 = 2;     % acceleration factor
c2 = 2;     % acceleration factor
% pso main program-------------------------------------start
maxite = 1000; % set max number of iterations
maxrun = 10;   % set max number of runs needed
funEval = zeros(maxrun,1);
for run = 1:maxrun
  run
  % pso initialization---------------------------------start
  for i = 1:n
    for j = 1:m
      x0(i,j) = round(LB(j)+rand()*(UB(j)-LB(j)));
    end
  end
  x = x0;        % initial population
  v = 0.1*x0;    % initial velocity
  for i = 1:n
    f0(i,1) = function(x0(i,:));
    funEval(run) = funEval(run) + 1;
  end
  [fmin0,index0] = min(f0);
  pbest = x0;                % initial pbest
  gbest = x0(index0,:);      % initial gbest
  % pso initialization-----------------------------------end
  % pso algorithm----------------------------------------start
  ite = 1;
  tolerance = 1;
  while ite<=maxite && tolerance>10^-12
    % update exponentially decreasing inertial weight
    w = wmin+(wmax-wmin)*exp(-ite/(maxite/25));
    %w=1 gives PSO without inertia weight
```

```
% pso velocity updates
for i = 1:n
  for j = 1:m
    v(i,j) = w*v(i,j)+c1*rand()*(pbest(i,j)-x(i,j))+c2*
       rand()*(gbest(1,j)-x(i,j));
  end
end
% pso position update
for i = 1:n
  for j = 1:m
    x(i,j) = x(i,j)+v(i,j);
  end
end
% handling boundary violations
for i = 1:n
  for j = 1:m
    if x(i,j) < LB(j)
      x(i,j) = LB(j);
    elseif x(i,j) > UB(j)
      x(i,j) = UB(j);
    end
  end
end
% evaluating fitness
for i = 1:n
  f(i,1) = function(x(i,:));
  funEval(run) = funEval(run) + 1;
end
% updating pbest and fitness
for i = 1:n
  if f(i,1) < f0(i,1)
    pbest(i,:) = x(i,:);
    f0(i,1) = f(i,1);
  end
end
% finding out the best particle
[fmin,index] = min(f0);
% storing best fitness
ffmin(ite,run) = fmin;
 % storing iteration count
ffite(run) = ite;
% updating gbest and best fitness
```

```
    if fmin < fmin0
      gbest = pbest(index,:);
      fmin0 = fmin;
    end
    % calculating tolerance
    if ite > 100;
      tolerance = abs(ffmin(ite-100,run)-fmin0);
    end
    ite = ite+1;
  end
  % pso algorithm----------------------------------------end
  gbest;
  fvalue = function([gbest(1),gbest(2),..,gbest(m)]); % size
      of gbest vector depends on m
  funEval(run) = funEval(run) + 1;
  fff(run) = fvalue;
  rgbest(run,:) = gbest;
  iterations(run)=ite;
end
% pso main program-----------------------------------------end
[bestfun,bestrun] = min(fff);
bestfun = floor(bestfun) % floor only necessary when output
   of 'function' is the integer number of iterations plus a
   small error term
best_variables = rgbest(bestrun,:)
toc
totalFunEval=sum(funEval)
totalIt=sum(iterations)
% Available from: https://www.researchgate.net/publication
   /296636431_Codes_in_MATLAB_for_
   Particle_Swarm_Optimization.
```

A.16 *MyNewtonMethod_3*

```
function [out]=myNewtonMethod_3(in)
% Newton algorithm to solve:
% F1= (10*x-1)^2 + (y-2)^2 = 0
% F2= (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) = 1
% with (multiple) solution (x,y) = (0.1, 2).
% Relaxation in Newton: relax1, relax2
% Jacobian modified through parameters: alf, bet, gam, eps
% Pure Newton for relax1=relax2=1, alf=10, bet=5, gam=10,
   eps=1
```

```
relax1 = in(1);
relax2 = in(2);
alf = in(3);
bet = in(4);
gam = in(5);
eps = in(6);
nmax = 1000;
n = 0;
x = -50.2;
y = 120.3;
F(1) = (10*x-1)^2 + (y-2)^2;
F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) - 1;
error = norm(F);
while error > 1.0E-16 && n < nmax
  n = n + 1;
  J(1,1) = 2*gam*(10*x-1);
  J(1,2) = 2*eps*(y-2);
  J(2,1) = 2*alf*(10*x-1)*(y-2)^2 + 5*y*pi*sin(5*x*y*pi);
  J(2,2) = 2*(10*x-1)^2*(y-2) + bet*x*pi*sin(5*x*y*pi);
  % Check if all entries in the Jacobian are finite numbers:
  if isnan(J(1,1)) ~= 0 || isinf(J(1,1)) ~= 0|| isnan(J(1,2)
     ) ~= 0 || isinf(J(1,2)) ~= 0 || isnan(J(2,1)) ~= 0||
     isinf(J(2,1)) ~= 0 || isnan(J(2,2)) ~= 0 || isinf(J
     (2,2)) ~= 0
    n = nmax;
    break
  end
  rcond = cond(J);
  if rcond < 1.0E-16 || rcond > 10^10
    n = nmax;
    break
  end
  delta = J\F';
  x = x - relax1*delta(1);
  y = y - relax2*delta(2);
  F(1) = (10*x-1)^2 + (y-2)^2;
  F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi) -1;
  error = norm(F);
end
out = n + error^0.1;
```

A.17 *MyNewtonMethod_4*

```
function [out]=myNewtonMethod_4(in)
% Newton algorithm to solve:
% F1= (10*x-1)^2 + (y-2)^2*exp(x*y) = 0
% F2= (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) = exp(x*y)
% with (multiple) solution (x,y) = (0.1, 2).
% Relaxation in Newton: relax1, relax2
% Jacobian modified through parameters: alf, bet
% Pure Newton for relax1=relax2=1, alf=10, bet=5
relax1 = in(1);
relax2 = in(2);
alf = in(3);
bet = in(4);
nmax = 1000;
n = 0;
x = -50.2;
y = 120.3;
F(1)=(10*x-1)^2 + (y-2)^2*exp(x*y);
F(2)=(10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) - exp(x*y);
error = norm(F);
while error > 1.0E-16 && n < nmax
  n = n + 1;
  J(1,1) = 2*10*(10*x-1)+y*(y-2)^2*exp(x*y);
  J(1,2) = 2*(y-2)*exp(x*y)+x*(y-2)^2*exp(x*y);
  J(2,1) = 2*alf*(10*x-1)*(y-2)^2 + 5*y*pi*sin(5*x*y*pi)*exp
     (x*y)-y*cos(5*x*y*pi)*exp(x*y)-y*exp(x*y);
  J(2,2) = 2*(10*x-1)^2*(y-2) + bet*x*pi*sin(5*x*y*pi)*exp(x
     *y)-x*cos(5*x*y*pi)*exp(x*y)-x*exp(x*y);
  % Check if all entries in the Jacobian are finite numbers:
  if isnan(J(1,1)) ~= 0 || isinf(J(1,1)) ~= 0|| isnan(J(1,2)
     ) ~= 0 || isinf(J(1,2)) ~= 0 || isnan(J(2,1)) ~= 0||
     isinf(J(2,1)) ~= 0 || isnan(J(2,2)) ~= 0 || isinf(J
     (2,2)) ~= 0
    n = nmax;
    break
  end
  rcond = cond(J);
  if rcond < 1.0E-16 || rcond > 10^10
    n = nmax;
    break
  end
```

```
  delta = J\F';
  x = x - relax1*delta(1);
  y = y - relax2*delta(2);
  F(1) = (10*x-1)^2 + (y-2)^2*exp(x*y);
  F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) - exp(x
     *y);
  error = norm(F);
end
out = n + error^0.1;
```

A.18 *MyNewtonMethod_5*

```
function [out]=myNewtonMethod_5(in)
% Newton algorithm to solve:
% F1= (10*x-1)^2 + (y-2)^2*exp(x*y) = 0
% F2= (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) = exp(x*y)
% with (multiple) solution (x,y) = (0.1, 2).
% Relaxation in Newton: relax1, relax2
% Jacobian modified through parameters: alf, bet, gam, eps
% Pure Newton for relax1=relax2=1, alf=10, bet=5, gam=10,
   eps=1
relax1 = in(1);
relax2 = in(2);
alf = in(3);
bet = in(4);
gam = in(5);
eps = in(6);
nmax = 1000;
n = 0;
x = -50.2;
y = 120.3;
F(1)=(10*x-1)^2 + (y-2)^2*exp(x*y);
F(2)=(10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) - exp(x*y);
error = norm(F);
while error > 1.0E-16 && n < nmax
  n = n + 1;
  J(1,1) = 2*gam*(10*x-1)+y*(y-2)^2*exp(x*y);
  J(1,2) = 2*(y-2)*exp(x*y)+eps*x*(y-2)^2*exp(x*y);
  J(2,1) = 2*alf*(10*x-1)*(y-2)^2 + 5*y*pi*sin(5*x*y*pi)*exp
     (x*y)-y*cos(5*x*y*pi)*exp(x*y)-y*exp(x*y);
  J(2,2) = 2*(10*x-1)^2*(y-2) + bet*x*pi*sin(5*x*y*pi)*exp(x
     *y)-x*cos(5*x*y*pi)*exp(x*y)-x*exp(x*y);
  % Check if all entries in the Jacobian are finite numbers:
```

```
  if isnan(J(1,1)) ~= 0 || isinf(J(1,1)) ~= 0|| isnan(J(1,2)
     ) ~= 0 || isinf(J(1,2)) ~= 0 || isnan(J(2,1)) ~= 0||
     isinf(J(2,1)) ~= 0 || isnan(J(2,2)) ~= 0 || isinf(J
     (2,2)) ~= 0
    n = nmax;
    break
  end
  rcond = cond(J);
  if rcond < 1.0E-16 || rcond > 10^10
    n = nmax;
    break
  end
  delta = J\F';
  x = x - relax1*delta(1);
  y = y - relax2*delta(2);
  F(1) = (10*x-1)^2 + (y-2)^2*exp(x*y);
  F(2) = (10*x-1)^2*(y-2)^2 - cos(5*x*y*pi)*exp(x*y) - exp(x
     *y);
  error = norm(F);
end
out = n + error^0.1;
```

A.19 *MyNewtonMethod_3D*

```
function [out]=myNewtonMethod_3D(in)
% Newton algorithm to solve:
% F1= (10*x-1)^2 + (y-2)^2 +(5*z-5)^2 = 0
% F2= (10*x-1)^2*(y-2)^2 +(5*z-5)^2 - cos(5*x*y*pi) = 1
% F3= exp(x*y)+(5*z-5)^2 = exp(0.2)
% with (multiple) solution (x,y,z) = (0.1, 2, 1).
% Relaxation in Newton: relax1, relax2
% Jacobian modified through parameters: alf, bet
% Pure Newton for relax1=relax2=1, alf=10, bet=5
relax1 = in(1);
relax2 = in(2);
alf = in(3);
bet = in(4);
nmax = 1000;
n = 0;
x = -50;
y = 120;
z = 15;
F(1) = (10*x-1)^2 + (y-2)^2 +(5*z-5)^2;
```

```
F(2) = (10*x-1)^2*(y-2)^2 +(5*z-5)^2 - cos(5*x*y*pi) - 1;
F(3) = exp(z)-exp(1);
error = norm(F);
while error > 1.0E-16 && n < nmax
  n = n + 1;
  J(1,1) = 2*10*(10*x-1);
  J(1,2) = 2*(y-2);
  J(1,3) = 2*5*(5*z-5);
  J(2,1) = 2*alf*(10*x-1)*(y-2)^2 + 5*y*pi*sin(5*x*y*pi);
  J(2,2) = 2*(10*x-1)^2*(y-2) + bet*x*pi*sin(5*x*y*pi);
  J(2,3) = 2*5*(5*z-5);
  J(3,1) = 0;
  J(3,2) = 0;
  J(3,3) = exp(z);
  % Check if all entries in the Jacobian are finite numbers:
  Break=0;
  for i = 1:3
    for j = 1:3
      if (isnan(J(i,j)) ~= 0 || isinf(abs(J(i,j))) ~= 0)
        n = nmax;
        Break=1;
      end
    end
  end
  if Break == 1
    break
  end
  rcond = cond(J);
  if rcond < 1.0E-16  || rcond > 10^10
    n = nmax;
    break
  end
  delta = J\F';
  x = x - relax1*delta(1);
  y = y - relax2*delta(2);
  z = z - delta(3);
  F(1) = (10*x-1)^2 + (y-2)^2 +(5*z-5)^2;
  F(2) = (10*x-1)^2*(y-2)^2 +(5*z-5)^2 - cos(5*x*y*pi) -1;
  F(3) = exp(z)-exp(1);
  error = norm(F);
end
out = n + error^0.1;
```

A.20 ODE

```
% Discretization of y''(x) + alpha*y(x) = x, y(0)=1, y(1)=1
   for x in [0,1] using FDM, on a grid with 100 segments.
% The solution of the ODE is y(x)=10^4*x+cos(0.01*x)
   +(1-10^4-cos(0.01))*sin(0.01*x)/sin(0.01).
clear all
N = 101;
alpha = 1E-4;
L = zeros(N,1);
D = zeros(N,1);
R = zeros(N,1);
b = ones(N,1);
dx = 1/(N-1);
for i = 2:N-1,
  D(i) = -2/(dx*dx) +alpha;
  L(i) = 1/(dx*dx);
  R(i) = 1/(dx*dx);
  b(i) = (i-1)*dx;
end
D(1) = 1;
D(N) = 1;
```

A.21 SOR

```
function [it,xsol]=gssor(L,D,R,b,N,w1,w2,w3)
% solve Ax=b using Gauss-Seidel method
% input L, D, R: matrices that together form A
% input b: vector (1D) containing r.h.s.
% input N: dimension of problem
% output x: solution vector
% output it: number of iterations

% initial solution
xsol = zeros(N,1);
resid = 1E3;
it = 0;
while resid>1E-9
% increment iteration counter
  it = it+1;
  if it > 10000,
    break;
  end
```

```
% solution of previous iteration
  xsolold = xsol;
% new solution with Gauss-Seidel
  Sum = R(1)*xsol(2);
  xsol(1) = (b(1)-Sum)/D(1);
  xsol(1) = w1*xsol(1) + (1-w1)*xsolold(1);
  for i = 2:N-1,
    Sum = L(i)*xsol(i-1) + R(i)*xsol(i+1);
    xsol(i) = (b(i)-Sum)/D(i);
    if 2*floor(i/2) == i
      xsol(i) = w2*xsol(i) + (1-w2)*xsolold(i);
    else
      xsol(i) = w3*xsol(i) + (1-w3)*xsolold(i);
    end
  end
  Sum = L(N)*xsol(N-1);
  xsol(N) = (b(N)-Sum)/D(N);
  xsol(N) = w1*xsol(N) + (1-w1)*xsolold(N);
% solution is a number
  for i = 1:N
    if isnan(xsol(i)) ~= 0 || isinf(xsol(i)) ~= 0
      it = 10^5;
      break
    end
  end
% residual resid= ||Ax-b||;
  Sum = 0;
  i = 1;
  axi = D(i)*xsol(i) + R(i)*xsol(i+1);
  Sum =Sum + (axi -b(i))^2;
  for i = 2:N-1
    axi = L(i)*xsol(i-1) + D(i)*xsol(i) + R(i)*xsol(i+1);
    Sum = Sum + (axi -b(i))^2;
  end
  i = N;
  axi = L(i)*xsol(i-1) + D(i)*xsol(i);
  Sum = Sum + (axi -b(i))^2;
  resid = sqrt(Sum);
  if resid > 10^30
    it = 10^6;
  end
end
```

```
it = it+resid^0.1;
```

A.22 Test Functions

Test functions together with the settings for which PSO finds their minimum.

Ackley Function

```
% Ackley Function: nearly flat outer region and a large hole
    at the centre. The risk for optimization methods is to
   get stuck in one of the many local minima.
% 2-dimensional form
% Global minimum: f(0,0)=0
a = 20;
b = 0.2;
c = 2*pi;
sum1 = 0;
sum2 = 0;
for i = 1:2
        sum1 = sum1 + x(i)^2;
        sum2 = sum2 + cos(c*x(i));
end
y = -a*exp(-b*sqrt(sum1/2)) - exp(sum2/2) + a + exp(1);
% PSO: LB=-32.768, UB=32.768, swarm size=10
```

Bukin6 Function

```
% Bukin Function N.6: many local minima in a ridge.
% Global minimum: f(-10,1)=0.
y = 100 * sqrt(abs(x(2) - 0.01*x(1)^2)) + 0.01 * abs(x(1)
   +10);
% PSO: LB=[-15,-3], UB=[-5,3], swarm size=10
```

Three-Hump Camel Function

```
% Three-Hump Camel Function: three local minima.
% Global minimum: f(0,0)=0.
y = 2*x(1)^2 - 1.05*x(1)^4 + x(1)^6/6 + x(1)*x(2) + x(2)^2;
% PSO: LB=[-5,-5], UB=[5,5], swarm size=10
```

Easom Function

```
% Easom Function: several local minima. It is unimodal and
   the global minimum has a small area.
% Global minimum: f(pi,pi)=-1
y = -cos(x(1))*cos(x(2))*exp(-((x(1)-pi)^2+(x(2)-pi)^2));
% PSO: LB=[-100,-100], UB=[100,100], swarm size=10
```

Eggholder Function

```
% Eggholder Function: dificult to optimize due to the large
   number of local minima.
% Global minimum: f(512,404.2319)=-959.6407
y = -(x(2)+47)*sin(sqrt(abs(x(1)/2+x(2)+47))) - x(1)*sin(
   sqrt(abs(x(1)-(x(2)+47))));
% PSO: LB=[-512, -512], UB=[512, 512], swarm size=10
```

McCormick Function

```
% McCormick Function: plate-shaped.
% Global minimum: f(-0.54719,-1.54719)=-1.9133
y=sin(x(1)+x(2))+(x(1)-x(2))^2-1.5*x(1)+2.5*x(2)+1;
% PSO: LB=[-1.5,-3], UB=[4,4], swarm size=10
```

Schaffer Function N.2

```
% Schaffer Function N.2: many local minima.
% Global minimum: f(0,0)=0
y = 0.5 + (sin(x(1)^2-x(2)^2)^2-0.5)/(1+0.001*(x(1)^2+x(2)
   ^2))^2;
% PSO: LB=[-100,-100], UB=[100,100], swarm size=10
```

Schaffer Function N.4

```
% Schaffer Function N.4: many local minima
% Global minimum: f(0,1.25313)=0.292579
y=0.5+(cos(sin(abs(x(1)^2-x(2)^2)))^2-0.5)/(1+0.001*(x(1)^2+
   x(2)^2))^2;
% PSO: LB=[-100,-100], UB=[100,100], swarm size=10
```

Styblinski-Tang Function

```
% Styblinski-Tang Function: not convex.
% 2-dimensional form
% Global minimum: f(-2.903534,-2.903534)=-39.16599*2
y = (x(1)^4 - 16*x(1)^2 + 5*x(1) + x(2)^4 - 16*x(2)^2 + 5*x
   (2))/2;
% PSO: LB=[-5,-5], UB=[5,5], swarm size=10
```

Sphere Function

```
% Sphere Function: 2 local minima except for the global one.
    It is unimodal.
% 2-dimensional form
y = x(1)^2 + x(2)^2;
% PSO: LB=[-5.12,-5.12], UB=[5.12,5.12], swarm size=10
```

A.23 Discontinuous Function

```
% Function for which PSO fails to find the minimum
function [y] = discont(x)
        y = (x ~= 1.2);
end
```