Validation and verification of a third degree optimization method

Anders Levin
Jörgen Johannesson
Abstract
This combined master thesis in Mathematics and in Computer Science deals with a method for finding the local minimum of a unimodal function inside a given interval by using a fifth degree polynomial. This fifth degree polynomial is created from the function value and the first and second derivative values in the end-points of the interval.

In this report the presented method is derived mathematically to converge and it is then proven that the method has a convergence rate of three. Last is the method tested against two reference methods to see the usefulness of the method. To do this some software development methods are described in the report and some test strategies are given. The tests are done with six different functions and with three different implementations of the method.

The conclusions from the tests are that it is often better to use one of the reference methods instead of the presented method, even if the presented method has a better convergence rate, and that the method needs to handle when the found approximation always is on one side of the interval. We could also see from the tests that none of the methods were good on finding a correct approximation. Therefore, there exist needs for more secure methods.

It is therefore suggested in the report that a search for other interpolating functions ought to be carried out in order to improve the method. Also, it could be interesting to test against another method with even higher convergence rate. To do that, another numerical representation is needed and it would be interesting to see if that changes the outcome.

Keywords: Optimization, Convergence rate, Software development

Sammanfattning
Denna kombinerade magisteruppsats i matematik och datalogi handlar om en metod för att hitta ett lokalt minimum för en unimodal funktion inom ett intervall genom användning av ett femtegradspolynom. Femtegradspolynomet skapas med hjälp av interpolation baserad på funktionsvärdena samt första och andra derivatans värden i intervallets ändpunkter.

I rapporten härleds matematiskt att metoden konvergerar, följt av ett bevis för att metoden konvergerar med en konvergenshastighet av tre. Slutfinal testas metoden mot två referensmetoder för att se användningsbarheten. För detta beskrivs vissa mjukvaruutvecklingsmetoder och några teststrategier. Testen utförs med sex olika funktioner och med tre olika versioner av metoden.

Slutsatserna från testen visar att metoden inte är bättre att använda än referensmetoderna även om den har högre konvergenshastighet samt att metoden måste ta hänsyn till när den bara hittar nya approximationer på ena sidan av intervallet. Vi kunde även se från testerna att ingen av metoderna var bra på att ge en korrekt approximation, utan det finns behov av säkrare metoder för detta.

Det är därför föreslaget i uppsatsen att man borde försöka att hitta ett annat interpolations-polynom för att förbättra metoden. Man borde även testa mot en metod som har högre konvergenshastighet. För att kunna göra det behöver man titta på andra sätt att representera numeriska värden och det skulle kunna vara intressant för att se om man då skulle få ett annat resultat.

Nyckelord: Optimering, konvergenshastighet, mjukvaruutveckling
# INTRODUCTION

1.1 BACKGROUND ........................................................................................................ 1
1.2 AIMS ......................................................................................................................... 2
1.3 PROBLEM .................................................................................................................. 2
1.4 REPORT STRUCTURE ............................................................................................... 4

# DESCRIPTION OF THE METHOD

4.1 INTERPOLATION ERROR ........................................................................................... 14
4.2 THE CONVERGENCE ............................................................................................... 15

# PRELIMINARY THEORY

3.1 THE PROBLEM REVISITED ..................................................................................... 7
3.2 DIVIDED DIFFERENCES AND INTERPOLATION POLYNOMIALS ............................... 9

# CONVERGENCE AND RATE OF CONVERGENCE

6.1 TEST STRATEGIES ................................................................................................... 46
6.2 TEST CASES ............................................................................................................. 49
6.3 NUMERICAL METHOD IMPLEMENTATIONS ............................................................ 51
6.4 TEST RESULT .......................................................................................................... 53
6.5 CONCLUSIONS ......................................................................................................... 58
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>SUMMARY AND FINAL CONCLUSIONS</td>
<td>59</td>
</tr>
<tr>
<td>8</td>
<td>ACKNOWLEDGEMENT</td>
<td>61</td>
</tr>
<tr>
<td>9</td>
<td>REFERENCES</td>
<td>62</td>
</tr>
<tr>
<td>9.1</td>
<td>BOOKS</td>
<td>62</td>
</tr>
<tr>
<td>9.2</td>
<td>PAPERS</td>
<td>63</td>
</tr>
<tr>
<td>9.3</td>
<td>INTERNET RESOURCES</td>
<td>63</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background
Finding zeros of equations is an important mathematical problem. There are only a few classes of equations of the form \( f(x) = 0 \) that can be solved exactly. It is for instance possible to solve linear, quadratic, cubic and quartic equations exactly. One of the most used numerical methods to find a root of an equation is Newton-Raphson’s method, which requires that \( f \) is differentiable. This method is usually very efficient. Another method that is somewhat less efficient, but instead it does not use any derivatives of \( f \), is the secant method. These two methods converge with the rates 2 and \((1 + \sqrt{5})/2\) respectively. There are of course a great deal of other numerical methods to solve equations, some of which are presented by Traub (1964), Ostrowski (1966) and Ortega & Rheinboldt (1970). In their different works, they give examples of a number of methods for solving these types of problems.

It is however sometimes desirable to find the optimal solution to a problem. The kind of problems in which an optimal solution is required can stem from engineering or economics, for instance how to select equipment and operating conditions for the production of a given material so that the profit will be at maximum. Another example of an optimization problem is how to design a factory and a parking lot so that the walking distance from the car to the desk is minimized. This problem of finding the minimum or maximum point of functions is closely related to the problem of finding the zeroes of a function. The solution of these optimization problems is linked to the solution of \( f'(x) = 0 \).

There are many methods to solve minimization problems of a function \( f(x) \) of one variable. This type of optimization problems is a special case of multi-variable optimization problems. For an introduction to these problems, see Dennis & Schnabel (1983). Some of the methods are so called hill-climbing techniques. They all use some sort of search procedure. That is, an initial estimation of the solution is made and tested. Depending on the result a search is made to improve the initial estimation. This procedure is repeated until a local optimum is found.

Some methods use derivatives and others do not. Another difference between methods is the number of points that they use. For an example of a method, that minimizes a function of one variable and uses only function, but not derivative, values see Brent (1973) or Mifflin & Strodiot (1993).

Since we can approach this optimization problem in so many different ways, it is natural that the rates of convergence of these methods differ. The most common methods for finding the minimum of a function use low order interpolation polynomials for the sequential fitting of the objective function, that is they use quadratic or cubic polynomials. In this work however we use a polynomial of degree five. This polynomial is used because it has the highest degree when it is still possible to exactly calculate the zeros of the differentiated polynomial. The exact solution is then used as an approximation of the minimum point of the object function.

There are a large number of different analysed methods for optimization. Some of these methods use non-polynomial interpolating functions. Barzilai & Ben-Tal (1982) mention a few such methods. The algorithms they find to be the most interesting use the following interpolating functions:

\[ ax + b + r \log(x - c), \]
\[ ax^2 + bx + c + r \log(x - d), \]

\[ \frac{ax^2 + bx + c}{(dx + 1)^2} \]

and

\[ \frac{ax^2 + bx + c}{dx - 1}. \]

These methods have their own advantages and disadvantages. For the first two, for instance, the coefficients are difficult to calculate, but on the other hand, the minimum points of these equations are not too difficult to calculate. In their work, Barzilai & Ben-Tal have also included numerical examples of different methods minimizing two different functions. These tests make them draw the conclusion that algorithms based on more than two interpolating points are inefficient and that two point algorithms are faster than one-point algorithms. The results of their studies are the basis of our optimization algorithm. We will thus study an algorithm using two interpolation points.

### 1.2 Aims

The aim of this master thesis is to investigate a method that finds the minimum of a function on an interval. The studied function attains its minimum exactly once in the interval and the function is monotonously decreasing before this point and monotonously increasing after. The method uses two points, i.e. the endpoints of an interval, and an interpolation polynomial. To determine the interpolation polynomial we use the function values and the values of the first and second derivatives at the two points. The interpolation polynomial, \( P(x) \), thus is of the fifth degree. For the interpolation polynomial, it is also assumed that the second derivative of \( P(x) \) is not zero at any point in the interval.

It is critical for the method to have a high convergence rate. A high rate of convergence indicates that it will only need a few number of iterations to get a good approximation.

### 1.3 Problem

Therefore, the main questions are:

- Does the method converge with a convergence rate of three in theory?
- Does the method have practical use?

### 1.4 Report structure

We start with a general description of the investigated method in chapter [2].

Then, more details of the method and some preliminary theory are given in chapter [3]. In chapter [4] the convergence rate is discussed and proven for the investigated method. At the end of chapter [4] the first of our problem questions is answered.
In chapter 5 some software development methods are described before the software program is developed. The interfaces of the software system is located in appendix A. Then, in chapter 6 the results from the tests of the method are given. These test results will tell us if the method has a practical use. The test program output and convergence rate calculations are located in appendix B and C.

Last, the final discussion is in chapter 7 followed by an acknowledgement in chapter 8 and the reference list in chapter 9.
2 Description of the method

The algorithm presented in this work determines the point \( \mu \) at which a function \( f(x) \) attains its minimum. From start we know that \( f \) has only one minimum point inside the interval \([\alpha, \beta]\) and no maximum or inflexion points. Let \( P(x) \) be the polynomial of degree five that interpolates \( f(x) \) at the end points \( \alpha \) and \( \beta \) such that

\[ f^{(r)}(\alpha) = P^{(r)}(\alpha) \quad \text{and} \quad f^{(r)}(\beta) = P^{(r)}(\beta) \quad \text{for} \quad r = 0, 1, 2. \]

Denote the interpolation polynomial as

\[ P(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0. \]

To determine an approximate value of \( \mu \), solve the fourth degree equation attained when \( P \) is differentiated. Since we know that the studied function has only one minimum and no other extreme points in the interval \((\alpha, \beta)\) the interpolation polynomial we construct has one or three extreme values in this interval. This is a consequence of how we have constructed the interpolation polynomial, that is

\[ f'(\alpha) = P'(\alpha) < 0 \quad \text{and} \quad f'(\beta) = P'(\beta) > 0. \]

This in turn means that the quartic equation we get when differentiating \( P \) has one or three roots in \((\alpha, \beta)\). In the case there is more than one root found in \((\alpha, \beta)\) we will use a rule to determine which one of them that shall be used. The estimation of \( \mu \) in this work is denoted \( \xi \), that is \( P'(\xi) = 0 \) and \( \xi \in (\alpha, \beta) \).

To determine which side of \( \mu \) the root \( \xi \) is located we calculate \( f'(\xi) \). If \( f'(\xi) > 0 \), we conclude that \( f \) is increasing and thus \( \xi \in (\mu, \beta) \). If on the other hand \( f'(\xi) < 0 \), \( f \) must be decreasing and therefore \( \xi \in (\alpha, \mu) \). With the aid of these observations, we can get a new interval on which to study \( f \). The new interval limits are then determined such that if \( \xi \in (\alpha, \mu) \) the left limit is \( \xi \) and the right is \( \beta \), and if on the other hand \( \xi \in (\mu, \beta) \) the left limit is \( \alpha \) and the right is \( \xi \). These new limits will be used to derive a new interpolation polynomial and a new approximate value of \( \mu \). This is repeated until a sufficiently good estimation \( \xi \) of \( \mu \) has been found.

The equation attained when \( P(x) = a_5x^5 + a_4x^4 + a_3x^3 + a_2x^2 + a_1x + a_0 \) is differentiated can be solved either with the Newton-Raphson method or the known method for finding solutions of a quartic equation (Beyer, 1987). We show how to find the roots of a quartic or biquadratic equation since it is needed in our method. However, we observe that to solve a quartic equation it is also necessary to solve a cubic equation. Therefore, we first study the cubic equation

\[ y^3 + py^2 + qy + r = 0. \]

To begin with make the substitution \( y = x - p/3 \), which yields

\[ x^3 + ax + b = 0, \]

where \( a = \frac{1}{3}(3q - p^2) \) and \( b = \frac{1}{27}(2p^2 - 9pq + 27r) \). To find the solution of the cubic equation put
The solutions are now

\[ x_1 = A + B \]

\[ x_2 = \frac{-A + B}{2} + \frac{A - B}{2}i\sqrt{3} \]

\[ x_3 = \frac{-A + B}{2} - \frac{A - B}{2}i\sqrt{3}. \]

The solutions to the original equation are found by subtracting \( p/3 \) from these solutions. If \( p, q \) and \( r \) are real, the following can be said about the solutions:

If \( \frac{b^2}{4} + \frac{a^3}{27} > 0 \) there is one real root and two complex.

If \( \frac{b^2}{4} + \frac{a^3}{27} = 0 \) there are three real roots and at least two are equal.

If \( \frac{b^2}{4} + \frac{a^3}{27} < 0 \) there are three different real roots.

This solution is known as the Cardan solution after the Milanese doctor Girolamo Cardano. The solution of the quartic equation was found by Ludovico Ferrari. For more on these gentlemen and others see Struik (1987).

To know the solution of the cubic equation is a vital point in solving the quartic equation. Having found this solution, we can thus turn our attention to solve the quartic equation. The quartic equation on the form \( x^4 + ax^3 + bx^2 + cx + d = 0 \) has the following resolvent cubic equation

\[ y^3 + by^2 + (ac - 4d)y - ad + 4bd - c^2 = 0. \]

Let \( y \) be one of the roots to this equation and furthermore put

\[ R = \sqrt{\frac{a^2}{4} - b + y}. \]

We observe that \( R \) could be a complex number and if \( R \neq 0 \) we then put

\[ D = \sqrt{\frac{3a^2}{4} - R^2 - 2b + \frac{4ab - 8c - a^3}{4R}} \]

and
\[ E = \sqrt{\frac{3a^2}{4} - R^2 - \frac{2ab}{4R}} \].

If on the other hand \( R = 0 \) we put

\[ D = \sqrt{\frac{3a^2}{4} - 2b + 2\sqrt{y^2 - 4d}} \]

and

\[ E = \sqrt{\frac{3a^2}{4} - 2b - 2\sqrt{y^2 - 4d}} . \]

The roots to the quartic equation are now

\[
\begin{align*}
x_{1,2} &= -\frac{a}{4} + \frac{R}{2} \pm \frac{D}{2} \\
x_{3,4} &= -\frac{a}{4} - \frac{R}{2} \pm \frac{E}{2}.
\end{align*}
\]

In the next chapter more details of the algorithm are given, especially how the interpolation polynomial is constructed.
3 Preliminary theory

In this chapter, we will state some theorems and definitions concerning interpolation polynomials and divided differences, which later are used to determine an interpolation polynomial of the object function. We will show how to differentiate divided differences. To begin with, however, we give a clearer description of the algorithm, and the functions we study in this work.

3.1 The problem revisited

Here, we define the kind of functions that we are studying. Some of the points made in the previous chapter are also further explained. However, we begin by giving a definition of a unimodal function. These functions play a major part in our continued work.

Definition 3.1

\[ f \text{ is unimodal on } [\alpha, \beta] \text{ if, for all } x_0, x_1 \text{ and } x_2 \in [\alpha, \beta], x_0 < x_1 < x_2 \Rightarrow \]
\[ (f(x_0) \leq f(x_1) \Rightarrow f(x_1) < f(x_2)) \text{ and } (f(x_1) \geq f(x_2) \Rightarrow f(x_0) > f(x_1)) \] (Brent, 1973)

The definition says that \( f \) has a unique minimum in \([\alpha, \beta]\), or more loosely speaking \( f \) cannot have a "hump" between any two points \( x_0 \) and \( x_2 \). Figure 3.1 shows an example of a unimodal function. If we were to study functions, \( f(x) \), which has a maximum point in \([\alpha, \beta]\), we can instead choose to find the minimum point of the function \(-f(x)\).

![Figure 3.1: Unimodal function](image)

The above definition has the advantage that it does not use derivatives of \( f \). That is, we can apply this definition even when considering methods that do not use derivatives.

In the following work, \( f(x) \) is assumed to be unimodal. We also assume that \( f(x) \) is differentiable as many times as necessary. Now we are interested in finding a local minimum of \( f(x) \) in the interval \( \alpha \leq x \leq \beta \) and we have that \( f'(\alpha) < 0 \) and \( f'(\beta) > 0 \).

Let \( \mu \) be the point at which \( f(x) \) attains its minimum value in \([\alpha, \beta]\).

In the iteration method that we are about to analyse, the index \( i \) denotes the \( i \)'th iteration. The interval limits are denoted \( \alpha_i \) and \( \beta_i \). The interpolation polynomial on \([\alpha, \beta]\) is consequently denoted \( P_i(x) \). For practical reasons the interval \((\alpha, \beta)\) is usually scaled to \((0,1)\) to make the calculations easier. When an approximate minimum point \( (t) \) has been found, we then make a transformation back to the \( x \)-axis:

\[ x = t(\beta_i - \alpha_i) + \alpha_i . \]
This re-scaling makes the computation of the coefficients of the interpolation polynomial faster. Thus, the method is speeded up, since each one of the iteration steps profits from this re-scaling. This means that we have found a new minimum point in the interval \([\alpha, \beta]\). To increase the understanding of what this re-scaling means see Figure 3.2.

Letting \( f(t(\beta_i - \alpha_i) + \alpha_i) = \hat{f}(t) \) we construct an interpolation polynomial, \( \hat{P}_i(t) \), of degree five, such that \( \hat{P}_i^{(r)}(0) = \hat{f}^{(r)}(0) \) and \( \hat{P}_i^{(r)}(1) = \hat{f}^{(r)}(1) \) for \( r = 0, 1, 2 \). In doing this we use all the information we presently need for the unimodal function. This will give us an interpolation polynomial that is of the fifth degree. This enables us to calculate the coefficients to the following interpolation polynomial.

\[
\hat{P}_i(t) = a_0 t^5 + a_1 t^4 + a_2 t^3 + a_3 t^2 + a_4 t + a_5. \tag{3.1}
\]

It is here that we see why we did the re-scaling above. The coefficients \( a_0, a_1, \) and \( a_2 \) are easily calculated, and \( a_3, a_4 \) and \( a_5 \) can be determined without too much work. We can see that the coefficients will change with each iteration. The dependence of \( i \) in \( \hat{f} \) is suppressed. Thus, we get the following expression for the coefficients

\[
\begin{aligned}
a_0 &= \hat{f}(0) \\
a_1 &= \hat{f}'(0) \\
a_2 &= \frac{1}{2} \hat{f}''(0) \\
a_3 &= 10 \hat{f}(1) - 4 \hat{f}''(1) + \frac{1}{2} \hat{f}'''(1) - 10 \hat{f}(0) - 6 \hat{f}'(0) - \frac{5}{2} \hat{f}''(0) \\
a_4 &= -15 \hat{f}(1) + 7 \hat{f}''(1) - \hat{f}'''(1) + 15 \hat{f}(0) + 8 \hat{f}'(0) + \frac{5}{2} \hat{f}''(0) \\
a_5 &= 6 \hat{f}(1) - 3 \hat{f}'(1) + \frac{1}{2} \hat{f}''(1) - 6 \hat{f}(0) - 3 \hat{f}'(0) - \frac{5}{2} \hat{f}''(0).
\end{aligned} \tag{3.2}
\]

In order to find the minimum of \( f \), we use the interpolation polynomial and locate the minimum of it. Thus we consider \( \hat{S}_i(t) = \hat{P}_i'(t) \) and find the roots to this function. \( \hat{S}_i(t) \) is a polynomial of degree four and it is possible to find an exact solution to the equation \( \hat{S}_i(t) = 0 \). If \( \hat{S}_i(t) \) has more than one root within the interval \((0,1)\), then we use a rule to select one of them roots, and we call this root \( t_{i+1} \). We thus find a new endpoint to our interval at \( x_{i+1} = (\alpha_{i+1}, \beta_{i+1}) \) where \( \hat{S}_i(t_{i+1}) = 0 \) and \( x_{i+1} = t_{i+1}(\beta_i - \alpha_i) + \alpha_i \).

The endpoint that is on the same side of \( \mu \) as \( x_{i+1} \) is replaced by \( x_{i+1} \). To check this we calculate \( f'(x_{i+1}) \). If the derivative \( f'(x_{i+1}) > 0 \) we know that \( x_{i+1} \) is to the right of \( \mu \), since \( f \) is unimodal, on the other hand if \( f'(x_{i+1}) < 0 \), \( x_{i+1} \) is to the left of \( \mu \). We have:
\[
\alpha_{i+1} = \begin{cases} 
\alpha_i & \text{if } x_{i+1} > \mu \\
\chi_{i+1} & \text{if } x_{i+1} < \mu 
\end{cases}
\]

and

\[
\beta_{i+1} = \begin{cases} 
\chi_{i+1} & \text{if } x_{i+1} > \mu \\
\beta_i & \text{if } x_{i+1} < \mu 
\end{cases}
\]

where \( \alpha_0 \) and \( \beta_0 \) are the starting points.

### 3.2 Divided differences and interpolation polynomials

To enable us to continue, we first have to study other ways of deriving interpolation polynomials. To begin with, we look at Lagrange’s formula. A definition of divided differences is also stated and we will see how these can be used to construct interpolation polynomials.

From now on we denote \( \text{int}(a, b, c, \ldots) \) the smallest interval containing the real numbers \( a, b, c, \ldots \). When interpolating, the numbers \( a, b, c, \ldots \) are called node points or interpolating points.

All this will be used in the next chapter to find an expression for the error between the approximated point \( x_{i+1} \) and the optimal point \( \mu \).

**Theorem 3.1**

Let \( x_0, x_1, \ldots, x_n \) be distinct real numbers, and let \( f \) be a given real valued function with \( n+1 \) continuous derivatives on the interval \( I_x = \text{int}(x, x_0, x_1, \ldots, x_n) \) where \( x \) is some given real number. Also let \( p_n \) be the interpolation polynomial of degree \( n \), such that \( p_n(x_i) = f(x_i) \) for \( i = 0, 1, \ldots, n \). Then there exists a \( \xi \in I_x \) with

\[
f(x) - p_n(x) = f(x) - \sum_{j=0}^{n} f(x_j) l_j(x) = \frac{(x-x_0)\ldots(x-x_n)}{(n+1)!} f^{(n+1)}(\xi) \tag{3.3}
\]

where \( l_i(x) = \prod_{j \neq i} \frac{x-x_j}{x_i-x_j} \) for \( i = 0, 1, \ldots, n \).

A proof of this theorem is presented by Atkinson (1989).

Before we continue to study interpolating polynomials, we give the definition of divided differences.

**Definition 3.2**

The divided difference (of the order \( n \)) of the function \( f(x) \) is given by

\[
f[x_i] = f(x_i)
\]

\[
f[x_0, \ldots, x_n] = \frac{f[x_1, \ldots, x_n] - f[x_0, \ldots, x_{n-1}]}{x_n - x_0}.
\]

The following theorem is useful when studying interpolation polynomials.
Theorem 3.2
The interpolation problem of determining an \( n \)th degree polynomial which agrees with the values of a given function on a set of \( n+1 \) points always has a unique solution which can be expressed in the form

\[
f(x) = c_0 + c_1 (x-x_0) + \ldots + c_n (x-x_0) \ldots (x-x_{n-1}).
\]

A proof of this theorem can be found in Björck & Dahlquist (1974). From theorem 3.2 we have

\[
f(x) = c_0 + c_1 (x-x_0) + \ldots + c_n (x-x_0) \ldots (x-x_{n-1}) + C(x)(x-x_0) \ldots (x-x_n),
\]

where according to equation (3.3)

\[
C(x) = \frac{f^{(n+1)}(\xi)}{(n+1)!}.
\]

Note that \( C(x) \) is bounded when \( f^{(n+1)}(x) \) is continuous on \( \text{int}(x_0,\ldots,x_n,x) \). For \( x = x_0 \) we get \( f(x_0) = c_0 \). Since \( f[x_0,x]=\left(f(x) - f(x_0)\right)/(x-x_0) \), we get the following expression

\[
f[x_0,x] = c_1 + c_2 (x-x_1) + \ldots + c_n (x-x_1) \ldots (x-x_{n-1}) + C(x)(x-x_1) \ldots (x-x_n).
\]

We now see that \( f[x_0,x_1] = c_1 \). Continuing this way we get \( c_k = f[x_0,\ldots,x_{k-1}] \) for all \( k = 0,1,\ldots,n \).

In our continued work, we want to study the error formula in terms of divided differences. We know that our interpolation polynomial can be written

\[
p_n(x) = c_0 + c_1 (x-x_0) + \ldots + c_n (x-x_0) \ldots (x-x_{n-1}).
\]

Let \( s \) be a real number distinct from the node points \( x_0, x_1, \ldots, x_n \). We now construct an interpolation polynomial \( p_{n+1} \), of degree \( n+1 \), such that \( p_{n+1}(x_i) = f(x_i) \) for \( i = 0,1,\ldots,n \) and \( p_{n+1}(s) = f(s) \). We get

\[
p_{n+1}(s) = p_n(s) + (s-x_0) \ldots (s-x_n) f[x_0,\ldots,x_n,s],
\]

where \( p_n(s) \) is the \( n \)th degree polynomial interpolating \( f \) at \( x_0, x_1, \ldots, x_n \) and \( s \), and where \( (s-x_0) \ldots (s-x_n) f[x_0, x_1, \ldots, x_n, s] \) only depends on the values \( f \) attains at the points \( x_0, x_1, \ldots, x_n \) and \( s \). Since \( p_{n+1}(s) = f(s) \), let \( x = s \) and move \( p_n(x) \) to the left hand side of the equation to obtain

\[
f(x) - p_n(x) = g_n(x) f[x_0,\ldots,x_n,x],
\]

where \( g_n(x) = (x-x_0) \ldots (x-x_n) \). Comparing this with equation (3.3) we see that

\[
f[x_0,\ldots,x_n,x] = \frac{f^{(n+1)}(\xi)}{(n+1)!}, \tag{3.4}
\]
for some $\xi \in \text{int}(x_0, x_1, \ldots, x_n, x)$. This relation will be used when we study the problem of how to differentiate a divided difference.

The following theorem extends the definition of divided differences to the case in which some or all of the nodes coincide.

**Theorem 3.3 (Hermite-Gennochi)**

Let $x_0, \ldots, x_n$ be distinct, and let $f(x)$ be $n$-times continuously differentiable on the interval $\text{int}(x_0, \ldots, x_n)$. Then

$$f[x_0, \ldots, x_n] = \int_{x_0}^{x_1} \cdots \int_{x_{n-1}}^{x_n} f^{(n)}(y_0 x_0 + \cdots + y_n x_n) dy_1 \cdots dy_n$$

where the integration is carried out over the $n$-dimensional region

$$\tau_n = \left\{ (y_1, \ldots, y_n) : \text{all } y_i \geq 0, \sum_{i=0}^{n} y_i \leq 1 \right\},$$

and $y_0$ is given by

$$y_0 = 1 - \sum_{i=0}^{n} y_i.$$

For a proof of this theorem, see Atkinson (1989).

In the following, we study how to differentiate a divided difference. This will be very useful later in our work, when we wish to find a recursive expression for the errors of our method. For more details on this see Kelly (1967). We may write

$$f[x_0, x] = f[x, x_0] = \frac{f[x_0] - f[x]}{x_0 - x}.$$

Letting $x_0 = x + \epsilon$ we have $f[x + \epsilon, x] = (f[x + \epsilon] - f[x]) / \epsilon$ and taking the limit as $\epsilon$ approaches zero gives us $f'[x] = f[x, x]$. Therefore more generally

$$f'[c, x] = f[c, x, x]. \quad (3.5)$$

Also with $c = x_0, \ldots, x_n$ we have $f'[x_0, \ldots, x_n, x] = f[x_0, \ldots, x_n, x, x]$. Likewise if $u_1, \ldots, u_n$ are functions of $x$,

$$f'[c, u_1, \ldots, u_n] = \sum_{i=1}^{n} f[c, u_1, \ldots, u_n, u_i] \frac{du_i}{dx}.$$

It follows that if $u_i = x$ then we have the relation

$$f'[c, x, \ldots, x] = n f[c, x, \ldots, x]. \quad (3.6)$$
Also from equation (3.5) and equation (3.6) we have the following expression for the \(r\)th derivative of a divided difference
\[
f^{(r)}[c, x] = r! f[c, x, \ldots, x]_{r+1\text{ times}}
\] (3.7)

Now we turn our attention to the error function \(R_r(x) = f(x) - p_n(x) = g_n(x)f[x_0, \ldots, x_n, x]\), where \(g_n(x) = (x - x_0)\ldots(x - x_n)\), without considering any rescaling. We wish to find a way to calculate
\[
R^{(r)}_r(x) = \frac{d^r}{dx^r} \left[ g_n(x)f[x_0, \ldots, x_n, x] \right]
\] (3.8)

We write the \(r\)th derivative of a product as, with \(D = d/dr\),
\[
D^r(uv) = uD^r v + rDvD^{r-1}v + \ldots + vD^r u = \sum_{i=0}^r \binom{r}{i} D^i u D^{r-i} v.
\]
The error function (3.8) therefore takes the form
\[
R^{(r)}_r(x) = \sum_{i=0}^r \binom{r}{i} g^{(i)}_n(x) \frac{d^{r-i}}{dx^{r-i}} f[x_0, \ldots, x_n, x]
\] (3.9)
and if we then use expression (3.7) for the \(r\)th derivative of a divided difference we get
\[
R^{(r)}_r(x) = \sum_{i=0}^r \binom{r}{i} g^{(i)}_n(x) f[x_0, \ldots, x_n, x]_{r-i+1\text{ times}}
\] (3.10)

In order to find an expression for the error without divided differences we need to use Rolle’s theorem and for clarity we state this theorem.

**Theorem 3.3 (Rolle)**

Let \(f\) be differentiable in \((a,b)\) and continuous in \([a,b]\). If \(f(a) = f(b)\) then there exists at least one point \(\xi\) in \((a,b)\) such that \(f’(\xi) = 0\).

A proof of this basic theorem can be found in any standard textbook in calculus, such as Hellström et al (1991).

We know from earlier that \(R_r(x) = f(x) - p_n(x)\). We differentiate this expression \(r\) times and form a function \(G(x)\) such that
\[
G(x) = f^{(r)}(x) - p^{(r)}_n(x) - R^{(r)}_r(x).
\]
The function \(G(x)\) has zeroes at the \(n + 2\) points \(x_0, \ldots, x_n, x\). According to Rolle’s theorem \(G’(x)\) has \(n + 1\) zeroes in the interval \(\text{int}(x_0, \ldots, x_n, x)\). Continuing like this we find that \(G^{(n+1)}(x) = 0\) at one point in the interval \(\text{int}(x_0, \ldots, x_n, x)\). Differentiating \(G(x)\) \(n + 1\) times using equation (3.10) yields
\[ G^{(n+1)}(x) = f^{(n+r+1)}(x) - ((n+r+1)! g_n(x) f[x_0, \ldots, x_n, x, \ldots, x] + \text{\( n+r+2 \) times}) \]

\[ (n+r+1)! g_n'(x) f[x_0, \ldots, x_n, x, \ldots, x] + \text{\( n+r+1 \) times} \]

\[ (n+r+1)! \frac{(n+1)! f[x_0, \ldots, x_n, x, \ldots, x]}{r+1 \text{ times}}. \]

From Rolle’s theorem we have that there exist a \( \xi \in \text{int}(x_0, \ldots, x_n, x) \) such that

\[ G^{(n+1)}(\xi) = f^{(n+r+1)}(\xi) - (n+r+1)! f[x_0, \ldots, x_n, x, \ldots, x] = 0. \]

Rewriting this expression, we find that

\[ f[x_0, \ldots, x_n, x, \ldots, x] = \frac{f^{(n+r+1)}(\xi)}{(n+r+1)!}. \]

Hence, we can rewrite (3.9) and get an expression for the error without divided differences

\[ R^{(r)}_T(x) = \sum_{i=0}^{r} \frac{r!}{(n+r-i+1)!} g_n^{(i)}(x) f^{(n+r-i+1)}(\xi_i). \]

The above formula is in most cases not particularly nice to write in an expanded form, but in the case \( r = 1 \) we have

\[ R^{(r)}_T(x) = g'_n(x) \frac{f^{(n+1)}(\xi)}{(n+1)!} + g_n(x) \frac{f^{(n+2)}(\xi_0)}{(n+2)!}. \] (3.11)

We have in this chapter given some more details about the interpolation polynomial we use in this work. The method itself has also been further explained. Finally some theory concerning interpolation polynomials and divided differences has been given.
4 Convergence and rate of convergence

In this chapter, we study the convergence and the rate of convergence of our method. To begin with, we give an expression for the interpolation error. This expression is then used to find a recurrence formula for the error between the optimal point and the approximated point. By using the recurrence formula it will be possible to show that the method converges and at what rate.

4.1 Interpolation error

To begin with, we study the error between our interpolation polynomial and the object function. This will be very useful in our continued work. The dependence of \( i \) in \( P_i \) will in the following theorems be suppressed. In the more common case of interpolation, where each point is only considered once, the error has a somewhat different expression. However, the proof of our theorem is based on the proof of the more common case. For a proof of this error, see Johansson (1995). Our proof use theorem 3.3

**Theorem 4.1**

Let \( f \) be at least six times differentiable. In addition, let \( P \) be an interpolation polynomial on the interval \([\alpha, \beta]\), such that \( f^{(r)}(\alpha) = P^{(r)}(\alpha) \) and \( f^{(r)}(\beta) = P^{(r)}(\beta) \) for \( r = 0, 1, 2 \). The error \( R_f(x) = f(x) - P(x) \) has the following form:

\[
 f(x) - P(x) = \frac{f^{(6)}(\xi)}{6!} w(x)
\]

where \( w(x) = (x - \alpha)^3(x - \beta)^3 \).

**Proof**

First we construct a function \( F \) such that \( F(x) = f(x) - P(x) - K w(x) \), where \( f(x) \) is the function we are interpolating, \( P(x) \) is the interpolation polynomial and \( K \) is a constant.

Take an arbitrary \( \eta \in (\alpha, \beta) \). It is now possible to construct \( K \) such that \( F(\eta) = 0 \). According to Rolle’s theorem there are numbers \( \xi_{11} \in (\alpha, \eta) \) and \( \xi_{12} \in (\eta, \beta) \) such that \( F'(\xi_{11}) = F'(\xi_{12}) = 0 \). Since we also have \( F'(\alpha) = F'(\beta) = 0 \) there exist numbers \( \xi_{2i} \in (\xi_{1i}, \xi_{1i+1}) \) such that \( F^{(i)}(\xi_{2i}) = 0 \) for \( i = 0, 1, 2 \) where \( \xi_{10} = \alpha \) and \( \xi_{13} = \beta \). Furthermore \( F^{(i)}(\alpha) = F^{(i)}(\beta) = 0 \) so there are numbers \( \xi_{3i} \in (\xi_{2i}, \xi_{2i+1}) \) such that \( \hat{F}^{(i)}(\xi_{3i}) = 0 \) for \( i = 0, 1, 2, 3 \) where \( \xi_{20} = \alpha \) and \( \xi_{24} = \beta \). This implies that there are \( \xi_{4i} \in (\xi_{3i}, \xi_{3i+1}) \) such that \( F^{(i)}(\xi_{4i}) = 0 \) for \( i = 0, 1, 2 \) and therefore there exist numbers \( \xi_{5i} \in (\xi_{4i}, \xi_{4i+1}) \) such that \( F^{(i)}(\xi_{5i}) = 0 \) for \( i = 0, 1 \). Finally we can find a \( \xi \in (\xi_{50}, \xi_{51}) \) such that \( F^{(6)}(\xi) = 0 \). Thus, we have

\[
 F^{(6)}(\xi) = f^{(6)}(\xi) - K 6! = 0,
\]

that is
Theorem 4.1 is proved.

Now we are in a position where we can study the convergence of the method and the rate of convergence. This will be done in the following section.

4.2 The convergence

The goal of the following section is to show that our method converges with the third rate of convergence. The dependence of $i$ in $P_i$ will no longer be suppressed. We also assume that the interpolation polynomial $P_i(x)$ has the two following properties. First $P_i'(\alpha_i)$ and $P_i'(\beta_i)$ are non-zero, second $|P_i'(x)| \leq K < \infty$ for all $x \in (\alpha_i, \beta_i)$ for some positive constant $K$. The basic ideas behind the following theorems come from Barzilai & Ben-Tal (1982). We start by defining the rate of convergence.

**Definition 4.1**

Let $x_0, x_1, x_2, \ldots$ be a sequence converging to $\mu$, and let $\varepsilon_n = x_n - \mu$. The rate of convergence is defined as the supremum of the nonnegative numbers $p$ satisfying

$$0 \leq \lim_{n \to \infty} \left| \frac{\varepsilon_{n+1}}{\varepsilon_n^p} \right| = C < \infty.$$  

Barzilai & Ben-Tal (1982) use the largest number instead of supremum. However, this is not correct to use for this kind of problem. If we were to use their definition, we would later have difficulties in proving the rate of convergence. Tamir (1979) use the above definition of convergence rate. To begin with, we show that the errors of our method satisfy a recurrence formula. In the proof of the following theorem, we use the mean value theorem, and for clarity, we therefore state this theorem here.

**Theorem 4.2** (Mean Value)

Let $f(x)$ be continuous in $[a,b]$, and let it be differentiable in $(a,b)$. Then there is at least one point $\xi$ in $(a,b)$ for which $f(b) - f(a) = f'(\xi)(b-a)$.

A proof of this basic theorem can be found in any standard textbook in calculus such as Hellström et al (1991).

**Theorem 4.3**

Let $\mu$ be the minimum point of a unimodal function $f(x)$ and let $P_i(x)$ be the fifth degree polynomial such that $P_i^{(k)}(\alpha_i) = f^{(k)}(\alpha_i)$ and $P_i^{(k)}(\beta_i) = f^{(k)}(\beta_i)$ where $k = 0, 1, 2$ and $\alpha_i$ and $\beta_i$ are the two endpoints of our interval at iteration step $i$. Also let $w_i(x) = (x - \alpha_i)^3 (x - \beta_i)^3$ be the weight function defined for the interval $(\alpha_i, \beta_i)$. Moreover let $x_0 = \alpha_0$ and $x_1 = \beta_0$ and let $x_{i+1}$ be the minimum point of $P_i(x)$ in $(\alpha_i, \beta_i)$. If $|P_i''| \neq 0$ on $(\alpha_0, \beta_0)$ then the errors, $\varepsilon_i = x_i - \mu$ satisfy the recursion formula

$$\varepsilon_{i+1} = M_i (\varepsilon_{i-1}^2 \varepsilon_i^3 + \varepsilon_{i-1}^3 \varepsilon_i^2) + N_i \varepsilon_{i-1}^3 \varepsilon_i^3$$  

(4.2)
where \( M_i = -\frac{3Y(\xi(\mu))}{P''_i(\theta(x_{i+1}))} \) and \( N_j = \frac{Z(\eta(\mu))}{P''_j(\theta(x_{j+1}))} \), and

\[
Y(x) = \frac{f^{(6)}(x)}{6!}, \quad Z(x) = \frac{f^{(7)}(x)}{7!}
\]

and where \( \alpha_{i+1} = \alpha_i \) and \( \beta_{i+1} = x_{i+1} \) or \( \alpha_{i+1} = x_{i+1} \) and \( \beta_{i+1} = \beta_i \), \( \xi(x), \eta(x) \in \text{int}(x_0, \ldots, x_n, x) \) and \( \theta(x_{i+1}) \in \text{int}(\mu, x_{i+1}) \).

**Proof**

Differentiating equation (4.1) and using formula (3.11) we get

\[
f'(x) = P'_i(x) + Y(\xi_i(x))w'(x) + Z(\eta_i(x))w(x),
\]

(4.3)

where \( Y \) and \( Z \) are defined as above. Now put \( x = \mu \) in (4.3) and use the fact that

\[
P'_i(\mu) = P'_i(\mu) - P'_i(x_{i+1}) = -\varepsilon_{i+1} P''_i(\theta(x_{i+1})).
\]

Hence

\[
0 = f'(\mu) = -\varepsilon_{i+1} P''_i(\theta(x_{i+1}))
\]

\[
+ 3Y(\xi_i(\mu))(\mu - x_{i+1})^2(\mu - x_i)^3 + (\mu - x_{i+1})^3(\mu - x_i)^3
\]

\[
+ Z(\eta_i(\mu))(\mu - x_{i+1})^3(\mu - x_i)^3.
\]

Letting \( M_i \) and \( N_i \) be as stated above, we have now proved the theorem.

The above theorem will be used when proving the convergence of the method. To begin with, we study a special case for the convergence and therefore make the following definition.

**Definition 4.2**

An optimization process where the approximated values tend to the optimum point from only one side is called a one-sided optimization process (OSOP).

We want to know if our method converges, that is, does the sequence \( \{x_i\} \) converge to \( \mu \). Thus, we want to know if the recursion formula of the error found in theorem 4.3 tends to zero when the number of iterations tend to infinity. To begin with, we prove the convergence of the method if it is an OSOP. Later in this chapter, we will give a more general proof of the convergence. To prove that the method converges we assume that the OSOP generates values to the left of \( \mu \). This means that we get a monotonous increasing sequence. This assumption is made for simplicity and the case where the OSOP generates values to the right of \( \mu \) is treated similarly, but then the sequence is monotonous decreasing. To begin with, we state the following theorem.

**Theorem 4.4**

If a sequence \( a_n, n = 1, 2, 3, \ldots \), is bounded and monotonous for \( n > n_0 \) then \( \lim_{n \to \infty} a_n \) exists.

A proof of this theorem can be found in Hyltén-Cavallius & Sandgren (1966). We can now prove the following theorem.
Theorem 4.5

The monotonous sequence \( \{\alpha_i\}_{i=0}^{\infty} \) that is generated by the OSOP converges to \( \mu \).

Proof

According to theorem 4.4 we know that the limit exists, and we assume that \( \lim_{i \to \infty} \alpha_i = \rho \) where \( \rho \neq \mu \). The case where \( \rho > \mu \) is not possible since \( \mu \) is an upper bound for \( \{\alpha_i\}_{i=0}^{\infty} \). Thus, we can assume that \( \rho \leq \mu \).

If \( \rho < \mu \) we have that \( \mu - \rho = \alpha > 0 \). We assume that for each \( \epsilon > 0 \) exists a positive number \( N \) such that for \( i > N \), \( \rho - \epsilon \leq \alpha_i \leq \rho < \mu \), is true. We also assume that \( |P_i''(x)| \leq K < \infty \) for all \( x \in (\alpha_i, \beta_i) \) for some positive constant \( K \). Theorem 4.2 now yields

\[
P_i'(\alpha_{i+1}) - P_i'(\alpha_i) = P_i''(\xi)(\alpha_{i+1} - \alpha_i),
\]

for some \( \xi \in (\alpha_i, \alpha_{i+1}) \). Since \( P_i'(\alpha_{i+1}) = 0 \) and \( P_i'(\alpha_i) = f'(\alpha_i) \) for \( i > N \) we have

\[
\frac{|f'(\alpha_i)|}{K} < (\alpha_{i+1} - \alpha_i) < \epsilon,
\]

(4.4)

where the last inequality is a consequence of that \( \alpha_i \) and \( \alpha_{i+1} \in [\rho - \epsilon, \rho] \). Since \( f \) is unimodal with minimum at the point \( x = \mu \) and since \( f' \) is continuous on \((\alpha_0, \beta_0)\) there exists an \( \epsilon_i > 0 \) such that \( |f'(x)| > \epsilon_i \) when \( \rho - \epsilon \leq \alpha_i \leq \rho \). But according to (4.4) we must have \( |f'(\alpha_i)| < K\epsilon \) and choosing \( \epsilon = \epsilon_i / (2K) \) we have a contradiction. This means that we have \( \rho \geq \mu \) and therefore \( \rho = \mu \), which shows that the OSOP converges to \( \mu \).

It has now been shown that the method converges if it is an OSOP. For a more general proof of the convergence, the assumption of an OSOP will no longer be necessary. We also remind ourselves that the sequence \( \{x_n\} \) contains the chosen zeros \( x_{i+1} \) of \( P_i(x) \), that lies in the interval \((\alpha, \beta)\). To prove the convergence the following definition is useful.

Definition 4.3

\( l \) is a cluster point of \( \{x_n\}_{n=0}^{\infty} \) if, given \( \epsilon > 0 \) and given \( N \), \( \exists n > N \) such that

\[
|x_n - l| < \epsilon.
\]

(Royden, 1988)

It will also be necessary to use the following theorems, and proofs can be found in Hyltén-Cavallius & Sandgren (1968).

Theorem 4.6

A sequence is convergent if and only if it is bounded and it has only one cluster point.

The theorem says that if a sequence has a limit \( l \), then \( l \) is a cluster point, but the converse is usually not true.
Theorem 4.7
From every infinite bounded sequence, it is possible to pick a convergent subsequence.

The convergence of the method without the assumption of an OSOP can now be shown with the aid of the two theorems above.

Theorem 4.8
The sequence \( \{x_i\} \) generated as above converges to \( \mu \).

Proof
The interval \([\alpha_0, \beta_0]\) is compact and we have that \([\alpha_0, \beta_0] \supseteq [\alpha_i, \beta_i] \supseteq \ldots\), i.e. \( \{x_i\} \) is a sequence on a compact interval. It is thus possible to pick a convergent subsequence from \( \{x_i\} \). Let \( \{\delta_i\} \) be the elements of \( \{x_i\} \) that lies in the interval \([\alpha_0, \mu]\) and let \( \{\phi_i\} \) be the elements in the interval \([\mu, \beta_0]\). The sequence \( \{\delta_i\} \) is monotonously increasing and \( \{\phi_i\} \) is monotonously decreasing. These sequences converge, according to theorem 4.5, since they are monotonous and bounded. Assume that \( \lim_{i \to \infty} \delta_i = \delta \) and \( \lim_{i \to \infty} \phi_i = \phi \). This means that there are four different cases which can occur. These are:

I. \( \delta < \mu < \phi \)
II. \( \delta < \mu = \phi \)
III. \( \delta = \mu < \phi \)
IV. \( \delta = \mu = \phi \).

Case I: Let \( \varepsilon > 0 \) and assume that \( N \) is sufficiently large so that \( \delta_i \in I_i \) and \( \phi_i \in J_i \) for \( i > N \) where \( I_i = (\delta - \varepsilon, \delta + \varepsilon) \) and \( J_i = (\phi - \varepsilon, \phi + \varepsilon) \). Let \( x_{i+1} \) be the chosen zero of \( P_i'(x) = 0 \) on the interval \([\alpha_0, \beta_0]\). We have \( |P_i''(x)| \leq K \) on the interval \([\alpha_0, \beta_0]\), where \( K \) is a positive constant. If we let \( K' \) be another positive constant, then

\[
0 < K' \leq |P_i'(\phi + \varepsilon)| = |P_i'(\phi + \varepsilon) - P_i'(x_{i+1})| = (\phi + \varepsilon - x_{i+1})K,
\]

where \( \xi \in (x_{i+1}, \phi + \varepsilon) \). Now pick \( \varepsilon < K' / (2K) \). This yields \( x_{i+1} < \phi - \varepsilon \). Thus \( x_{i+1} \) is outside the interval \( J_i \) and must therefore be in the interval \( I_i \), since the sequences \( \{\delta_i\} \) and \( \{\phi_i\} \) together contain all the points of \( \{x_i\} \). We have that \( |P_i'(\alpha_i)| > k' > 0 \) and \( k < |P_i''(x)| < k'' < \infty \) on \([\alpha_0, \beta_0]\), where \( k, k' \) and \( k'' \) are positive constants. We now have

\[
k' < |P_i'(\alpha_i)| = |P_i'(x_{i+1}) - P_i'(\alpha_i)| = (x_{i+1} - \alpha_i)P_i''(\eta) < k''(x_{i+1} - \alpha_i)
\]

where \( \eta \in I_i = (\delta - \varepsilon, \delta + \varepsilon) \). Since also \( x_{i+1} \) and \( \alpha_i \in I_i \) it is possible to do yet another estimation

\[
k''(x_{i+1} - \alpha_i) < k''(\delta + \varepsilon - \delta + \varepsilon) = k''2\varepsilon.
\]

This means that \( 2\varepsilon > k'/ k'' \). But, since it is possible for us to pick \( \varepsilon \) in the interval \( 0 < \varepsilon < K' / (2K) \) we get a contradiction. So the assumption that \( \delta < \mu < \phi \) is false.
Case II: Now assume $\delta < \mu = \varphi$. Just like above we let $\delta_i \in I_i$ and $\varphi_i \in J_i$ for $i > N$. We also have that $|P_i'(\alpha_i)| > k' > 0$ and $k < |P_i''(x)| < k'' < \infty$ where $k$, $k'$ and $k''$ are positive constants. If $x_{i+1} \in I_i$, we get as above for each $\varepsilon$, $0 < \varepsilon < K'/2K$:

$$k' < |P_i'(x_{i+1}) - P_i'(\alpha_i)| < k''2\varepsilon.$$  

Just like above it is possible to get a contradiction by choosing a sufficiently small $\varepsilon$. This means that $x_{i+1} \not\in I_i$ and therefore $\delta$ cannot be a cluster point. The assumption that $\delta < \mu = \varphi$ is false.

Case III: This case is shown false in a similar way as in case II.  

Case IV: Since the three previous assumptions all are false, the only case that remains must be true. Thus, the only possibility remaining is $\delta = \mu = \varphi$. We have now shown that the sequence $\{x_i\}$ converges to $\mu$.

Above we have shown that our interpolation algorithm converges to $\mu$. Now we want to examine with what rate the algorithm converges. To do this we continue to study (4.2). However, the way in which (4.2) presently looks it is no good to us, it needs to be re-written. After this re-writing, we will in theorem 4.10 show that our algorithm has a convergence rate of three. To begin with, we replace (4.2) by a more useful difference equation.

Theorem 4.9

Let $f$ be a unimodal function with minimum at $\mu$ and let $P_i$ be the fifth degree polynomial interpolating $f$ with minimum at $x_{i+1}$. Furthermore let $M_i$ and $N_i$ be defined as in theorem 4.3. If $|P_i''| > K$ on $[\alpha_i, \beta_i]$ and if $f^{(6)}(x)$ and $f^{(7)}(x)$ are continuous and if $M_i \to M \neq 0$ when $i \to \infty$ then with $\varepsilon_i = x_i - \mu$,

$$\varepsilon_{i+1} = A_i\varepsilon_i^2\varepsilon_{i-1}^{\frac{3}{2}}, \quad (4.5)$$

where

$$A_i = M_i\left[1 + \frac{\varepsilon_i}{\varepsilon_{i-1}}\right] + N_i\varepsilon_i.$$  

Furthermore $A_i \to M$.

Proof

Since $|P_i''| > K$ and $f^{(6)}(x)$ and $f^{(7)}(x)$ are continuous the sequences $M_i$ and $N_i$ are bounded. Equation (4.2) now gives

$$\frac{\varepsilon_{i+1}}{\varepsilon_i} = M_i\left(\varepsilon_{i-1}^2\varepsilon_i^2 + \varepsilon_{i-1}^3\varepsilon_i\right) + N_i\varepsilon_{i-1}^3\varepsilon_i^2 \to 0, \quad (4.6)$$

when $i \to \infty$, since the sequence $\{x_i\}$ converges to $\mu$. Rewriting (4.2) in the form
\[ \varepsilon_{i+1} = \varepsilon_i^2 \varepsilon_{i-1}^3 \left[ M_i + M_i \frac{\varepsilon_i}{\varepsilon_{i-1}} + N_i \varepsilon_i \right], \]

we see by (4.6) that (4.5) holds with

\[ A_i = M_i \left[ 1 + \frac{\varepsilon_i}{\varepsilon_{i-1}} \right] + N_i \varepsilon_i, \]

and it follows that \( A_i \to M \). Thus theorem 4.9 is proved.

We are now in a position where we are able to state and prove the following theorem about the rate of convergence of our algorithm.

**Theorem 4.10**

Let \( f \) be a unimodal function with minimum at \( \mu \in (\alpha_0, \beta_0) \) and let \( P_i \) be the fifth degree polynomial interpolating \( f \) with minimum at \( x_{i+1} \). If \( |P^*| > K > 0 \) on \( [\alpha_0, \beta_0] \) and if \( f^{(6)}(x) \) and \( f^{(7)}(x) \) are continuous and if \( M_i \to M \) when \( i \to \infty \), the sequence \( \{x_i\} \) generated by our interpolation algorithm converges to the solution \( \mu \) with the rate of convergence equals three.

**Proof**

We start by re-scaling (4.5) by letting \( \delta_i = \varepsilon_i / a \) and \( B_i = A_i / a^4 \), where \( a = \beta_0 - \alpha_0 \). Taking logarithms in this re-scaled version of (4.5) yields with \( y_i = \log |\delta_i| \) and \( C_i = \log |B_i| \)

\[ y_{i+1} - 2y_i - 3y_{i-1} = C_i, \quad i = 1, 2, \ldots \tag{4.7} \]

and we assume that \( M \neq 0 \). The case when \( M_i \to 0 \) will not be studied here. It is clear that that the method will converge faster if \( M_i \to 0 \). The expression \( \log |B_i| \) does not exist if \( B_i = 0 \) for some \( j \). This case is not interesting to study, since if \( B_i = 0 \) then from (4.5) we have \( \varepsilon_{i+1} = 0 \) and then the exact solution would have been found. Due to the fact that \( C_i \to \log |M / a^4| \) when \( i \to \infty \), we have that for all \( |M / a^4| > \varepsilon > 0 \) there exists an \( i_0 \) such that for \( i > i_0 \), we have

\[ \log |M / a^4| - \varepsilon < |C_i| < \log (|M / a^4| + \varepsilon). \]

For \( i < i_0 \), we have a finite number of bounded \( C_i \), \( |C_i| \leq \max_{i < i_0} |C_i| \). Re-scale equation (4.7) and let

\[ u_i = \frac{y_i}{S + \log |Ma^4|} \quad \text{and} \quad D_i = \frac{C_i}{S + \log |Ma^4|} \]

where \( S = -1 \) if \( |M / a^4| < 1 \) and \( S = 1 \) otherwise. We have from (4.7)

\[ u_{i+1} - 2u_i - 3u_{i-1} = D_i, \quad i = 1, 2, \ldots \tag{4.8} \]
with the starting conditions \( u_0 \) and \( u_1 \). The homogenous problem of (4.8), \( u_{i+1} - 2u_i - 3u_{i-1} = 0 \), has the solution \( u_i = c_13^i + c_2(-1)^i \). For the particular solution we let

\[
    u_i = \sum_{j=0}^{i-1} k_j D_{i-1-j}.
\]

Using this expression in (4.8) yields

\[
    \sum_{j=0}^{i} k_j D_{i-j} - 2\sum_{j=0}^{i-1} k_j D_{i-1-j} - 3\sum_{j=0}^{i-2} k_j D_{i-2-j} = D_i.
\]

Re-writing and changing the indices, we get

\[
    k_0 D_i + (k_1 - 2k_0) D_{i-1} + \sum_{j=0}^{i-2} (k_{j+2} - 2k_{j+1} - 3k_j) D_{i-2-j} = D_i, \quad i = 1, 2, \ldots
\]

It is easily seen that \( k_0 = 1 \) and \( k_1 = 2 \) and that

\[
    k_{j+2} - 2k_{j+1} - 3k_j = 0
\]

gives a solution. By using the z-transform on the above expression, we find that

\[
    k_j = \frac{3}{4} 3^j + \frac{1}{4} (-1)^j,
\]

where \( j \) is a natural number. A particular solution is then

\[
    u_i = \sum_{j=0}^{i-1} \left( \frac{3}{4} 3^j + \frac{1}{4} (-1)^j \right) D_{i-1-j},
\]

and thus when we add the homogenous and particular solution we find that (4.8) has the general solution

\[
    u_i = c_1 3^i + c_2 (-1)^i + \sum_{j=0}^{i-1} \left( \frac{3}{4} 3^j + \frac{1}{4} (-1)^j \right) D_{i-1-j}.
\]  

(4.9)

To show that the method converges we have to study \( D_i \) a bit more closely. Assuming that \( \frac{|M|}{a^4} > 1 \) we have

\[
    D_i = \frac{\log|B_i|}{1 + \log|Ma^{-4}|}.
\]  

(4.10)

From the earlier arguments above, we have for (4.10) that
\[
\frac{\log|B_i|}{1 + \log|M/a^4|} < \frac{\log|Ma^{-4} + \varepsilon|}{1 + \log|Ma^{-4}|} = \frac{\log|Ma^{-4} + \varepsilon|}{\log|M/a^4|}.
\]

Since \(\log\) is an increasing function we have that \(\log(|M/a^4| + \varepsilon) < \log|eM/a^4|\) if \(\varepsilon < \varepsilon' = (e-1)|M/a^4|\). We have assumed that \(\varepsilon < |M/a^4|\), so we know that all \(\varepsilon\) are smaller than \(\varepsilon'\) and we thus have
\[
\frac{\log|B_i|}{1 + \log|Ma^{-4}|} < 1.
\]

We can also make the following estimation
\[
\frac{\log|B_i|}{1 + \log|Ma^{-4}|} > \frac{\log|Ma^{-4} - \varepsilon|}{1 + \log|Ma^{-4}|}.
\]

Since \(\log x < x - 1\) for all \(x > 1\) we have that \(1 + \log|M/a^4| < |M/a^4|\). Furthermore \(\log|M/a^4| - \varepsilon > 0\) if \(\varepsilon < \varepsilon'' = |M/a^4| - 1\), which is a positive quantity by assumption. Thus for all \(\varepsilon < \varepsilon''\) we have
\[
\frac{\log|B_i|}{1 + \log|Ma^{-4}|} > \frac{\log|Ma^{-4} - \varepsilon|}{|Ma^{-4}|} > 0.
\]

We have thus shown that for a sufficiently large \(i\), \(0 < D_i < 1\) when \(|M/a^4| > 1\). In a similar way we see that when \(|M/a^4| < 1\) we have
\[
\frac{\log|B_i|}{-1 + \log|Ma^{-4}|} > \frac{\log|Ma^{-4} + \varepsilon|}{-1 + \log|Ma^{-4}|} > 0
\]

and
\[
\frac{\log|B_i|}{-1 + \log|Ma^{-4}|} < \frac{\log|Ma^{-4} - \varepsilon|}{-1 + \log|Ma^{-4}|} < 1
\]

for all sufficiently small \(\varepsilon\). The final case that needs to be examined is when \(|M/a^4| = 1\). We have with \(S=1\)
\[
\frac{\log|B_i|}{1 + \log|Ma^{-4}|} = \log|\varepsilon| < 1,
\]

if \(\varepsilon < e - 1\). On the other hand, we also have
We have thus shown that $|D_i| < 1$ for all $i > i_0$. We can return to (4.9) and divide $u_i$ with $3^i$, which yields

$$\frac{u_i}{3^i} = c_1 + \frac{c_2 (-1)^i}{3^i} + \frac{1}{3^i} \sum_{j=0}^{i-1} \frac{3^j}{4} D_{i-1-j} + \frac{1}{3^i} \sum_{j=0}^{i-1} \frac{1}{4} (-1)^j D_{i-1-j}.$$  

Re-writing this expression and taking absolute values yields

$$\left| \frac{u_i}{3^i} - c_1 - \frac{c_2 (-1)^i}{3^i} \right| \leq \frac{1}{4} \sum_{j=0}^{i-1} \left| \frac{3^j}{4} D_{i-1-j} \right| + \frac{1}{4} \sum_{j=0}^{i-1} \left| \frac{(-1)^j}{3^j} \right|.$$  

(4.11)

Studying the first of the two terms in the right-hand expression, we see that this can be calculated as

$$\frac{1}{4} \sum_{j=0}^{i-1} \frac{3^j}{4} = \frac{1}{4} \left(1 - \left(\frac{3}{4}\right)^i\right),$$  

(4.12)

that tends to $3/8$ as $i \to \infty$. Since the estimated series in (4.12) is convergent then $\frac{1}{4} \sum_{j=0}^{i-1} \frac{3^j}{4} D_{i-1-j}$ is also convergent. From the second term in (4.11) we have

$$\frac{1}{4} \sum_{j=0}^{i-1} \frac{(-1)^j}{3^j} \leq \frac{1}{4} \left(\frac{1}{3^i}\right) \to 0,$$

when $i \to \infty$. Then the sequence $u_i / 3^i$ converges when $i \to \infty$. Thus there exists a constant $c$ such that $y_i / 3^i \to c$ when $i \to \infty$. Furthermore, we have

$$3^i (c - \epsilon') < y_i < 3^i (c + \epsilon'),$$

for all $\epsilon' > 0$ if $i > i_1$. Since we know that $y_i \to -\infty$ when $i \to \infty$, we see that $c < 0$.

To determine the convergence rate we recall from definition 4.1 that the rate of convergence $p$ is the supremum of the nonnegative numbers $p$ such that

$$0 \leq \lim_{i \to \infty} \left\{ \frac{|e_{j+1}|}{|e_j|^p} \right\}$$

is a finite constant. Therefore we study $y_{i} - p y_{j-1}$ and try to find a $p$ such that $y_i - p y_{j-1} < \infty$. We can do the following approximation

$$y_i - p y_{j-1} < 3^i (c + \epsilon') - p 3^{i-1} (c - \epsilon') = 3^{i-1} (c (3 - p) + \epsilon' (3 + p)).$$

We see that the supremum of the numbers $p$ is at least 3. The rate of convergence for our method then is at least three.

We have proved that the method converges and that it does so with rate at least three. Thus, we have shown what we set out to do in this chapter.
5 System Development

In the previous chapters, we have proven that the method described in chapter converges and that it does so with a convergence rate of three.

To test the method in real life we created a test implementation. In this chapter, we first describe some software development methods and then the test system that was implemented. The next chapter will describe the results from running the test implementation for some functions.

5.1 System Development Method

This part starts with some descriptions of software development methods. It is followed by a description of the software development method we used for our system implementation.

5.1.1 Methods

Software development is an engineering discipline and is not only about creating a software system that solves a specific problem. Software development is also about producing the software system in a well-engineered fashion and in a cost-efficient way. For a software system to be well-engineered, Sommerville (1992) state these four key-attributes that the software system should have:

1. The software system should be maintainable.
2. The software system should be reliable.
3. The software system should be efficient.
4. The software system should offer an appropriate interface.

To comply with all these attributes is not easy, as some of the points are exclusive. Optimising the balance between these attributes depends on the software system use. A software system for a cell phone may be restricted to small memory. Then it may be necessary to optimise efficiency with trade-offs for maintainability and interface.

To help creating cost-efficient and well-engineered software systems different software development methods have been created. These are for example ‘Exploratory Programming’, ‘Prototyping’ and ‘Formal transformation’ (Sommerville 1992).

Kang et al (1992) says that a software development method should identify:

1. The development activities.
2. The artefacts that should be created.
3. The available resources.
4. A development plan (process) linking the activities, artefacts and resources together.

The ‘Waterfall’ method

The first software development method a software developer usually meets is the ‘Waterfall’ method. It is based on the hypothesis that all system development is a document driven linear step-by-step method.

The steps in the ‘Waterfall’ software development method are:

---

1 Software system will be used for a computer program or a set of computer programs solving a problem.

2 Kruchten (2000, page 41) defines artifact as “…a piece of information that is produced, modified, or used by a process.”
1. Requirements
   Identify and describe the requirements of the software system in a way that both system developer and end-user can understand. Defines what the software system developed should do in a requirement specification document.

2. System and software design
   System design defines the environment for the system (hardware, operating system, protocols, etc.) and creates program architecture of executable programs. The result is a software design that spread the requirements over program units in executable programs.

3. Implementation and unit testing
   Realisation of the software design. Each unit is tested to follow its design specification before it is complete.

4. Integration and system testing
   Test that the different executable programs in the program architecture work together in the way the system design wanted. Also, test to make sure that all of the requirements are fulfilled.

5. Operation and maintenance
   Updating the system for new end-user demands, correcting errors found during the system operation. This also includes keeping the system available and running.

The linear structure of the ‘Waterfall’ method is to complete one step before moving on to the next step. This makes it easy to define where the development currently is in the software development process.

However, this does not work in real life practice. The requirements may change during the design, the design may change because of implementation issues, the implementation may change cause of an integration fault, etc. Software development is therefore not linear but incremental and involves sequence of steps (Sommerville 1992).

The Rational Unified Process method
Both Booch (1994) and Rumbaugh (2001) have described incremental object-oriented software development methods. These methods has been merged and further developed into one software development method framework called ‘The Rational Unified Process’ (RUP) as described in Kruchten (2000).

The activities in a RUP method are grouped together into the following workflows or activity groups:
1. Project management.
3. Requirements.
4. Analysis and design.
5. Implementation.
6. Test.
7. Configuration and change management.

These activities are not separated over time. Instead, all activities are included in each incremental, known as an iteration. The length and number of iterations may vary depending on the size of the software development effort and the development teams experience in the problem area.

In one iteration, the time distribution between the activities depends on where in the development process the iteration exists. In the first iterations the testing may only be 1% while in the last construction iterations the testing may be 95% or more.

The business engineering activities connects the real world with the software that shall be developed. From the business activities, the need of a software system comes and the requirements for such a system. In the requirement activities the business requirements are analysed and transformed into a system view, where the requirements are distributed over a set of software components.

In the analysis activities the domain analysis and scenario planning is done. Domain analysis tries to identify the important classes and objects for the domain area. These activities could also include examine already existing systems and their solutions.

Scenario planning is the main action among the analysis activities. In the scenario planning the essential scenarios, or use cases, is identified and analysed. Scenarios are based on a storyboarding technique similar to practices in television and movie industry. The use of scenarios for analysis was first formalised by Jacobsen (Booch 1994). The purpose of scenarios is to establish the behaviour for each class and to illustrate how the systems elements co-operate together as a team.

The design activities consist of the high-level and low-level design. The high-level design includes dividing a software component into layers, modules etc. Architecture patterns can be helpful like for example the pipe-filter architecture pattern as described by Shaw & Garlan (1996).

The low-level design consists of designing the individual components in the architecture. This includes design of the classes and the interaction among them as described by Kruchten (1995). Gamma et al (1995) describes several design patterns that can be used in this activity. More problem specific design pattern also exists, for example for distributed systems.

The implementation activities are to realise the design by implementing the classes and test them. What is realised in an iteration is decided in the plan activities. As different requirements are included in each iteration, some classes may need to be reworked.

The test activities tests the architecture components together to verify that the system does what it was intended for and within the requirements.
The project management activities handle in what order functions should be implemented, who should do it and when. The configuration and change management activities keep track of the releases and the changes between them.

A RUP software development process have four different phases. These phases indicate the progress of the software development process. Each phase have entry and exit criteria that must be achieved that is called a milestone. When all four phases are completed, the software system will either be terminated or extended and updated by repeating over the four phases again.

Each of the phases is divided into one or several iterations as in Figure 5.1. The management phases are the following (Kruchten 2000 and Booch 1994):

1. Inception
   The vision of a product or a need is transformed into a software development project. The inception seeks to establish the core requirements for the software system.

2. Elaboration
   The target for this phase is to analyse the problem and create a design an architecture that shall solve the problem. The analysis also allows being able to create a plan for the two next phases.

3. Construction
   Build the system in increments. Each increment shall contain more functionality then the previous increment.

4. Transition
   Put the software system to work by training the end-user, marketing, installing and maintain the software system.

**Extreme programming**
Jeffries et al (2000) describes a software development method called ‘Extreme programming’ (XP) developed by Kent Beck. It is a simple incremental method based on a set of common-sense rules. The essential ideas for XP are:
• **User stories**

User stories are used to create a description of what shall happen in different situations and to state the requirements. The user stories are also easier to understand for a non-technical person than a technical requirement specification, as they are written in the end-users vocabulary. Because the user stories hold the requirements, they are the foundation for how the system is expanded and evolved, as user story is ranked in importance. User stories are related to the scenarios in RUP as both encapsulate requirements. Nevertheless, the user stories differ from RUP scenarios, as they do not show how a problem is solved. Instead, they only describe the problem. The user stories do not cover anything related to algorithm or database layout or the interaction between classes.

• **Test first philosophy**

From the user stories, test cases and test implementations is created. The idea is to emphases the tests and to make sure that all implemented user stories still works. The tests first philosophy also helps with the requirement analysis, as a user story that is added can generate more user stories. To further imply the test first philosophy, test implementation should be created before the implementation of what the test case cover.

• **Pair programming**

The idea with pair programming is that two persons working together can write more and better implementations then when working each on there own. One result from working together is that both programmers know what the implementation does. This makes it easier to replace a member of the team.

• **Continuously Integration**

To discover faults early between different parts, integration is needed all the time. This also helps to test the system regularly. The common solution to this is ‘Nightly builds’ and automated testing with the help of a script language etc.

• **Communication**

All communication should be direct face to face in XP. The software development team consists of developers and at least one business expert. The business expert creates the use cases and is the development teams contact person for all questions regarding the requirements and the functionality of the system.

Also in XP there exist iterations. One iteration consists of a selection of the user stories created. These user stories are graded in importance where the most important shall be implemented first.

Martin (2000) describes XP as the minimal implementation of RUP with add-ons. Smith (2001) points out some similarities between RUP and XP, but he also points out several indifferences. The main indifference is that XP is not designed to scale for large projects.

To support this indifference Smith reasons that to be able to scale, not all code can be managed by all developers. If the project scales enough, the communication would be a major difficulty. Other reasons he gives are refactoring, where XP misses the architectural overview of the system.

One other major indifference between XP and RUP is that XP only concerns the software development while RUP looks also at the business case.
5.1.2 Method choice

As said before, all software development is incremental. Therefore, the ‘Waterfall’ method is not interesting as a development method, as it does not support incremental development.

Cockburn (1999) says that there is a cost in choosing a too rigorous software development method. This gives the idea that keeping the method simple and small is extremely good. RUP may in this case be a too rigorous software development method. XP on the other hand is small, simple and easy to use. It puts the focus on the software development and does not imply any limitations on the generated documentation.

XP, on the other hand, have the restrictions with pair programming and direct communication, which are hard to fulfil in our specific case. Pair programming is hard when the software development team consists of just one single person. The direct communication is hard when the project team is distributed geographically.

Therefore, we use a minimal RUP implementation that is close to XP. RUP for its system development process and XP for the fundamental low level tasks. The problem that XP cannot scale is no problem for us.

From RUP the following parts are modified:

- Some development views are disregarded.
  From the ‘4+1’-view model by Kruchten (1995), the process view and the physical view are not used. The reason is that the software system is executed on a single processor system and the views would not include any interesting information.

- Documentation
  The documentation is really on the basic level. No special documentation will be generated except from what is in this report. All other documentation is in the form of ‘Javadoc’ like comments within the code (see internet resource Sun).

- Some workflows are ignored
  Both the business and deployment workflows are ignored as our software system only lives to test the convergence rate. There is no real need to look at the background and the future.

The modifications from the XP software development method are the following:

- No pair programming.
  Pair programming is disregarded because of the reason given previously.

- Communication.
  The direct communication is limited to randomly occurring meetings and telephone conferences as well as sending mail.

- No role usage
  The XP roles are not used. That is because both the customer and the manager roles are both missing. Some parts of there responsibilities are therefore taken care of by ourselves.
5.2 System development

Now we will describe the requirements found and the design issues on a system for finding a minimum of a mathematical function. This section also describes how the design is implemented. First, there is a description of what the system should do and a statement of the requirements. Then the design specific issues are discussed, as how the system is divided into components etc.

5.2.1 System requirements

The general requirements on a software system for finding a function minima are these. Some of these requirements are general and others are more specific to mathematical software systems. The system:

- shall be efficient
- shall return the correct solution
- shall be function independent
- shall be method independently
- should be independent of the system platform

For requirement 1, efficient means that it should need to use as few iterations as possible and by that be as fast as possible. Few iterations should be given by using a fast converging method as the method in chapter 2. The system should also be as memory efficient as possible.

Correct means that it should calculate the function values correct for any function and value. It also means that the calculated approximation should be less than the allowed error. Last, the system shall not calculate an approximation that does not hold for the requirements that the method states. As for the tested method, the interval endpoints shall always have function derivative values with opposite signs.

The system should not be limited to be able to calculate only minima for some sorts of functions. Instead, the system should be able to calculate minima for every possible sort of functions, only restricted by that the functions must follow the given requirements for the method used.

There should also be a possibility to use the system independently of the method. The systems architecture should support more than one method. If we want to compare two different methods, we should be able to use the same system architecture and only change the method calculation.

Last, we would like to be able to move the system from one platform to another without making to many changes to the system. For example, we should be able to move from a PC environment running Windows as operation system to a Unix version as Solaris on a workstation.
5.2.2 System specification

**Requirement analysis**

The first requirement says that the system should be efficient. That can be in sense of speed. The speed depends on the method. If the method have a low convergence rate, the system must iterate much before the approximation is good enough. On the other hand, the calculations are often very easy, and each iteration takes a shorter time to perform. If the method instead has a high convergence rate, the system would only need few iterations to get an adequate approximation. Each calculation is instead often more complex and each iteration takes more time.

The system can also be efficient in other ways. The system can be memory efficient and be able to run on computers with limited memory resources. We do not state any limitations on needed memory, as we demand the underlying operation system to handle the memory. That does not prevent us to be operating system independent, as requirement 5 states when most of today’s operating systems support virtual memory.

Both sorts of the above efficiencies are needed. The time efficiency because the user does not want to wait for a long period before the system gives a result back. The memory efficiency because, if the system uses more memory then there is available, the operating system will interrupt the execution. The efficiency will be compared by the time from call until we get a result back from the system.

That the system should calculate a correct value is obvious. There is no need of a malfunction system. Therefore, it is an important question that must be satisfied. Correct calculation is connected to the function independence. To get a correct solution we must calculate the function values correct for all functions. We can not be sure that all calculations are correct. A computer always uses an approximation to represent values (see Johansson 1995). We must therefore allow minor errors. These errors can of course not be larger then the allowed error.

Function independence is also important when we want to be able to use the system for different functions. The function independence makes the system more useful for the user and this feature is quite important. The function independence may not be as important as speed efficiency and calculation correctness but more important then memory efficiency.

From function-independence is the step to method independence not far. By method independence, we mean that the system should be changeable so it is able to use a certain method. We do not want the system to support one method only. This will also make comparisons between different methods possible and that makes it a quite important requirement.

Last, we have the requirement of platform independence. We would like to use the system on any available platform. We cannot guarantee that the system works on every platform but it will work on a major part of available platforms.

To rank the requirements is impossible but we see that some of these requirements are less important then others. We have platform independence and memory efficient as minor and correctness and execution efficiency as major requirements.

**Implementation method choice**

When choosing implementation method there are three ways to implement the system. The option of functional programming does not fit our problem why we do not look at that method here.
Then we have two options, we could implement it in an imperative way, or we could implement it in an object-oriented way. Regardless of which method we choose the remaining choices depend on this choice.

The imperative method is characterised by sequential execution of statements, variables are memory locations, and assignment is used to change the values. The property of variables as memory location makes the imperative method memory efficient. The drawback for the imperative method is the von Neuman bottleneck that restricts manipulation on multiple dataobjects at the same time. Most programming languages support the imperative method (Louden 1993).

The object-oriented approach is based on objects, which consist of datamembers. To the object is a set of methods associated to access and modify the datamembers. The datamembers and methods are the object attributes.

The objects are grouped into classes that represent all objects with the same attributes. Between classes, there could exist inheritance. For example, one class could be a generalisation or specialisation of a class with similar methods.

Object orientation also gives us polymorphism. Polymorphism means that the objects that an operation can work on can be of different classes. Objects could call a method within another object without knowing exactly which class that object is made of. Because methods and data are connected through the class definition, object-oriented programming are accused of not being memory efficient (Louden 1993).

With the functionality of inheritance and polymorphism, the object-oriented approach is suitable to make system that should be adaptable to different methods as our. The imperative approach does not support this as easy as object-orientation does when the method is more tightly connected into the system. This led us to choose an object-oriented approach for the system development.

That will also give the ability to use a standard interface to a mother system that is independent on whether one using Newton-Raphsons method, our third order convergent method or any other optimization method. That will make it easy to compare different methods for finding a minimum of a function, when the only thing we need to do is to change the method.

Choosing program language

To make the choice of language easier we made our choice from commonly used languages. Together with the restriction that the language should support object-orientation gives us to choose among C++, Java, and Ada. All these three languages satisfy requirement 5.

C++ satisfy requirement 5 as platform independent due to its availability on almost every possible platform. There are versions that act differently because different implementations. There exists a standard version that hopefully all versions should follow. C++ strongest point is its speed while the weakest are the pointers.

Java is platform independent because it uses a virtual machine. The virtual machine interprets an intermediate code called bytecode that are the same for all platforms. The virtual machine is different for all platforms but should act the same for them all. The virtual machine slows Java down compared to C++, else Java have removed many bad parts from C++.

Ada is platform independent in code and one have to rebuild the system on every platform. We do not see a problem when the system is not supposed to be moved from one platform to another between executions. Ada’s strongest point is its strict typing that prevents typecasting.
For requirement 2, we would like to use a hard typed language as Ada. That gives a security among types so that we do not convert floating numbers to integers and then back again losing significant precision. We need to make sure that our calculations are correct. The strong typing makes the language less speed efficient and therefore not supporting requirement 1 totally.

Instead, we should use a fast language as C++ that satisfy requirement 1. C++ is not hard typed but we can use it and still get correct answers. We have to take into consideration how to calculate approximations and function values.

Java ends up between Ada and C++, and is not preferable unless you want a system that can be run on any platform. That is because Java lacks speed when it is interpreted and it is not hard typed.

This led to that we chose to use C++ as programming language.

### 5.2.3 System design

The system has two major tasks: to calculate the function value and to calculate a new approximation. These two tasks should have a common interface against the system that uses methods in our system. That gives us a system that looks like Figure 5.2. The ‘User system’, let’s say our user interface, uses for a given function methods that the ‘Library system’ provides for finding a minimum. The library system can then be divided into two major parts, one for each task. To this two parts we add a third part that is the framework that interfaces the surrounding system.

There could exist methods that need access to some external libraries that contain functions and methods as the commercial NAG or one out of the free PACK series that contain for example LINPACK and LAPACK, as referenced by Freeman & Philips (1992). These can be of any number and we choose to connect them to the method part. This will give us a system architecture as in Figure 5.3.

In Figure 5.4 the library part is dashed because it does not need to be present in all implementations of a method. One may include that functionality into the method part. The other three parts will always be present. The framework will be the same for all implementation. The method and function part are implementations of the strategy design pattern to be able to test different algorithms and functions (Gamma et al 1994).
The framework part is the connection between the other parts and that makes it practical to let it contain and handle the System State. Then all parts have access to it easily if they need to know something about the state.

There is always a possibility that something goes wrong. In that case, the function part and the method part will throw exceptions. The function part takes care of errors like a mistyped function string like ‘x*4+’ in the general function implementation, while both the method and the function parts cares for some calculation errors like division with zero. The library part does not have any specified error handling defined by our system, because the library part is requested to handle all errors by itself, or the method that uses the libraries. That is because we cannot handle all types of errors that can occur.

The system will also include a data part that is sent between the interface and the method part. That is because the method part should not store any values. That will give the possibility to use decorator design pattern on the method to use another method in some cases (Gamma et al 1994). All this will give a system that looks like Figure 5.4.

**Tasks for each part**

Each part of the system has a number of responsibilities to handle. Here we give a short description of the responsibilities for each part. The design of each part is deeper described in section 5.2.5 (function part), section 5.2.6 (method part) and in section 5.2.7 (framework and data parts).

**Function part**

The function part solves the third requirement of function independence. It therefore handles the following responsibilities:

- Check function-strings and builds syntax-tree for them with the general function implementation. If there is an error in the function-string, throw an exception.
- Calculate function values and function derivative.
- Create statistics to be able to know the number of function evaluations and the number of operations for each function evaluation.
Method part

The method part solves the fourth requirement of method independence. Its responsibility is to:
  • Calculate a new approximation based on an old approximation value. If something goes wrong then throw an exception as a signal of error.
  • Be able to support different methods.

Library part

These parts supply functions that the method part uses in its calculation of a new approximation. Preferably this is a standard package like NAG or LAPACK, but could also be one that is specific to the method implementation.

Framework part

The framework glues together the user system, the function part, and the method part. It has the following responsibilities:
  • Hold the state of the system.
  • Control execution.
  • Transfer messages between the other parts.

Data part

Data part holds any vital data for the system and method.

5.2.4 Scenarios

A scenario consists of a top-level of objects that take place in the scenario. These objects send messages or function-calls to each other. This is shown as arrows from the sending object to the receiving object with a name of the message/function-call as well as a number indicating in what number order the call was sent. If the message/function-call may be done recursively then a star is added to the message/function-call name.

The interactions that we chose to study are these:
  • Starting up the system
  • Parse of a function-string in the supplied general function implementation
  • Calculation of a new approximation

These scenarios were chosen since these functions are the main or the hardest function for each part. The starting of the system creates every part and is one major task for the interface part, parse of function-strings are the hardest task for the Function-part as calculation are for the Method-part.
Starting up the system

1. The user system creates an instance of a method implementation
2. The method calls method ‘createLibraries’
   5. The method function ‘createLibraries’ creates instances of library parts if these are used
6. The user system creates an instance of the framework
7. The interface creates an instance of the data part
8. The user system set the method in the framework
9. The framework informs the method instance of its existence
10. The user system creates a function implementation.
11. The user system assigns the function implementation to the framework.
12. The user system creates a data part instance
13. The user system initiate the data part by calling ‘setAttribute’ for each value that the method needs
14. The framework is state is set by the created data part
1. The user system calls the method ‘GetFunction()’ in the framework, that returns the general function part implementation.
2. The user system calls ‘SetFunction(…)’ in the function part implementation with a mathematical function as a string.
3. The function part deletes old parsetree (Tree).
4. The parsetree recursively deletes all subtrees.
5. Then it creates a new root for the parsetree (Tree).
6. The parsetree creates subtrees as needed.
7. The function part resets the statistics.

The user system knows if an error was detected if an exception is thrown.
Calculation of a new approximation

In this scenario, we do not use any libraries. The use of libraries would be done in point 5.
1. User system calls framework method ‘Calculate()’. This will start the iterations. For each iteration:
   2. The framework calls the ‘Init(…)’ function if it is the first iteration.
   3. The method implementation gets the current state from the data object.
   4. The method implementation calls the ‘CalculateDerivative(…)’ method to get some function evaluations done.
   5. The framework forwards the call to the function part.
   6. The function part calls ‘CalculateDerivative(…)’ recursively. For each ‘CalculateDerivative(…)’ with degree equal to zero or one, call the syntax tree method ‘Calculate(…)’ or ‘Derivative(…)’ with given x-value.
   7. If the function part implementation is the general function implementation, the Tree calls ‘Calculate(…)’ or ‘Derivative(…)’ recursively through the syntax tree.
   8. The method implementation stores the new approximation in the data part using method ‘SetAttribute(…)’.
9. The framework stores the new state returned by the method implementation.
10. The framework calls the ‘Calculate(…)’ function to calculate the next iteration value.
11. The method implementation gets the current state from the data object.
12. The method implementation calls the ‘CalculateDerivative(…)’ method to get some function evaluations done.
13. The framework forwards the call to the function part.
14. The function part calls ‘CalculateDerivative(…)’ recursively. For each ‘CalculateDerivative(…)’ with degree equal to zero or one, call the syntax tree method ‘Calculate(…)’ or ‘Derivative(…)’ with given x-value.
15. If the function part implementation is the general function implementation, the Tree calls ‘Calculate(…)’ or ‘Derivative(…)’ recursively through the syntax tree.
16. The method implementation stores the new approximation in the data part using method ‘SetAttribute(…)’.
17. The framework stores the new state returned by the method implementation.
18. The framework calls ‘CheckEndCondition(…)’ to see if the end condition is fulfilled.
19. The method implementation get the needed attributes from the data part to check the end condition.
20. If end condition is not fulfilled, go to step 10 again, else return the last approximation.

5.2.5 Function-part

Let us now look on the function part. Here is a description of what the function part does, and how it is done.

Functionality
This part primary deals with tasks like:

- calculate function values
- calculate function derivative values
- create statistics
To calculate function and function derivative values is the essential operations for
the function part. This can be done with direct implementations of the functions or
with the enclosed function parser.

The function part is also able to generate statistics of function and function
derivative evaluations and number of mathematical operations for each function
evaluation. This is used to be able to compare different methods to each other, see
chapter [6].

The enclosed parser also has the task to:
• signal error
• check the strings that contains the functions
• build a syntax tree

The need for checking the function strings is because not all function strings may
be syntactically correct. The systems parser cannot tell from a natural language what
it is meant to do. Instead, we have to use a special language like a strict mathematical
language. If the function string yet contain errors like three plus signs in a row, it must
still be able to signal to the user system that there was an error in the function string.

To check the function string we use the method of recursive decent as described in
decent is based on a grammar, and a grammar consist of four parts:
• one set of terminals
• one set of non-terminals
• one start non-terminal
• one set of productions

Terminals are the symbols from which strings are made. Non-terminals are
syntactical variables that represent a set of strings. One non-terminal is the start
symbol. That start symbol represents all strings possible to form from the grammar.
The productions are the way that terminals and non-terminals can be combined in a
proper way to form a string. Each production starts with a non-terminal followed by
an arrow and a sequence of terminals and non-terminals (Aho, Sethi &Ullman 1986).

Recursive decent work by transforming each production into a procedure or class
whose action/behaviour depends on the right side of the production. Terminals are
read while non-terminals are interpreted as calls to another procedure/class. To be
able to choose between different parts of the productions we use a look-a-head
character. The use of recursive decent will also give us the ability to build the syntax
tree at the same time as the syntax check. If there is an error, we throw an exception
that includes a string that describes the error.

Louden (1993) also describes another method, YACC (Yet Another Compiler
Compiler). It is available for almost every UNIX platform and generates a C program
that uses a bottom-up algorithm to parse the grammar. Based on that YACC is a
UNIX system and that it does not give us the syntax-tree, we choose to use recursive
decent as a parser.

The syntax tree is saved and used for calculation of function value. That is done by
traversing the syntax tree in-order. To calculate function derivative value of degree
one each projection also knows the rules for derivation, and calculates derivatives in
the same way function values are calculated. For function derivatives of degree two
and more, we use Ridders method of polynomial extrapolation as in Press et al (1992).
Function Grammatical

To be able to build a syntax tree we must have a grammar for our system parser. The grammar we designed is simple and has the following productions:

\[
\begin{align*}
<\text{Expr}> & \to <\text{Term}> [('+'|'-') <\text{Expr}>]^* \\
<\text{Term}> & \to <\text{Factor}> [('*'|'/') <\text{Term}>]^* \\
<\text{Factor}> & \to [(('+'|'-') <\text{Number}> | <X> | <\text{Function}> | <\text{ParE}>) '\^' <\text{Expr}>] \\
<\text{Number}> & \to [[<\text{Digits}>] ['.' <\text{Digits}>] [('e'|'E') [[('+'|'-')]]<\text{Digits}>] \\
<\text{Digits}> & \to ('0'|...|'9')^* \\
<X> & \to 'x' | 'X' \\
<\text{Function}> & \to <\text{FunName}> <\text{ParE}> \\
<\text{ParE}> & \to ('<\text{Expr}> ')') \\
<\text{FunName}> & \to '\sin' | '\cos' | 'tan' | 'asin' | 'acos' | 'atan' | 'sqr' | 'sqrt' | 'exp' | 'log' \\
\end{align*}
\]

The ‘+’ sign last in <Digits> production means one or more of those terminals inside the parentheses and the ‘*’ sign last in <Expr> and <Term> productions means zero or more of those terminals inside the parentheses. The ‘[xxx]’ means that the symbols inside are optional. The start production for the grammatical is <Expr>.

The syntax tree is built from these productions. The production <FunName> will support the mathematical functions of sin, cos, tan, arcsin (asin), arccos (acos), arctan (atan), square (sqr), squareroot (sqrt), exponatial (exp) and the natural logarithm (log).

For each production, there is also a matching derivative production as follow.

\[
\begin{align*}
<\text{Expr}'> & \to <\text{Term}'> [(('+'|'-') <\text{Expr}'>)] \\
<\text{Term}'> & \to <\text{Factor}'> [('*'|'+') <\text{Term}'> ' '* <\text{Factor}>' '* <\text{Term}'>')'] \\
<\text{Factor}'> & \to Note 1 \\
<\text{Number}'> & \to '0' \\
<X> & \to '1' \\
<\text{Function}'> & \to <\text{FunName}'> <\text{ParE}> ' '* <\text{ParE}> \\
<\text{ParE}'> & \to '(' <\text{Expr}'> ')') \\
<\text{FunName}'> & \to Note 2 \\
\end{align*}
\]

– Note 1: The number of projections are quite a few but are all based on the following equation: Da^b = \left(\frac{ba'}{a} + b' \ln a\right)a^b

– Note 2: The derivative function for each function can be found in Hellström et al (1991) or any other base calculus book.

Classes

When implementing Recursive decent, each production is transformed to a class. Inheritance and polymorphism makes this ideal. In that way, we can get the answers from different objects by using the same interface. This will make the syntax tree into a ‘Composite’ design pattern implementation (Gamma et al 1994).

We will also have a class that interfaces towards the framework. This class will count statistic values for all function evaluations. The parser class contains the syntax tree as well as a cache of calculated function and derivative values to improve the calculation time.
5.2.6 Method part

This section will deal with the internal design of the method part. As in the function part, we will start describing what this subsystem does, followed by the realisation.

Functionality

In this part, we calculate a new approximation based on an old approximation. That may need access to library part/parts, why these are included here as well. This part also does:

- Support libraries
- Initial method calculations
- Throw error exceptions
- Check end condition

The main concern for this part is the calculation of a new approximation. This calculation is method specific. Here the adaptation of the system is done to follow the method. This makes it important that it is easy to make the needed changes. The adaptability is an important consideration for this part. How to adapt the system is described later in section 5.2.8.

In some calculations, the need for methods found in other mathematical libraries is used. Then it is up to the method part to create and support these libraries. That support is initiation and removal.

If an error occur during the calculation, we need to be able to signal that an error occurred. That is done by throwing an exception.

The last part, check end-condition, is to see if the last approximation was better than the previous one, and if the approximation was good enough to stop.

One way to check if the approximation is good enough is done by comparing the difference between the last two iteration’s approximations. Another way to approximate the error by using the size of the interval. One commonly used method for multivariable functions is the max-norm, which is defined as

$$\|e\|_\infty = \max(\|e_1\|, \|e_2\|, \ldots, \|e_n\|).$$

The choice of end condition method is dependent on the method. For some methods that, for example complex numbers, the end condition can be calculated in another way. The default end condition check uses the first alternative by comparing the new approximation with the approximation from the previous iteration.

Classes

This part consists of only one class, the method class. The method class is more of an interface class that all methods must implement. The implemented methods can use functions that the framework supplies, and functions from own libraries. The class will also need to know what values that are stored in the data part.

The instantiation of the needed libraries can be done during creation of the method class or during calculation. The functions from the libraries are private to the method class.
5.2.7 Framework and data part

In section 5.2.5 and section 5.2.6 we have described the two main parts of our system. Here we describe the remaining framework and the data part.

**Functionality**
The parts that we have not discussed yet are the framework and the data part. The tasks for these parts are:
- Hold the state of the system.
- Store vital data.
- Glue the system together.
- Control execution.

The framework holds some part of the system’s state. The state that is stored in the framework is the global state of the system. The state for the method is stored in data part objects. The values in the data part are stored as name-value pairs, and are method dependent and change each time a new iteration is done.

**Classes**
The framework parts consist of two main classes, the framework class and the data part class. The framework class controls the state and handles the call flow to the method implementations. The data part class holds the values in a vector as name-value pairs.

5.2.8 Adjust the system to a calculation method

This section tells how to adjust the system to use different types of calculation methods. The adjustment is made by implementing some functions in the method class.

We will show how the adjustment is done by giving a small example template. This example will be an implementation of the Newton-Raphson method. In the method part we need to override at least one method, but it can be as many as six. The affected methods are:
- We need a constructor declaration. This is needed for all classes.
- We can name our method by overriding the ‘GetMethodName’ method.
- We can create library instances by overriding the ‘CreateLibraries’ method or they can be created in the constructor.
- Delete the library instances in a destructor or overriding the ‘DeleteLibraries’ method.
- If the user do not want to use the default end condition check, override the ‘CheckEndCondition’ method.
- If the method need to do some initial calculations, override the ‘Init’ method.
- We must override the ‘Calculate’ method.

**Example**
In our example, we do not need to override neither the library methods nor the ‘CheckEndCondition’ method. That is because we do not need to use any library or to use any other end condition check then the default end condition.
The Init method is not overridden either, because the Newton-Raphson method does not need any previous calculations. If the implemented method instead were the Secant method, some initial calculation would be needed.

This gives us the following needed source code:

```cpp
class NewtonRaphsonMethod : public Method {
    // Constructor
    NewtonRaphsonMethod() : Method() {}

    // Method name (Optional)
    CString GetMethodName() const { return "Newton-Raphson"; }

    // Calculate method
    NVData* Calculate( const NVData* state) const {
        double error = state->GetAttribute( MethodGlobals::Error );
        double appr  = state->GetAttribute( MethodGlobals::Approximation );
        double first = myParent->CalculateDerivate( appr, error, 1, Center );
        double second = myParent->CalculateDerivate( appr, error, 2, Center );

        // Copy current state
        NVData* newState = new NVData( state );
        newState->SetAttribute( MethodGlobals::Approximation, appr-first/second);
        return newState;
    }
}
```

5.3 System development results

In the previous section, we described the software development. In this section, we will evaluate the result from the software development. We start by comparing the requirements and the resulting software system and end with some general comments.

5.3.1 Requirement verification

In section 5.2.1 the requirements for the software system were given. The requirements were for efficiency, correctness, function independence, method independence and, platform independence.

The requirement for iteration efficiency depends mostly on the used method implementation. Therefore, see section 6.4 for method efficiency discussion.

The correctness requirement can be divided into two parts, function calculation correctness and method correctness. The function calculation correctness shall be within the allowed minimum error when the function and the first-degree derivative function value are calculated analytically in the general function implementation. For derivatives with degree larger than one, the use of Ridders method (Press et al 1992) should be enough, at least for not to high degree of derivative. For method correctness, see section 6.4.

Function independence is achieved in two ways. The general function can parse and interpretative simple functions within the limitations of the grammar. The second way is to implement a function directly. The last also gives the ability to create functions that behave differently in different areas of an interval as the function

\[
 f(x) = \begin{cases} 
  x^2 + \frac{2}{x} & x > 0 \\
  x + 2 & x \leq 0 
\end{cases}
\]

The idea behind the method independence is that the system should imply the same amount of complexity to all methods. The system should also help with common parts as data storing between iterations, checking the end-condition and halting the calculation when the method have iterated to many times.
Platform independence was to be accomplished with the help of the language choice of C++. However, during the development we made use of the C++ classes from the Microsoft Foundation Classes (MFC) library. To make the system more independent of the platform, these classes were changed and classes from the Standard Template Library (STL) were used instead.

5.3.2 General comments

As noted in the previous sections, several design patterns from Gamma et al (1994) have been used. At least those parts are thereby conformant with what is known as a good object-oriented design.

These design patterns are used to make the system adaptable with respect to both method and function. It is very easy to create a new method implementation and test its performance. As for the function given above, an method implementation can use one method until the error becomes less than a limit, then use another method to improve the approximation even more.

Because the function and method independence is built with the same design pattern (the Strategy design-pattern), is it as easy to change function, as it is to change method.
6 Numerical results

In this chapter, we describe how the test of the method is done and present the results we get. First, there is a description of different approaches to evaluate different numerical method implementations to how to make them comparable. Then there is a description of the different numerical method implementations that are tested and on what mathematical functions to test them on. Then last follows the results we got from doing the tests.

6.1 Test strategies

What test strategy can be used to compare two different numerical method implementations that solve the same problem? The test strategy must be based on the requirements. From section 5.2.1 we learn that there are five main requirements. In section 5.2.2 some test strategies have already been stated.

One important question is that the requirement is testable. The requirements for function independence and method independence are such requirements. They exist to able the system to test the efficiency of the different numerical method implementations. The operating system independence is not tested because all tests will be done on the same computer to make the result comparable. If we used different computers with different configurations, we could end up with different results for the same test.

We are left with two main variables for the test, efficiency and correctness.

6.1.1 Efficiency

Efficiency can be compared in several ways. The most obvious way is to compare the execution time. Other ways can be to compare the number of iterations or the number of function evaluations.

Execution time

Measuring execution time is usually done by counting processor clock ticks or the seconds for the execution. Most computer languages allow access to a clock and therefore it can be done from the inside of the program.

Comparison of efficiency by execution time is harder. The usual way of comparing the efficiency between two programs or algorithms is benchmarking. Benchmarking is based on a set of inputs that is representative for a common mix of applications for the problem. The program or algorithm that performs best is considered to be the best for all possible inputs (Aho & Ullman 1992).

Nevertheless, it is not certain that the program that performed best, really is the best program. The program could have executed all the time without any interrupts, and by that get more amounts of processor time and a higher benchmark score. To eliminate the effect of this, the average result is compared for a number of benchmarking executions.

Another issue when comparing algorithm implementations is that different implementations can do different things during the execution. Some implementations may write to a file or a database, which will increase the execution time dramatically.
Analytic investigation

A better way is to analyse the algorithm analytically. This often results in an expression describing the complexity of the problem as a function of the size in the worst or the average case. The worst case is the upper bound of the problem for some input, while the average case is what the problem generally needs (Cormen et al 1990).

For a sort problem, the size of the problem is expressed in the number of elements to order. If an algorithm with size $n$ is said to take $n^2$ time, then the same algorithm should take four times longer if the size doubles.

For a numerical method, it is not that easy to express the size of the problem. The problem size for a numerical method depends mostly on two factors, the number of iterations needed to calculate an good enough approximation and the number of function calculations that is performed in each iteration. We therefore get the following formula

$$T_m(n) = I_m(n) * t_m(n),$$

where $T$ is the total time, $I$ the number of iterations and, $t$ the time for each iteration, $m$ for the method and $n$ for the correctness in the approximation. The hard parts with this way are, how are the $I$ function and the $t$ function defined analytically.

The number of iterations ($I$) depends on the convergence rate for the method. A method with better convergence rate will use less iterations to get an approximation within the allowed error.

The execution time for an iteration ($t$) depends on the method implementation. This is often related to how many function and derivative calculations the method needs during the iteration, but other more complicated calculations in the method may also impact on the execution time.

Number of iterations

While both execution time and analytic analysis are not easy methods for comparing efficiency, we need another way. One way could be to count the number of iterations when testing a method implementation.

This is based on the assumption that $t(n)$ are considered constant for all method implementations. The method implementation that has least iterations is then considered the best method implementation as it has the highest convergence rate.

Number of function evaluations

Another way is to calculate the number of function evaluations that a method implementation needs before it can calculate a new approximation. This is just as easy as the iteration count. It is just to count each function evaluation.

This is based on the assumption that the function evaluations represent mainly the complexity for the method implementation. The method implementation does not add any or adds a constant amount of complexity that can be ignored.

Notable is that this procedure does not take into consideration the number of iterations. It is therefore a bad way to judge the convergence rate for the method. Instead, it shows how much the method depends on the function calculation efficiency. A method that needs few function calculations is more efficient than a method that uses more function calculations if the function calculation takes a large amount of time.
6.1.2 Correctness

When applying a numerical method, the method usually does not return the exact correct answer. Instead, it returns an approximation of the correct answer. In some other cases, it may not return any answer at all.

To test the correctness, we look at three different options. First, there is a comparison if the method acts as expected, then if the answer is correct. We also look on the restrictions for the method to ensure that they are fulfilled.

**Convergence rate correctness**

With this way at looking at correctness, we compare the convergence rate for each method against their known convergence rate. The convergence rate for a method is tested with the formula

\[
0 \leq \lim_{n \to \infty} \frac{|x_n - \alpha|}{(|x_{n-1} - \alpha|)^p} = C < \infty
\]

where \(x_n\) is an calculated approximation, \(x_{n-1}\) is the previously calculated approximation, \(\alpha\) is the correct answer and \(p\) is the known convergence rate. If the test is true then the method does act as expected. This is based on the definition of convergence rate (see Definition 4.1).

**Correct result**

This option looks at the correctness of the approximation. One way to know if an approximation is correct, is to compare the difference between the approximation and the correct result as

\[
\delta = |\alpha - x|
\]

where \(\delta\) is the difference, \(\alpha\) is the correct answer and \(x\) is the calculated approximation. A smaller \(\delta\) represents a better approximation then a larger one.

Another way is to compare the result after a given number of iterations. The method that gives the best approximation after executing \(n\) number of iterations is the best. This way will also reflect the convergence rate for the methods. A better approximation after \(n\) iterations will generally indicate a higher convergence rate. A method with a bad convergence rate could be ‘lucky’ and return a better approximation then a method with a better convergence rate.

**Restrictions**

Each method implies a number of restrictions. These restrictions for a method states for which functions the method can calculate an approximation. These restrictions can be for example on the function, on a library routine, or on a method constant. Generally, the fewer the restrictions are, the better it is.

For some methods, the restriction is only there to ensure that the method is used when it can calculate an approximation. In some cases, the method can calculate the correct approximation when the restrictions are broken, in some other the method cannot.
6.2 Test cases

To test a method and see how well it works we apply different types of functions on the method. That is to see if the method also in practice converges with the expected convergence rate. All minimum points of the different test functions are known, thus we can check if the method really converges with the desired rate. This means that it is possible to check if the rate of convergence of the method is slower or even quicker than in theory.

For a comparison, the functions are also tested with other methods. What we then compare is the number of iterations and the number of function evaluations in the different methods. This is done to determine if the method we use is the best to use on these kind of problems.

These tests help us to detect weaknesses and to make improvements in our method. The improvements have led to different versions of the optimization algorithm. We test the functions on all of these versions.

The first function we test is \( \cos x \) (TC 1) with the interval \([2, 4]\). The function is chosen because it is symmetric about the axis \( x = \pi \) and the first and second derivatives are easily calculated by hand. It is then possible for us to check if there are any basic errors in the implementation of the method. Furthermore, the methods we use for comparison have no difficulties in finding the minimum point. Thus, we are getting a first indication of how efficient our method is.

To study how well our method deals with polynomials, we study the polynomial

\[
x^6 - 12x^5 - 6x^4 + 80x^3 - 15x^2 + 300x + 1000 \quad (TC \ 2).
\]

A sixth degree polynomial is used since our method constructs a fifth degree interpolation polynomial. If a polynomial of degree lesser or equal to five were to be used the minimum point would be found within the first iteration. The test function has one minimum in the interval \([-4, -0.1]\) at \( x = -\sqrt{5} \). The same advantage as with the function \( \cos x \) goes here, that is the first and second derivatives are easily calculated by hand. Errors in our test program are therefore easily detected.
In this work, we study only functions that are unimodal on an interval. Therefore, it is of interest to study functions that are unimodal on the whole real axis. The advantage with the unimodal functions is that it is possible to use different starting intervals as long as the minimum point lies within the interval. The results of the different starting intervals can then be compared. The functions we study are

\[ 2\sqrt{1 + x^2} - x \quad (TC\ 3) \quad \text{and} \quad -\frac{\ln x}{x} \quad (TC\ 4). \]

The first of these unimodal functions attains its minimum at \( x = \sqrt{3} \) and the second function has its minimum at \( x = e \). The intervals tested with are for the first function [-4, 4] and for the second function [0.1, 4]. What is interesting with the later of the two functions is that the Newton-Raphson method does not converge with the starting value \( x = 4 \). In neither of the above functions, the minimum points are hard to find if one should wish to calculate them by hand.

It is also of interest to study how well our method works when the minimum point of the objective function lies a relative short distance away from a vertical asymptote. Studying such a function is perhaps a bit extreme but we can hopefully learn something from studying such a case. The function we have chosen to study is

\[ \frac{2x^2 - x}{x - 1} \quad (TC\ 5) \]

on the interval [1.1, 3]. It has a vertical asymptote at \( x = 1 \) and attains its minimum value at \( x = 1 + \sqrt{2/2} \). We also note that this function is not continuous on the whole real axis. The derivative at the left end point of our interval is a relatively large negative number and the derivative at the right end point is a relatively small positive number. It is interesting to see how our method deals with this fact.
The behaviour of our method when dealing with functions that have a horizontal asymptote is also interesting to study. That is one of the starting points of our method have a first derivative that is close to zero. A function that fulfils this requirement is

\[ xe^{-x^2} \] (TC 6).

It attains its minimum at \( x = -\sqrt{\frac{1}{2}} \) and tends to zero when \( x \) tends to infinity. The interval at which we study this function is \([-3, 0]\).

### 6.3 Numerical method implementations

For the tests, we have implemented three different methods. These are the Newton-Raphson method, the Secant method and the method from chapter 2. The last we have tested in three different implementations.

Each method is presented with a short description of the method, its implementation and any restrictions that the method may imply.

#### 6.3.1 Newton-Raphson

The Newton-Raphson method is a simple method for finding a root to a mathematical function. For finding the root \( x \) to a function \( f(x) \), the Newton-Raphson method says that

\[
x_{i+1} = x_i - \frac{f(x_i)}{f'(x_i)},
\]

where \( x_{i+1} \) is the new approximation and \( x_i \) is the current approximation.

As the method is derived from the Taylor series expansion of a function in the neighbourhood of a point, it has a good convergence rate near the root of the function. Press et al (1992) show that the convergence rate is quadratically. Far from the function root, the Newton-Raphson method can give inaccurate and meaningless results and even result in a loop state. Therefore, Newton-Raphson should only be used within limited intervals near an expected root.

For use as an optimisation method, the function to find the root for is the first derivative. This will of course also give any trapezium points for the function also as roots. The Newton-Raphson method can then be formulated as

\[
x_{i+1} = x_i - \frac{f'(x_i)}{f''(x_i)}.
\]

From this, we see that the availability of both a first and a second derivative of the function are required.
6.3.2 Secant Method

The Secant method is also a method for finding a root to a function. The method assumes that the function is approximately linear near the root of the function and the next approximation is the solution to the line between two points near the root. The new approximation is based on the equation

\[ x_{i+1} = x_i - \frac{f(x_i)(x_i - x_{i-1})}{f(x_i) - f(x_{i-1})}, \]

where \( x_{i+1} \) is the new approximation and \( x_i \) is the current approximation.

The Secant method differs from a false position method (or regula falsi) in that way that the new approximation replaces the oldest of the two points. The convergence rate for the Secant method can be shown to be of the “Golden ratio” of \( \frac{1 + \sqrt{5}}{2} \approx 1.618 \) (Press et al 1992).

To implement the method is also very easy. Compared to the Newton-Raphson method, the Secant method does not need second derivatives. The Secant method instead needs to know two points near the root. If only one point is known, another can be generated by using the Newton-Raphson method for just one iteration, but then the second derivative needs to be known. An alternative is to take a random point near the known point.

6.3.3 Method implementation 1

This method is an optimisation method based on the first and second derivatives in two points enclosing the local maximum or the local minimum. The method is described in chapter 2 and is more complicated than the Newton-Raphson and the Secant methods are.

The implementation of the method first calculates the first derivative polynomial for the scaled interpolating function. Using this first derivative polynomial, the implementation tries to find the roots to the polynomial by using the numerical Laguerre’s method for finding the roots as described in Press et al 1992.

If the polynomial returns any complex roots, they will be disregarded. From the real roots, the root that lies between zero and one will be selected and rescaled back to be the new approximation of the root.

6.3.4 Method implementation 2

Our second implementation of the method from chapter 2 acts in the same way as the first implementation. The only difference is that the solving of the first derivative polynomial. In this second implementation, we changed the Laguerre’s method with an analytical method that solves the first derivative polynomial exact. The analytical method is described previously in chapter 2 and in Beyer 1987.

When solving the first derivative polynomial analytically, we must use complex numbers during the calculations. The implementation of the complex number calculations is based on the STL complex type, see Stroustrup (1998).
6.3.5 Method implementation 3

When observing the results from the first two implementations, we saw that after a few iterations, the new approximation always replaced the approximation from the iteration before.

In our third implementation of the method, we added a check. If the first derivative in the new approximation point was much smaller than the derivative on the other side of the interval, then the other side was also replaced with a new guessed point somewhere between the new approximation and the old point.

The guessed point is the first tenth of the interval between the point to replace and the new approximation that have a correct derivative sign.

6.4 Test result

To run the tests, we have used a normal PC work station equipped with an Intel Pentium III 450 MHz processor and ~200 Mb of RAM memory running Microsoft Windows 98 as operating system. The test system described in section 5.2 were developed as a command line program that does not need any more input from the user then the given on the command line.

The test cases from section 6.2 were executed 100 times for each method implementation as described in section 6.3. The results from the test cases were stored in log text files after the execution. Because the reference methods only use one start approximation, these tests are run twice, each using one of the interval endpoints as the start approximations. Which endpoint that the test execution has used as a start value is given by a suffix after the method name. The suffix low means that the start approximation was based on the lower valued endpoint, that is the left endpoint, while the suffix high means the other higher valued endpoint. An example of this is ‘Secant low’.

The test results for each run were then filtered and compiled with the help of some scripts and resulted in the output in appendix B.
6.4.1 Execution time

The execution time is the difference between the start and the end time of an approximation calculation. The results in Table 6.1 are the average time over the 100 test executions.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>0,32</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>0,23</td>
</tr>
<tr>
<td>Secant low</td>
<td>0,42</td>
</tr>
<tr>
<td>Secant high</td>
<td>0,46</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>1,08</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>2,18</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>0,87</td>
</tr>
</tbody>
</table>

Table 6.1: Execution time (ms)

From Table 6.1 we can see that the implemented reference methods were quicker then all the different method implementations. This may be because of implementation issues, but can also be a result of the more complex calculations in the last methods.

The results for some test case and method pair may be looking faulty. These values are marked in italic in the table. They are the results of that the method failed in the first iteration. The reason could be that the method diverged instead or that no root was found inside the interval.

6.4.2 Iterations

To be able to compare the number of iterations both the reference methods and the different method implementations need to generate correct approximation results. The tests that did not generate a correct approximation are marked in italic in the table below.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>7</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>5</td>
</tr>
<tr>
<td>Secant low</td>
<td>5</td>
</tr>
<tr>
<td>Secant high</td>
<td>5</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>5</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>5</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>3</td>
</tr>
</tbody>
</table>

Table 6.2: Number of Iterations

From the values in Table 6.2 we can see that the different implementations of the method from chapter 2 are using fewer iterations then the reference methods. We can also see that the optimized third implementation can use fewer iterations then the other two implementations, but in some cases, as in test case 3, the opposite occurs during the calculation of the new endpoint.
6.4.3 Function evaluations

The last way to judge the efficiency of the methods is to look at the number of function evaluations. During the calculations, the calculations of function and derivative values were counted and recorded. The result was as in Table 6.3.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>0/14</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>0/10</td>
</tr>
<tr>
<td>Secant low</td>
<td>0/12</td>
</tr>
<tr>
<td>Secant high</td>
<td>0/12</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>13/18</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>15/18</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>7/16</td>
</tr>
</tbody>
</table>

Table 6.3: Number of function/derivative evaluations

These numbers are not comparable, because they depend on the number of iterations. Therefore, we look at the average number of derivatives and function evaluations for each iteration. This results in Table 6.4 for derivative evaluations for each iteration and Table 6.5 for function evaluations for each iteration.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>2</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>2</td>
</tr>
<tr>
<td>Secant low</td>
<td>2.4</td>
</tr>
<tr>
<td>Secant high</td>
<td>2.4</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>3.6</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>3.6</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>5.33</td>
</tr>
</tbody>
</table>

Table 6.4: Number of derivative evaluations/iteration

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>0</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>0</td>
</tr>
<tr>
<td>Secant low</td>
<td>0</td>
</tr>
<tr>
<td>Secant high</td>
<td>0</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>2.6</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>3</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>2.33</td>
</tr>
</tbody>
</table>

Table 6.5: Number of function evaluations/iteration

From Table 6.4 and Table 6.5 we see that the tested method uses function evaluations as well as more derivative evaluations than the reference methods. This indicates that if the function and derivative evaluations take a considerable time, then would the reference methods be a much better and a much more efficient choice.
6.4.4 Correctness

To look at the correctness of the tests we calculate the difference between the approximation and the correct answer.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
<th>Correct results</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
<td>TC 2</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>4,10E-10</td>
<td>5,00E-10</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>4,10E-10</td>
<td>4,47</td>
</tr>
<tr>
<td>Secant low</td>
<td>4,10E-10</td>
<td>5,00E-10</td>
</tr>
<tr>
<td>Secant high</td>
<td>4,10E-10</td>
<td>N.A.</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>2,4E-09</td>
<td>5,00E-10</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>5,4E-09</td>
<td>5,00E-10</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>4,10E-10</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

Table 6.6: Error of the last approximation

In Table 6.6 the error of the approximation is given. From the table, we can see that some start values for the reference methods results in wrong and in some case incorrect approximations.

It is also interesting to notice that the method implementations approximations are either equally or less accurate as the reference methods approximations. This is a result of the end condition rule in the implementation. The end condition rule have been set to be

\[ |x_n - x_{n-1}| < 10^{-11}, \]

where \( x_n \) is the latest approximation, and that \( x_{n-1} \) is the previous approximation. With this end condition, we think that we at least should be able to get an approximation with a maximum error of \( \pm 10^9 \) from our method implementations.

From Table 6.6 we can see examples of the importance of the start approximation for the two reference methods. Both solve two out of three when starting from the lower approximations, while only one out of six for the higher start approximation.

We can also from Table 6.6 see that none of the methods were able to find a correct approximation for test case 6. Only the method implementations were able to find a correct approximation for test case 3, but also that the method implementations can also stop too early to be able to give an acceptable approximation. This happens in test case 1 for the two first implementations and in test case 5 for the method implementation 2. The reason for this behaviour is the same that we saw before creating the last implementation. We can also see that the Newton-Raphson method in test case 2 finds a solution. The found solution is the conjugate solution for the function and is a local maximum.
6.4.5 Convergence rate

To be able to see if the method implementations have the same convergence rate as the method should, we have used the formula from definition 4.1. The calculated series of values for each test case, method and iteration that found a solution give an indication of the upper bound for the values. The test cases where the method implementation does not find a solution is not considered because the approximation could be of any value, even infinity, and that would not mean anything. This leaves us with 19 out of 42 possible values to consider found in the previous section. The calculations are done in appendix C and Table 6.7 contains the upper bound for the results.

<table>
<thead>
<tr>
<th>Method Implementation</th>
<th>Test case</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>TC 1</td>
</tr>
<tr>
<td>Newton-Raphson low</td>
<td>0,80066</td>
</tr>
<tr>
<td>Newton-Raphson high</td>
<td>0,4063359</td>
</tr>
<tr>
<td>Secant low</td>
<td>2,1871144</td>
</tr>
<tr>
<td>Secant high</td>
<td>0,1339951</td>
</tr>
<tr>
<td>Method impl. 1</td>
<td>1,557E15</td>
</tr>
<tr>
<td>Method impl. 2</td>
<td>3,724E15</td>
</tr>
<tr>
<td>Method impl. 3</td>
<td>0,3501991</td>
</tr>
</tbody>
</table>

Table 6.7: Upper bound for calculated values

From the calculations for the test case and method, we can see that the last iterations would not have a correct convergence rate. This is typically a result of numerical errors generated by the method implementation and the number representation as well as from the convergence rate calculations. As the approximation approaches the correct answer, the error inflicted of the number representation gets increasingly significant and consequently the convergence rate will decrease. This is also illustrated in the test cases where the method implementations stopped too early. In this cases the upper bound value is as high (or higher) then the ignored values. This is an indication that the methods does not have the expected convergence rate in this context.

In some other cases, the series is to short to be able to read something from it. That is the case for the third method implementation for test case 1. After removing the last iteration because of the significant error, there exists only one remaining value.

6.4.6 Restrictions

We also wanted to take a quick look at what restrictions each method set on the mathematical problem. All of our tested methods require that the function must be able to calculate the first derivative. For the tested method, we also need to be able to calculate the function value and the second derivative.

The reference methods also needed start values that were near the solution in order to converge to the correct solution. Otherwise, the method could diverge instead.

The tested method needed an interval that contained at least one minimum. That means that the functions first derivative at the end points of the interval must have opposite signs.
6.5 Conclusions

In the previous sections we have looked at some tests strategies and test cases and presented the results from executing these tests. What have we learned from that?

Initially, we first used a general function implementation as described in section 5.2.5 which interpreted the mathematical function from a string. This resulted in that the system behaved nondeterministic and the results were different between different runs. The reason behind this nondeterministic behaviour was the caching of calculations that was done. Instead, we implemented each test case as an own class. This change resulted in an improvement for the reference methods, because they were more depending on the derivative calculation and that was in this case improved.

After this change, the tests were executed again. The results were recorded from the tests as in appendix B. From our tests, we have concluded that the reference methods are faster and therefore also more efficient then the tested method, even tough the tested method has a third degree of convergence. This is illustrated in Table 6.1. One indication of the reason for this is given in Table 6.4 and Table 6.5 as the tested method needs many more function and derivative evaluations than the reference methods. Another reason is that the tested method implies more calculations that are mathematical by the method itself than the reference methods as seen in section 6.4.3.

We could also see in Table 6.6 that regardless of the method, the numbers of correct results were few. The reference methods solved as an average little less than half of the test cases (2.5 of 6). The tested method implementations were slightly better with an average of 2.66 of six test cases. One implementation even solved more than half of the test cases.

These results were not as good as we had hoped from the start. We had hoped that the tested methods would have solved more test cases than the reference methods. We knew from the start that some of the selected test cases are very hard to find a solution for. But the first two test cases we think is quite easy. For these two test cases we expected that, regardless of method, the correct solution should have been found. That was not the case and that is of cause a disappointing result, but it also illustrates some of the weaknesses of the tested methods.

The reasons for implementing and executing the tests were to see if the methods had the expected convergence rate. In a few cases, we could see in the results indications that the method implementations 1 and 2 do not have the expected convergence rate for the situation where the approximation is found only on one side of the interval. Otherwise, it is very hard to say from our results whether the methods have the correct convergence rate or not because the numbers of values in the number series are quite few. In some cases, the numbers of values were only one or two values after ignoring the last iteration. We could also see that iterating more would not have solved this, as the error in the computers numerical representation would inflict too much error into the calculations.

When looking at the numbers we can see some parts that support that the method has a third rate of convergence. That is that compared to the reference methods, the method uses less iterations as seen in Table 6.2. This indicates that the method converges faster then the reference methods. As the reference method Newton-Raphson has a second rate of convergence, the tested method should have a better rate than that. However, we can say if the method does not need to have a third degree convergence rate in some implementations. The method needs to handle the case where the new approximation is always on the same side as the previous. If this case is handled, then the method has the possibility to have a convergence rate of three.
7 Summary and Final Conclusions

In this report, we have derived that a method for finding the minimum of a unimodal function has a third rate of convergence. To find the optimal point $\mu$ we use an interpolation polynomial, $P(x)$, of degree five. When constructing this polynomial we use the function value at two points and the first and second derivatives at these points. The points are such that they are placed on each side of $\mu$. For the interpolation polynomial, we also assume that $|P''| > D > 0$ in the studied interval.

To be able to prove that the method has a third rate of convergence we begin by showing how to differentiate divided differences. An expression for the interpolation error is also derived and is found to be $f(x) - P(x) = f^{(6)}(\xi)w(x)/6!$. By using the method for differentiating divided differences on this expression we find a recurrence formula for how the error $\varepsilon_i$ in respect to $\mu$ changes. We get the following expression

$$\varepsilon_{i+1} = M_i \left( e_{i+1} e_{i}^2 + e_{i+1} \varepsilon_{i+1} + e_{i} \varepsilon_{i+1} \right) + N_i e_{i} \varepsilon_{i+1} \varepsilon_{i}^2.$$

Next, we show that the error $\varepsilon_i$ tends to zero as $i \to \infty$. First, we study a special case to prove the convergence. In this case, it is assumed that the estimated value of $\mu$ always occurs to the left of $\mu$. When this special case has been proved we also prove the more general case where the estimated values can be attained on both sides of $\mu$.

Then, we re-write the recurrence formula and use this to prove that the method converges with rate at least three. To prove that the method has at least convergence rate three we use definition 4.1.

With this result, the first of our two problems were solved. To answer the second problem, we tested the method against two reference methods. The reference methods used were the Newton-Raphson and the Secant methods. The test was done by implementing and running a number of test cases in a small computer program. The results in section 6.4 indicate that the method has a better convergence rate then the two reference methods.

During the testing, we found out that the method needed some improvements in the case where the approximation always comes on the same side of the interval. We tested this case with a solution that guessed a new approximation in the other side of the interval. We think this implementation does show a more correct convergence rate then the other implementations. However, it was very hard to tell if a method had the expected convergence rate of three or not, because the methods found an answer approximation within a small number of iterations.

We also noticed in Table 6.1 that the reference methods were usually faster than the method. The reason for this behaviour is that the reference methods needed less computational power. This can be seen in Table 6.3, Table 6.4 and Table 6.5 where the reference methods use less function and derivative calculations than the method.

Another reason is that the test cases resulted in few iterations for both the method as well as for the reference methods and this makes the higher convergence rate for the method not as important. If the method should have been better than the reference methods, it would have needed to use fewer computations in each iteration or to have a higher convergence rate than three to reduce the number of iterations even more.

With that we have showed that the derived method has a convergence rate of three, but the complexity of the method makes it not practical useful.
The approach of using a fifth degree polynomial as an interpolating function is clearly not fruitful. For future studies of this subject we suggest that a search for other interpolating functions, \( g(x) \), with the conditions \( f^{(r)}(\alpha) = g^{(r)}(\alpha) \) and \( f^{(r)}(\beta) = g^{(r)}(\beta) \) for \( r = 0, 1, 2 \), is carried out. The aim being to find interpolating functions \( g(x) \), in which the coefficients are easy to find and where the solution \( g'(x) = 0 \) also is easily found.

We think that it could have been interesting to compare the tested method against another method with a higher convergence rate. We did not try to find such a method, because the tested method needed very few iterations. If another method with a higher convergence rate were to be tested, the result would have been that even fewer iterations are needed to find an equally good approximation. From that result, it would be impossible to draw any conclusions.

To be able to perform such test, better approximations are needed. That is not possible without changing the numerical representation. Changing the numerical representation could also result in a decreasing efficiency. This depends on slow calculations due to new complexity of the system that another numerical representation would imply.
8 Acknowledgement

The authors would like to thank Habil. Dr. Prof. Antanaz Zilinskas for all the help and inspiration received during this work. We would also like to thank Mathias Hedenborg and Börje Nilsson for the help, the support and the pushing to get this report finally completed.

We would also like to thank Cap Gemini, Europolitan and Gatespace for letting us use different software to implement the test program and to write this report.
9 References

9.1 Books
Shaw Mary & Garlan David 1996 *Software architecture – Perspective on an emergin dicipline* Upper Saddle River: Prentice Hall, Inc.


### 9.2 Papers


Cockburn Alistair 1999 *A Methodology Per Project* Humans and Technologies Inc. webpage: [http://member.aol.com/humansandt/papers/methyperproject/methyperproject.html](http://member.aol.com/humansandt/papers/methyperproject/methyperproject.html)


### 9.3 Internet resources


Appendix A  System interfaces

The developed system has four important public interfaces. Here we give these interfaces and the public methods they declare. Nothing of the internals of the implementation is given. For a deeper understanding see the supplied code. The comments in the code are made to conform to the JavaDoc style.

A.1 MethodInterface interface

```cpp
/**
 * The different states a calculation can be in
 */
enum CalculationStates {
    NotInitiated,
    Initiated,
    Halted,
    Solved,
    Failed
};

/**
 * MethodInterface is the interface that connects the
 * method with the function and controls the iterations.
 */
class MethodInterface {
    public:
        /**
         * Constructor
         * @param step The number of iteration between
         * halts.
         */
        MethodInterface( int step = -1 );

        /**
         * Destructor.
         * The method object is not cleaned up in this method.
         */
        virtual ~MethodInterface();

        /**
         * Changes the method that is used.
         * @param method The new method to use
         * @return The old method used
         */
        Method* SetMethod( Method& method );

        /**
         * Get the method object that is used.
         * @return The method that is used.
         */
        const Method* GetMethod() const;

        /**
         * Start calculations. The first time the method Init
         * is called in the method.
         * @exception If something fails
         */
        double Calculate() throw (Exception);

        /**
         * Reset calculation state to 'NotInitiated'. The state can be
         * 'NotInitiated', 'Initiated', 'Solved'.
         */
        void Reset();

        /**
         * Changes the current state with a new value
         */
```
** @param state The new state
**/  
void SetState( NVData& state );

/**
 ** Return the object that holds current state
 ** @return The state object
 **/  
const NVData* GetState() const;

/**
 ** Changes the number of iterations between halts.
 ** @param step The number of calculations before stop.
 ** Use -1 to stop only when calculation state
 ** is 'Solved'.
 **/  
void SetStep( int step );

/**
 ** Get the number of iterations between halts.
 ** @return The number
 **/  
int GetStep() const;

/**
 ** Check if the calculation state is 'Solved'
 ** @return True if calculation state is 'Solved'.
 **/  
bool IsSolved() const;

/**
 ** Check if the calculation state is 'Failed'
 ** @return True if calculation state is 'Failed'.
 **/  
bool IsFailed() const;

/**
 ** Check if the calculation state is 'Halted'
 ** @return True if calculation state is 'Halted'.
 **/  
bool IsHalted() const;

/**
 ** Return the current iteration
 ** @return The current iteration
 **/  
int GetIteration() const;

/**
 ** Return the Function that the method uses to calculate
 ** function and derivative values with
 ** @return The function object
 **/  
const Function* GetFunction() const;

/**
 ** Change the function object
 ** @param function The new function
 ** @return The old function
 **/  
Function* SetFunction( Function& function );

/**
 ** Get the number of times the function has been evaluated.
 ** @return A number.
 **/  
int GetFunctionEvaluations() const;
/**
 * Get the number of times a derivative have been evaluated.
 * @return A number.
 */
int GetDerivativeEvaluations() const;

/**
 * Get the number of operations to evaluate a function
 * @return A number
 */
int GetOperations() const;

/**
 * Convenience method. Calculate a derivative value.
 * @param value The x value
 * @param error The minimum error for the calculation
 * @param degree The derivative degree
 * @param type The derivative type
 * @return The function value at value
 * @exception ProcessException If function does not have a value
 */
double CalculateFunction( double value, double error ) const;

/**
 * Convenience method. Calculate a derivative value.
 * @param value The x value
 * @param error The minimum error for the calculation
 * @param degree The derivative degree
 * @param type The derivative type
 * @return The function value at value
 * @exception ProcessException If function does not have a value
 */
double CalculateDerivative( double value, double error, int degree, const DerivativeType type ) const;

};  // class MethodInterface

A.2 Method interface
/**
 * The Method class declares an interface that
 * all method implementations should conform to.
 * class Method {
 * public:
 */
/**
 * Constructor
 */
Method();

/**
 * Destructor
 */
virtual ~Method();

/**
 * Set the MethodInterface that uses this method.
 * Used by Interface class to set the ref to it.
 * @param parent The interface to use
 */
void SetParent( const MethodInterface* parent );

/**
 * Return a name of the method. If not overridden it
 * returns the string "Unknown".
 * @return The given name
 */
virtual String GetMethodName() const;
/**
 * This method should be overridden if external libraries are used and is not instanciated directly.
 */
virtual void CreateLibraries();

/**
 * This method should be overridden if external libraries are used and is not deleted directly.
 */
virtual void DestroyLibraries();

/**
 * Only called before the calculation is started.
 **
 * @param data The current state
 * @return The initiated state
 */
virtual NVData* Init( const NVData* theData);

/**
 * Calculate an iteration value. The data object is used to store the state of the calculations.
 **
 * Operation that must be overridden
 **
 * @param data The current state
 * @return The new state.
 * @exception MethodException To indicate an error.
 */
virtual NVData* Calculate( const NVData* theData) throw( MethodException ) = 0;

/**
 * Check if the end condition is fullfilled.
 **
 * @param previous The state before calculation
 * @param after The state after calculation
 * @param condition The end condition
 * @return True if calculations should stop.
 * @exception MethodException Indication of an error.
 */
virtual bool CheckEndCondition( const NVData* previous, const NVData* after, const double condition ) const throw( MethodException );

}; //class Method

A.3 Function interface

/**
 * DerivativeType is the different numerical derivatives that exists.
 */
enum DerivativeType {
    Left,
    Center,
    Right
};

/**
 * Function defines the general interface that the Method part have to calculate function and derivative values. The Function class also collects statistics of evaluations.
 */
class Function {
public:
/**
 * Constructor
 */
Function();

/**
 * Destructor
 */
virtual ~Function();
A.4 NVData

/**
** A NVRecord is a Name-Value record
**/

class NVRecord {
public:

  /**
   ** The name
   **/
  String myName;

  /**
   ** The value
   **/
  double myValue;

  /**
   ** Constructor
   **/
};
** @param name The name of the record
** @param value The value for the record
**/
NVRecord( const String& name, const double& value );
/**
** Copy constructor
**
** @param source The NVRecord to copy
**/
NVRecord( const NVRecord& source );
/**
** No arg constructor
**/
NVRecord();
};
/**
** The class NVData holds a list of NVRecords.
** To iterate over the list use NVDataIterator.
**/
class NVData {
public:
/**
** Constructor. Creates a empty list.
**/
NVData();
/**
** Copy constructor. Performs a deep copy
** of the list
**
** @param source The list to copy
**/
NVData(const NVData& source );
/**
** Destructor. Cleans up the list and
** destroys all NVRecords.
**/
~NVData();
/**
** Creates or changes a NVRecord.
**
** @param name  The name of the NVRecord
** @param value The value the NVRecord should hold.
**/
void SetAttribute( const String& name, const double& value );
/**
** Return the value stored in the NVRecord with name.
**
** @param name  The name of the NVRecord.
** @return The value in the NVRecord
** @exception NVDataException If no NVRecord with name found.
**/
double GetAttribute( const String& name) const
    throw( NVDataException );
/**
** Removes and destroys the NVRecord with name
**
** @param name  The name of the NVRecord.
** @exception NVDataException If no NVRecord with name found.
**/
void RemoveAttribute( const String& name) throw (NVDataException);
Appendix B  Output from test cases
This appendix contains all output from the test runs. In these tables the columns are:
  N is the number of times the same data comes up.
  I is number of iterations
  FE is number of function evaluations.
  DE is number of derivative evaluations.
  State is any of Solved or Failed.
  Approximations is a list of all approximations

B.1 Output from test case 1
Test run for: Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 7, 0, 14, Solved, 3.141592654, 3.141592654, 3.141592654, 3.140943912, 3.266186278, 2.467893675, 4.185039863
Total Time: 32
Stop Newton-Raphson Lower execution

*********
Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 5, 0, 10, Solved, 3.141592654, 3.141592654, 3.141592387, 3.15087294, 2.842178718
Total Time: 23
Stop Newton-Raphson Upper execution

*********
Start Secant Lower execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 5, 0, 12, Solved, 3.141592654, 3.141592654, 3.141592395, 3.145895397, 3.120336387
Total Time: 42
Stop Secant Lower execution

*********
Start Secant Upper execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 5, 0, 12, Solved, 3.141592654, 3.141592654, 3.14159269, 3.14124347, 3.166882432
Total Time: 46
Stop Secant Upper execution

*********
Start Method 1 execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 5, 13, 18, Solved, 3.141592656, 3.141592656, 3.141592654, 3.141592659, 3.142646788
Total Time: 108
Stop Method 1 execution

*********
Start Method 2 execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 5, 15, 18, Solved, 3.141592659, 3.141592659, 3.141592653, 3.141592659, 3.142646787
Total Time: 218
Stop Method 2 execution

*********
Start Method 3 execution:
N, I, FE, DE, State, Approximations
----------------------------------------------
100, 3, 7, 16, Solved, 3.141592654, 3.141592654, 3.142646788
Total Time: 87
Stop Method 3 execution
**B.2 Output from test case 2**

Test run for:
Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------
100, 8, 0, 16, Solved, -2.236067977, -2.236067977, -2.236068026, -2.236282967, -2.250507746, -2.362708754, -2.672011695, -3.212157689
Total Time: 33
Stop Newton-Raphson Lower execution

******
Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------
100, 8, 0, 16, Solved, 2.236067977, 2.236067977, 2.236068427, 2.236800133, 2.266109372, 2.447751289, 2.915197431, 3.791814672
Total Time: 32
Stop Newton-Raphson Upper execution

******
Start Secant Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------
100, 10, 0, 22, Solved, -2.236067977, -2.236067977, -2.236067978, -2.236068506, -2.236197231, -2.239968879, -2.26835719, -2.363217645, -2.548579349, -2.868448328
Total Time: 75
Stop Secant Lower execution

******
Start Secant Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------
Process exception: Method exhaustion.
1, 1000, 0, 2002, Failed
Total Time: 72
Stop Secant Upper execution

******
Start Method 1 execution:
N, I, FE, DE, State, Approximations
------------------------------------------
100, 4, 6, 13, Solved, -2.236067977, -2.236067977, -2.236069, -2.244386185
Total Time: 77
Stop Method 1 execution

******
Start Method 2 execution:
N, I, FE, DE, State, Approximations
------------------------------------------
100, 4, 6, 12, Solved, -2.236067977, -2.236067977, -2.236069, -2.244386185
Total Time: 177
Stop Method 2 execution

******
Start Method 3 execution:
N, I, FE, DE, State, Approximations
------------------------------------------
Method exception: No root inside interval!
1, 4, 8, 16, Failed
Total Time: 1
Stop Method 3 execution

**B.3 Output from test case 3**

Test run for:
Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------
Method exception: Approximation have reached +/- Infinity
1, 4, 0, 8, Failed
Total Time: 1
Stop Newton-Raphson Lower execution

******
Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
Method exception: Approximation have reached +/- Infinity
1,  4,  0,  8,  Failed
Stop Newton-Raphson Upper execution

*********
Start Secant Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 6, 0, 14, Solved, 3.128381719e+015, 3.128381719e+015, 4.171175626e+015, -75714.45832, -
303119.3109, 72.8963706
Total Time: 50
Stop Secant Lower execution

*********
Start Secant Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 6, 0, 14, Solved, -128767187.2, -128767187.2, -515071878.2, 874.9281984, 1174.216356, -
3.866221064
Total Time: 47
Stop Secant Upper execution

*********
Start Method 1 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 6, 12, 20, Solved, 0.5773502692, 0.5773502692, 0.5773504291, 0.5697260916, 0.8900022288,
2
Total Time: 122
Stop Method 1 execution

*********
Start Method 2 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
Method exception: No roots found.
1,  2,  3,  7,  Failed
Total Time: 1
Stop Method 2 execution

*********
Start Method 3 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 8, 18, 28, Solved, 0.5773502692, 0.5773502692, 0.5773502692, 0.5773502692, 0.5773502693,
0.5697260916, 0.8900022288, 2
Total Time: 193
Stop Method 3 execution

B.4 Output from test case 4
Test run for:
Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 16, 0, 32, Solved, 2.718281828, 2.718281828, 2.718281469, 2.718226703, 2.692012139,
2.541032621, 2.221690191, 1.816279561, 1.41556839, 1.067778524
Total Time: 74
Stop Newton-Raphson Lower execution

*********
Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
Method exception: Approximation have reached +/- Infinity
1,  2,  0,  4,  Failed
Total Time: 1
Stop Newton-Raphson Upper execution

*********
Start Secant Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 21, 0, 44, Solved, 2.718281828, 2.718281828, 2.718281706, 2.718226703, 2.715870887,
2.693227501, 2.609662893, 2.439641472, 2.197251959, 1.916967634
Total Time: 152
Stop Secant Lower execution

*********
Start Secant Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 1, 0, 4, Solved, -2.794638594
Total Time: 15
Stop Secant Upper execution

*********
Start Method 1 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 711, 3553, 2136, Solved, 2.718281829, 2.718281829, 2.718281828, 2.718281829, 2.718281854, 2.718282434, 2.718285244, 2.718291918, 2.718303388, 2.718320155
Total Time: 17042
Stop Method 1 execution

*********
Start Method 2 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
Method exception: No roots found.
1, 737, 3680, 2211, Failed
Total Time: 358
Stop Method 2 execution

*********
Start Method 3 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
Method exception: No root inside interval!
1, 5, 14, 24, Failed
Total Time: 2
Stop Method 3 execution

B.5 Output from test case 5

Test run for:
Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 11, 0, 22, Solved, 1.707106781, 1.707106781, 1.707106781, 1.707097389, 1.70601655, 1.696005999, 1.660156231, 1.59108925, 1.500221727, 1.406003931
Total Time: 52
Stop Newton-Raphson Lower execution

*********
Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
Method exception: Approximation have reached +/- Infinity
1, 5, 0, 10, Failed
Total Time: 1
Stop Newton-Raphson Upper execution

*********
Start Secant Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 13, 0, 28, Solved, 1.707106781, 1.707106781, 1.707106722, 1.707081182, 1.70601655, 1.696005999, 1.660156231, 1.59108925, 1.500221727, 1.406003931
Total Time: 92
Stop Secant Lower execution

*********
Start Secant Upper execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 7, 0, 16, Solved, -5.786820201e+045, -5.786820201e+045, 6.424661027e+014, -4.992964379e+014, 23948369.02, -3227.471888, 61.33333333
Total Time: 59
Stop Secant Upper execution

*********
Start Method 1 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 63, 317, 193, Solved, 1.707106781, 1.707106781, 1.707106781, 1.707124827, 1.707319904, 1.707991084, 1.711548444, 1.714623217
Total Time: 1291
Stop Method 1 execution

Start Method 2 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 65, 327, 199, Solved, 1.707122665, 1.707122665, 1.707122665, 1.707122665, 1.707122665, 1.707122665, 1.707122665, 1.707122665
Total Time: 2447
Stop Method 2 execution

Start Method 3 execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 5, 16, 32, Solved, 1.707106781, 1.707106781, 1.707106767, 1.705675307, 2.962184981
Total Time: 134
Stop Method 3 execution

B.6 Output from test case 6

Test run for:
Start Newton-Raphson Lower execution:
N, I, FE, DE, State, Approximations
------------------------------------------------------------
100, 47, 0, 94, Solved, 0.5, 0.5, 0.5, 0.4999999999, 0.4999999999, 0.4999999999, 0.4999999999, 0.4999999999, 0.4999999999
Total Time: 227
Stop Newton-Raphson Lower execution

Start Newton-Raphson Upper execution:
N, I, FE, DE, State, Approximations
Method exception: Approximation have reached +/- Infinity
1, 1, 0, 2, Failed
Stop Newton-Raphson Upper execution

Start Secant Lower execution:
N, I, FE, DE, State, Approximations
Process exception: Method exhaustion.
1, 1000, 0, 2002, Failed
Total Time: 76
Stop Secant Lower execution

Start Secant Upper execution:
N, I, FE, DE, State, Approximations
Method exception: Approximation have reached +/- Infinity
1, 1, 0, 4, Failed
Total Time: 1
Stop Secant Upper execution

Start Method 1 execution:
N, I, FE, DE, State, Approximations
Method exception: No roots found.
1, 1, 2, 4, Failed
Total Time: 1
Stop Method 1 execution

Start Method 2 execution:
N, I, FE, DE, State, Approximations
Process exception: Method exhaustion.
Stop Method 2 execution

*********
Start Method 3 execution:
N, I, PE, DE, State, Approximations

Method exception: No roots found.
1, 1, 2, 4, Failed
Stop Method 3 execution
### Appendix C  Calculations of convergence rate

#### C.1 TC1 - Newton-Raphson method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3.1415927</td>
<td>n.a.</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>2,438E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>2,438E+09</td>
</tr>
<tr>
<td>N-3</td>
<td>3.1409439</td>
<td>0.0006487</td>
<td>0.0009747</td>
</tr>
<tr>
<td>N-4</td>
<td>3.2661863</td>
<td>0.1245936</td>
<td>0.0417907</td>
</tr>
<tr>
<td>N-5</td>
<td>2.4678937</td>
<td>0.673699</td>
<td>0.2745137</td>
</tr>
<tr>
<td>N-6</td>
<td>4.1850399</td>
<td>1.0434472</td>
<td>0.6187638</td>
</tr>
<tr>
<td>N-7</td>
<td>2</td>
<td>1.1415927</td>
<td>0.80066</td>
</tr>
</tbody>
</table>

#### C.2 TC1 – Newton-Raphson method, Upper end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3.1415927</td>
<td>n.a.</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>2,438E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>3.1415924</td>
<td>2.666E-07</td>
<td>5771.863</td>
</tr>
<tr>
<td>N-3</td>
<td>3.1508729</td>
<td>0.0092803</td>
<td>0.0030954</td>
</tr>
<tr>
<td>N-4</td>
<td>2.8421787</td>
<td>0.2994139</td>
<td>0.1035184</td>
</tr>
<tr>
<td>N-5</td>
<td>4</td>
<td>0.8584073</td>
<td>0.4063359</td>
</tr>
</tbody>
</table>

#### C.3 TC1 – Secant method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3.1415927</td>
<td>n.a.</td>
<td>1,618</td>
</tr>
<tr>
<td>N</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>632650,88</td>
</tr>
<tr>
<td>N-2</td>
<td>3.1415924</td>
<td>2,586E-07</td>
<td>18,880902</td>
</tr>
<tr>
<td>N-3</td>
<td>3.1458954</td>
<td>0.0043027</td>
<td>0.0017427</td>
</tr>
<tr>
<td>N-4</td>
<td>3.1203364</td>
<td>0.0212563</td>
<td>2,1871144</td>
</tr>
<tr>
<td>N-5</td>
<td>2</td>
<td>1.1415927</td>
<td>0.0171567</td>
</tr>
</tbody>
</table>

#### C.4 TC1 – Secant method, Upper end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3.1415927</td>
<td>n.a.</td>
<td>1,618</td>
</tr>
<tr>
<td>N</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3.1415927</td>
<td>4.102E-10</td>
<td>632650,88</td>
</tr>
<tr>
<td>N-2</td>
<td>3.1415927</td>
<td>3,641E-08</td>
<td>445,59878</td>
</tr>
<tr>
<td>N-3</td>
<td>3.1412435</td>
<td>0.0003492</td>
<td>0.0142745</td>
</tr>
<tr>
<td>N-4</td>
<td>3.1668824</td>
<td>0.0252898</td>
<td>0.1339951</td>
</tr>
<tr>
<td>N-5</td>
<td>4</td>
<td>0.8584073</td>
<td>0.0323764</td>
</tr>
</tbody>
</table>
### C.5 TC1 – Method implementation 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3,1415927</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3,1415927</td>
<td>2,41E-09</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3,1415927</td>
<td>2,41E-09</td>
<td>1,721E+17</td>
</tr>
<tr>
<td>N-2</td>
<td>3,1415927</td>
<td>4,102E-10</td>
<td>3,492E+19</td>
</tr>
<tr>
<td>N-3</td>
<td>3,1415927</td>
<td>6,41E-09</td>
<td>1,557E+15</td>
</tr>
<tr>
<td>N-4</td>
<td>3,1426468</td>
<td>0,0010541</td>
<td>5,4724788</td>
</tr>
</tbody>
</table>

### C.6 TC1 – Method implementation 2

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3,1415927</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3,1415927</td>
<td>5,41E-09</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3,1415927</td>
<td>5,41E-09</td>
<td>3,416E+16</td>
</tr>
<tr>
<td>N-2</td>
<td>3,1415927</td>
<td>5,898E-10</td>
<td>2,637E+19</td>
</tr>
<tr>
<td>N-3</td>
<td>3,1415927</td>
<td>5,41E-09</td>
<td>3,724E+15</td>
</tr>
<tr>
<td>N-4</td>
<td>3,1426468</td>
<td>0,0010541</td>
<td>4,6187786</td>
</tr>
</tbody>
</table>

### C.7 TC1 – Method implementation 3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>3,1415927</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>3,1415927</td>
<td>4,102E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>3,1415927</td>
<td>4,102E-10</td>
<td>5,94E+18</td>
</tr>
<tr>
<td>N-2</td>
<td>3,1426468</td>
<td>0,0010541</td>
<td>0,3501991</td>
</tr>
</tbody>
</table>

### C.8 TC2 – Newton-Raphson method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>-2,236068</td>
<td>n.a.</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>2,001E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>-2,236068</td>
<td>4,85E-08</td>
<td>212471,32</td>
</tr>
<tr>
<td>N-3</td>
<td>-2,236283</td>
<td>0,000215</td>
<td>1,0493228</td>
</tr>
<tr>
<td>N-4</td>
<td>-2,250508</td>
<td>0,0144398</td>
<td>1,0310905</td>
</tr>
<tr>
<td>N-5</td>
<td>-2,362709</td>
<td>0,1266408</td>
<td>0,9003536</td>
</tr>
<tr>
<td>N-6</td>
<td>-2,672012</td>
<td>0,4359437</td>
<td>0,6663658</td>
</tr>
<tr>
<td>N-7</td>
<td>-3,212158</td>
<td>0,9760897</td>
<td>0,4575631</td>
</tr>
<tr>
<td>N-8</td>
<td>-4</td>
<td>1,763932</td>
<td>0,3137083</td>
</tr>
</tbody>
</table>

### C.9 TC2 – Secant method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>-2,236068</td>
<td>n.a.</td>
<td>1,618</td>
</tr>
<tr>
<td>N</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>559950,06</td>
</tr>
<tr>
<td>N-2</td>
<td>-2,236068</td>
<td>5,002E-10</td>
<td>559189,34</td>
</tr>
<tr>
<td>N-3</td>
<td>-2,236069</td>
<td>5,285E-07</td>
<td>7,1658115</td>
</tr>
<tr>
<td>N-4</td>
<td>-2,236197</td>
<td>0,0001293</td>
<td>1,0345024</td>
</tr>
<tr>
<td>N-5</td>
<td>-2,239969</td>
<td>0,0039009</td>
<td>1,0207923</td>
</tr>
<tr>
<td>N-6</td>
<td>-2,268357</td>
<td>0,0322892</td>
<td>1,0081149</td>
</tr>
<tr>
<td>N-7</td>
<td>-2,363218</td>
<td>0,1271497</td>
<td>0,9083964</td>
</tr>
<tr>
<td>N-8</td>
<td>-2,548579</td>
<td>0,3125114</td>
<td>0,8348767</td>
</tr>
<tr>
<td>N-9</td>
<td>-2,868448</td>
<td>0,6323804</td>
<td>0,6559683</td>
</tr>
</tbody>
</table>
### C.10 TC2 – Method implementation 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>-2,236068</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>4,003E+18</td>
</tr>
<tr>
<td>N-2</td>
<td>-2,236069</td>
<td>1,023E-06</td>
<td>467516920</td>
</tr>
<tr>
<td>N-3</td>
<td>-2,244386</td>
<td>0,0083182</td>
<td>1,7765366</td>
</tr>
</tbody>
</table>

### C.11 TC2 – Method implementation 2

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>-2,236068</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>-2,236068</td>
<td>4,998E-10</td>
<td>4,003E+18</td>
</tr>
<tr>
<td>N-2</td>
<td>-2,236069</td>
<td>1,023E-06</td>
<td>467516920</td>
</tr>
</tbody>
</table>

### C.12 TC3 – Method implementation 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>0,5773503</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>5,77E-01</td>
<td>1,037E-11</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>5,77E-01</td>
<td>1,037E-11</td>
<td>9,292E+21</td>
</tr>
<tr>
<td>N-2</td>
<td>5,77E-01</td>
<td>1,599E-07</td>
<td>2,537E+09</td>
</tr>
<tr>
<td>N-3</td>
<td>5,70E-01</td>
<td>0,0076242</td>
<td>0,3608258</td>
</tr>
<tr>
<td>N-4</td>
<td>8,90E-01</td>
<td>0,312652</td>
<td>0,249465</td>
</tr>
<tr>
<td>N-5</td>
<td>2</td>
<td>1,4226497</td>
<td>0,1085843</td>
</tr>
</tbody>
</table>

### C.13 TC3 – Method implementation 3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>0,5773503</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>5,77E-01</td>
<td>1,037E-11</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>5,77E-01</td>
<td>1,037E-11</td>
<td>9,292E+21</td>
</tr>
<tr>
<td>N-2</td>
<td>5,77E-01</td>
<td>1,037E-11</td>
<td>9,292E+21</td>
</tr>
<tr>
<td>N-3</td>
<td>5,77E-01</td>
<td>1,104E-10</td>
<td>7,715E+18</td>
</tr>
<tr>
<td>N-4</td>
<td>5,70E-01</td>
<td>0,0076242</td>
<td>0,0002491</td>
</tr>
<tr>
<td>N-5</td>
<td>8,90E-01</td>
<td>0,312652</td>
<td>0,249465</td>
</tr>
<tr>
<td>N-6</td>
<td>2</td>
<td>1,4226497</td>
<td>0,1085843</td>
</tr>
<tr>
<td>N-7</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### C.14 TC4 – Newton-Raphson method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>2,7182818</td>
<td>n.a.</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>2,7182818</td>
<td>4,59E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>2,7182818</td>
<td>4,59E-10</td>
<td>2,178E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>2,7182815</td>
<td>3,595E-07</td>
<td>3552,6847</td>
</tr>
<tr>
<td>N-3</td>
<td>2,7176564</td>
<td>0,0006254</td>
<td>0,9189959</td>
</tr>
<tr>
<td>N-4</td>
<td>2,6920121</td>
<td>0,0262697</td>
<td>0,9062709</td>
</tr>
<tr>
<td>N-5</td>
<td>2,5410326</td>
<td>0,1772492</td>
<td>0,8361541</td>
</tr>
<tr>
<td>N-6</td>
<td>2,2216902</td>
<td>0,4965916</td>
<td>0,7187626</td>
</tr>
<tr>
<td>N-7</td>
<td>1,8162796</td>
<td>0,9020023</td>
<td>0,6103573</td>
</tr>
<tr>
<td>N-8</td>
<td>1,4155684</td>
<td>1,3027134</td>
<td>0,531508</td>
</tr>
<tr>
<td>N-9</td>
<td>1,0677785</td>
<td>1,6505033</td>
<td>0,4782072</td>
</tr>
</tbody>
</table>
### C.15 TC4 – Secant method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>2,7182818</td>
<td>n.a.</td>
<td>1,618</td>
</tr>
<tr>
<td>N</td>
<td>2,7182818</td>
<td>4,59E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>2,7182818</td>
<td>4,59E-10</td>
<td>590164.8</td>
</tr>
<tr>
<td>N-2</td>
<td>2,7182817</td>
<td>1,225E-07</td>
<td>70,062603</td>
</tr>
<tr>
<td>N-3</td>
<td>2,7182267</td>
<td>5,513E-05</td>
<td>0,9516625</td>
</tr>
<tr>
<td>N-4</td>
<td>2,7158709</td>
<td>0,0024109</td>
<td>0,9483621</td>
</tr>
<tr>
<td>N-5</td>
<td>2,6932275</td>
<td>0,0250543</td>
<td>0,9392785</td>
</tr>
<tr>
<td>N-6</td>
<td>2,6096629</td>
<td>0,1086189</td>
<td>0,9094683</td>
</tr>
<tr>
<td>N-7</td>
<td>2,4396415</td>
<td>0,2786404</td>
<td>0,8586661</td>
</tr>
<tr>
<td>N-8</td>
<td>2,197252</td>
<td>0,5210299</td>
<td>0,8001295</td>
</tr>
<tr>
<td>N-9</td>
<td>1,9169676</td>
<td>0,8013142</td>
<td>0,7456064</td>
</tr>
</tbody>
</table>

### C.16 TC4 – Method implementation 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>2,7182818</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>2,7182818</td>
<td>5,41E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>2,7182818</td>
<td>5,41E-10</td>
<td>3,417E+18</td>
</tr>
<tr>
<td>N-2</td>
<td>2,7182818</td>
<td>4,59E-10</td>
<td>5,592E+18</td>
</tr>
<tr>
<td>N-3</td>
<td>2,7182818</td>
<td>5,41E-10</td>
<td>2,9E+18</td>
</tr>
<tr>
<td>N-4</td>
<td>2,7182819</td>
<td>2,554E-08</td>
<td>3,247E+13</td>
</tr>
<tr>
<td>N-5</td>
<td>2,7182824</td>
<td>6,055E-07</td>
<td>1,15E+11</td>
</tr>
<tr>
<td>N-6</td>
<td>2,7182852</td>
<td>3,416E-06</td>
<td>1,52E+10</td>
</tr>
<tr>
<td>N-7</td>
<td>2,7182919</td>
<td>1,009E-05</td>
<td>3,325E+09</td>
</tr>
<tr>
<td>N-8</td>
<td>2,7183034</td>
<td>2,156E-05</td>
<td>1,007E+09</td>
</tr>
<tr>
<td>N-9</td>
<td>2,7183202</td>
<td>3,833E-05</td>
<td>382948731</td>
</tr>
</tbody>
</table>

### C.17 TC5 – Newton-Raphson method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1,7071068</td>
<td>n.a.</td>
<td>2</td>
</tr>
<tr>
<td>N</td>
<td>1,7071068</td>
<td>1,865E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>1,7071068</td>
<td>1,865E-10</td>
<td>5,361E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>1,7071068</td>
<td>1,865E-10</td>
<td>5,361E+09</td>
</tr>
<tr>
<td>N-3</td>
<td>1,7070974</td>
<td>9,392E-06</td>
<td>2,1147347</td>
</tr>
<tr>
<td>N-4</td>
<td>1,7055016</td>
<td>0,0021052</td>
<td>2,119257</td>
</tr>
<tr>
<td>N-5</td>
<td>1,6753661</td>
<td>0,0317407</td>
<td>2,0895792</td>
</tr>
<tr>
<td>N-6</td>
<td>1,5809775</td>
<td>0,1261293</td>
<td>1,995191</td>
</tr>
<tr>
<td>N-7</td>
<td>1,4467692</td>
<td>0,2603375</td>
<td>1,8609828</td>
</tr>
<tr>
<td>N-8</td>
<td>1,3196122</td>
<td>0,3874946</td>
<td>1,7338257</td>
</tr>
<tr>
<td>N-9</td>
<td>1,2201921</td>
<td>0,4869147</td>
<td>1,6344056</td>
</tr>
</tbody>
</table>
### C.18 TC5 – Secant method, Lower end point

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1.7071068</td>
<td>n.a.</td>
<td>1.618</td>
</tr>
<tr>
<td>N</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>1029561.3</td>
</tr>
<tr>
<td>N-2</td>
<td>1.7071067</td>
<td>5.919E-08</td>
<td>92,327798</td>
</tr>
<tr>
<td>N-3</td>
<td>1.7070812</td>
<td>2.56E-05</td>
<td>1,5911571</td>
</tr>
<tr>
<td>N-4</td>
<td>1.7060166</td>
<td>0.0010902</td>
<td>1,5904538</td>
</tr>
<tr>
<td>N-5</td>
<td>1.696006</td>
<td>0.0111008</td>
<td>1,5853914</td>
</tr>
<tr>
<td>N-6</td>
<td>1.6601562</td>
<td>0.0469506</td>
<td>1,5654491</td>
</tr>
<tr>
<td>N-7</td>
<td>1.5910893</td>
<td>0.1160175</td>
<td>1,531939</td>
</tr>
<tr>
<td>N-8</td>
<td>1.5002217</td>
<td>0.2068851</td>
<td>1,4848203</td>
</tr>
<tr>
<td>N-9</td>
<td>1.4060039</td>
<td>0.3011029</td>
<td>1,4426749</td>
</tr>
</tbody>
</table>

### C.19 TC5 – Method implementation 1

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1.7071068</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>2.874E+19</td>
</tr>
<tr>
<td>N-2</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>2.874E+19</td>
</tr>
<tr>
<td>N-3</td>
<td>1.7071069</td>
<td>1.518E-07</td>
<td>5,332E+10</td>
</tr>
<tr>
<td>N-4</td>
<td>1.7071248</td>
<td>1.805E-05</td>
<td>25833358</td>
</tr>
<tr>
<td>N-5</td>
<td>1.7073199</td>
<td>0.0002131</td>
<td>1864174.8</td>
</tr>
<tr>
<td>N-6</td>
<td>1.7079911</td>
<td>0.0008843</td>
<td>308195,95</td>
</tr>
<tr>
<td>N-7</td>
<td>1.7093602</td>
<td>0.0022534</td>
<td>7,7282,273</td>
</tr>
<tr>
<td>N-8</td>
<td>1.7115484</td>
<td>0.0044417</td>
<td>25716,012</td>
</tr>
<tr>
<td>N-9</td>
<td>1.7146232</td>
<td>0.0075164</td>
<td>10459,471</td>
</tr>
</tbody>
</table>

### C.20 TC5 – Method implementation 2

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1.7071068</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>1.7071068</td>
<td>1.588E-05</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>1.7071227</td>
<td>1.588E-05</td>
<td>3.964E+09</td>
</tr>
<tr>
<td>N-2</td>
<td>1.7071227</td>
<td>1.588E-05</td>
<td>3.964E+09</td>
</tr>
<tr>
<td>N-3</td>
<td>1.7071227</td>
<td>1.588E-05</td>
<td>3.964E+09</td>
</tr>
<tr>
<td>N-4</td>
<td>1.7071227</td>
<td>1.591E-05</td>
<td>3.944E+09</td>
</tr>
<tr>
<td>N-5</td>
<td>1.7071235</td>
<td>1.671E-05</td>
<td>3.407E+09</td>
</tr>
<tr>
<td>N-6</td>
<td>1.7071434</td>
<td>3.663E-05</td>
<td>340204220</td>
</tr>
<tr>
<td>N-7</td>
<td>1.7073191</td>
<td>0.0002124</td>
<td>3824353,3</td>
</tr>
<tr>
<td>N-8</td>
<td>1.7079904</td>
<td>0.0008837</td>
<td>307775,74</td>
</tr>
<tr>
<td>N-9</td>
<td>1.7093597</td>
<td>0.0022529</td>
<td>77280,631</td>
</tr>
</tbody>
</table>

### C.21 TC5 – Method implementation 3

<table>
<thead>
<tr>
<th>Iteration</th>
<th>Values</th>
<th>Error</th>
<th>Convergence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correct</td>
<td>1.7071068</td>
<td>n.a.</td>
<td>3</td>
</tr>
<tr>
<td>N</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>n.a.</td>
</tr>
<tr>
<td>N-1</td>
<td>1.7071068</td>
<td>1.865E-10</td>
<td>2.874E+19</td>
</tr>
<tr>
<td>N-2</td>
<td>1.7071068</td>
<td>1.419E-08</td>
<td>6.534E+13</td>
</tr>
<tr>
<td>N-3</td>
<td>1.7056753</td>
<td>0.0014315</td>
<td>4,8364439</td>
</tr>
<tr>
<td>N-4</td>
<td>2,962185</td>
<td>1.2550782</td>
<td>0,0007241</td>
</tr>
</tbody>
</table>