Optimizing SOR using derivative-free optimization

Bachelor’s Project Mathematics
July 2019
Student: J.J.W. van der Meulen
First supervisor: R. Luppes
Second assessor: A. Sterk
Abstract

When a function is not differentiable, numerical methods can be used to compute the minimum and maximum of a function. These numerical methods are not limited to computing minima of functions, they can also be used to find optimal parameters for iterative methods. This study is mainly concerned with using numerical methods to find optimal relaxation parameters for the iterative method called "Successive over-relaxation". It will research the robustness and efficiency of 3 numerical methods called "The Downhill Simplex Method", "Powell’s Method" and "Simulated Annealing". These 3 methods have a fundamentally different approach to finding minima and hence will give different results when applied to SOR. In this study we will see that Powell’s Method is not suitable for finding optimal parameters for SOR. Simulated Annealing does well, but is so computationally expensive that it is useless in many situations. The Downhill Simplex Method turns out to be by far the best optimization method for our problem, as it is both the fastest and the most robust method.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Contents</td>
<td>3</td>
</tr>
<tr>
<td>1 Introduction</td>
<td>5</td>
</tr>
<tr>
<td>2 Preliminaries</td>
<td>6</td>
</tr>
<tr>
<td>2.1 Rastrigin function</td>
<td>6</td>
</tr>
<tr>
<td>2.2 Successive Over Relaxation</td>
<td>6</td>
</tr>
<tr>
<td>3 Powell’s Method applied to SOR and the Rastrigin Function</td>
<td>7</td>
</tr>
<tr>
<td>3.1 Introduction to Powell’s Method</td>
<td>7</td>
</tr>
<tr>
<td>3.2 Rastrigin Function</td>
<td>9</td>
</tr>
<tr>
<td>3.3 Powell’s Method applied to SOR with 3, 4 and 6 parameters (N=100, ODE)</td>
<td>9</td>
</tr>
<tr>
<td>3.4 Conclusion Powell’s Method applied to SOR (N=100, ODE)</td>
<td>11</td>
</tr>
<tr>
<td>4 Downhill Simplex Method (DSM) applied to SOR</td>
<td>12</td>
</tr>
<tr>
<td>4.1 Introduction to The Downhill Simplex Method</td>
<td>12</td>
</tr>
<tr>
<td>4.2 The Rastrigin function</td>
<td>13</td>
</tr>
<tr>
<td>4.3 SOR with 3, 6 and 10 parameters (N=100, ODE)</td>
<td>14</td>
</tr>
<tr>
<td>5 Simulated Annealing applied to SOR and Rastrigin</td>
<td>15</td>
</tr>
<tr>
<td>5.1 Introduction to Simulated Annealing</td>
<td>15</td>
</tr>
<tr>
<td>5.2 The Rastrigin function</td>
<td>16</td>
</tr>
<tr>
<td>5.3 Simulated Annealing applied to SOR (N=100, ODE)</td>
<td>16</td>
</tr>
<tr>
<td>5.4 Conclusion Simulated Annealing applied to SOR (N=100, ODE)</td>
<td>17</td>
</tr>
<tr>
<td>6 Comparison of the Methods applied to SOR (N=100, ODE)</td>
<td>17</td>
</tr>
<tr>
<td>7 Changing the ODE discretization</td>
<td>18</td>
</tr>
<tr>
<td>7.1 Increasing the subintervals</td>
<td>18</td>
</tr>
<tr>
<td>7.1.1 The Downhill Simplex Method</td>
<td>18</td>
</tr>
<tr>
<td>7.1.2 Powell’s Method</td>
<td>19</td>
</tr>
<tr>
<td>7.1.3 Simulated Annealing</td>
<td>20</td>
</tr>
<tr>
<td>7.1.4 Comparison of the methods</td>
<td>20</td>
</tr>
<tr>
<td>7.2 non-equidistant grids</td>
<td>20</td>
</tr>
<tr>
<td>7.2.1 Random grid</td>
<td>21</td>
</tr>
<tr>
<td>7.2.2 grid with dense edges</td>
<td>22</td>
</tr>
<tr>
<td>7.2.3 grid with dense centre</td>
<td>22</td>
</tr>
<tr>
<td>7.2.4 Comparison of the methods for non-equidistant grids</td>
<td>23</td>
</tr>
<tr>
<td>8 Changing the ODE</td>
<td>23</td>
</tr>
<tr>
<td>8.1 Equidistant grid</td>
<td>23</td>
</tr>
<tr>
<td>8.2 random grid</td>
<td>24</td>
</tr>
<tr>
<td>8.3 grid with dense edges</td>
<td>24</td>
</tr>
<tr>
<td>8.4 SOR with 6 parameters</td>
<td>25</td>
</tr>
<tr>
<td>8.5 Harder ODE</td>
<td>25</td>
</tr>
<tr>
<td>8.6 Conclusion</td>
<td>25</td>
</tr>
<tr>
<td>9 Solving a PDE</td>
<td>26</td>
</tr>
<tr>
<td>9.1 The Downhill Simplex Method</td>
<td>26</td>
</tr>
<tr>
<td>9.2 Lowering K</td>
<td>27</td>
</tr>
</tbody>
</table>
1 Introduction

It is often desired to know the minimum of a function, take for example a company that wants to minimize expenses and maximize profit. Or a bridge engineer that wants to know the minimum amount of cable that needs to be used in order to meet the safety requirements. When possible the minimization is done analytically by means of differentiation, but not every function is differentiable. When this is the case derivative-free optimization methods can be used to determine the minimum of a function.

These optimization methods are not limited to computing the minimum of regular continuous functions, take for example a situation where the function is an iterative method. The input of this function are some parameters on which the iterative method depend, and the output is the number of iterations needed for the iterative method to converge. In this case there are no derivatives available to compute directly, hence a derivative free optimization method has to be used to find optimal parameters that lead to fastest convergence of the iterative method.

The Bachelor theses ”Methods of optimization for numerical Algorithms ” by S.J Petersen [6] and ”Optimizing Parameters of Iterative Methods” by E.I. Maquelin [2] address this problem. The thesis by S.J Petersen primarily focuses on The Downhill Simplex Method and Powell’s method applied to functions and algorithms depending on one or two variables. The thesis by E.I. Maquelin continues this research by investigating the efficiency of these methods when applying it to iterative processes that depend on more than two variables. In this paper she thoroughly researches Newton’s Method and briefly discusses SOR.

This BSc. thesis will mainly focus on the optimization of SOR and will do so by means of 3 different optimization methods, and determine which of these 3 methods is most effective at doing so. The 3 optimization methods considered are The Downhill Simplex Method, Powell’s method using Golden Section Search or Parabolic interpolation and lastly the Simulated Annealing method. These numerical optimization methods will be tested on their efficiency and most importantly their robustness. Where robustness is defined as the ability to find the global minimum starting from various initial points. Each optimization method is briefly introduced, explaining the workings and adaptability of them. After this introduction they are applied to the Rastrigin Function. This is a test function designed to challenge an optimization method. Applying the optimization methods to it will be helpful to understand where the method might fail, and how it can be adapted to overcome any failure. After this, each method will be extensively applied to SOR. SOR will be tested with different numbers of parameters and will be applied to solve the discretization of a number of ODE’s and PDE’s.

After applying our optimization methods to many different problems, we will be able to conclude with the advantages and disadvantages of each method and give a general recommendation on how to use these methods to find optimal parameters for SOR. This general recommendation can be employed in fluid dynamics, where the flow of a fluid is simulated by means of solving the discretized system. Time will be separated in small time steps, where for each of these time steps a matrix problem with dimension exceeding a million will be solved using SOR or more advanced linear solvers. The scale of these matrices causes them to be computationally expensive to solve, hence finding optimal relaxation parameters is important. Currently it is not known what the optimal parameters are for these matrices, hence the optimization methods will be employed to find out.

Throughout this BSc. thesis MATLAB is used to implement the optimization methods and the relevant MATLAB code used can be found in Appendix A.
2 Preliminaries

The primary focus of this BSc. thesis is to investigate how derivative-free optimization methods react to finding optimal parameters for the iterative method SOR. In order to test our optimization methods they will be applied to a test function called the Rastrigin Function. In this section SOR and the Rastrigin function will be briefly introduced.

2.1 Rastrigin function

The Rastrigin function is designed to have many local minima as can be seen in figure 1. Due to steep gradients towards these minima it is difficult for optimization methods to find the global minimum. This makes it a perfect function to test the robustness of an optimization method. The n-dimensional Rastrigin function is defined as follows:

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

Figure 1: The 2-dimensional Rastrigin function

It has a global minimum at \( x = [0, 0, \ldots, 0] \), where \( f(x) = 0 \). In this BSc. thesis the 4-dimensional version of the Rastrigin function will be used. MATLAB-code is in Appendix A.1.

2.2 Successive Over Relaxation

SOR is an iterative method that solves matrix equations of the form \( Ax = b \); it is an adaptation of the Gauss-Seidel method. The Gauss-Seidel method works as follows: Firstly, an initial guess is needed, then rewrite each equation of the system, solving equation i for the i’th unknown. Now start by computing \( x_1 \), using the initial guess. From there on solve the other \( x_i \)'s using the most recent values of the unknowns. Repeat this process until convergence is achieved[1]. SOR is an adaption of this method by means of the use of a relaxation parameter which is often in \((0, 2)\). In SOR, during the Gauss-Seidel iteration the i’th element in the solution vector is composed of a combination of the i’th element of the new and the old solution vector. The ratio that determines how much of the old vector is kept and how much of the new vector is added is determined by the so called relaxation parameter \( \omega \). For certain matrices it is known how to compute the optimal value of this relaxation parameter that leads to fastest convergence[1]. We could decide to add more parameters that take turns in between each Gauss-Seidel iteration. An example is SOR with 2 \( \omega \)'s, where \( \omega_1 \) controls the odd-numbered vector elements and \( \omega_2 \) controls the even-numbered vector elements. In general, for multiple relaxation parameters it is not known which values of the parameters lead to fastest convergence, hence optimization methods can be used to find them.
The SOR algorithm

Here follows a description of the SOR algorithm in formula form.

Iteration rule by Gauss-Seidel:

$$\hat{x}_i^{(m+1)} = \frac{1}{a_{ii}} (b_i - \sum_{j=1}^{i-1} a_{ij} x_j^{(m+1)} - \sum_{j=i+1}^{n} a_{ij} x_j^{(m)})$$

$i = 1 \ldots n$

Relaxation:

$$x_i^{(m+1)} = \omega \hat{x}_i^{(m+1)} + (1 - \omega) x_i^{(m)}$$

Under-relaxation: $0 < \omega < 1$

Over-Relaxation: $1 < \omega < \infty$

Gauss-Seidel: $\omega = 1$

ODE

The problem that SOR will be used on in the first 6 sections concerns solving the ordinary differential equation:

$$y''(x) + \alpha y(x) = x \quad y(0) = y(1) = 1$$

Where $\alpha = 10^{-4}$. The ODE will be discretized by using the finite difference method [2]. The interval [0,1] will discretized into 100 subintervals where $y_i$ denotes the approximate solution at point $x_i$. The approximation of the second derivative by the finite difference method is

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

where $h$ denotes the step size. The discretization gives a system of the form:

$$\begin{bmatrix}
-2+\alpha & 1 & & & & \\
1 & -2+\alpha & 1 & & & \\
& \ddots & \ddots & \ddots & & \\
& & 1 & -2+\alpha & 1 & \\
& & & & 1 & -2+\alpha \\
\end{bmatrix}
\begin{bmatrix}
x_1 \\
x_2 \\
\vdots \\
x_{99} \\
x_{101} \\
\end{bmatrix}
= \frac{1}{h^2}
\begin{bmatrix}
1 \\
0.01 \\
\vdots \\
0.99 \\
1 \\
\end{bmatrix}$$

SOR will be used to solve this system. For different relaxation parameters $\omega$ the amount of iterations SOR needs to converge changes. Because of the nature of the problem no derivatives can be taken and hence we need to use derivative-free optimization methods to find optimal $\omega$ values to achieve fastest convergence. Relevant MATLAB code can be found in Appendix A.2 and A.3.

3 Powell’s Method applied to SOR and the Rastrigin Function

3.1 Introduction to Powell’s Method

Powell’s method is a derivative-free minimization technique that attempts to find the minimum of a function through 1-dimensional line searches[3]. In this Bsc thesis we will study Powell’s method using two different 1-dimensional line searches that are known as: Golden Section Search and Parabolic Interpolation. Powell’s method using parabolic interpolation is known as Coggin’s method. Powell’s method is constructed by adding a clever adaptation to an intuitive method known as the taxi cab method. The taxi cab method, like any other optimization method, starts at an initial guess. From there it finds the minimum along the first
basis direction by means of a 1-dimensional search. After that it proceeds by minimizing in the other basis
directions[4]. Now repeat this process until convergence is achieved. An example of the taxi cab method
finding a minimum is depicted on the left side of figure 2.

The problem with this method is that it can be extremely inefficient. Take for example a function whose
shape is a long narrow valley not parallel to any of the unit vectors[2]. When the taxi cab method descents
down this valley it will end up taking a series of very small steps. To improve the direction set, Powell’s
method adds a vector in the direction of the valley, while making sure that the vectors in the direction set
are ‘non interfering’. Where ‘non interfering’ means that the minimization in 1 direction is not being undone
by minimization in another direction. This would be very inefficient and it could even lead to a situation
where the algorithm gets stuck in a loop of endlessly cycling through the direction set.

There is however a problem with this version of Powell’s method. The process of replacing the direction
vectors could cause the set to contain linear dependent vectors. When this happens the algorithm would
look for a minimum in a subspace of the original space, and thus loose robustness. This problem is solved
by removing the direction which contributed to the largest decrease in function value. This may sound
contradictory, but doing this gives the highest chance of maintaining linear independence in the direction
set, which is most important. An example of Powell’s method finding a minimum is depicted on the right
side of figure 2.

Figure 2: A sequence of points generated by the taxicab method (left) and Powell’s method (right)
Source: [3]

Adaptation of Powell’s Method

The way the Golden Section search and Parabolic Interpolation find a minimum on a line is by narrowing
down an interval until the minimum is captured in an interval that meets the tolerance. Therefore, before
these methods can start their line search they first need an initial interval. The way they find this interval
is by looking to the right or the left of the point \(x_0\) in increasing steps until the function value between 2
successive points, \(x_1\) and \(x_2\) increases. If it looked to the right and this happens the bracketed interval is
\([x_0, x_2]\), and if it looked to the left it is \([x_2, x_0]\). By default the algorithm starts by looking to the right, but
it looks to the left when the first point generated to the right of \(x_0\) is going uphill. The way this bracketing
process can by manipulated is by adjusting the step size. Choosing a small step size provides more precision
around \(x_0\), but might only contain a local minimum. Choosing a large step size will often give a larger search
interval which has both advantages and disadvantages. The advantage is that a larger search interval has
a higher chance that the global minimum is contained within it. The disadvantage is that a larger interval
might contain points at which the iterative methods do not converge, which can cause much instability,
causing Powell’s method to diverge. The relevant MATLAB codes can be found in Appendices A.4 to A.7.
3.2 Rastrigin Function

Since the Rastrigin function has many local minima, it is a great example to show why the bracketing is important. In the table below can be seen that when the step size is too low, the bracketing will not create an interval big enough to contain the global minimum. This results in an immediate jump to a local minimum. However, when taking a larger step size i.e. causing a larger search interval, the global minimum is found to the precision that the tolerance requires. The table only shows the results for Powell’s Method using the the Golden Section Search, this is because results for Coggin’s Method were identical.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Golden section search)</th>
<th>Minimum Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1]</td>
<td>[0.995, 0.995, 0.995, 0.995]</td>
<td>3.9798</td>
<td>0.1</td>
</tr>
<tr>
<td>[1,1,1,1]</td>
<td>-10^-4, [0.0737, 0.1701, 0.1701, 0.1701]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[10,10,10,10]</td>
<td>[9.9487, 9.9487, 9.9487, 9.9487]</td>
<td>398</td>
<td>0.1</td>
</tr>
<tr>
<td>[10,10,10,10]</td>
<td>10^-4, [0.1507, 0.1507, 0.1507, 0.1507]</td>
<td>1.8 x 10^-7</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 1: Powell’s Method applied to the Rastrigin function. The global minimum is at [0,0,0,0].

3.3 Powell’s Method applied to SOR with 3, 4 and 6 parameters (N=100, ODE)

The iterative method ‘Successive Over-Relaxation’ with 3 or more parameters is a much different function to optimize than for example the Rastrigin function. This is because SOR is very sensitive to its input. A small change in the input could mean the difference between convergence of divergence of SOR. It is because of this sensitivity that optimizing SOR is much harder than a general smooth function. The bracketing will proof to be very important in order to achieve convergence. In this section Powell’s Method will be applied to SOR with 3, 4 and 6 parameters. Doing this will show how both versions of Powell’s method when we increase the amount of relaxation parameters. The 3 parameter version of SOR is constructed such that \( \omega_1 \) controls the boundaries of the solution vector. Where \( \omega_2 \) and \( \omega_3 \) control the values in between the boundaries such that \( \omega_2 \) controls the even-numbered vector elements and \( \omega_3 \) controls the odd-numbered vector elements. Versions of SOR with more than 3 parameters are constructed similarly where \( \omega_1 \) controls the boundaries, and the other parameters take turns in between the boundaries.

SOR with 3 parameters

Applying Powell’s Method to SOR with 3 parameter is relatively successful as both versions of Powell’s method are able to converge to the suspected global minimum value of 481 iterations. Coggin’s Method shows to be more sensitive to the step size as it only converged if it was 0.1. The Golden Section version of Powell’s method seems to be better for the SOR with 3 omega’s, this is the same conclusion that was made in E. Maquelin’s thesis. However in that BSc. thesis it was concluded that Coggin’s Method diverges for almost any initial point not close to the minimum, but that can be fixed by choosing the right step size, which had not yet been considered.
<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Golden Section)</th>
<th>Minimum Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1]</td>
<td>[0.0708, 1.9034, 1.9773]</td>
<td>481</td>
<td>0.1</td>
</tr>
<tr>
<td>[1,1,1]</td>
<td>[0.0752, 1.9032, 1.9774]</td>
<td>481</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[0.0604, 1.9389, 1.9415]</td>
<td>483</td>
<td>1</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[-0.0005, 1.6037, 9.7899]</td>
<td>divergence</td>
<td>0.1</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[0.1562, 1.8984, 1.9821]</td>
<td>496</td>
<td>1</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Coggins’s method)</th>
<th>Minimum Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1]</td>
<td>[0.1045, 1.8978, 1.9831]</td>
<td>489</td>
<td>0.1</td>
</tr>
<tr>
<td>[1,1,1]</td>
<td>[2.2549, 1.9550, 2.5563]</td>
<td>divergence</td>
<td>1</td>
</tr>
<tr>
<td>[0,0.5,1.5,2]</td>
<td>[0.0756, 1.8858, 1.9966]</td>
<td>481</td>
<td>0.1</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[-0.0936, 0.1754, -7.2095]</td>
<td>divergence</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 2: Powell’s method used with both Golden Section search and Interpolation on the iterative SOR method with 3 parameters. N = 100, ODE.

SOR with 4 parameters

Adding one extra parameter should cause more difficulty for the methods. This is because SOR needs perfectly balanced parameters to even converge at all. Adding one extra parameter makes this balancing more difficult. From the tables we can see that the Golden Section version performs worse. Coggin’s Method appears to converge from all of the initial points provided the correct step size.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Golden Section)</th>
<th>Minimum Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1]</td>
<td>[1.9468, 2.1843, 1.5588, 1.7817]</td>
<td>divergence</td>
<td>0.1</td>
</tr>
<tr>
<td>[1,1,1,1]</td>
<td>[0.0640, 1.8957, 1.9386, 1.9914]</td>
<td>497</td>
<td>0.01</td>
</tr>
<tr>
<td>[0.0,5,1.5,2]</td>
<td>[0.0290, 2.0772, 1.8275, 1.8741]</td>
<td>1023</td>
<td>1</td>
</tr>
<tr>
<td>[0.0,5,1.5,2]</td>
<td>[0.5320, 1.9861, 1.8149, 1.9945]</td>
<td>772</td>
<td>0.1</td>
</tr>
<tr>
<td>[0.0,0,0]</td>
<td>[0.5791, 1.9810, 1.9422, 1.8968]</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>[0.0,0,0]</td>
<td>[0.5499, 0.9981, 2.9068, 1.2550]</td>
<td>Divergence</td>
<td>0.1</td>
</tr>
<tr>
<td>[0.0,0,0]</td>
<td>[0.0421, 1.8721, 1.9387, 2.0078]</td>
<td>700</td>
<td>0.01</td>
</tr>
<tr>
<td>[0.2,2,2]</td>
<td>[0.0600, 1.9367, 1.9317, 1.9527]</td>
<td>481</td>
<td>0.01</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Coggins’s method)</th>
<th>Minimum Value</th>
<th>Step Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1]</td>
<td>[1.7046, 2.3700, 1.3651, 1.6078]</td>
<td>divergence</td>
<td>0.1</td>
</tr>
<tr>
<td>[1,1,1,1]</td>
<td>[0.0763, 1.9764, 1.9428, 1.9010]</td>
<td>483</td>
<td>0.01</td>
</tr>
<tr>
<td>[0.0,5,1.5,2]</td>
<td>[0.0942, 1.9648, 1.9310, 1.9233]</td>
<td>487</td>
<td>0.1</td>
</tr>
<tr>
<td>[0.0,5,1.5,2]</td>
<td>[1.7796, 0.3202, 2.1382, 1.8710]</td>
<td>481</td>
<td>0.01</td>
</tr>
<tr>
<td>[0.0,0,0]</td>
<td>[0.0421, 1.8721, 1.9387, 2.0078]</td>
<td>divergence</td>
<td>0.1</td>
</tr>
<tr>
<td>[0.0,0,0]</td>
<td>[0.5791, 1.9810, 1.9422, 1.8968]</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>[0,2,2,2]</td>
<td>[0.0925, 1.9177, 1.9285, 1.9733]</td>
<td>489</td>
<td>0.1</td>
</tr>
</tbody>
</table>

Table 3: Powell’s method used with both Golden section search and Interpolation on the iterative SOR method with 4 parameters. N=100, ODE.
SOR with 6 parameters

When adding 2 more parameters the golden section variant seems to lose most of its robustness, while Coggin’s retains it. Convergence to the minimum from outside the appropriate interval seems impossible. In the construction of the table many step sizes have been tried, and the best results were given by stepsize = 0.01.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Golden Section)</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1,1,1]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>[1.5,1.5,1.5,1.5,1.5,1.5]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>[0.0,5.1,1.5,2,2]</td>
<td>[0.0717, 1.9089, 2.0012, 2.0232, 1.8969, 1.8773]</td>
<td>482</td>
</tr>
<tr>
<td>[0.2,2.2,2.2]</td>
<td>[0.0753, 1.7910, 1.9436, 2.0409, 1.9770, 1.9643]</td>
<td>481</td>
</tr>
<tr>
<td>[0.0,0,0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>[3,3,3,3,3]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Powell’s Method (Coggin’s method)</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1,1,1]</td>
<td>[1.0273, 1.9712, 1.8971, 1.9165, 1.8702, 1.7878]</td>
<td>533</td>
</tr>
<tr>
<td>[1.5,1.5,1.5,1.5,1.5,1.5]</td>
<td>[0.1028, 2.0165, 1.8194, 1.9710, 1.9254, 1.9778]</td>
<td>486</td>
</tr>
<tr>
<td>[0.0,5.1,1.5,2,2]</td>
<td>[0.1265, 1.7798, 1.9340, 2.0470, 1.9525, 2.0066]</td>
<td>500</td>
</tr>
<tr>
<td>[0.2,2.2,2.2]</td>
<td>[0.0953, 1.8130, 1.9274, 2.0354, 1.9345, 2.0030]</td>
<td>481</td>
</tr>
<tr>
<td>[0.0,0,0,0,0]</td>
<td>[0.1059, 2.0167, 1.8194, 1.9707, 1.9253, 1.9780]</td>
<td>483</td>
</tr>
<tr>
<td>[3,3,3,3,3]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
</tbody>
</table>

Table 4: Powell’s method used with both Golden section search and Interpolation on the iterative SOR method with 6 parameters. N=100, ODE. Step size has been set to 0.01.

3.4 Conclusion Powell’s Method applied to SOR (N=100, ODE)

By applying the two variants of Powell’s Method to SOR we can conclude that the sensitive nature of SOR causes the step size to be of vital importance to the robustness of Powell’s Method. Providing the correct step size and an initial point, Coggin’s Method proved to be robust for SOR with 3, 4 and 6 parameters. This was not the case for the Golden Section variant as it lost much of its robustness as a result of adding the additional parameters.

As we have already mentioned before, the reliance on a proper bracketing step size can be explained by the fact that SOR needs very specific parameter values to converge. Suppose we are in the process of a 1-dimensional line search around the point \( x_0 \) and we take a step size of 1. To make a bracket around \( x_0 \) the algorithm looks at the function value at the point \( (x_0 + 1) \). Since SOR is very sensitive to the parameter values and needs very specific values to converge it is most likely that for this point SOR will not converge, and as a result \( (x_0 + 1) \) is the upper bound. By the same reasoning \( (x_0 - 1) \) will be the lower bound resulting in the search interval \([x_0 - 1, (x_0 + 1)]\). For our 1-dimensional optimization methods this is a terrible initial interval. If there even exists a region of parameter values in the search interval for which SOR achieves convergence it might be so small that the line search is not able to find it, it could skip over it. It is precisely for this reason that in most cases a small step size is better suited for our optimization problem, since then the line searches have a higher chance to be given a reasonable search interval. Taking a step size that is too small will also cause problems, as that could result in such a small search interval that there is actually no optimization being done anymore. It is because of these reasons that the step size is of vital importance for the convergence of Powell’s Method.
4 Downhill Simplex Method (DSM) applied to SOR

4.1 Introduction to The Downhill Simplex Method

The Downhill Simplex Method, also known as the Nelder-Mead Method, was created by John Nelder and Roger Mead in 1965 [5]. The main idea of the method is rolling an initial simplex downhill until the stopping criteria are met. The method is derivative-free as it only uses function evaluations.

The algorithm works for an N-dimensional optimization problem as follows. First the algorithm requires an initial guess, then using that initial guess the algorithm creates \( N \) more points which together with the initial guess form the initial simplex consisting of \( N + 1 \) points. These \( N \) points are generated by simply adding each of the \( N \) basis vectors to the initial guess. One can scale these basis vectors to obtain a larger or smaller initial simplex.

Given the initial simplex, the algorithm will perform a series of steps in attempt to roll the simplex downhill. These steps involve reflections, contractions and expansions as shown in figure 3. The figure caption contains a brief description of how the algorithm decides to contract or expand. Performing these steps will guarantee that the method converges to a local minimum, unfortunately this is not always the global minimum.

There are 2 stopping criteria to the Downhill Simplex Method. The first is a tolerance on the drop in function value. When after a step the drop in function value is low enough the method will be terminated. The second is a tolerance in the change of minimum vector. When 2 successive minima are close enough to each other, then the method will be terminated.

Adaptability of Downhill Simplex Method

Three ways of adapting DSM are done by the scaling of the contraction, the reflection and the expansion. During the algorithm, when the algorithm suspects the minimum lies within the simplex, it performs a contraction of the simplex. This contraction can be scaled by a constant. In the same way the reflection and expansion constant can be adjusted. Changing the parameters can influence the speed and robustness of the method. Consider for example a case where the initial guess is far away from the minimum. In that case DSM might need many steps to go towards it. The number of steps could be reduced by taking a higher expansion parameter, but this might cause a loss of accuracy. In general scaling these 3 constants (reasonably) will only cause small changes in the robustness of the method. If the minimum is not found when applying DSM to a certain function, it is unlikely that adjusting these 3 parameters will fix it. There is however one more way to adjust DSM that can make or break the robustness of it. That is the scaling of the initial Simplex. Similar to the step size for Powell’s Method, it is crucial for DSM that the initial simplex is not too large and not too small. Too large will cause each of the vertices to be on points where SOR diverges, causing the method to diverge. Too small will cause the search area to be too small, thus missing the global minimum. Because this scalar is the most important one, it will be shown in the tables where we apply DSM to SOR. In the tables it is found under the column: ‘scalefactor’.

MATLAB has a build in function called ‘fminsearch’ that takes a function to be minimized and an initial guess. This function uses DSM, but does not have the required adaptability that we need to thoroughly research the optimization method. Hence an adaptable version of DSM is used made by Fuchang Gao and Lixing Han. The MATLAB code can be found in Appendix A.8.
4.2 The Rastrigin function

As we can see in the tables The Downhill Simplex has trouble converging to the global minimum. With a small scale factor it immediately converges into the closest local minimum it can find. Lower minima can be found by taking a larger scale factor, causing some local minima to be omitted. However, increasing this scale factor does not cause the method to converge to the global minimum.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>Scalefactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,0,0,0]</td>
<td>[0,0,0,0]</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>[1,2,3,4]</td>
<td>[1.9899, 1.9899, 2.9848, 3.9797]</td>
<td>32</td>
<td>1</td>
</tr>
<tr>
<td>[1,2,3,4]</td>
<td>[0.0001, 0.0002, 1.9898, 0.9949]</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[9.9487, 9.9487, 9.9487, 9.9487]</td>
<td>397</td>
<td>1</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[-0.0004, -2.9849, 0.0003, -2.9851]</td>
<td>17</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 5: The Downhill Simplex Method applied to the Rastrigin function
4.3 SOR with 3, 6 and 10 parameters (N=100, ODE)

After having applied DSM to SOR with 3, 6 and 10 parameters we can see that it is very consistent and reliable. Provided the right size of the the initial simplex, DSM was robust for all 3 variants of SOR tested. The computing time when DSM was applied to SOR with 10 parameters was remarkably low. With initial guess \([0, 0, ..., 0, 0]\) it found the minimum in less than 5 seconds. However from applying DSM to the Rastrigin function we have seen that the method is susceptible to converging to local minima, which we can see has happened also when applying it to SOR. DSM consistently converges to minimum values of around 500 iterations while we know from Powell’s method that the minimum for 3 parameters is at most 481 iterations.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>Scalefactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1]</td>
<td>1.4014, 1.9284, 1.9503</td>
<td>501</td>
<td>1</td>
</tr>
<tr>
<td>[0,1,2]</td>
<td>0.6817, 1.9117, 1.9674</td>
<td>500</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>1.5043, 1.9276, 1.9511</td>
<td>505</td>
<td>1</td>
</tr>
<tr>
<td>[5,5,5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>1</td>
</tr>
<tr>
<td>[5,5,5]</td>
<td>1.2227, 1.9145, 1.9646</td>
<td>502</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 6: Downhill Simplex applied to SOR with 3 parameters. N=100, ODE.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>Scalefactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1,1]</td>
<td>1.5650, 1.9413, 1.9310, 2.0179, 1.9249, 1.8857</td>
<td>506</td>
<td>1</td>
</tr>
<tr>
<td>[0,0.5,1.5,2.2,5]</td>
<td>0.9061, 1.8540, 1.9474, 2.0432, 1.9394, 1.9218</td>
<td>501</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0,0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0,0,0,0]</td>
<td>1.3816, 1.9527, 1.9049, 1.9730, 1.9242, 1.9434</td>
<td>503</td>
<td>2</td>
</tr>
<tr>
<td>[5,5,5,5,5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>10</td>
</tr>
<tr>
<td>[5,5,5,5,5]</td>
<td>0.9593, 1.9375, 1.8915, 2.0034, 1.9095, 1.9589</td>
<td>506</td>
<td>20</td>
</tr>
</tbody>
</table>

Table 7: Downhill Simplex applied to SOR with 6 parameters. N=100, ODE.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>Scalefactor</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,...,1]</td>
<td>-</td>
<td>507</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,...,0]</td>
<td>-</td>
<td>divergence</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,...,0]</td>
<td>-</td>
<td>501</td>
<td>2</td>
</tr>
<tr>
<td>[2,2,...,2]</td>
<td>-</td>
<td>500</td>
<td>2</td>
</tr>
<tr>
<td>[0.2,0.4,...,1,8,2]</td>
<td>-</td>
<td>500</td>
<td>2</td>
</tr>
<tr>
<td>[3,3,...,3]</td>
<td>-</td>
<td>512</td>
<td>5</td>
</tr>
<tr>
<td>[1,0,1,...,0,1,0]</td>
<td>-</td>
<td>506</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 8: Downhill Simplex applied to SOR with 10 parameters. N=100, ODE.

The results have shown that DSM gives entirely different results bases on which initial point is chosen. This suggests that in order to increase the chance to find the global minimum, DSM should be applied to many different initial points. An obvious suggestion is doing this by applying DSM to a rough grid around the suspected global minimum. Such a DSM grid search will be performed later in this study.
5 Simulated Annealing applied to SOR and Rastrigin

5.1 Introduction to Simulated Annealing

Simulated annealing is a probabilistic derivative-free method for finding the extreme values of a function. The name and inspiration come from a technique in material science called annealing. In annealing a metal is heated up, after which it is cooled in a controlled fashion. This process increases the ductility and reduces the hardness of the metal. In simulated annealing this process is imitated by having a temperature component that starts off high and cools down slowly. The algorithm is initiated by an initial guess, after which it randomly generates a neighbouring point. Now the algorithm chooses by chance whether or not to accept this point. When this point is better, i.e. it has a lower function value, it accepts it always. But when it is worse, i.e. it has a higher function value, there is still a chance it will accept this point based on the temperature. If the temperature is high, the chance to accept a worse point is relatively high, namely 50 percent. But as the temperature goes down, so does the chance to accept. The gradually cooling aspect of simulated annealing causes the algorithm to search actively at the start at many places, but as the temperature cools down it is forced to look more narrow until stopped by the algorithm.

This is where the advantage lies for simulated annealing compared to, for example, the Downhill Simplex Method. DSM is very susceptible to converging to local minima. This is because in every step of the algorithm it wants to improve on the previous step. Consequently, when it descents down a local minimum, it is likely that it will continue down it. Doing so it might never look around for other, possibly better, minima. Simulated annealing is designed to solve this problem by occasionally accepting worse answers, in the hope that it might lead to better ones later. But since the method needs to converge at some point, this willingness to accept worse answers is reduced over time by means of the temperature variable.

Adaptation of Simulated Annealing

Simulated annealing can be adapted in many ways to suit the function it is trying to minimize, this can be done by adjusting parameters accordingly. Examples of parameters are: the initial temperature, the rate of cooling, the neighbouring point generating function, stopping criteria and more. The most important parameter is the generator function, this generator can be adapted in a simple way. The way it works is that the generator draws a random number from the normal distribution, and adds it to 1 row in the vector. Before it adds the drawn number it multiplies it by a generator scalar according to the problem, it is in this scaling where the adaptability lies. Take for example the Rastrigin function, if the scalar is too low, then the generated points are always very close by, and hence when it lands in a local minimum the chance to get out is very low. Because this generator scalar is the most important parameter in Simulated Annealing, it will be shown in the tables later in this section.

There are 2 stopping criteria for simulated annealing: the stopping temperature and a maximum amount of consecutive rejections. The stopping temperature is the temperature at which the algorithm stops. The maximum of consecutive rejections stops the algorithm if a neighbouring point has been generated too many times but has never been accepted.

The randomness of the problem comes from 2 aspects: the random generation of a neighbouring point, and the random chance to accept that point. Because of this the method does not give the same answer for the same initial guess every time. This means that in general it is wise to run the algorithm from an initial point a number of times, and then select the best minimum found.

The MATLAB implementation of Simulated Annealing can be found in Appendix A.9
5.2 The Rastrigin function

Applying simulated annealing to the Rastrigin Function shows that the generator scalar is crucial in finding the global minimum which is $[0, 0, 0, 0]$. Similar to Powell’s Method it converges to local minima when the relative step size is too small, but works perfectly when the step size is large enough.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Simulated Annealing</th>
<th>Minimum Value</th>
<th>Scalar</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1]</td>
<td>[-1.9899, 4.9747, -1.9899, -0.9950]</td>
<td>33.8328</td>
<td>$\frac{1}{10}$</td>
</tr>
<tr>
<td>[1,1,1,1]</td>
<td>$10^{-3} \cdot [0.1468, -0.0034, -0.0864, -0.1684]$</td>
<td>2 $\cdot 10^{-5}$</td>
<td>1</td>
</tr>
<tr>
<td>[1,2,3,4]</td>
<td>[3.9797, 3.9798, 2.9848, 4.9747]</td>
<td>65.6668</td>
<td>$\frac{1}{10}$</td>
</tr>
<tr>
<td>[1,2,3,4]</td>
<td>[-0.0010, 0.0006, -0.002, 0.0003]</td>
<td>3 $\cdot 10^{-5}$</td>
<td>1</td>
</tr>
<tr>
<td>[1,10,100,1000]</td>
<td>[-0.0036, -0.0038, -0.0028, 0.0011]</td>
<td>0.0072</td>
<td>10</td>
</tr>
</tbody>
</table>

Table 9: Simulated Annealing applied to the Rastrigin function. The global minimum is at $[0,0,0,0]$.

5.3 Simulated Annealing applied to SOR (N=100, ODE)

When applying Simulated Annealing to SOR the algorithm is ran a maximum of 20 times, and the best minimum was shown in the table. When the algorithm found a minimum below 500 it was stopped and in the table is shown under ‘tries’ how many tries it took to reach a minimum below 500.

From the tables we can see that Simulated Annealing loses much of its efficiency when it is applied to SOR with more parameters. It works perfectly for SOR with 3 parameters as it finds the minimum on the first try. For the SOR variant with 4 parameters it is having more trouble. For the 6-dimensional SOR it finds the minimum when it starts close and waits long enough. It took 30 minutes to find the minimum starting from the point $[1, 1, 1, 1, 1, 1]$. Although it is successful at finding the minimum, 30 minutes is simply too inefficient to call this a good result for Simulated Annealing.

The reason why Simulated Annealing performs so poorly for 6 parameters can be explained by the fact that the algorithm is simply adding random numbers to the vector and hopes it works. This method can work fine when the dimension is low, but when the dimension becomes higher the chances become slim that the right number is added to just the right row. Simulated annealing was also tested on the 10-dimensional SOR, but the table is not shown as it could not find a single minimum in a reasonable amount of time. For the tables for 11 and 12 the scalar column has been left out, since it has been set to 1 for all initial points.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Simulated Annealing</th>
<th>Minimum Value</th>
<th>Scalar</th>
<th>Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1]</td>
<td>[0.0632, 1.9077, 1.9734]</td>
<td>487</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[0.1060, 1.9254, 1.9535]</td>
<td>492</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>[2,-2,2]</td>
<td>[0.0704, 1.8983, 1.9829]</td>
<td>485</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>[10,10,10]</td>
<td>[0.0917, 1.9055, 1.9745]</td>
<td>491</td>
<td>2</td>
<td>1</td>
</tr>
</tbody>
</table>

Table 10: Simulated Annealing applied to SOR with 3 parameters
Table 11: Simulated Annealing applied to SOR with 4 parameters

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Simulated Annealing</th>
<th>Minimum Value</th>
<th>Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1]</td>
<td>0.0649, 1.9073, 1.9387, 1.9740</td>
<td>496</td>
<td>4</td>
</tr>
<tr>
<td>[0,0,0,0]</td>
<td>0.0626, 1.9283, 1.9347, 1.9595</td>
<td>498</td>
<td>7</td>
</tr>
<tr>
<td>[2,-2,-2,-2]</td>
<td>0.0692, 1.8706, 2.0362, 1.8780</td>
<td>733</td>
<td>20</td>
</tr>
<tr>
<td>[10,10,10,10]</td>
<td>0.1147, 1.9706, 1.9417, 1.9066</td>
<td>496</td>
<td>4</td>
</tr>
</tbody>
</table>

Table 12: Simulated Annealing applied to SOR with 6 parameters

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Simulated Annealing</th>
<th>Minimum Value</th>
<th>Tries</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1,1,1,1,1,1]</td>
<td>0.0915, 1.9076, 1.7541, 2.0147, 2.0282, 1.9986</td>
<td>498</td>
<td>20</td>
</tr>
<tr>
<td>[0,0,0,0,0,0]</td>
<td>0.0872, 1.8919, 1.9282, 1.9997, 1.8684, 2.0176</td>
<td>495</td>
<td>15</td>
</tr>
<tr>
<td>[2,-2,-2,-2,-2]</td>
<td>[not relevant]</td>
<td>divergence</td>
<td>20</td>
</tr>
<tr>
<td>[10,10,10,10,10]</td>
<td>[not relevant]</td>
<td>divergence</td>
<td>20</td>
</tr>
</tbody>
</table>

5.4 Conclusion Simulated Annealing applied to SOR (N=100, ODE)

We can conclude that Simulated Annealing works well for low-dimensional variants of SOR. This is simply because the computing time required is low and it can try enough times until it hits a minimum. However, for higher-dimensional variants of SOR this is much harder since just the right numbers need to be added to just the right rows. Another problem is that for the 6-dimensional case the algorithm was run 20 times from the initial point [1, 1, 1, 1, 1, 1]. For 19 out of 20 times the algorithm found nothing, but 1 time it hit just the right spot. This means that while the program is running it could not be known if it was going to result in anything. As a consequence the method is unreliable and unpredictable, since one just has to wait with no idea if it will stumble upon a minimum.

6 Comparison of the Methods applied to SOR (N=100, ODE)

Now that all 3 optimization methods have been applied to SOR it is time to compare them and see how they did. First a quick summary is given of each method after which they will be compared.

First Powell’s Method was applied to SOR. We have seen that both variants of Powell’s Method worked well for the 3 parameter version of SOR, but when more parameters were added to SOR only Coggin’s method could retain its robustness. When SOR was scaled up to 10 parameters Powell’s Method would only converge when starting very close to the minimum. After Powell’s Method we applied DSM to SOR with up to 10 parameters. The same results were shown for each of the SOR variants, namely the method achieved complete robustness with good speed for SOR with up to 10 parameters. The only downside of DSM was that it always converge to the local minimum values around 500, and not the desired (suspected) global minimum value. For SOR with 3 parameters we know from Powell’s Method that the global minimum is at most 481, but DSM could not find it. Finally we applied Simulated Annealing to SOR, this method is effective for the 3 dimensional versions of SOR but showed much difficulty with the 6 dimensional version. It proved to be unreliable and the computing time simply took too long compared to the other methods. This is why Simulated Annealing was not even attempted on SOR with 10 parameters. Now if we compare the 3 methods we quickly see that all 3 methods worked well on the 3-dimensional version of SOR, but as SOR was scaled up only DSM and Coggin’s Method could retain their robustness. Powell’s Method was able to find the global minimum from some initial points close to the minimum, while DSM would always get stuck in a local minimum evaluated at around 500. In general, when no information is known about the position of the minimum, DSM is best as it will find a local minimum for most reasonable initial points. When it is known within close proximity where the global minimum lies, Powell’s Method should definitely be tried as it could find the global minimum. The dependence of Powell’s Method on its initial point suggests that applying it to a grid of initial points could be effective. This will be done later in this study.
7 Changing the ODE discretization

In this section we will study how the optimization methods react when we change the way we discretize our ODE. We will change the discretization in 2 ways. First we will look how the optimization methods and SOR react when we increase the amount of equidistant subintervals to higher amounts. In this part we will primarily focus on the speed and efficiency of the methods, not necessarily on the robustness. This is because increasing the amount of subintervals does not really change the SOR problem. It only makes it more expensive and hence it will not change the robustness. After studying the influence of the increase of subintervals we will study how the methods react when non-equidistant discretization grids are taken, for example a grid that is more concentrated around the edges of the interval [0,1]. In this part the robustness of the methods will again be investigated, as taking an non-equidistant grid creates a completely different SOR problem, and hence the robustness of the methods may be affected. From here onward the tables will not contain a column for the bracketing step size, generator function scalar and the initial simplex scalar anymore. In the construction of the tables many constants have been tried and the best results are shown.

7.1 Increasing the subintervals

If we increase the number of subintervals of the discretization of the ODE gradually from 100 to 2000, we get better and better approximations of the solution of the ODE. However, this comes at a cost, namely SOR function evaluations become more expensive since the dimension of the linear equation it is solving increases. Because of this we want to know which method is most efficient, i.e. uses the lowest amount of function evaluations. In the previous section our focus was primarily in researching the robustness of the optimization methods, but in this section our primary focus will be on the efficiency of the methods. Because we want to focus on the efficiency we will apply our optimization methods to SOR with 3 parameters, starting with initial guess [1.5,1.5,1.5]. We do this because the methods are robust under these conditions and hence we can fully focus on the efficiency of the methods. The reason we know the methods are robust under these conditions is because for the construction of each table a grid search was done. In each grid search DSM was started from 11³ different initial points, those points were constructed on a [0:0.2:2] spaced grid in 3 dimensions. After applying DSM on these points the lowest minimum found is the same as the minimum found after simply starting from the initial point [1.5,1.5,1.5]. From this grid search we can already conclude that the robustness of the methods applied to SOR does not deteriorate as the amount of subintervals increases. The reason that DSM is used for this grid search is because in the previous section of this thesis it appeared to be faster and more robust.

In the construction of the tables the residual allowance term of SOR is lowered from $10^{-9}$ to $10^{-3}$ to speed up the comparison, this is why for the regular 100 subintervals we found a lower minimum value of 253 for SOR with 3 parameters, instead of the 481 found in the previous section.

7.1.1 The Downhill Simplex Method

From the table we can see that as the amount of subintervals increase, each function evaluation of SOR becomes more expensive. For 100 subintervals each function evaluation takes about 0.0016 seconds, and for 1000 subintervals it takes around 0.11 seconds, that is almost 70 times longer. This is because the matrix is 10 times bigger. That means that each Gauss-Seidel iteration is 10 times more work. Also the required number of iterations for higher a number of intervals is simply higher.
### Table 13: The Downhill Simplex Method applied to SOR with 3 parameters starting from initial guess [1.5,1.5,1.5]. #Subintervals is the number of subintervals used in the discretization. Minimum Iterations is the minimum found by the method. Func. Eval. is the number of function evaluation used by the method. Time is the time the method took in seconds. Time/Func. Eval. is the average amount of seconds each function evaluation of SOR took. The DSM scalefactor is set to 1.

<table>
<thead>
<tr>
<th>#Subintervals</th>
<th>Minimum Iterations</th>
<th>#Func. Eval</th>
<th>Time</th>
<th>Time/#Func. Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>253</td>
<td>363</td>
<td>0.06</td>
<td>0.0016</td>
</tr>
<tr>
<td>[200]</td>
<td>601</td>
<td>198</td>
<td>2</td>
<td>0.01</td>
</tr>
<tr>
<td>[500]</td>
<td>1504</td>
<td>172</td>
<td>11</td>
<td>0.064</td>
</tr>
<tr>
<td>[800]</td>
<td>2400</td>
<td>432</td>
<td>36</td>
<td>0.083</td>
</tr>
<tr>
<td>[1000]</td>
<td>3000</td>
<td>431</td>
<td>49</td>
<td>0.11</td>
</tr>
<tr>
<td>[2000]</td>
<td>6054</td>
<td>185</td>
<td>79</td>
<td>0.4</td>
</tr>
</tbody>
</table>

Table 13: The Downhill Simplex Method applied to SOR with 3 parameters starting from initial guess [1.5,1.5,1.5]. #Subintervals is the number of subintervals used in the discretization. Minimum Iterations is the minimum found by the method. Func. Eval. is the number of function evaluation used by the method. Time is the time the method took in seconds. Time/Func. Eval. is the average amount of seconds each function evaluation of SOR took. The DSM scalefactor is set to 1.

### 7.1.2 Powell’s Method

For Powell’s Method we can see in table 14 that they found roughly the same minimum values as DSM did, which confirms the assumed robustness of both methods. We see that both versions of Powell’s method behave relatively similar in terms of efficiency, they both take about the same amount of time for all the tested amount of subintervals. The time needed per SOR evaluation is roughly the same as in table 13, which is expected since it does not depend much on the method used. For all subintervals the Powell’s Methods are slower than DSM as they require more function evaluations. We can also see that the global minimum values are suspicious in tables 14/15. There seems to be a perfect relation where the global minimum value is 3 times the amount of subintervals. This could be caused by the SOR code, but after careful review it does not appear to be.

### Table 14: The Golden Section version of Powell’s Method applied to SOR with 3 parameters with an increasing amount of subintervals starting from initial guess [1.5,1.5,1.5]. The step size has been set to 0.01.

<table>
<thead>
<tr>
<th>#Subintervals</th>
<th>Minimum Iterations</th>
<th>#Func. Eval</th>
<th>Time</th>
<th>Time/#Func. Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>255</td>
<td>916</td>
<td>2</td>
<td>0.002</td>
</tr>
<tr>
<td>[200]</td>
<td>544</td>
<td>587</td>
<td>6</td>
<td>0.01</td>
</tr>
<tr>
<td>[500]</td>
<td>1466</td>
<td>605</td>
<td>32</td>
<td>0.05</td>
</tr>
<tr>
<td>[800]</td>
<td>2404</td>
<td>893</td>
<td>85</td>
<td>0.1</td>
</tr>
<tr>
<td>[1000]</td>
<td>3000</td>
<td>803</td>
<td>120</td>
<td>0.15</td>
</tr>
<tr>
<td>[2000]</td>
<td>6000</td>
<td>1191</td>
<td>499</td>
<td>0.42</td>
</tr>
</tbody>
</table>

Table 14: The Golden Section version of Powell’s Method applied to SOR with 3 parameters with an increasing amount of subintervals starting from initial guess [1.5,1.5,1.5]. The step size has been set to 0.01.

### Table 15: Coggin’s Method applied to SOR with 3 parameters with an increasing amount of subintervals starting from initial guess [1.5,1.5,1.5]. The step size has been set to 0.01

<table>
<thead>
<tr>
<th>#Subintervals</th>
<th>Minimum Iterations</th>
<th>#Func. Eval</th>
<th>Time</th>
<th>Time/#Func. Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>310</td>
<td>617</td>
<td>2</td>
<td>0.003</td>
</tr>
<tr>
<td>[200]</td>
<td>601</td>
<td>750</td>
<td>13</td>
<td>0.017</td>
</tr>
<tr>
<td>[500]</td>
<td>1472</td>
<td>512</td>
<td>29</td>
<td>0.057</td>
</tr>
<tr>
<td>[800]</td>
<td>2400</td>
<td>509</td>
<td>58</td>
<td>0.1</td>
</tr>
<tr>
<td>[1000]</td>
<td>3000</td>
<td>576</td>
<td>114</td>
<td>0.2</td>
</tr>
<tr>
<td>[2000]</td>
<td>6000</td>
<td>956</td>
<td>425</td>
<td>0.45</td>
</tr>
</tbody>
</table>

Table 15: Coggin’s Method applied to SOR with 3 parameters with an increasing amount of subintervals starting from initial guess [1.5,1.5,1.5]. The step size has been set to 0.01.
7.1.3 Simulated Annealing

In table 16 we can see that Simulated Annealing finds the same minima as the previous methods, but uses much more function evaluations. For some subintervals it requires almost 10 times more function evaluations than the other methods. As we have seen before, Simulated Annealing is somewhat inconsistent because of its randomness. This can be seen from the fact that it completed the optimization problem faster with 2000 subintervals than it did for the case with 1000 subintervals. This can easily happen when it gets lucky and finds the right trajectory quickly. We can see that for 1000 and 2000 intervals the time per function evaluation of Simulated Annealing is slightly lower than in the other methods. This is because in Simulated Annealing a lot of very bad \( \omega \) values are tried, when this happens SOR quickly realises that these parameters lead to divergence and it stops the algorithm prematurely, and hence some function evaluations are done quickly.

<table>
<thead>
<tr>
<th>#Subintervals</th>
<th>Minimum Iterations</th>
<th>#Func. Eval</th>
<th>Time</th>
<th>Time/#Func. Eval</th>
</tr>
</thead>
<tbody>
<tr>
<td>[100]</td>
<td>267</td>
<td>2155</td>
<td>13</td>
<td>0.006</td>
</tr>
<tr>
<td>[200]</td>
<td>574</td>
<td>1576</td>
<td>30</td>
<td>0.02</td>
</tr>
<tr>
<td>[500]</td>
<td>1500</td>
<td>2016</td>
<td>122</td>
<td>0.06</td>
</tr>
<tr>
<td>[800]</td>
<td>2400</td>
<td>2338</td>
<td>265</td>
<td>0.11</td>
</tr>
<tr>
<td>[1000]</td>
<td>3042</td>
<td>8036</td>
<td>1025</td>
<td>0.13</td>
</tr>
<tr>
<td>[2000]</td>
<td>6000</td>
<td>3000</td>
<td>925</td>
<td>0.31</td>
</tr>
</tbody>
</table>

Table 16: Simulated Annealing applied to SOR with 3 parameters with an increasing amount of subintervals starting from initial guess \([1.5,1.5,1.5]\)

7.1.4 Comparison of the methods

When applying the 3 methods to SOR with increasing intervals it is quickly seen that The Downhill Simplex Method performs the best out of all the methods, because it did the job faster. For 2000 intervals it was more than 5 times faster than all other methods. As expected we have seen that Simulated Annealing performed very poorly. In the previous section we had already seen that the method relies on trying a lot of points, and does so with not too much strategy. This means it uses a somewhat unnecessary amount of function evaluations and hence it is not suited to be used on a problem where speed is required. Powell’s Method performed better than Simulated Annealing but it was not as fast DSM. This leads to the conclusion that DSM is the most efficient of the 3 methods for our problem.

7.2 non-equidistant grids

For equidistant grids we approximated the second derivative by \( y''(x_i) \approx \frac{1}{h^2}(y_{i+1} - 2y_i + y_{i-1}) \). A more generalised version for non-equidistant grid approximates the second derivative by the following formula:

\[
y''(x_i) \approx \frac{1}{\Delta x_l(\Delta x_r + \Delta x_l)} \cdot y_{i-1} - \left( \frac{1}{2\Delta x_l(\Delta x_r + \Delta x_l)} + \frac{1}{2\Delta x_r(\Delta x_r + \Delta x_l)} \right) \cdot y_i + \frac{1}{\Delta x_r(\Delta x_r + \Delta x_l)} \cdot y_{i+1}
\]

where \( \Delta x_r \) is the midpoint between \([x_{i-1}, x_i]\) and \( \Delta x_l \) is the midpoint between \([x_{i-1}, x_i]\). With this formula we can now apply SOR to a non-equidistant discretization, and study how the optimization methods react to finding optimal parameters. In the previous section robustness of the methods was assumed but here it is again thoroughly investigated. This is because taking the non-equidistant grid creates a completely different matrix equation that SOR has to solve, with different optimal parameters as before. In this section we will not test The Simulated Annealing Method as it is continually outperformed by the other methods. For Powell’s Method only Coggin’s Method is tested as no big difference was found between the 2 versions of Powell’s Method and when there was a difference, Coggin’s Method performed better.
All the non-equidistant grids tested in this section will contain 100 subintervals. The SOR residue is set back to $10^{-6}$, because for 100 subintervals this is computationally reasonable. The MATLAB code can be found in Appendix A.10.

grid searches

Before we continue we should briefly discuss grid searches. From this point onward we will apply grid searches to our problem in order to most effectively use our optimization methods. During a grid search we apply an optimization method to a grid of initial points. This grid is constructed such that it covers a wide area around the suspected global minimum. These grid searches take some time, but in the end will provide the best results our optimization methods have to offer. In the tables a grid search is denoted as ‘grid 0:0.5:2’, meaning the optimization method was used on a grid where all 3 dimensions are tested from 0 to 2 with steps of 0.5. The word grid will be used interchangeably from now on but from context it will be clear whether it refers to a grid search or a discretization grid.

7.2.1 Random grid

The first non-equidistant discretization grid that will be considered is constructed by simply drawing 100 numbers uniformly between [0,1] and setting these drawn numbers as nodes. This results in a somewhat evenly spread grid with constantly different lengths in between nodes, which makes it a perfect test case to see how the optimization methods and SOR react to this.

From the tables we can see that the Downhill Simplex Method seems to find local minima from the given initial points. We know these are local minima since a grid search was conducted that found the suspected global minimum of 871. We have seen that for the equidistant case that DSM would converge to the global minimum from most given Initial Points within the [0,2]^3 interval. This is not the case for the random grid.

Coggin’s Method was not able to find the suspected global minimal value of 871. It managed to find local minima starting from [0.5,0.5,0.5] and [1.5,1.5,1.5]. In attempt of finding the global minimum, Coggin’s Method was also applied to a grid of dimensions [0:0.5:2] but the global minimum was not found. The best initial point of that grid was the point [0,0,0], hence the result of the grid search is the same as the result from starting at [0,0,0].

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.5,1.5,1.5]</td>
<td>[1.0159, 1.9732, 1.9731]</td>
<td>898</td>
<td>198</td>
<td>4</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[1.6010, 1.9754, 1.9721]</td>
<td>953</td>
<td>259</td>
<td>11</td>
</tr>
<tr>
<td>[5,5,5]</td>
<td>[0.7158, 1.9764, 1.9712]</td>
<td>937</td>
<td>151</td>
<td>5</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>[0.0262, 1.9727, 1.9750]</td>
<td>871</td>
<td>25000</td>
<td>1334</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Coggin’s Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.5,1.5,1.5]</td>
<td>1.4931, 1.9736, 1.9739</td>
<td>907</td>
<td>2148</td>
<td>16</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
<td>4733</td>
<td>339</td>
</tr>
<tr>
<td>[0.5,0.5,0.5]</td>
<td>0.2963, 1.9791, 1.9684</td>
<td>949</td>
<td>2428</td>
<td>101</td>
</tr>
<tr>
<td>[5,5,5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>270</td>
<td>0.1</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>1.4931, 1.9736, 1.9739</td>
<td>907</td>
<td>60000</td>
<td>3000</td>
</tr>
</tbody>
</table>

Table 17: Downhill Simplex and Coggins applied to SOR. Where SOR is solving the discretized system using the random grid
7.2.2 grid with dense edges

The next grid we will consider is one with dense edges. It is constructed such that there are 90 nodes contained in the intervals $[0,0.2]$ and $[0.8,1]$ and the other 10 nodes are contained in $[0.2,0.8]$.

We can see from the tables that this is an easy problem for DSM as it finds the global minimum from all the initial points. A grid search was performed and DSM the global minimum for all 125 initial points. Even when starting at $[5,5,5]$ it manages to find the global minimum.

Powell’s Method converges from almost all initial conditions to local minima close to the global minimum. A grid search was performed where Powell’s Method managed to converge to the global minimum.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1.5,1.5,1.5]$</td>
<td>0.0818, 1.9209, 1.9475</td>
<td>361</td>
<td>311</td>
<td>2</td>
</tr>
<tr>
<td>$[0,0,0]$</td>
<td>0.0815, 1.9366, 1.9316</td>
<td>361</td>
<td>305</td>
<td>5</td>
</tr>
<tr>
<td>$[5,5,5]$</td>
<td>0.0820, 1.9085, 1.9604</td>
<td>361</td>
<td>255</td>
<td>2</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>0.0821, 1.9279, 1.9403</td>
<td>361</td>
<td>30000</td>
<td>227</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Coggin’s Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1.5,1.5,1.5]$</td>
<td>0.0760, 1.8877, 1.9832</td>
<td>362</td>
<td>439</td>
<td>4</td>
</tr>
<tr>
<td>$[0,0,0]$</td>
<td>0.1457, 1.9371, 1.9307</td>
<td>369</td>
<td>529</td>
<td>36</td>
</tr>
<tr>
<td>$[0.5,0.5,0.5]$</td>
<td>0.1188, 1.9334, 1.9342</td>
<td>365</td>
<td>691</td>
<td>31</td>
</tr>
<tr>
<td>$[5,5,5]$</td>
<td>not relevant</td>
<td>divergence</td>
<td>166</td>
<td>0.1</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>0.0817, 1.9251, 1.9574</td>
<td>361</td>
<td>300000</td>
<td>2500</td>
</tr>
</tbody>
</table>

Table 18: Downhill Simplex and Coggin’s applied to SOR. Where SOR is solving the discretized system using the grid with dense edges

7.2.3 grid with dense centre

The last discretization grid we will test is a grid with a dense centre. 90 nodes are contained in the interval $[0.4,0.6]$ and 10 nodes in the $[0,0.4]$ and $[0.6,1]$. From the tables we can see that DSM seems to constantly converge close to the global minimum. Continuing the trend of this section, Powell’s Method again was not able to find the suspected global minimum that was found by the Downhill Simplex Grid search.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1.5,1.5,1.5]$</td>
<td>0.0338, 1.9428, 1.0194</td>
<td>1108</td>
<td>257</td>
<td>8</td>
</tr>
<tr>
<td>$[0.5,0.5,0.5]$</td>
<td>0.0344, 1.9315, 2.0319</td>
<td>1105</td>
<td>281</td>
<td>7</td>
</tr>
<tr>
<td>$[5,5,5]$</td>
<td>0.0319, 1.9335, 1.0296</td>
<td>1106</td>
<td>263</td>
<td>10</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>0.0333, 1.9455, 2.0166</td>
<td>1105</td>
<td>5000</td>
<td>1264</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Coggin’s Method</th>
<th>Minimum Value</th>
<th>#Func. Evals</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[1.5,1.5,1.5]$</td>
<td>0.1005, 1.9128, 2.0339</td>
<td>2508</td>
<td>1405</td>
<td>47</td>
</tr>
<tr>
<td>$[1,1,1]$</td>
<td>1.3691, 1.9765, 1.9837</td>
<td>1278</td>
<td>588</td>
<td>49</td>
</tr>
<tr>
<td>$[0.5,0.5,0.5]$</td>
<td>0.1005, 1.9327, 2.0304</td>
<td>1128</td>
<td>668</td>
<td>217</td>
</tr>
<tr>
<td>$[0,0,0]$</td>
<td>not relevant</td>
<td>divergence</td>
<td>2827</td>
<td>183</td>
</tr>
<tr>
<td>$[5,5,5]$</td>
<td>not relevant</td>
<td>divergence</td>
<td>199</td>
<td>0.2</td>
</tr>
<tr>
<td>grid 0:0.5:2</td>
<td>0.1005, 1.9327, 2.0304</td>
<td>1128</td>
<td>75000</td>
<td>18600</td>
</tr>
</tbody>
</table>

Table 19: Downhill Simplex and Coggin’s Method applied to SOR. Where SOR is solving the discretized system using the grid with dense centre.
7.2.4 Comparison of the methods for non-equidistant grids

We have applied Powell’s Method and DSM to 3 different optimization problems related to solving the ODE on different non-equidistant grids and DSM proved to be superior in each case. DSM found lower minima than Powell’s Method in each instance and did so faster as well. Powell’s Method applied to a grid of initial conditions did not give the desired results for each of the 3 grids.

8 Changing the ODE

So far we have applied SOR to a discretized system of the ODE $y''(x) + 10^{-4} \cdot y(x) = x$. Since our goal is to investigate how the optimization methods react to different optimization problems concerning SOR, we will change the ODE such that the discretized system is harder to solve. We do this by changing the ODE to:

$$10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$$

By increasing the difficulty of the ODE it is expected that SOR needs low $\omega$ values in order to converge. In this section we investigate whether or not DSM is able to find these optimal parameters. We will only discuss DSM since it was the only method that had relevant results. Powell’s Method could not find a single minimum unless the initial point was extremely close to it. We will solve the ODE on 3 different grids all containing 100 subintervals: The equidistant grid, the random grid and the grid with dense edges. Since these 3 grids give a different matrix equations to solve for SOR, they are ideal to test DSM on. After the tests on the 3 non-equidistant grids, we will do 2 more tests on the equidistant grid, one using SOR with 6 parameters and one where we again increase the difficulty of the ODE. The MATLAB code can be found in Appendix A.11.

8.1 Equidistant grid

From the tables we can see that DSM finds reasonable minima when starting close to or on top of $[0,0,0]$. A grid search was done which gives the suspected global minimum of $[1.0000, 0.1250, 0.1250]$, which was found by DSM when started from $[0,0,0]$. When DSM was started from $[1.5,1.5,1.5]$ it found parameters for which SOR converges but they were not close to optimal. The reason why it is hard to find optimal parameters starting from $[1.5,1.5,1.5]$ is because SOR does not converge for any parameters close to it. Since the ODE is harder now SOR needs to underrelax, the optimal parameters seem to be $[1.0125,0.125]$. When DSM constructs its initial simplex from $[1.5,1.5,1.5]$ it creates 3 more points, namely $[2.5,1.5,1.5]$, $[1.5,2.5,1.5]$ and $[1.5,1.5,2.5]$, for all 4 points on the initial simplex the method diverges, which makes for an impossible start for DSM.

The reader may have noticed that the global minimum found is suspicious. It might be that because of our methods used it ended up on the precise fractions $1$ and $\frac{1}{8}$, and hence the minimum found might not be global. If we display more decimals the minimum $[1.000, 0.1250, 1.250]$ is rather $[1.000007, 0.124999, 0.124999]$ and hence these are not precise fractions. When $0.0001$ is added to the 3 vector components the function evaluation goes from 101 to 135, showing that the omega’s are very sensitive at this point and it is clearly a local minimum.

Another Downhill Simplex grid search was performed that looked into Initial points starting with (partly) negative $\omega$ values but this grid search found no new results.
<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.5,1.5,1.5]</td>
<td>[1.6339, 0.0123, 0.8372]</td>
<td>1243</td>
<td>1323</td>
<td>10</td>
</tr>
<tr>
<td>[1,1,1]</td>
<td>[1.9181, 0.1213, 0.1200]</td>
<td>170</td>
<td>368</td>
<td>3</td>
</tr>
<tr>
<td>[0.5,0.5,0.5]</td>
<td>[0.2575, 0.1240, 0.1238]</td>
<td>139</td>
<td>559</td>
<td>2</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[1.0000, 0.1250, 0.1250]</td>
<td>101</td>
<td>428</td>
<td>3</td>
</tr>
<tr>
<td>grid [0:2,0:0.5,0:0.5]</td>
<td>[1.0000, 0.1250, 0.1250]</td>
<td>101</td>
<td>1000 000</td>
<td>3000</td>
</tr>
<tr>
<td>grid [-0.5:2,-0.1:0.5,-0.1:0.5]</td>
<td>[1.0000, 0.1250, 0.1250]</td>
<td>101</td>
<td>1000 000</td>
<td>3000</td>
</tr>
</tbody>
</table>

Table 20: Downhill Simplex applied to SOR. Where SOR is solving the ODE $10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ on an equidistant grid containing 100 subintervals.

8.2 random grid

From the table we can see that the suspected global minimum value is 2524 iterations, which is found from 2 different initial points. DSM finds the minimum from [1.5,1.5,1.5] but not from [1,1,1] and [0.5,0.5,0.5]. There does not seem to be a theoretical explanation for why this is the case other than for the fact that in general these optimization methods have a luck component. Sometimes they simply hit the right spot and sometimes they do not. In this case DSM is simply fortunate when started from [1.5,1.5,1.5], as a vertex landed in the correct spot causing it to find the minimum.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.5,1.5,1.5]</td>
<td>[1.9676, 0.0795, 0.0710]</td>
<td>2524</td>
<td>567</td>
<td>19</td>
</tr>
<tr>
<td>[1,1,1]</td>
<td>not relevant</td>
<td>divergence</td>
<td>173</td>
<td>0.5</td>
</tr>
<tr>
<td>[0.5,0.5,0.5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>152</td>
<td>0.4</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[1.9908, 0.0795, 0.0710]</td>
<td>2524</td>
<td>534</td>
<td>9</td>
</tr>
<tr>
<td>grid [0:2,0:0.5,0:0.5]</td>
<td>[1.9908, 0.0795, 0.0710]</td>
<td>2524</td>
<td>534</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 21: Downhill Simplex applied to SOR. Where SOR is solving the ODE $10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ on the random grid with 100 subintervals.

8.3 grid with dense edges

From the table we can see that the discretization grid with dense edges has resulted in a severe decrease in robustness of DSM as it needs to start very close to the minimum in order to find it. Just as for the equidistant grid there is no better place to start than on [0,0,0].

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.5,0.5,0.5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>175</td>
<td>0.05</td>
</tr>
<tr>
<td>[0.1,0.1,0.1]</td>
<td>not relevant</td>
<td>divergence</td>
<td>163</td>
<td>0.4</td>
</tr>
<tr>
<td>[0.05,0.05,0.05]</td>
<td>[0.0347, 0.0207, 0.0207]</td>
<td>1175</td>
<td>188</td>
<td>2</td>
</tr>
<tr>
<td>[0.0,0]</td>
<td>[0.0286, 0.0238, 0.0180]</td>
<td>1159</td>
<td>268</td>
<td>4</td>
</tr>
<tr>
<td>grid [0:2,0:0.5,0:0.5]</td>
<td>[0.0286, 0.0238, 0.0180]</td>
<td>1159</td>
<td>268</td>
<td>6000</td>
</tr>
</tbody>
</table>

Table 22: Downhill Simplex Method applied to SOR. Where SOR is solving the ODE $10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ on the grid with dense edges containing 100 subintervals.
8.4 SOR with 6 parameters

Now we test the changed ODE on an equidistant grid with 100 subintervals, using SOR with 6 parameters. As we can see from the table Downhill Simplex does well in this case compared to the SOR with 3 parameters on an equidistant grid. It comes close to the suspected global minimum of 151 iterations when started from multiple initial points. In some sense it does even better than the 3 parameter case since from [1,1,1,1,1,1] it came very close to the minimum unlike SOR with 3 parameters started on [1,1,1], which did not come nearly as close to the global minimum. This could be explained by the fact that DSM can actually benefit from more dimensions as the simplex will have more vertices and thus could provide a more stable path.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Min. Val.</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[1.5,1.5,1.5,1.5,1.5,1.5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>730</td>
<td>4</td>
</tr>
<tr>
<td>[1,1,1,1,1,1]</td>
<td>[0.2097, 0.1194, 0.1135, 0.1297, 0.1045, 0.1357]</td>
<td>154</td>
<td>1838</td>
<td>12</td>
</tr>
<tr>
<td>[0.5,0.5,0.5,0.5,0.5,0.5]</td>
<td>[0.9173, 0.1495, 0.1332, 0.0968, 0.1264, 0.1218]</td>
<td>161</td>
<td>783</td>
<td>6</td>
</tr>
<tr>
<td>[0,0,0,0,0,0]</td>
<td>[0.2123, 0.1434, 0.1175, 0.1052, 0.1241, 0.1351]</td>
<td>151</td>
<td>882</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 23: Downhill Simplex Method applied to SOR with 6 parameters. Where SOR is solving the ODE $10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ on the grid with dense edges containing 100 subintervals.

8.5 Harder ODE

We now change some of the coefficients of our ODE to go from $10^{-3} \cdot y''(x) + 3 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ to $10^{-4} \cdot y''(x) + 10 \cdot y'(x) + 10^{-4} \cdot y(x) = x$. This results in a even harder ODE as the solution of the discretized system becomes very chaotic. In practice, grid refinement will be used to find a more decent solution of the ODE, but since we want to make the matrix problem as difficult as possible we stick to a rough grid.

DSM has a much harder time finding the global minimum in this case, for the first time it does not find the lowest minimum when started from [0,0,0]. A grid search was performed using DSM and it found the suspected minimum of [0.1087, 0.1157, 0.0001] where SOR needs 293 iterations.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0.5,0.5,0.5]</td>
<td>not relevant</td>
<td>divergence</td>
<td>124</td>
<td>1</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[0.1294, 0.1001, 0.0002]</td>
<td>332</td>
<td>1298</td>
<td>12</td>
</tr>
<tr>
<td>[0.1294, 0.1001, 0.0002]</td>
<td>[0.2262, 0.1.161, 0.0001]</td>
<td>298</td>
<td>218</td>
<td>2</td>
</tr>
<tr>
<td>grid [0-0.4,0:0.2,0:0.2]</td>
<td>[0.1087, 0.1157, 0.0001]</td>
<td>293</td>
<td>not measured</td>
<td>1046</td>
</tr>
</tbody>
</table>

Table 24: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the ODE $10^{-4} \cdot y''(x) + 10 \cdot y'(x) + 10^{-4} \cdot y(x) = x$ on the equidistant grid containing 100 subintervals.

8.6 Conclusion

After changing the ODE we can see that it is harder for our optimization methods to find the optimal parameters. Powell’s method does not find a single minimum, and DSM needs to start relatively close to the minimum to find it. However it consistently found the minimum when starting from [0,0,0]. The most reliable method is to apply DSM on a grid, which has consistently found us the lowest minima for all problems.
9 Solving a PDE

In this section we will be concerned with solving a 2-dimensional PDE. This will be done by discretizing the x,y plane using an equidistant 2-dimensional grid where the x-axis is indexed by i and the y-axis is indexed by j. The x-axis and the y-axis are split equidistantly ($hx = hy$) with 15 intervals, giving a total of 256 grid points. The derivatives are approximated by the finite difference method, like we did for the ODE. The approximations we will use for the second derivatives are:

$$u_{xx}(x_{i,j}) \approx \frac{1}{(hx)^2} (y_{i+1,j} - 2y_{i,j} + y_{i-1,j}), \quad u_{yy}(x_{i,j}) \approx \frac{1}{(hy)^2} (y_{i,j+1} - 2y_{i,j} + y_{i,j-1})$$

The approximations we will use for the first derivatives are:

$$u_{x}(x_{i,j}) \approx \frac{1}{2} \cdot \frac{1}{hx} (y_{i+1,j} - y_{i-1,j}), \quad u_{y}(x_{i,j}) \approx \frac{1}{2} \cdot \frac{1}{hy} (y_{i,j+1} - y_{i,j-1})$$

In this section first we will solve the PDE:

$$K \nabla u + 5u_x + 3u_y + 10^{-4}u = x + y, \quad u(0,y) = u(1,y) = u(x,0) = u(x,1) = 0$$

Initially K is set to $10^{-4}$. After solving for $K = 10^{-4}$, we will make the PDE more difficult by changing it to $10^{-5}$ and later $10^{-7}$. For this PDE we know the discretization is difficult to solve for SOR, hence in order for SOR to achieve convergence it needs to underrelax heavily. We will only use DSM since Powell’s method has proven not to work for solving the harder ODE and thus it will probably not help for our PDE either. In this section the SOR residual is set to $10^{-3}$. The relevant MATLAB codes are given in Appendix A.12 and A.13.

9.1 The Downhill Simplex Method

DSM is able to find a local minimum starting from [0,0,0], and finds an even lower minimum starting from [1,0,0]. However it needs an extensive Downhill Simplex grid search to find the lowest minimum value at 972 iterations. For this PDE SOR is extremely sensitive to its parameters, for clarity we have shown only the first 4 digits of the parameters in the table but in reality these parameters have up to 10 relevant digits: changing the 10th decimal place of a parameter causes the SOR iterations to go up by over a hundred. This extreme sensitivity makes it very hard for the Downhill Simple Method to converge as there is only a very small parameter space for which SOR converges at all. This means that in order to find the optimal parameters, one needs to know roughly where they are which is a significant downside of DSM.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0,0,0]</td>
<td>[0.9830, 0.0001, 0.2060]</td>
<td>1175</td>
<td>255</td>
<td>4</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
<td>263</td>
<td>0.1</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[1.9815, 0.0001, 0.0251]</td>
<td>1027</td>
<td>272</td>
<td>4</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[0.7056, 0.0018, 0.0012]</td>
<td>7412</td>
<td>133</td>
<td>2</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
<td>124</td>
<td>0.02</td>
</tr>
<tr>
<td>[0,0,0]</td>
<td>[1.2718, 0.0252, 0.0001]</td>
<td>972</td>
<td>250000</td>
<td>1800</td>
</tr>
</tbody>
</table>

Table 25: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the PDE with $K = 10^{-4}$.
9.2 Lowering $K$

Now we lower $K$ to $10^{-5}$ in order to increase the difficulty for SOR to solve the discretized system. For this harder PDE it is very difficult for DSM to find optimal parameters, from $[0,0,0]$ it could not find parameters for which SOR even converges. Since we know the difficulty of this matrix for SOR, we know that the optimal parameters are very small. Since we do not know how small, an exponential Downhill Simplex search grid is performed that tries all combinations of initial points starting from $10^{-4}$ down to $10^{-9}$ with logarithmic steps. This initial Downhill Simplex search grid could be seen as a rough scouting mission trying to find the correct order of magnitude of the initial points that DSM should use. Running the exponential grid resulted in a minimum of 1689 iteration when DSM was initiated at the point $[10^{-5}, 10^{-6}, 10^{-6}]$, giving a rough idea of where DSM should start. With that obtained knowledge a more precise Downhill Simplex grid search was performed around this initial point which found a lower minimum of 1638.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0,0,0]$</td>
<td>not relevant</td>
<td>divergence</td>
<td>34</td>
<td>3</td>
</tr>
</tbody>
</table>
| grid$[10^{-4} : \frac{1}{10} : 10^{-9},$
  $10^{-4} : \frac{1}{10} : 10^{-9},$
  $10^{-4} : \frac{1}{10} : 10^{-9}]$ | [0.982668, 0.017342, 0.0000001] | 1689         | 10000 | 273  |
| grid$[10^{-6} : 10^{-9},$
  $10^{-6} : 10^{-6} : 10^{-5},$
  $10^{-6} : 10^{-6} : 10^{-5}]$ | [1.04476, 0.003580, 0.0000005] | 1638         | 50000 | 1500 |

Table 26: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the PDE with $K = 10^{-5}$

To test the effectiveness of the exponential grid search followed up by a refined grid search we again lower $K$, this time from $10^{-5}$ to $10^{-7}$. The exponential search found a minimum of 2226 values when it initiated DSM at $[10^{-10}, 10^{-10}, 10^{-9}]$, hence a refined grid search was done around these points. This refined grid found a minimum of 1762 when starting from the point $[10^{-11}, 10^{-11}, 9 \cdot 10^{-11}]$.

<table>
<thead>
<tr>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
<th>#Func. Eval.</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>$[0,0,0]$</td>
<td>not relevant</td>
<td>divergence</td>
<td>34</td>
<td>3</td>
</tr>
</tbody>
</table>
| grid$[10^{-8} : \frac{1}{10} : 10^{-10},$
  $10^{-5} : \frac{1}{10} : 10^{-10},$
  $10^{-5} : \frac{1}{10} : 10^{-10}]$ | [0.9827188732, 0.0000000001, 0.0172831267] | 2226         | 7000  | 50   |
| grid$[10^{-11} : 10^{-10} : 10^{-10},$
  $10^{-11} : 10^{-10} : 10^{-10},$
  $10^{-11} : 10^{-10} : 10^{-10}]$ | [0.97854897401, 0.02145102600, 0.00000000009] | 1762         | 50000 | 618  |

Table 27: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the PDE with $K = 10^{-7}$

We can see that the exponential grid search is very effective, as it narrows down a large search range for DSM. When it is roughly known where DSM should start its searches a refined grid can be build that found the suspected global minimum. For both test cases the refined search grid resulted in a lower minimum value.
10 Disturbing the parameters

Another way of studying the robustness of our methods is by artificially changing the values of the parameters. This can simply be done by using relaxation parameter $\omega_2 + 1$, instead of the input $\omega_2$. This would cause the optimal value of $\omega_2$ to shift one unit downwards. In the same way one could multiply or divide the parameter input. We'll apply SOR to the PDE from section 9.1 with $K = 10^{-4}$.

As we can see DSM can handle the shifts, but only after the initial guess is corrected for it. Similar to the case without the parameter shifting, DSM needs to start close to the minimum in order to find it. It does not matter that $\omega_2$ is negative and $\omega_3$ is positive, DSM can precisely find the optimal parameters values for $\omega_2$ and $\omega_3$ around $10^3$ and $10^{-3}$.

<table>
<thead>
<tr>
<th>Shift</th>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No shift</td>
<td>[0,0,0]</td>
<td>[0.9830, 0.0001, 0.0206]</td>
<td>1175</td>
</tr>
<tr>
<td>$\omega_2 + 1$</td>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>$\omega_2 + 1$</td>
<td>[-1,0,0]</td>
<td>[0.9830, -0.9999, 0.0206]</td>
<td>1175</td>
</tr>
<tr>
<td>$\omega_2 + 10^3, \omega_3 - 10^3$</td>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>$\omega_2 + 10^3, \omega_3 - 10^3$</td>
<td>[0,-10^3,10^3]</td>
<td>[0.9830, 0.0001-10^3, 0.0206+10^3]</td>
<td>1175</td>
</tr>
</tbody>
</table>

Table 28: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the PDE with $K = 10^{-4}$. Some parameters are shifted, these values are shown in the first column.

For division we see the same pattern. As long as DSM starts close to the minimum it manages to find it. For a shift of $\omega_2/10^6$, [0,0,0] was not close enough. When this shift was corrected by starting from [0,10,0], DSM had no problem finding the minimum again.

<table>
<thead>
<tr>
<th>Shift</th>
<th>Initial Point</th>
<th>Downhill Simplex Method</th>
<th>Minimum Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>No shift</td>
<td>[0,0,0]</td>
<td>[0.9830, 0.0001, 0.0206]</td>
<td>1175</td>
</tr>
<tr>
<td>$\omega_2/10$</td>
<td>[0,0,0]</td>
<td>[0.9481, 0.0011, 0.0208]</td>
<td>1165</td>
</tr>
<tr>
<td>$\omega_2/10^2$</td>
<td>[0,0,0]</td>
<td>[0.7235, 0.0918, 0.0248]</td>
<td>1052</td>
</tr>
<tr>
<td>$\omega_2/10^6$</td>
<td>[0,0,0]</td>
<td>not relevant</td>
<td>divergence</td>
</tr>
<tr>
<td>$\omega_2/10^6$</td>
<td>[0,10,0]</td>
<td>[1.0018, 9.9981, 0.0229]</td>
<td>1126</td>
</tr>
</tbody>
</table>

Table 29: Downhill Simplex Method applied to SOR with 3 parameters. Where SOR is solving the PDE with $K = 10^{-4}$. Some parameters are shifted, these values are shown in the first column.

From tables 28-29 and the previous parts of this study, it is clear that it does not matter for DSM where the optimal parameters lie. When the SOR problem is hard enough it needs to start close to the minimum. In table 25, where we solved this PDE without the parameter shift, we can see that DSM could not converge starting from the point [0,1,0]. Hence, it is also not possible to converge from [0,0,0] when the second entry is shifted down by 1 unit.
11 Conclusion

After having applied multiple optimization methods to multiple test cases it is time to decide which method is generally best to find optimal parameters for SOR. First a short summary of the results is given.

We started by applying both versions of Powell’s Method to the Rastrigin function and found that provided the right step size they found the global minimum with no problems. After that we applied it to SOR with multiple parameters and observed that as the number of parameters got higher more problems arose. From this point onward we only tested Coggin’s Method as it turned out to be the better version of Powell’s Method.

Next we applied The Downhill Simplex Method to the Rastrigin function and observed that it was unable to find the global minimum, despite taking a big scaling factor. When we applied DSM to SOR we observed that it was very effective in case of 3 parameters but not as much for 6 parameters. For 3 parameters it found local minima fast and effective but for 6 parameters the algorithm became less robust, in addition the method is computationally expensive.

Next we increased the amount of subintervals of the discretization in order to study the efficiency of the methods. When we applied the 3 optimization methods we saw that DSM was by far the fastest as for a discretization of 2000 subintervals it was more than 5 times as fast compared to the other methods.

Then instead of taking an equidistant grid, we took 3 different non-equidistant grids which again gave us 3 new matrix problems for SOR to solve, and thus 3 new optimization problems for our methods. At this point we decided to only test DSM and Coggin’s Method as these 2 methods were the only contenders left for a possible best method. When we applied these 2 methods to our non-equidistant grids we saw that DSM performed much better than Coggin’s Method as it was faster and more robust. From this point we decided to only use DSM for our upcoming optimization problems as it appeared to be better in all circumstances.

Next we changed the ODE by adding a first derivative, which made the problem significantly harder for SOR to solve. We solved this ODE on 3 different grids and saw that DSM had a hard time finding the optimal parameters but appeared to be successful every time when starting from the initial point [0,0,0]. For each of the 3 grids this minimum found from [0,0,0] was confirmed to be the global minimum by an extensive Downhill Simplex grid search.

Then we changed the ODE into a more difficult PDE. DSM could not find the global minimum from [0,0,0], as a Downhill Simplex grid search was performed that found a better minimum. The PDE was made more difficult by lowering K, which caused DSM to fail finding a triple of parameters that would cause SOR to converge. An exponential Downhill Simplex grid search was performed to get an idea of where Downhill should start, and successfully found a good initial point. Then a more refined grid search was done around that point which lead to finding the suspected global minimum. After this method seemed successful it was tested again on an even harder PDE where K was lowered to $10^{-7}$ and again it appeared successful.

Finally a quick study was done to see how DSM reacted when we artificially changed the optimal parameter values and found that DSM has no problem finding the optimal values as long as it starts close to the minimum.
The underlying aim of this study was to give a general recommendation on how to find optimal parameters for SOR problems encountered in fluid dynamics. The dimensions of this matrix problem are extremely high so optimal parameters could greatly reduce computation time. From this study it is clear that the recommended strategy is using an exponential DSM grid search to find out the scale of the parameters if this is not known already. When it is known roughly where the minimum lies a refined DSM grid search should be performed to closely look around, hopefully finding the global minimum.

Simulated Annealing is simply too expensive when the matrix dimensions of the problem are high, hence it should not be used. The current version of Powell’s method is not really suited for this optimization problem under any circumstances, as it is always outperformed by DSM. As suggested, there might be room to improve Powell’s Method to make it a better fit for our optimization problem.

12 Suggestions for further research

In this study, versions of Powell’s Method, Simulated Annealing and DSM have been used that are designed to optimize any given function. That means they are generic function optimizers with no special interest in solving SOR. The code of these optimization methods are at most 2 pages, which means they are relatively basic and there is much room for improvement. A suggestion for further research is improving these optimization methods such that they are better suited for optimizing SOR. A method that needs clear improvement is Powell’s Method. Powell’s Method started out promising in our study, but as the matrix equations became harder, it started to consistently diverge and couldn’t find anything close to the global minimum anymore. This was because Powell’s Method had problems with the sensitivity of SOR regarding the $\omega$ parameters. When the trajectory of Powell’s method started to go off track, it hardly ever came back to converge successfully. This could be improved by adding a ‘memory’ to Powell’s Method, such that when it starts to diverge it can stop its search and start again at a point where it was earlier, this time searching in a different direction.

Another interesting experiment is using DSM to optimize itself. Since DSM relies on 4 parameters, the method itself could be used to find optimal parameters for itself. This is a computationally challenging process, but it might bring to light new information about DSM.

Another suggestion would be to try new optimization methods to optimize SOR. There are many different optimization algorithms, many of which have not been tried yet in this study. In [7] a review is given of many different derivative-free optimization methods.
13 References


A.1 Rastrigin Function

```matlab
% 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.2 SOR to solve ODE

```matlab
function [it,xsol]=gssor(L,D,R,b,N,x)
% 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

% This version of SOR uses 3 parameters
% More parameters can be added manually

% initial solution
xsol = zeros(N,1);
resid = 1E3;
it = 0;
while resid>1E-3
    % 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) = x(1)*xsol(1) + (1-x(1))*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) = x(2)*xsol(i) + (1-x(2))*xsolold(i);
        else
```
xsol(i) = x(3)*xsol(i) + (1-x(3))*xsolold(i);
end
Sum = L(N)*xsol(N-1);
xsol(N) = (b(N)-Sum)/D(N);
xsol(N) = x(1)*xsol(N) + (1-x(1))*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
it = it+ resid^0.1;

A.3 Discretization of equidistant 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

%The number of subintervals + 1
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);
\[ b(i) = (i-1) \cdot dx; \]

\[ D(1) = 1; \]

\[ D(N) = 1; \]

\section*{A.4 Powell's Method}

\begin{verbatim}
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;
end
end
\end{verbatim}
D = eye(n);
ips = ip;
if ip && n == 2
    figure(abs(ip));
end
[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
xo = x0;
yo = y0;
it = 0;
nS = 1;

% Step size constant used by the bracketing
stp = 0.01;
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, stp);
            Ot = Ot*problem;
        else
            [stepsize, x, Ot, nS1] = coggins(S, xo, D(:, i), 0*ips, problem, tol, mxit, stp);
        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, stp);
        yo = yo*problem;
    else
        [stepsize, xo, yo, nS1] = coggins(S, x, D(:, n), 0*ips, problem, tol, mxit, stp);
        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

% Displaying the trajectory (x0) and its corresponding evaluation (y0)
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.5 Bracketing the 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
    if y0 <= y
      stepsize = stepsize;
      j = j+1;
    else
      stepsize = stepsize;
      j = j+1;
    end
  end
end
if nS > 1000
  warning('Too many function evaluations');
end
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
x2 = x;
x1 = x0+stepsize*(j-1)*d;

A.6 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(:);
d = 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);
return
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
end
A.7 Coggin’s Method

```matlab
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)
        if nargin < 5 || isempty(problem)
            problem = -1;
        end
        if nargin < 7 || isempty(mxit)
            mxit = 50*(1+4*(ip>0));
        end
    end
```

ip = 0;
end
if nargin < 5 || isempty(problem)
    problem = -1;
end
if nargin < 6 || isempty(tol)
    tol = 1e-5;
end
if nargin < 7 || isempty(mxit)
    mxit = 50*(1+4*(~(ip>0)));
end
d = d(:);
nd = d'*d;
if nargin < 8 || isempty(stp)
    stepsize = 0.5*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;
A.8 The Downhill Simplex Method

```matlab
function [xmin, fmin, ct]=ANMS(myfunction, xinit, tol, max_feval)

% Adaptive Nelder-Mead Simplex Algorithm for
% solving the unconstrained optimization problem:
% min f(x).
% It uses the adaptive parameters introduced in the following paper:
% Fuchang Gao and Lixing Han
% "Implementing the Nelder-Mead simplex algorithm with adaptive parameters"
% It also uses a relatively large initial simplex (The initial simplex
% used in the numerical experiments in the above paper is small in order
% to compare ANMS and FMINSEARCH).

% REMARK:
% If you use ANMS as a local search method in combination with a
% metaheuristic method, you may want to use a smaller initial simplex.
% by Fuchang Gao and Lixing Han
% coded August, 2010
% slightly updated August, 2015
%
% Inputs:
% xinit--initial guess
% tol--tolerance for termination (Recommended value: 10^-4)
% max_feval--maximum number of function evaluations
% myfunction--objective function. (In myfunction(x), x is a ROW vector)
%
% Outputs:
% xmin--approximate optimal solution at termination. It is a row vector.
% fmin--minimum function value at termination
% ct--number of function evaluations at termination
%
% x0=xinit(:)';  % x0 is a row vector.
myfunction = fcnchk(myfunction);
dim=max(size(x0));  % dimension of the problem

% set up adaptive parameters
alpha=1;  beta=1+2/dim;  gamma=0.75-0.5/dim;  delta=1-1/dim;
% Construct the initial simplex: Large initial simplex is used.
% This scalefactor can be changed manually.
```
% scalefactor = min(max(max(abs(x0)),1),10);
D0=eye(dim);
D0(dim+1,:)=(1-sqrt(dim+1))/dim*ones(1,dim);
for i=1:dim+1
    X(i,:)=x0+scalefactor*D0(i,:);
    FX(i)=feval(myfunction,X(i,:));
end;
ct=dim+1;
[FX,I]=sort(FX);
X=X(I,:);
% Main iteration
while max(max(abs(X(2:dim+1,:)-X(1:dim,:)))) >= scalefactor*tol
    if ct>max_feval
        break;
    end
    M=mean(X(1:dim,:)); % Centroid of the dim best vertices
    xref=(1+alpha)*M-alpha*X(dim+1,:);
    Fref=feval(myfunction,xref);
    ct=ct+1;
    if Fref<FX(1)
        % expansion
        xexp=(1+alpha*beta)*M-alpha*beta*X(dim+1,:);
        Fexp=feval(myfunction,xexp);
        ct=ct+1;
        if Fexp < Fref
            X(dim+1,:)=xexp;
            FX(dim+1)=Fexp;
        else
            X(dim+1,:)=xref;
            FX(dim+1)=Fref;
        end;
    else
        if Fref<FX(dim)
            % accept reflection point
            X(dim+1,:)=xref;
            FX(dim+1)=Fref;
        else
            if Fref<FX(dim+1)
                % Outside contraction
                xoc=(1+alpha*gamma)*M-alpha*gamma*X(dim+1,:);
                Foc=feval(myfunction,xoc);
                ct=ct+1;
                if Foc<=Fref
                    X(dim+1,:)=xoc;
                    FX(dim+1)=Foc;
                else
                    % shrink
                    for i=2:dim+1
                        X(i,:)=X(1,:)+delta*(X(i,:)-X(1,:));
                    end;
                end
            else
                % Inside contraction
                for i=1:dim
                    X(i,:)=X(i,:)+delta*(X(i,:)-X(1,:));
                end;
            end
        else
            % Shrink
            for i=2:dim+1
                X(i,:)=X(1,:)+delta*(X(i,:)-X(1,:));
            end;
        end
    end
end;
function [minimum,fval] = anneal(loss, parent, options)

% ANNEAL Minimizes a function with the method of simulated annealing
% (Kirkpatrick et al., 1983)

% ANNEAL takes three input parameters, in this order:
% LOSS is a function handle (anonymous function or inline) with a loss
% function, which may be of any type, and needn’t be continuous. It does,
% however, need to return a single value.
% PARENT is a vector with initial guess parameters. You must input an
% initial guess.
% OPTIONS is a structure with settings for the simulated annealing. If no
% OPTIONS structure is provided, ANNEAL uses a default structure. OPTIONS
% can contain any or all of the following fields (missing fields are
% filled with default values):
% Verbosity: Controls output to the screen.
% 0 suppresses all output
% 1 gives final report only [default]
% 2 gives temperature changes and final report
% Generator: Generates a new solution from an old one.
% Any function handle that takes a solution as input and
% gives a valid solution (i.e. some point in the solution
% space) as output.
% The default function generates a row vector which
% slightly differs from the input vector in one element:
% @(x) (x+(randperm(length(x))==length(x))*randn/100)
% Other examples of possible solution generators:
% @(x) (rand(3,1)): Picks a random point in the unit cube
% @(x) (ceil([9 5].*rand(2,1))): Picks a point in a 9-by
% discrete grid
% Note that if you use the default generator, ANNEAL only
% works on row vectors. For loss functions that operate
% on
% column vectors, use this generator instead of the
default:
% @(x) (x(:,')+(randperm(length(x))==length(x))*randn/100)
% InitTemp: The initial temperature, can be any positive number.
% Default is 1.
% StopTemp: Temperature at which to stop, can be any positive number
% smaller than InitTemp.
% Default is 1e-8.
% StopVal: Value at which to stop immediately, can be any output
% of
% LOSS that is sufficiently low for you.
% Default is -Inf.
% CoolSched: Generates a new temperature from the previous one.
% Any function handle that takes a scalar as input and
% returns a smaller but positive scalar as output.
% Default is @(T) (.8*T)
% MaxConsRej: Maximum number of consecutive rejections, can be any
% positive number.
% Default is 1000.
% MaxTries: Maximum number of tries within one temperature, can be
% any positive number.
% Default is 300.
% MaxSuccess: Maximum number of successes within one temperature, can
% be any positive number.
% Default is 20.

% Usage:
% [MINIMUM,FVAL] = ANNEAL(LOSS,NEWSOL,[OPTIONS]);
% MINIMUM is the solution which generated the smallest
% encountered
% value when input into LOSS.
% FVAL is the value of the LOSS function evaluated at MINIMUM.
OPTIONS = ANNEAL();

OPTIONS is the default options structure.

Example:
The so-called "six-hump camelback" function has several local minima in the range \(-3 \leq x \leq 3, -2 \leq y \leq 2\). It has two global minima, namely
\[ f(-0.0898, 0.7126) = f(0.0898, -0.7126) = -1.0316. \]
We can define and minimise it as follows:
\[
camel = @(x,y)(4-2.1*x.*x.^2+ x.*x.^4/3).*x.^2+x.*y+4*(y.^2-1).*y.^2;
\]
\[
loss = @(p)camel(p(1),p(2));
\]
\[
[x f] = ANNEAL(loss,[0 0])
\]

We get output:

Initial temperature: 1
Final temperature: 3.21388e-007
Consecutive rejections: 1027
Number of function calls: 6220
Total final loss: -1.03163

\[
x = -0.0899 0.7127
\]
\[
f = -1.0316
\]

Which reasonably approximates the analytical global minimum (note that due to randomness, your results will likely not be exactly the same).

Reference:

joachim.vandekerckhove@psy.kuleuven.be

$Revision: v5 $ $Date: 2006/04/26 12:54:04 $
error('MATLAB:anneal:badOptions',...  
'Input argument ''options'' is not a structure')
end
fs = {'CoolSched','Generator','InitTemp','MaxConsRej',...  
'MaxSuccess','MaxTries','StopTemp','StopVal','Verbosity'};
for nm=1:length(fs)
  if ~isfield(options,fs{nm}), options.(fs{nm}) = def.(fs{nm}); end
end

% main settings
newsol = options.Generator; % neighborhood space function
Tinit = options.InitTemp; % initial temp
minT = options.StopTemp; % stopping temp
cool = options.CoolSched; % annealing schedule
minF = options.StopVal;
max_consec_rejections = options.MaxConsRej;
max_try = options.MaxTries;
max_success = options.MaxSuccess;
report = options.Verbosity;
k = 1; % boltzmann constant

% counters etc
itry = 0;
success = 0;
finished = 0;
consec = 0;
T = Tinit;
initenergy = loss(parent);
oldenergy = initenergy;
total = 0;
if report==2, fprintf(1,'\n T = %7.5f, loss = %10.5f\n',T,oldenergy); end
while ~finished;
  itry = itry+1; % just an iteration counter
  current = parent;
  % % Stop / decrement T criteria
  if itry >= max_try || success >= max_success;
    if T < minT || consec >= max_consec_rejections;
      finished = 1;
      total = total + itry;
      break;
    else
      T = cool(T); % decrease T according to cooling schedule
      if report==2, % output
        fprintf(1,'\n T = %7.5f, loss = %10.5f\n',T,oldenergy);
      end
      total = total + itry;
      itry = 1;
      success = 1;
    end
  end
end
newparam = newsol(current);
newenergy = loss(newparam);

if (newenergy < minF),
    parent = newparam;
    oldenergy = newenergy;
    break
end

if (oldenergy - newenergy > 1e-6)
    parent = newparam;
    oldenergy = newenergy;
    success = success+1;
    consec = 0;
else
    if (rand < exp((oldenergy - newenergy)/(k*T)))
        parent = newparam;
        oldenergy = newenergy;
        success = success+1;
    else
        consec = consec+1;
    end
end

minimum = parent;
fval = oldenergy;
if report;
    fprintf(1, '
 Initial temperature: \t%g
', Tinit);
    fprintf(1, ' Final temperature: \t%g
', T);
    fprintf(1, ' Consecutive rejections: \t%i
', consec);
    fprintf(1, ' Number of function calls: \t%i
', total);
    fprintf(1, ' Total final loss: \t%g
', fval);
end

A.10 Discretization of nonequidistant ODE

% Discretization of y''(x) + alpha*y(x) = x, y(0)=1, y(1)=1 for x in [0,1]
% using FDM, on a nonequidistant grid. 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

% Manually implement grid. Example: Equidistant grid
% with 100 subintervals from 0 to 1
grid = 0:0.01:1

% Example grid with dense centre containing 101 subintervals
grid1 = 0:0.09:0.4;  \%Left
grid2 = 0.41:0.002:0.59;  \%Dense centre
grid3 = 0.6:0.09:1;  \%Right
grid = [grid1 grid2 grid3] \% The grid
N = length(grid);
A.11 Discretization of the harder ODE

```matlab
alpha = 1E-4;
beta = 1E-3;
gamma = 3;
L = zeros(N,1);
D = zeros(N,1);
R = zeros(N,1);
b = ones(N,1);

for i = 2:N-1,
    DeltaXL = (grid(i) - grid(i-1));
    DeltaXR = (grid(i+1) - grid(i));
    L(i) = beta*(1/(0.5 * DeltaXL*(DeltaXL+DeltaXR)))-gamma/(DeltaXR+DeltaXL);
    R(i) = beta*(1/(0.5 * DeltaXR*(DeltaXL+DeltaXR)))+gamma/(DeltaXR+DeltaXL);
    D(i) = -(L(i)+R(i)) + alpha;
    b(i) = grid(i);
end
D(1) = 1;
D(N) = 1;
```
A.12 Discretization of the PDE

% Solve: \( k \Delta \phi + U \frac{\partial \phi}{\partial x} + V \frac{\partial \phi}{\partial y} + \alpha \phi(x,y) = x+y, \)
% for \( x,y \) in \([0,1]\)
% boundary conditions: \( \phi(0,y)=\phi(1,y)=1 = \phi(x,0)=\phi(x,1)=1 \)
% using FDM, on an equidistant grid with \( M \times N \) segments

clear all

% physics
alpha = 1E-4;
kdif = 1.0E-4;
Uvel = 5;
Vvel = 3;

% grid: \( M \times N \) segments
M = 16;
N = 16;
dx = 1/(M-1);
dy = 1/(N-1);

% halo grid: boundaries at 2 and \( N+1 \), internal points 3:N
Lx = zeros(M+2,N+2);
Ly = zeros(M+2,N+2);
D = zeros(M+2,N+2);
Rx = zeros(M+2,N+2);
Ry = zeros(M+2,N+2);
b = ones(M+2,N+2);

D(2,:) = 1;
D(M+1,:) = 1;
D(:,2) = 1;
D(:,N+1) = 1;
for j = 3:N,
    for i = 3:M,
        D(i,j) = -2*kdif/(dx*dx) - 2*kdif/(dy*dy) + alpha;
        Lx(i,j) = 1*kdif/(dx*dx);
        Rx(i,j) = 1*kdif/(dx*dx);
        Ly(i,j) = 1*kdif/(dy*dy);
        Ry(i,j) = 1*kdif/(dy*dy);
    end
end

Rx(i,j) = Rx(i,j) + Uvel/(2*dx);
Lx(i,j) = Lx(i,j) - Uvel/(2*dx);
Rx(i,j) = Rx(i,j) + Vvel/(2*dy);
Ly(i,j) = Ly(i,j) - Vvel/(2*dy);
b(i,j) = (i-2)*dx + (j-2)*dy;

end
end
function [it]=gssor2d(Lx, Ly, D, Rx, Ry, b, M, N, x);

% solve Ax=b using SOR method
% input A: matrix , 2D arrays for diagonals
% input b: vector, 2D array for r.h.s.
% input M, N: dimensions of problem
% output x: solution vector, stored in 2D array
% output it: number of iterations

% initial solution
xsol = zeros(M+2, N+2);

w1 = x(1);
w2 = x(2);
w3 = x(3);

% halo grid included

resid = 1E3;
it = 0;

% determine omega’s only once
% omega’s can be added manually
for j = 2 : N+1,
    for i = 2 : M+1,
        if i == 2 || j == 2 || i == M+1 || j == N+1
            wapply(i, j) = w1;
        else
            if 2*floor((i+j)/2) == i+j
                wapply(i, j) = w2;
            else
                wapply(i, j) = w3;
            end
        end
    end
end

while resid > 1E-3,
    % increment iteration counter
    it = it + 1;
    if it > 10000,
        break;
    end

% solution of previous iteration
% xsolold = xsol;

% new solution with Jacobi
    for j = 2 : N+1,
        for i = 2 : M+1,
% solution of previous iteration
xsolold = xsol(i,j);

sum = Lx(i,j)*xsol(i-1,j)+Rx(i,j)*xsol(i+1,j)+
+Ly(i,j)*xsol(i,j-1)+Ry(i,j)*xsol(i,j+1);
xsol(i,j) = (b(i,j)-sum)/D(i,j);

% apply relaxation
xsol(i,j) = wapply(i,j)*xsol(i,j) + (1-wapply(i,j))*xsolold;

% test if solution is a decent number
if isnan(xsol(i,j)) ~= 0 || isinf(abs(xsol(i,j))) ~= 0
    it = 10^-5;
    break
end

% residual resid = ||Ax-b||;
sum=0;
Nmid=floor(N/2);
Mmid=floor(M/2);
% for j =2: N+1,
% for i =2: M+1,
% estimate total residual from mid-grid
for j=Nmid-1: Nmid +1
    for i=Mmid-1: Mmid +1
        axij = Lx(i,j)*xsol(i-1,j)+Rx(i,j)*xsol(i+1,j)+
        +Ly(i,j)*xsol(i,j-1)+Ry(i,j)*xsol(i,j+1);
        axij = axij + D(i,j)*xsol(i,j);
        sum = sum + (axij -b(i,j))^2;
    end
end
% estimate total residual from mid-grid
sum = sum *(N/3) *(M/3);
resid = sqrt(sum);

% test if residual is a decent number
if resid >1.0E100,
    it =10^-6;
    break
end

% %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

% print number of iterations and relative difference
%
%('iteraties: %2.0f rel. error: %12.8 E \n',it, resid);
end