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

Methods of Optimization for Numerical Algorithms

Bachelor's Project Mathematics

July 2017

Student: S.J. Petersen

First supervisor: dr.ir. R. Luppes

Second assessor: dr. A.E. Sterk

Contents

Chapter 1

Introduction

Optimizing functions is one of the most common problem of mathematics and also very often required in real world problems. While for well defined functions, this is as easy as calculating the gradient and finding the zero, we also want to be able to do this for functions that are not defined in a way that we can actually take the derivative. An example of this is the optimization of the number of iterations numerical algorithms require, depending on some tuning variables.

To do this we will analyse different methods of numerical minimization and optimization. The first of which, the Downhill Simplex method, is entirely self contained, whereas the second method, Powell's method, makes use of one-dimensional optimization methods.

Our main goal is to find a robust way of optimizing algorithms, by making use of these optimization methods.

Chapter 2

The Downhill Simplex Method

The first method of numerical minimization we will consider is called the Downhill Simplex method, also often referred to as Amoeba and the Nelder-Mead method. It is not to be confused with the Simplex method from econometrics, as these are entirely different.

2.1 The basics of the Downhill Simplex method

The Downhill Simplex method is an algorithm for multidimensional minimization. It is special in that it does not use any one-dimensional minimization algorithm. Another important aspect of this method is that it does not require the derivatives of the functions it tries to minimize.

The Downhill Simplex method has been given its name due to its use of a simplex. In N dimensions, a simplex is a geometrical figure consisting of $N + 1$ points and connecting lines, planes. For example, in two dimensions a simplex is a triangle and in three dimensions it is a tetrahedron. Note that the definition of a simplex does not require the points to be evenly distributed, meaning that the triangle and tetrahedron do not have to be the regular ones.

The Downhill Simplex method works as an algorithm with a starting point. The starting point however, has to be a starting simplex. So we have to define a starting simplex, by choosing the initial $N + 1$ points. It is however still possible to work with an initial point consisting of a single point P_0. We do this by then defining the other N points as

$$P_i = P_0 + \lambda e_i \qquad i = 1, \ldots, N$$

where e_i is the unit vector it usually indicates. One can freely choose the value of λ and even choose to use different λ_i's for each vector direction. This however defeats the purpose of using a single starting point.

Using the starting point or simplex, the Downhill Simplex method takes steps every iteration, in order to find a (local) minimum. We can categorize the steps this algorithm can take in four distinct actions. The algorithm chooses which action to take according to a couple of different calculations and criteria. A detailed description of how this works will follow in the next section. The options available are the following.

Reflection: The algorithm can take the highest point out of the $N+1$ points that defines the simplex and mirror it to the opposite side of the simplex, in a manner that conserves its volume. This is the most common step for the simplex, as will be made clear in the next section.

Expansion: The simplex can expand in order to move in the 'right' direction. An example is expanding beyond a low point, such that this low point is included in the volume of the simplex.

Contraction: When the simplex approaches a minimum or enters a 'valley', the simplex will contract itself in the transposed direction of the valley. Multiple contraction, also referred to as N-dimensional contraction, is possible when the simplex tries to pass through something we can visualize as the eye of the needle.

In figure 2.1 on the following page, the above described steps are visualized in two dimensions.

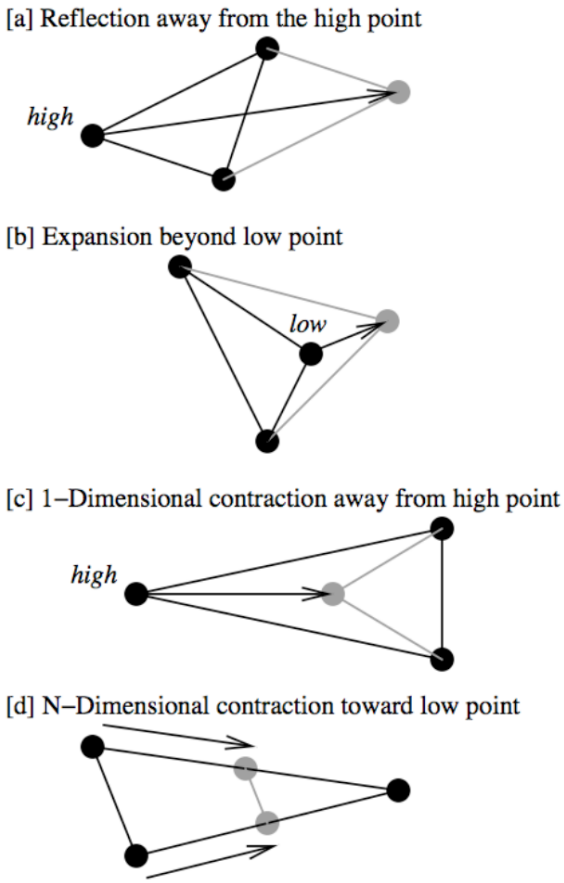

Figure 2.1: Possible movements of the simplex in the downhill simplex method, C.C. Kankelborg [2]

2.2 The algorithm of the Downhill Simplex method

Remember that we start the algorithm with a simplex defined by $N+1$ points, P_i for $i = [0, 1, \ldots, N]$. Now we can denote some of the P_i as $P^{(M)}$, $P^{(m)}$ and $P^{(\mu)}$, for which the function respectively attains its maximum value, its minimum value and its second largest (immediately after the maximum) value. Next to these values, we will also need the centroid of the point $P^{(k)}$, defined as

$$P_c^{(k)} = \frac{1}{N} \sum_{i=0, i \neq k}^{N} P_i$$

Now we can describe the algorithm in a number of steps.

1. Determine $P^{(M)}$, $P^{(m)}$ and $P^{(\mu)}$

2. Our goal is to find a minimum, from now on reffered to as the minimizer of P, P^*. Using the simplex we have, we calculate as an approximation of P^*, the point

$$\bar{P} = \frac{1}{N+1} \sum_{i=0}^{N} P_i$$

Now we check if $\bar{P}$ is a good approximation of P^*. For this we want the standard deviation of the function values $f(P_0, \ldots, P_N)$ from

$$\bar{f} = \frac{1}{N+1} \sum_{i=0}^{N} f(P_i)$$

have an upper bound by some tolerance or error ϵ, typical values of ϵ and other variables introduced in this section can be found after the steps.

$$\frac{1}{N+1} \sum_{i=0}^{N} \left(f(P_i) - \bar{f} \right)^2 < \epsilon$$

If this is not the case, $\bar{P}$ is not a good approximation of P^* and we will generate a new point for the simplex by reflecting $P^{(M)}$ with respect to $P_c^{(M)}$. We call this point P_r

$$P_r = (1+\alpha) P_c^{(M)} - \alpha P^{(M)}$$

Here $\alpha \geq 0$ is the reflection factor.

3. reflection When $f\left(P^{(m)}\right) \leq f(P_r) \leq f\left(P^{(\mu)}\right)$, $P^{(M)}$ is replaced with P_r and the algorithm starts again with the new simplex. Due to the reflection always being generated and the shape of most valleys (where going in the opposite side of the highest value goes towards the minimum), this is the most common situation.

4. expansion When $P_r \leq P^{(m)}$, the reflection step has produced a new minimizer, meaning the minimizer could lie outside the simplex being considered, therefore the simplex must be expanded after the reflection (so the reflection still happens). We define P_e by

$$P_e = \beta P_r + (1 - \beta) P_c^{(M)}$$

Here β is the expansion factor. Two possibilities arise at this point.

- $f(P_e) < f\left(P^{(m)}\right)$, then $P^{(M)}$ is replaced by P_e.
- $f(P_e) \geq f\left(P^{(m)}\right)$, then $P^{(M)}$ is replaced by P_r.

5. Contraction When $P_r > P^{(\mu)}$, then the minimizer probably lies within the simplex and we want to apply a contraction step. There are two approaches we can take to do this. If $f(P_r) < f\left(P^{(M)}\right)$, we generate a contraction point by

$$P_{co} = \gamma P_r + (1 - \gamma) P_c^{(M)}$$

where $\gamma \in (0, 1)$. Otherwise this contraction point is generated by

$$P_{co} = \gamma P^{(M)} + (1 - \gamma) P_c^{(M)}$$

Now depending on the value of $f(P_{co})$, two things can happen. If $f(P_{co}) < f\left(P^{(M)}\right)$ and $f(P_{co}) < f(P_r)$, we replace $P^{(M)}$ with P_{co}. Otherwise, N new points will be created, by halving their distance to $P^{(m)}$.

As mentioned before, there are typical values for α, β and γ for the downhill simplex algorithm. This is where it varies from the Nelder-mead method, as the Nelder-Mead method does not have typical values for these variables. As such we could say that the downhill simplex algorithm is part of the set of Nelder-Mead algorithms. The typical values for the downhill simplex method are $\alpha = 1$, $\beta = 2$ and $\gamma = \frac{1}{2}$.

Chapter 3

Applying the Downhill Simplex Method

Now that we know how the downhill simplex algorithm works, we will apply this method to some practical examples, to study the strong and weak points. For this we will make use of Matlab with the built-in function $fminsearch$. This function is based on the downhill simplex algorithm and knowing Matlab, it will have some tricks to improve its robustness, allowing to study the weak points that can not easily be solved.

The tests we will give the downhill simplex method are of several forms. First we will try the method on functions that are well-defined and of which we know where the global minimum is, allowing us to test the performance of the downhill simplex method. Afterwards we will try to use this method to optimize two algorithms that are based on fixed point iteration, in order to test the method in a 'realistic manner'.

3.1 Test functions

We will apply the downhill simplex method to two testing functions. These functions are the Rosenbrock function and the Rastrigin function. For both of these functions the global minimum is known. This and other properties of these functions allow us to test the behaviour of the downhill simplex method.

3.1.1 The Rosenbrock function

The Rosenbrock function is given by:

$$f(x, y) = (a - x)^2 + b\left(y - x^2\right)^2$$

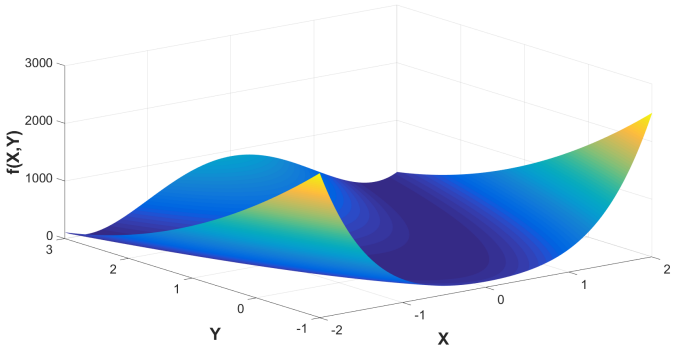

Figure 3.1: The Rosenbrock function in 2 dimensions

This function has a global minimum at $f(a, a^2) = 0$. The values that are generally used for this function are $a = 1$ and $b = 100$. These are also the values used in this study.

As we can see in figure 3.1, the Rosenbrock function has a wide valley, with a small gradient. This creates a challenge for optimization algorithms.

When testing the *fminsearch* algorithm for decent initial guesses, everything works well and the valley appears to be no challenge for this method. When the initial guess is located far from the actual minimum, the algorithm struggles. This can also be seen in table 3.1.

Table 3.1: Behaviour of the Downhill Simplex method when applied to the Rosenbrock function

Initial guess (x,y)	Output Downhill Simplex	Time (seconds)
(2, 2)	(1.0000, 1.0000)	0.008903
(10, 10)	(1.0000, 1.0000)	0.002608
(-10, -10)	(1.0000, 1.0000)	0.002101
(-100, -100)	(1.0000, 1.0001)	0.002276
(-10000, 10000)	(-68.3127, 4666.3312)	0.003979

As one can see, when we are far away from the minimum, in this case at the point $(-10000, 10000)$, the Downhill simplex method does not converge to the minimum, but rather it thinks the minimum is at an entirely different place.

The time that is shown in table 3.1 will be used later, so that we can compare the Downhill Simplex method with other methods.

3.1.2 The Rastrigin function

The Rastrigin function is given by:

$$f(x, y) = 20 + \big(x^2 - 10\cos(2\pi x)\big) + \big(y^2 - 10\cos(2\pi y)\big)$$

The global minimum for this function is at $(x, y) = (0, 0)$, $f(0, 0) = 0$.

As we can see in figure 3.2 on the following page, this function has a ton of local minima. Therefore the optimization algorithms have a lot of trouble finding the global minimum.

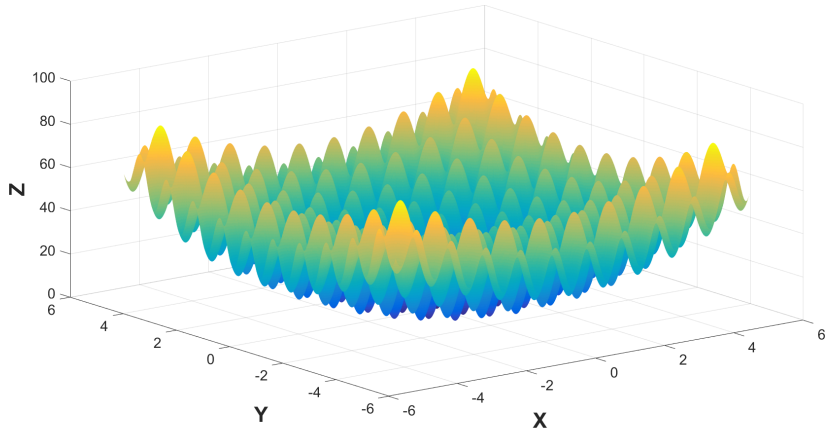

Figure 3.2: The Rastrigin function in 2 dimensions

When applying the Downhill Simplex method on the Rastrigin function, results are highly dependent on the initial guess. The Downhill Simplex method can not distinguish between a local minimum and global minimum. As such it converges to a local minimum close to the initial guess. This behaviour can be seen in table 3.2.

Table 3.2: Behaviour of the Downhill Simplex method when applied to the Rastrigin function

Initial guess (x,y)	Output Downhill Simplex	Time (seconds)
(0, 0)	(0, 0)	0.002704
(1, 1)	(0.9950, 0.9950)	0.001976
(1.5, 1.5)	(1.9899, 1.9899)	0.001708
(2, 2)	(1.9899, 1.9899)	0.010765
(-5, -5)	(-4.9747, -4.9747)	0.002198
(100, 100)	(-0.9950 , -0.9950)	0.003308

As one can see in this table, the Downhill Simplex method finds a local minimum relatively close to the initial guess, when the initial guess is close to the minimum. When we go far away though, it seems the behaviour is better than expected. For the initial guess of $(100, 100)$, the methods results in an output (proposed minimum) of $(-0.9950, -0.9950)$. This is relatively close to the global minimum at $(0, 0)$.

An explanation for this could be that in the beginning stages, there are a lot of expansion steps, making the simplex grow, allowing the algorithm to 'skip' some of the local minima close to the global minimum.

Once again the time will be used later to make a comparison to other methods.

3.2 Optimizing algorithms using the Downhill Simplex method

Now that we know how the Downhill Simplex method performs on the Rosenbrock and Rastrigin functions, we will try and apply this method to optimize numerical algorithms. The algorithms we will try to optimize in this section are algorithms that solve equations using succesive substitution, also known as fixed point iteration.

3.2.1 Fixed point iteration in one dimension

In this section the Downhill Simplex method is used to optimize the variables a and b used in the following script.

```
Nit = 0;
x = 1;
error = 1000;
while error > 1.0E-8
    Nit = Nit + 1;
    xold = x;
    x = x + a * (sin(x) + x*x - 2);
    x = b*x + (1-b)*xold;
    error = abs(x-xold);
end
```

This is a script that solves the equation $\sin(x)+x^2 = 2$ using succesive substitution under relaxation. The goal is to optimize the variables a and b to get the lowest number of iterations Nit.

Defining this script as a function of a and b, with the number of iterations as its output is the first step. In this way we have defined a function $Nit(a, b)$, over which we can try to run $fminsearch$. Notice that we cannot determine the partial derivatives with respect to a and b.

Running $fminsearch$ on $Nit(a, b)$ gives results that raise more questions. For some initial guesses fminsearch finds what appears to be a solution, while for most starting guesses it just gives the initial guess as its answer. In order to know what is going on, we have to analyse the behaviour of the function.

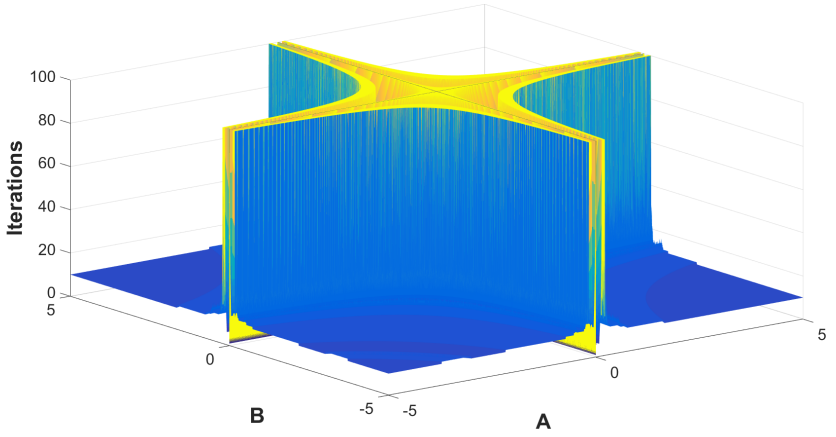

Figure 3.3: Behaviour of the function

In figure 3.3 a plot is shown of the function, limited to 100 iterations. The first to be noticed is that outside of the star-like pattern, the number of iterations gradually lowered. Unable to explain this from mathematical theory, I decided to look at the solution for x the script got at some of these points, i.e. the x that would solve $\sin(x) + x^2 = 2$. The value this x took outside of the star-like formation turned out to be NaN, which means not a number. The script makes the value of x go to infinity really quickly and when x becomes NaN, the error can not be computed and as such the script thinks it is done.

In order to solve this problem, a check is added as to whether or not x is actually a number and if x is not a number, we define the number of iterations to be NaN as well. The result is shown in figure 3.4.

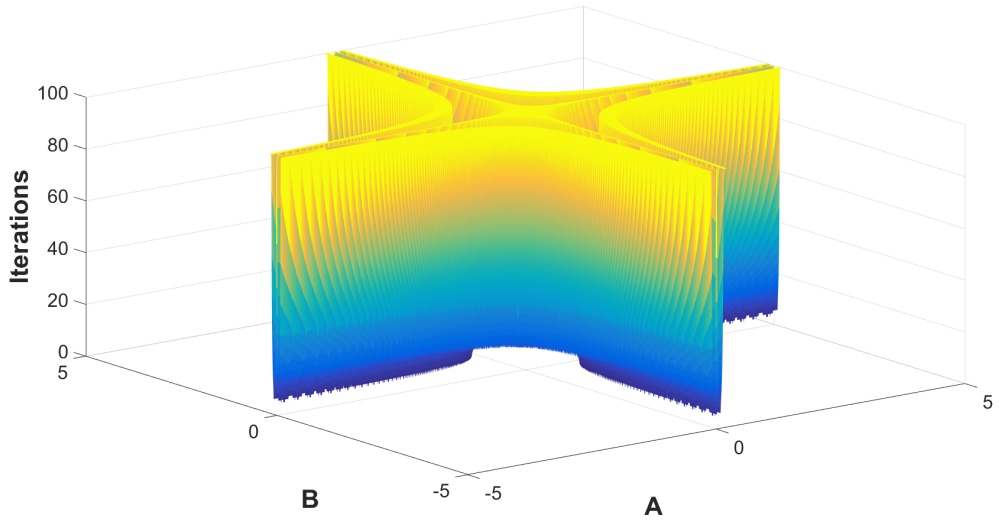

Figure 3.4: Behaviour of the function after alterations

When we turn the figure in such a way that we look from the top down as depicted in figure 3.5, we can see a very interesting figure.

The colour-coding is in such a way that the darker the color, the lower the number of iterations. We can see that there are two types of valleys in this figure. the top-left and bottom-right are very similar to each other, as are the top-right and bottom-left. A very unfortunate downside of the Downhill Simplex method is encountered here. When the initial guess is in the top-right or bottom-left valley, the algorithm finds a local minimum inside those valleys, making it so it will not find the global minimum in another valley. This behaviour can be seen in table 3.3.

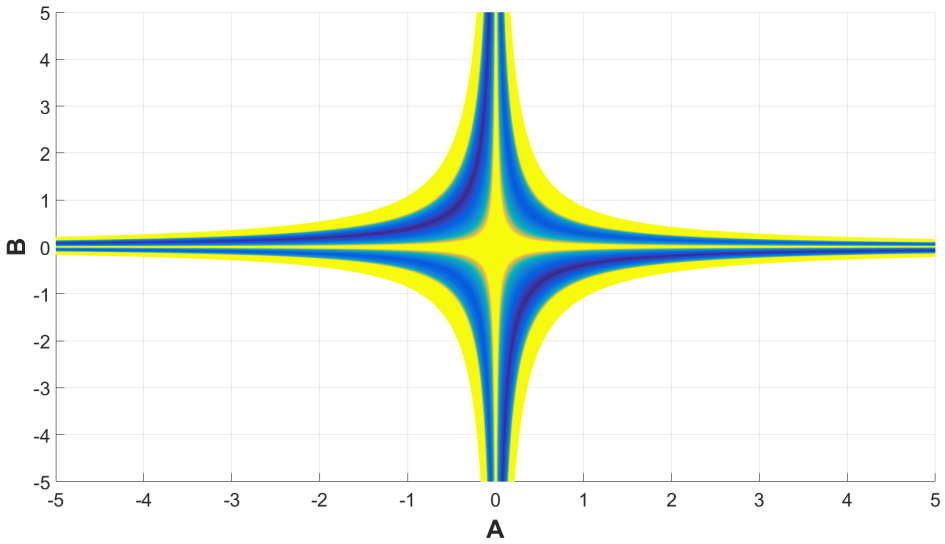

Figure 3.5: Behaviour of the function from top down

Table 3.3: Behaviour of the Downhill Simplex method when applied to successive substitution

Initial guess (a,b)	Output Downhill Simplex	Time (seconds)
(0.5, 0.5)	(0.5328, 0.5203)	0.062708
(-0.5, 0.5)	(-0.6312, 0.6062)	0.012486
(0.5, -0.5)	(0.6312, -0.6062)	0.026526
(-0.5, -0.5)	(-0.5328, -0.5203)	0.005909
(100, 100)	(100,100)	0.009560

The structure of the darkest parts of figure 3.5 is reminiscent of the equation

$$b = -\frac{1}{a}$$

However when we take that exact equation, we end up in the yellow parts rather than the blue parts. As such, we can try plotting the results when

$$b = -\frac{1}{q \cdot a}$$

Plotting this results in figure 3.6.

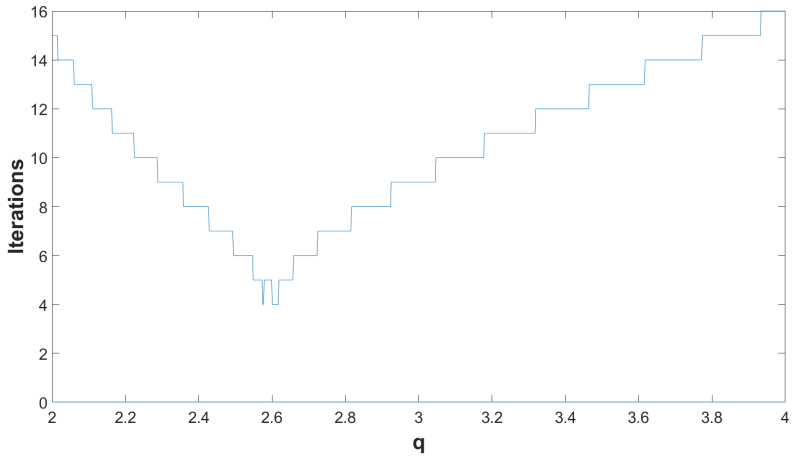

Figure 3.6: The number of iterations for different values of q

The structure we see in figure 3.6 does not change for different values of a, as long as $a \neq 0$. This means that for this specific script there is a relation between the optimal a and b. What is also interesting to note, is that this relation still holds when we change the equation we try to solve. However the optimal values of q do change.

3.2.2 Fixed point iteration in multiple dimensions

In this section we will try to use the Downhill Simplex method to optimize an algorithm for fixed point iterations in three dimensions. It is an algorithm that solves the equations:

$$x = f(y, z) = \frac{1}{3}\cos(yz) + \frac{1}{6}$$

$$y = g(x, z) = \frac{1}{9}\sqrt{x^2 + \sin(z) + 1.06} - 0.1$$

$$z = h(x, y) = \frac{-1}{20}e^{-xy} - \frac{10\pi - 3}{60}$$

Pseudo code for this algorithm is as follows:

```
Nit = 0;
x = 2, y = 2, z = 2;
error = 1000;
while error > 1.0E-5
    Nit = Nit + 1;
    xold = x;
    yold = y;
    zold = z;
    x = a * f(y,z);
    y = b * g(x,z);
    z = h(x,y);
    error = sqrt((x-xold)^2 + (y-yold)^2 + (z-zold)^2);
end
```

The actual code used for this system, an alteration on a script written by Alain G. Kapitho [3], can be found in appendix A.9.

For this code we once again want to optimize the number of iterations Nit, where a and b are the variables we want to optimize. In order to do so we define the number of iterations Nit as a function of a and b. The reason we will not add a third variable c for the third equation, is that by keeping the number of variables at two allows us to make a plot that shows us exactly what is happening. This can be found in figure 3.7 on the following page.

Once again we can take a look at this figure from top down, in order to get a better idea of what is going on. The resulting figure is figure 3.8.

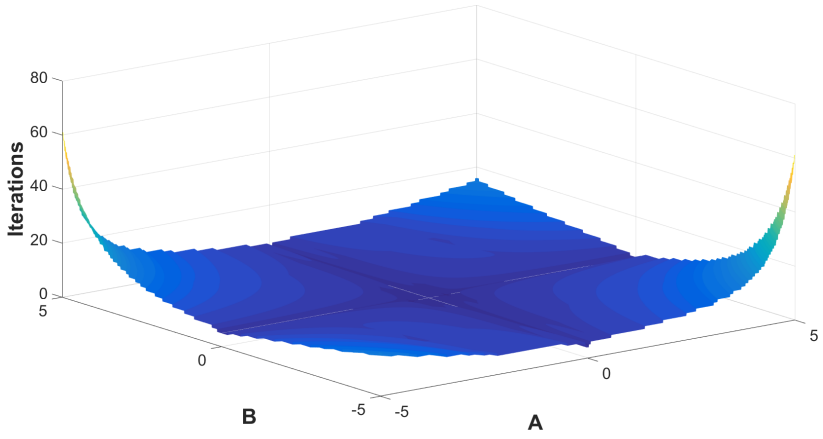

Figure 3.7: Behaviour of multidimensional fixed point iteration

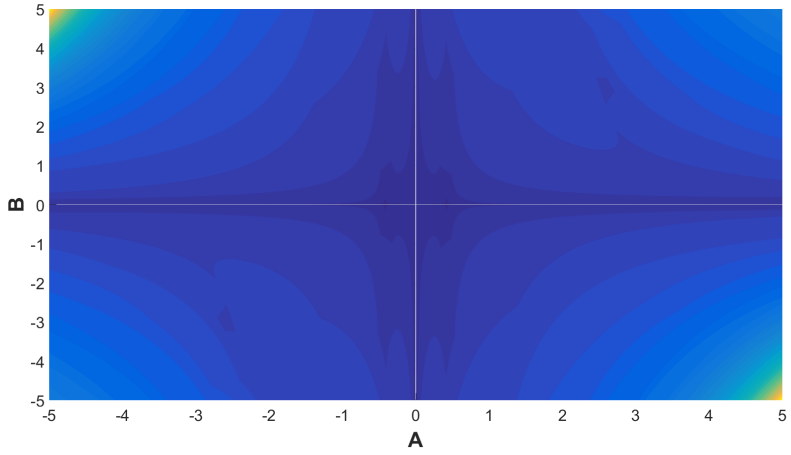

Figure 3.8: Behaviour of multidimensional fixed point iteration, topdown

The algorithm was modified in such a way that the number of iterations is NaN when $a = 0$ or $b = 0$. This was done because the algorithm misbehaves in this case.

When testing the Downhill Simplex method on this function, the results are rather disappointing. Looking at the behaviour of this function one would assume that an optimization algorithm should have no issues finding the minimum. In reality though, the minimum was only found when the initial guess was sufficiently close.

The behaviour can be seen in table 3.4

Table 3.4: The behaviour of the Downhill Simplex method when applied to multidimensional fixed point iteration

Initial guess (a,b)	Output Downhill Simplex	Functionvalue	Time (seconds)
(0.1, 0.1)	(0.1000, 0.1000)	4	0.002534
(-0.1, 0.1)	(-0.1000, 0.1000)	4	0.001951
(1, 1)	(1, 1)	6	0.003093
(-1, 1)	(-1, 1)	6	0.001982
(5, 4)	(5, 4)	13	0.003402
(4,-5)	(-0.0000, -4.2500)	4	0.002884

For this table the function values are shown for the output of the Downhill Simplex method.

Quite often the initial guess supplied to the Downhill simplex method was given as the output as well meaning the method did not really do anything. Looking at the theory behind the Downhill Simplex method allow us to get a better understanding of why this is happening.

In section 2.2, we can see that the Downhill Simplex algorithm checks that it has found minimum by checking that the standard deviation of function values of its points is sufficiently small,

$$\frac{1}{N+1}\sum_{i=0}^{N}\left(f\left(P_i\right)-\bar{f}\right)^2<\epsilon$$

When analysing figure 3.8, we can see that there are a lot of areas where the figure is entirely flat. This results in the standard deviation being exactly zero, meaning that the algorithm thinks it has found the solution.

Hence, the Downhill Simplex method does not perform well when applied to algorithms similar to this multidimensional fixed point iteration algorithm.

Chapter 4

Powell's Methods of Optimization

In this chapter we will discuss our second optimization algorithm, Powell's method. In contrast to the simplex method, Powell's method does make use of algorithms that do one-dimensional optimization/minimization. As such it is necessary for us to first analyze one-dimensional optimization methods.

4.1 One-dimensional optimization

With the restriction of not having a gradient or in case of one dimensional optimization, the (directional) derivative, there are two methods of one-dimensional minimization we will consider. These are the Golden Section Search method and Parabolic Interpolation.

The Golden Section search is relatively slow, but it is very robust as it does not make any assumptions with respect to the behaviour of the function that it is trying to optimize. Parabolic Interpolation is on the opposite side of the spectrum in that it can be a lot faster for well behaving functions, but it is not as robust. This is because Parabolic Interpolation assumes the function can be approximated by a parabolic function near its minimum. This works very well for a lot of functions, however for functions where this behaviour is not valid, Parabolic Interpolation will simply not find a minimum.

As the goal is to have a robust algorithm, not necessarily an algorithm that is as fast as possible, we will apply the Golden Section Search as our main method for one-dimensional optimization/minimization. In the testing phase we will also try to use Parabolic interpolation and see how it performs compared to Golden Section Search.

4.1.1 Golden Section search

Golden section search is based on the bisection method, where we will repeatedly bisect an interval into subintervals and determine in which subinterval there is a minimum.

We start out by bracketing a minimum:
Definition 4.1.1. We say a minimum of function $f(x)$ is bracketed by two points a, b, $a < b$, when there is a point c, such that $a < c < b$ with $f(c) < f(a)$ and $f(c) < f(b)$.

We will not handle methods for bracketing a minimum, though the code used to do this can be found in appendix A.15.

When we have bracketed this minimum, we can choose a new point x in the interval (a, b). Depending on the function value $f(x)$ at this new point x, we can choose between two new intervals to evaluate for a minimum.

Although this theory is valid for any x we choose, for simplicity we assume x is chosen in the interval (c, b). Then one of the following things will happen.

- If $f(c) < f(x)$, we will look at the interval (a, x)
- If $f(x) < f(c)$, we will look at the interval (c, b)

This behaviour is visualised in figure 4.1 on the following page, where the colored lines are the newly choosen intervals.

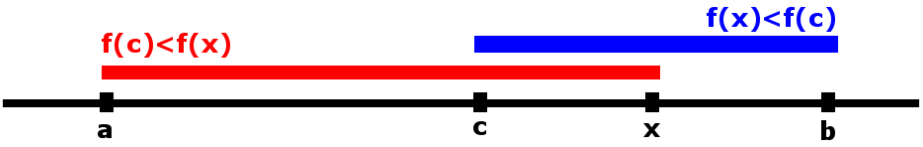

Figure 4.1: Choosing a new interval

The algorithm simply stops when the interval becomes small enough.

This algorithm works to find a local minimum no matter how we choose the x as long as $x \in (a, b)$. However, you can end up at different local minima depending on how x is chosen. We do however want this algorithm to always work as well as it possibly can. This means that we want the algorithm to finish as quickly as possible, even when we always have to go to the larger interval of our two possible new intervals.

This means that we want the two possible new intervals to be of the same size. In order to achieve this, we can define the distances as fractions of the total interval. We define w to be the fraction ac of ab:

$$w = \frac{c - a}{b - a}$$

In a similar way, we can define z to be the fraction cb of ab:

$$z = \frac{x - c}{b - a}$$

This means that the two possible intervals can be defined in terms of w and z:

- The first interval can be defined as $w + z$
- The second interval can be defined as $1 - w$

When also taking account our wish for these intervals to be of the same size, we arrive at the following equation:

$$w + z = 1 - w \Rightarrow z = 1 - 2w$$

Usually, c is the result of the previous iteration, meaning that w is already chosen optimal. As such, x should be the same fraction from c to b as c was from a to b:

$$\frac{z}{1-w} = w$$

Combining that $z = 1 - 2w$ and $\frac{z}{1-w} = w$, we get that

$$w^2 - 3w + 1 = 0 \rightarrow w = \frac{3 - \sqrt{5}}{2}$$

This means that when all points are choosen optimally, w and $1 - w$ are the fractions of the Golden Ratio. This is why this method was named the Golden Section search.

4.2 Multidimensional optimization

As mentioned before, Powell's methods perform multidimensional optimization, using one-dimensional optimizations. This is done by performing one-dimensional optimization on a number of directions and continue doing so until a minimum is found.

The way of choosing the directions can vary significantly. First we will handle the most basic way of choosing the directions.

- Begin with the initial guess and minimize in the direction of the standard basisvector e_1.
- From the resulting point, minimize in the direction of e_2.
- Continue in this way until e_N.
- Keep cycling through $e_1, \ldots, e_N$ until there is nu longer significant change.

This method works well for a lot of functions, in that it is successful in finding a minimum. This method is however very inefficient. An example would be a two-dimensional function that has a narrow valley, in a direction that is not the

same as either e_1 or e_2. What this method does in such a case, is take a small in direction e_1, then take a small step in direction e_2, then take a small step direction e_1 again. This goes on and on with very small steps. Eventually it will find the minimum in the valley, but it takes a very long time.

By changing the way we choose directions, we can make this method a lot more robust and a lot faster. From the example we can for example imagine that we would want to search in the direction of the valley.

4.2.1 Powell's conjugate direction method

Powell introduced a new way of choosing the directions on which to search for a minimum. We start out the same way we did before, starting at a point x_0 and minimizing in the direction of every basis-vector e_i once. After these N line searches, we have arrived at some point x_N. Now the direction of the vector $x_N - x_0$ is added at the end of the cycle. The most successful basis vector, i.e. the basis vector that contributes most to the direction of $x_N - x_0$, is removed from the cycle.

Now we start our cycle again from the point we have arrived at. These cycles continue until the algorithm has found a minimum.

As one can see, this method of taking directions solves some of the weaknesses from the most basic way of choosing directions that we handled before.

Taking the same example we used earlier, a two-dimensional function with a narrow valley that is not in the direction of e_1 or e_2, the following would happen. First we take small steps in directions e_1 and e_2 again. Then we define our new direction $x_2 - x_0$. For a straight valley, this is the direction of the valley. Optimizing in this direction results in the minimum.

So rather than a lot of small steps, this algorithm takes two small steps and then one big step in order to find the minimum.

Chapter 5

Applying Powell's Method

Now that we have explained the theory behind Powell's method of optimization, we will subject this method to the same tests as the Downhill Simplex method. This allows us to analyse the performance of Powell's method, but it also enables us to make a comparison between these methods.

In order to apply Powell's method, we will be making use of a number of scripts that have been shared on the Mathworks forum by Argimiro R. Secchi in 2001 [4]. The scripts can be found in appendix A.14, A.15, A.16, and A.17.

5.1 Test functions

Once again we will use the Rosenbrock function and the Rastrigin function as our tests for Powell's method. For the finer details on the Rosenbrock function, readers can look back at section 3.1.1 and for the Rastrigin function at section 3.1.2.

5.1.1 Rosenbrock function

Powell's function performs very well on the Rosenbrock function, finding the global minimum at $(x, y) = (1, 1)$, even when the initial guess is suboptimal.

Table 5.1: Behaviour of Powell's method with golden section search when applied to the Rosenbrock function

Initial guess (x,y)	Output Powell	Time (seconds)
(2, 2)	(1.0000, 1.0000)	0.003912
(10, 10)	(1.0000, 1.0000)	0.003784
(-10, -10)	(1.0000, 1.0000)	0.003050
(-100, -100)	(1.0000, 1.0000)	0.002306
(-10000,10000)	(1.0000, 1.0000)	0.002189

Table 5.2: Behaviour of Powell's method with parabolic interpolation when applied to the Rosenbrock function

Initial guess (x,y)	Output Powell (Coggins)	Time (seconds)
(2, 2)	(1.0000, 1.0001)	0.002198
(10, 10)	(1.0001, 1.0001)	0.002483
(-10, -10)	(1.0000, 1.0000)	0.002599
(-100, -100)	(1.0000, 1.0000)	0.002445
(-10000,10000)	(0.9999, 1.0000)	0.002398

In table 5.1, we see the behaviour of Powell's method with golden section search on the Rosenbrock function. In table 5.2, we see the behaviour of Powell's method with parabolic interpolation on the Rosenbrock function.

For the Rosenbrock function we see that performance for both golden section search and parabolic interpolation is great. The weakness of the Downhill Simplex method where it does not converge to the minimum when we go too far away from the minimum is not present here either.

When we compare the speed to that of the Downhill Simplex method, in table 3.1 on page 12, we see that the difference in speed is almost negligible, as all these times are inside a small margin of error.

5.1.2 Rastrigin function

When applying Powell's methods to the Rastrigin function, things become very interesting.

Table 5.3: Behaviour of Powell's method using golden section search when applied to the Rastrigin function

Initial guess (x,y)	Output Powell	Time (seconds)
(0, 0)	1e-4*(-0.2372, -0.2372)	0.434990
(1, 1)	(0.9950, 0.9950)	0.001439
(1.5, 1.5)	(0.9950, 0.9950)	0.000520
(2, 2)	(1.9899, 1.9899)	0.003093
(-5, -5)	(-4.9747, -4.9747)	0.001602
(100, 100)	1e-4*(0.1987, 0.1987)	0.433081

Table 5.4: Behaviour of Powell's method using parabolic interpolation when applied to the Rastrigin function

Initial guess (x,y)	Output Powell (coggins)	Time (seconds)
(0, 0)	(0, 0)	0.000484
(1, 1)	(0.9950, 0.9950)	0.001647
(1.5, 1.5)	(0.9950, 0.9950)	0.000725
(2, 2)	(1.9899, 1.9899)	0.001446
(-5, -5)	(-4.9747, -4.9747)	0.001374
(100, 100)	1e-13*(-0.0940, -0.1174)	0.001570

Looking at tables 5.3 and 5.4, one can see that the Powell method using golden section search fails to find the minimum when it starts out at the minimum. It does however get quite close to the minimum. Nevertheless, this behaviour is rather unexpected.

Another thing that grabs the attention is that just like the Downhill Simplex method, Powell's method also works rather well when we start from a point that is far away from the minimum. Both the golden section search, but especially the quadratic interpolation variant get very close to the global minimum when we start at $(100, 100)$.

Comparing methods, the obvious winner is Powell's method using quadratic interpolation. This method assumes that the function it is trying to minimize behaves like a quadratic function. As the Rastrigin function is a quadratic function of sorts, it makes sense that this method works relatively well.

After looking at the test functions, we can say performance of Powell's methods and the Downhill Simplex method are very similar, with some exceptions. The fastest method seems to be Powell's method using quadratic interpolation. Now it is time to see how Powell's method performs on algorithms.

5.2 Applying Powell's method to algorithms

We use the same algorithms as we did in section 3.2. For details the reader is recommended to look back at section 3.2.

5.2.1 Algorithm 1

The first algorithm is successive substitution under relaxation, essentially fixed point iteration in one dimension. As expected, results here are very similar to the results when using the Downhill Simplex method, at least when we are talking about Powell's method using golden section search.

Table 5.5: Behaviour of Powell's method using golden section search when applied to successive substitution

Initial guess (a,b)	Output Powell	Time (seconds)
(0.5, 0.5)	(1.2391, 0.2230)	0.447472
(-0.5, 0.5)	(-0.6037, 0.6372)	0.003661
(0.5,-0.5)	(2.4091, -0.1585)	0.020487
(-0.5,-0.5)	(-0.8961, -0.2991)	0.015762
(100,100)	error	

When looking at table 5.5 and keeping in mind the shape of figure 3.5 on page 17, if we start inside of the figure, we stay in the quadrant the initial guess is in. This

Table 5.6: Behaviour of Powell's method using parabolic interpolation when applied to successive substitution

Initial guess (a,b)	Output Powell (coggins)	Time (seconds)
(0.5, 0.5)	Coggins failure	
(-0.5, 0.5)	Coggins failure	
(0.5,-0.5)	Coggins failure	
(-0.5,-0.5)	Coggins failure	
(100,100)	Coggins failure	

means that the local minimum is found, but the global minimum is only found when the initial guess is at the correct quadrant. This is the same as with the Downhill Simplex method.

When we start outside of the star-like pattern, in this case with the initial guess of $(100, 100)$, Powell's method gives an error, as it can not calculate a minimum, here the Downhill Simplex method thought the minimum was at the initial guess.

When looking at the results of the parabolic interpolation method in table 5.6, we see why this method is known to be less robust. Parabolic interpolation fails to bring results, it returns a failure that has to do with exceeding the maximum number of iterations. This suggests the parabolic interpolation algorithm gets stuck in an infinite loop.

5.2.2 Algorithm 2

The second algorithm uses fixed point iteration in three dimensions. Performance in this case is very interesting. Remember that the Downhill Simplex was done very quickly for most initial guesses, giving the initial guess as the output.

Performance of Powell's method with golden section search is similar in this case, but it does not return the input as the output. Instead, Powell's method seems to try very hard to find at least something, always with the same amount of success. Especially when starting far away from the minimum, Powell's method seems to struggle to find a minimum. This can all be seen in table 5.7

Table 5.7: Behaviour of Powell's method with golden section search on multidimensional fixed point iteration

Initial guess (a,b)	Output Powell	Functionvalue	Time (seconds)
(0.1, 0.1)	(0.5125, 0.5088)	4	0.018173
(-0.1, 0.1)	(0.4930, 0.6882)	4	0.021067
(1, 1)	(1.8264, 1.8180)	6	0.003093
(-1, 1)	(0.0089, -0.0032)	3	0.514112
(5, 4)	(5.0699, 4.0692)	13	0.006379
(4,-5)	(4.0630, -4.9201)	19	0.011121

When switching to parabolic interpolation rather than golden section search, results become a lot more interesting, as is seen in tabe 5.8.

Table 5.8: Behaviour of Powell's method with parabolic interpolation on multidimensional fixed point iteration

Initial guess (a,b)	Output Powell (Coggins)	Functionvalue	Time (seconds)
(0.1, 0.1)	(0.1000, 0.1000)	4	0.003493
(-0.1, 0.1)	(-0.1000, 0.1000)	4	0.000868
(1, 1)	(-0.1250, 1.0000)	4	0.001627
(-1, 1)	(NaN, NaN)		119.680230
(5, 4)	(0.0156, 4.0000)	4	0.003194
(4,-5)	(0.0003, -5.0000)	4	0.002146

Now, for most initial guesses, the output is a pretty close to the actual minimum. There was one initial guess during the testing phase that did not work for parabolic interpolation and that is the initial guess of $-1, 1$. Here the parabolic interpolation algorithm exceeds the maximum number of iterations time after time and eventually, after a large amount of time, the result is NaN for both a and b.

All in all, looking at the results tells us that it is highly dependent on the problem which method has more success. Both the Downhill Simplex method and Powell's method with golden section search are robust in giving an output, but they do not always give the best results. Powell's method with parabolic interpolation on the other hand has the tendency to fail completely, but it is also the fastest method when it does work. The speed of the other two algorithms is very similar.

Chapter 6

Combining Methods

The goal of this chapter is to use the Downhill Simplex method and Powell's methods to create an algorithm that has a higher chance of finding the global minimum rather than a local minimum. The reason for wanting to combine both methods is that there might be cases where one method does work and the other doesn't. When combining these methods, we know for sure the algorithm has the highest possible chance of success.

6.1 Using a grid

Using algorithms that are good at finding local minima in order to find a global minimum is a challenging task. We will attempt to do this by using a grid.

By using a grid and running both algorithms with each grid point as an initial guess, we get two local minima for each of these initial guesses, one supplied by the Downhill simplex method, the other supplied by Powell's methods. By selecting the minimum of these local minima we get our best guess of the global minimum.

One can imagine that using this method for a large, dense grid can get very computationally expensive. Another downside of using a grid, is that even if we choose a very large and dense grid, there is still a possibility that the global minimum lies outside our grid.

6.1.1 Choosing the grid

In order to make use of a grid as efficient and robust as possible, we want to choose our grid in an efficient manner. As one can imagine there are a lot of ways to choose a grid. One can choose a grid that is very large and where the grid points are sparse, you can choose to have a very horizontal oriented grid, or a vertical oriented one. Let us start out by defining a grid at all.

When using an initial guess x_0, we can define our grid to go a distance p in every direction from this x_0, with a distance dp between each grid point. This results in a grid like in figure 6.1

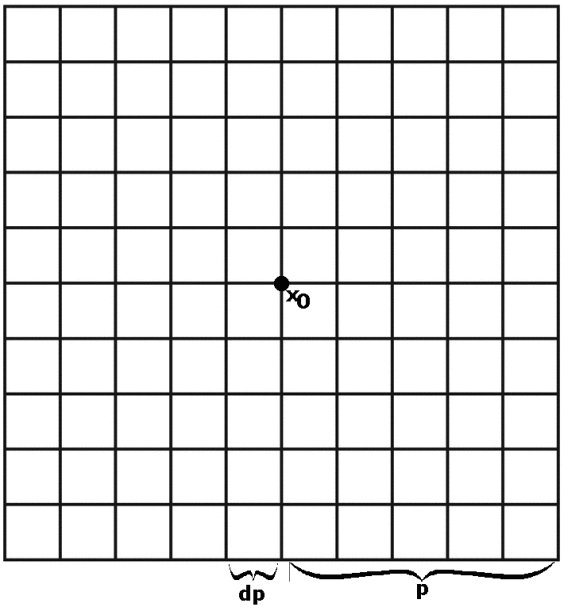

Figure 6.1: Basic way of choosing a grid

As we already know the behaviour of our test functions, this basic grid is sufficient for us to test the performance and behaviour of using a grid. We will be testing the use of a grid on the Rastrigin function and also on the algorithm with successive substitution.

6.1.2 Testing a basic grid

For this section we will be running both te Downhill Simplex method and Powell's method with the golden section search on a grid, where we know that the minimum is already within the grid.

Rastrigin function

When having a grid from -5 to 5 for both x and y, with grid points every 0.25, the resulting output is, as expected, $(0, 0)$. The interesting part is seeing how long this takes. The answer for this grid is 8.892410 seconds. This is considerably longer than with a single initial guess. This was expected as well.

It is also interesting to see how this function behaves when the initial guess is not one of the grid points. To do so we adjust our grid and place it from -2 to 4 with grid points every 0.75. Now the resulting output is $(-0.1426, -0.0522)$. The function value at this point is 0.0005. This means that although we did not get to our minimum, we did get very close to it. The difference between this and the actual global minimum can be explained by tolerances in our optimization algorithms. It took 0.750029 seconds to run the algorithm this time.

Successive substitution

Another interesting testing case for the grid is the successive substitution algorithm. We take the same initial grid we did in the previous case. The resulting point is $(1.5667, -0.2438)$. In fact there were two points supplied that were both on the line where the global minimum lies, it simply chose the first one, given by Powell's method. The point supplied by the Downhill Simplex method is $(1.5667, -0.2438)$. The time it takes to make this evaluation is 4.520256 seconds.

As expected, when the grid is chosen efficiently, using a grid like this works very well indeed. Where this method runs into problems is when individual iterations of the optimization algorithms take a long time, as these algorithms are run quite often for every grid point, resulting in running these algorithms over the grid taking a very long time.

It is also very interesting to see that even when running over a grid, the approximation of the global minima found by both methods do vary ever so slightly. This means that it was indeed a good idea to combine the methods, as this results in the best approximation of the global minimum.

Now that we know that using a well chosen grid, while being computationally expensive, does work, we want to find a way to always choose our grid decent, without having a very large very dense grid, as this would be too computationally expensive.

Finding the most efficient ways to choose this grid is something that can be done with further research that will not be handled now. One can however imagine that such a method would involve starting out with a large grid and making it smaller until a minimum is found.

Chapter 7

Conclusion

In order to optimize and minimize functions that are not defined in a way that we can take the gradient, we have analysed two numerical methods of finding minima, namely the Downhill Simplex method and Powell's methods.

Both methods are decently good at finding a local minimum, but finding a global minimum can only be done with a well chosen initial guess. In order to remedy this, we introduced a grid on which to run these algorithms. Using a well chosen grid works as expected and finding a global minimum in this way is possible.

Downsides of using a grid is that it is computationally expensive. Choosing the grid in an efficient manner is something for which more research can be done.

Bibliography

[1] A. Quarteroni, R., and Saleri, F. *Numerical Mathematics*. Springer, 2000.

[2] Kankelborg, C. Multidimensional minimization. Lecture Notes, 2009. http://solar.physics.montana.edu/kankel/ph567/examples/Octave/minimization/amoeba/NotesPh567.pdf.

[3] Kapitho, A. Fixed-point for functions of several variables. http://m2matlabdb.ma.tum.de/download.jsp?Search=181&SearchCategory=AuthorWithFiles&AuthorName=Alain+Kapitho&MP_ID=412, 2006. Accessed July 18, 2017.

[4] Tonel, G. Unconstrained optimization using powell. https://nl.mathworks.com/matlabcentral/fileexchange/15072-unconstrained-optimization-using-powell?focused=5091405&tab=function. Accessed July 18, 2017.

[5] W.H. Press, S.A. Teukolsky, W. V., and FLannery, B. *Numerical Recipes in C*. Press Syndicate of the University of Cambridge, 2002.

Appendix A

Matlab code

A.1 Rosenbrock

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

A.2 Rastrigin

```
%This script defines the rastrigin function globally for easy acces 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));
end
```

A.3 Rosenbrockplot

```
%This function makes a plot of the rosenbrock functions on the interval
%where X goes from -2 to 2 and Y goes from -1 to 3.

clear all
a=5;
X=linspace(-2,2,400*a+1);
Y=linspace(-1,3,400*a+1);
no=0;

for i = 1:length(X)
    for j = 1:length(Y)
        it(j,i) = rosenbrock([X(i),Y(j)]);
        no=no+1;
    end
end

surf(X,Y,it,'EdgeColor','none','LineStyle','none','FaceLighting','phong');
set(gca,'fontsize',24)
xlabel('X','FontSize',32,'FontWeight','bold')
ylabel('Y','FontSize',32,'FontWeight','bold')
zlabel('f(X,Y)','FontSize',32,'FontWeight','bold')
```

A.4 Rastplot

```
%This function is made to make a plot of the Rastrigin function on the
%standard interval where x and y go from -5.12 to 5.12.

X = linspace(-5.12,5.12,1001);
Y = linspace(-5.12,5.12,1001);
for i = 1:length(X)
    for j = 1:length(Y)
        Z(j,i) = rastrigin([X(i),Y(j)]);
    end
end
surf(X,Y,Z,'EdgeColor','none','LineStyle','none','FaceLighting','phong');
set(gca,'fontsize',22)
xlabel('X','FontSize',32,'FontWeight','bold')
ylabel('Y','FontSize',32,'FontWeight','bold')
zlabel('Z','FontSize',32,'FontWeight','bold')
```

A.5 Scriptsolve

```
%This script is a function for solving sin(x) + x^2 = 2 using succesive
%substitution under relaxation and its output is the number of iterations
%it uses to solve this equation.
function it = scriptsolve(z)
    p=100;          %maximum number of iterations allowed
    Nit = 0;
    x = 1;
    error = 1000;
    while error > 1.0E-8
        if Nit < p
            Nit = Nit + 1;
            xold = x;
            x = x + z(1)*(sin(x) + x^2 -2);
            x = z(2)*x + (1-z(2))*xold;
            error = abs(x-xold);
        else
            Nit = p;
            break
        end
    end
    %After using succesive substitution, the following part is meant to
    %verify that this method actually found a solution to sin(x) + x^2 = 2.
    if isnan(x)==1 || abs((sin(x)+x^2-2)^2) > 1.0E-8
        it = NaN(1);
    else
        it = Nit;
    end
end
```

A.6 Plotje

```
%This script is created to make a plot of the behaviour of a function at
%the interval where both x and y go from -a to a.
clear all
%Begin by defining the variables and vectors that are needed.
a=5;
X=linspace(-a,a,400*a+1);
Y=linspace(-a,a,400*a+1);
no=0;
%Creating the functionvalue matrix for every value of X and Y
for i = 1:length(X)
    for j = 1:length(Y)
        it(j,i) = scriptsolve([X(i),Y(j)]);
        no=no+1;
        if mod(no,1000)==0
            %disp(no)
        end
    end
end
%plotting the functionvalue matrix against the vectors X and Y using the a
%surf plot.
surf(X,Y,it,'EdgeColor','none','LineStyle','none','FaceLighting','phong');
set(gca,'fontsize',24)
xlabel('A','FontSize',32,'FontWeight','bold')
ylabel('B','FontSize',32,'FontWeight','bold')
zlabel('Iterations','FontSize',32,'FontWeight','bold')
```

A.7 Scriptsolve2

```
%Alteration of the scriptsolve function, b = -1/q*a
%In this script, a is set at 1, there have been versions of this script
%where a was variable, for testing purposes, it turned out that within
%normal values of a, nothing changes.
function it = scriptsolve2(Q)
    p=100;          %maximum number of iterations allowed
    Nit = 0;
    x = 1;
    a=1;
    b=-1/(Q*a);
    error = 1000;
    while error > 1.0E-8
        if Nit < p
            Nit = Nit + 1;
            xold = x;
            x = x + a*(sin(x) + x^2 -2);
            x = b*x + (1-b)*xold;
            error = abs(x-xold);
        else
            Nit = p;
            break
        end
    end
    %After using succesive substitution, the following part is meant to
    %verify that this method actually found a solution to sin(x) + x^2 = 2.
    if isnan(x)==1 || abs((sin(x)+x^2-2)^2) > 1.0E-8
        it = NaN(1);
    else
        it = Nit;
    end
end
```

A.8 Plotje2

```
%This script was made to plot the behaviour of the scriptsolve function
%when the two variables are dependent on eachother such that b = 1/(q*a)

Q=linspace(2,4,1000);
z=zeros(length(Q));
for i=1:length(Q)
    z(i)=scriptsolve2(Q(i));
end

%Plotting the results
plot(Q,z);
set(gca,'fontsize',24)
xlabel('q','FontSize',32,'FontWeight','bold')
ylabel('Iterations','FontSize',32,'FontWeight','bold')
```

A.9 Fixed_Point_Systems

```
function k = fixed_point_systems(z1)
N = 1000;
z=[1; 1; 1];
z(1) = z1(1);
z(2) = z1(2);
x0 = [2,2,2];

if z(1)==0 || z(2)==0
    k=NaN(1);
else
%Fixed_Point_Systems(x0, N) approximates the solution of a system of non-
%linear equations F(x) = 0 rewritten in the form x = G(x) starting with an
%initial approximation x0. The iterative technique is implemented N times
%The user has to enter the function G(x)at the bottom
%The output consists of a table of iterates whereby each column displays
%the iterates of x1, x2, ..., xn until the maximum number of iterations is
%reached or the stopping criterion is satisfied.

%Author: Alain G. Kapitho
%Date  : Jan. 2006

%The script was slightly altered by introducing the vector z, dependent on
%two variables. It was made into a function, with these variables as input
%and the number of iterations as output
n = size(x0,1);
if n == 1
   x0 = x0';
end

i = 1;
x(:,1) = x0;
```

```
tol = 1e-05;
while i <= N
   x(:,i+1) = z.*G(x(:,i));
   if abs(x(:,i+1)-x(:,i)) < tol                %stopping criterion
      k =i;
      x = x';
      return
   end
   i = i+1;
end
if abs(x(:,i)-x(:,i-1)) > tol
   x = x';
   k=NaN(1);
else if i > N
        x = x';
        k = N;
    end
end
end

%this part has to be changed accordingly with the specific function G(x)
function y = G(x)
y = [(1/3)*cos(x(2)*x(3))+1/6;
   (1/9)*sqrt(x(1)^2 + sin(x(3)) + 1.06) - 0.1;
   (-1/20)*exp(-x(1)*x(2)) - (10*pi - 3)/60];
```

A.10 Plotje3

```
%This script was made to make a plot of the behaviour of the
%fixed_point_systems script. it does so by plotting The number of
%iterations over X and Y.

clear all
a=5;
X=linspace(-a,a,400*a+1);
Y=linspace(-a,a,400*a+1);
no=0;

for i = 1:length(X)
    for j = 1:length(Y)
        it(j,i) = fixed_point_systems([X(i),Y(j)]);
        no=no+1;
        if mod(no,1000)==0
            disp(no)
        end
    end
end

surf(X,Y,it,'EdgeColor','none','LineStyle','none','FaceLighting','phong');
set(gca,'fontsize',24)
xlabel('A','FontSize',32,'FontWeight','bold')
ylabel('B','FontSize',32,'FontWeight','bold')
zlabel('Iterations','FontSize',32,'FontWeight','bold')
```

A.11 Minimum

```
function minimum = minimum(fun1,x0,p,int,e)
%This script was made to run the powell method of optimization on a grid.
%We start out by defining the grid:
m = min(x0(1),x0(2)) - p;
M = max(x0(1),x0(2)) + p;
X=m:int:M;
Y=m:int:M;
%Starting value for our found minimum, infinity:
min2=[x0(1),x0(2),Inf(1)];
for i = 1:length(X)
    for j = 1:length(Y)
        value = fun1([X(i),Y(j)]);
        %Setting the value to Inf if functionevaluation returns NaN.
        if isnan(value)==1
            N = Inf(1);
        else
            %Trying to use powell's method only for decent (working)
            %initial guesses.
            try
                min1 = powell(fun1,[X(i),Y(j)],0,e);
            catch
                warning('Problem using powell.  Terminating function.');
                min1 = [X(i),Y(j)];
            end
            N = fun1(min1);
        end
        %Setting a new minimum when it is smaller than the currently known
        %one
        if N <= min2(3)
            min2 = [min1(1),min1(2),N];
        end
```

```
    end
end
minimum=min2;
```

A.12 Minimum2

```
function minimum = minimum2(fun1,x0,p,int)
%This script was made to run fminsearch on a grid.
%We start out by defining the grid:
m = min(x0(1),x0(2)) - p;
M = max(x0(1),x0(2)) + p;
X=m:int:M;
Y=m:int:M;
%Starting value for our found minimum, infinity:
min2=[x0(1),x0(2),Inf(1)];
for i = 1:length(X)
    for j = 1:length(X)
        value = fun1([X(i),Y(j)]);
        %Setting the value to Inf if functionevaluation returns NaN.
        if isnan(value)==1
            N = Inf(1);
        else
            %Trying to use fminsearch only for decent (working) initial
            %guesses.
            try
                min1 = fminsearch(fun1,[X(i),Y(j)]);
            catch
                warning('Problem using function.  Terminating function.');
                min1 = [X(i),Y(j)];
            end
            N = fun1(min1);
        end
```

```
        %Setting a new minimum when it is smaller than the currently known
        %one
        if N < min2(3)
            min2 = [min1(1),min1(2),N];
        end
    end
end
minimum=min2;
```

A.13 Minimum3

```
function minimal = minimum3(fun1,x0,p,int,m)
%This function makes use of minimum.m and minimum2.m to make the best
%approximation of a minimum within the grid size.
min1 = minimum(fun1,x0,p,int,m)
min2 = minimum2(fun1,x0,p,int)
if min2(3)<min1(3)
    minimal = min2;
else
    minimal = min1;
end
```

A.14 Powell

```
function [xo,Ot,nS]=powell(S,x0,ip,method,Lb,Ub,problem,tol,mxit)
%   Unconstrained optimization using Powell.
%
%   [xo,Ot,nS]=powell(S,x0,ip,method,Lb,Ub,problem,tol,mxit)
%
%   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 two 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]=aurea(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]=aurea(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.15 Bracket

```
function [x1,x2,nS]=bracket(S,x0,d,problem,stepsize)
%   Bracket the minimum (or maximum) of the objective function
%   in the search direction.
%
%   [x1,x2,nS]=bracket(S,x0,d,problem,stepsize)
%
%   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 three 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.16 Coggins

```
function [stepsize,xo,Ot,nS]=coggins(S,x0,d,ip,problem,tol,mxit,stp)
%   Performs line search procedure for unconstrained optimization
%   using quadratic interpolation.
%
%   [stepsize,xo,Ot,nS]=coggins(S,x0,d,ip,problem,tol,mxit)
%
%   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 three 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.17 Aurea

```
function [stepsize,xo,Ot,nS]=aurea(S,x0,d,ip,problem,tol,mxit,stp)
%   Performs line search procedure for unconstrained optimization
%   using golden section.
%
%   [stepsize,xo,Ot,nS]=aurea(S,x0,d,ip,problem,tol,mxit,stp)
%
%   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('aurea requires three 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 Aurea: reached maximum number of iterations!');
elseif ip,
  plot(stepsize,Ot,'r*');
end
```