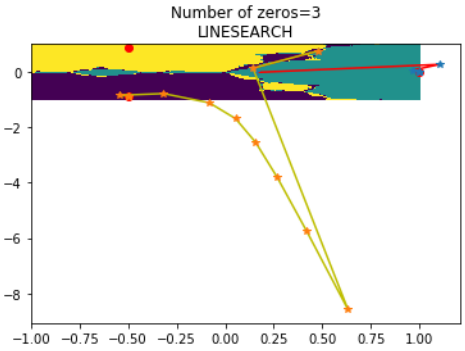
LUND UNIVERSITY
Faculty of Science

MASTER THESIS

Investigation of Line Search Globalization and Scaling Aspects of Newton's Method in Two Industrial Implementations

Amira El Gamal

Supervised by Prof. Dr. Claus Führer



November 6, 2017

Lund University

Abstract

Faculty of Science
Centre for Mathematical Sciences
Numerical analysis

Master in Science

Investigation of Line Search Globalization and Scaling Aspects of Newton's Method in Two Industrial Implementations

by Amira El Gamal

Simulating complex physical systems often requires solving systems of nonlinear algebraic equations. One of the most frequently used numerical methods to solve systems of nonlinear equations is Newton's method with its advantage of quadratic local convergence. However, Newton's method does not guarantee global convergence. This raises the need for combining Newton's method with a globalization strategy. One more problem that affects Newton's method convergence is caused by large differences in the scales of the iteration variables as well as the residuals. Although the Newton iteration is affine invariant, the termination criteria and norm calculations are not. This in turn affects the convergence. In this thesis, we address topics of Newton's method globalization using line search and the scaling of both variables and residuals from theoretical and implementation perspective.

Acknowledgment

I gratefully thank my supervisor Claus Führer for his support and guidance during the thesis work. I thank him also for his continuous encouragement during different stages of my education at Lund University. I would like also to thank my teachers, Carmen Arévalo, Gustaf Söderlind, Philip Birken for their interesting courses and enjoyable lectures in the beautiful field of numerical analysis. I would like also to thank Anna-Maria Persson twice, once as a wonderful teacher and another as a helpful and dedicated student principal.

I thank the department of mathematics, faculty of science, Lund University for giving me the opportunity to study for the master degree in mathematics which was always a dream for me. I also thank Modelon AB for giving me the opportunity to make a practical master thesis. Special thanks to Agnes Ramle and Iakov Nakhimovski for their help during the thesis work. I also thank my country Egypt for the free of charge education from the primary education to MSc degree in computer science.

Last but not least, I would like to thank my best friend and my husband Amr for his containment and support since we met. I like to thank my wonderful lovely kids Reem and Ismail for their patience and love. Finally, I like to express my endless gratitude to my parents Halim and Huda who faced many life difficulties to raise me up.

Contents

1 Introduction **1**
- 1.1 FMUs and Modelica Models . . . 2
- 1.2 Thesis Organization . . . 2

2 Newton's Method: Background and Software Aspects **5**
- 2.1 The Algorithm . . . 5
  - 2.1.1 Linear Model Construction . . . 6
  - 2.1.2 Descent Property of Newton's Direction . . . 7
  - 2.1.3 The Calculation of the Jacobian . . . 9
- 2.2 Convergence Results . . . 10
- 2.3 Software . . . 13
  - 2.3.1 KINSOL (SUNDIALS) . . . 13
  - 2.3.2 OCT-NLESOL (Modelon) . . . 19
- 2.4 Summary . . . 23

3 Globalization Strategy **25**
- 3.1 The Line Search Algorithm . . . 25
- 3.2 Exact Line Search . . . 26
  - 3.2.1 Exact Line Search Methods that do not use Derivatives 27
  - 3.2.2 Exact Line Search Methods That Use Derivatives . . . 28
- 3.3 Inexact Line Search . . . 29
  - 3.3.1 The Backtracking Algorithm . . . 31
- 3.4 Line Search in KINSOL . . . 33
  - 3.4.1 Own Modifications to KINSOL . . . 38
- 3.5 Summary . . . 40

4 Scaling **41**
- 4.1 Scaling of Independent Variables and Residuals . . . 42
- 4.2 Affine Invariant Lipschitz Constant . . . 45

4.2.1 Affine Covariance 46
4.2.2 Affine Contravariance 49
4.3 Software 50
4.3.1 Scaling and Termination Criteria in KINSOL 50
4.3.2 Scaling and Termination Criteria in OCT-NLESOL . . 52
4.4 Summary 53

5 Experiments **55**
5.1 Line Search Experiments 56
5.1.1 The Effect of Line Search Globalization on Newton's Method Convergence 56
5.1.2 Changing the Line Search Parameters 60
5.1.3 Tracing a Point With and Without Line Search 63
5.2 Scaling Experiments 65
5.2.1 Experiments Based on Deuflhard's Approach 66
5.2.2 Residual Scaling of Industrial Models 71
5.3 Summary 73

6 Conclusion and Future work **75**

Bibliography **76**

Chapter 1

Introduction

This thesis was initiated by Modelon AB which often has to face models described by physical quantities covering a wide scale of magnitude. Scaling issues might impact performance of numerical solvers which is one of the topics of this thesis.

One of the products of Modelon AB is OCT, Optimica Compiler Toolkit, a framework which contains a Modelica compiler and several numerical solvers. One of those solvers is a nonlinear equations solver, we will refer to it in this thesis as OCT-NLESOL . This solver is written in C and it is based on the generic open source package KINSOL. KINSOL is a part of a collection of software packages called SUNDIALS, SUite of Nonlinear and DIfferential/ALgebraic equation Solvers. SUNDIALS is developed by Lawrence Livermore National Laboratory, Department of Energy, United States. Despite that OCT-NLESOL is based on KINSOL, it is a customized solver with its own implementation to different functions and constants needed for solving a system of nonlinear equations.

As OCT-NLESOL uses KINSOL line search subroutine as it is, we got a question from Modelon AB of how KINSOL line search subroutine works. Also, we got another question of how the scaling of variables affects the termination criteria and Newton's method convergence. So in this thesis, we investigate the implementation of Newton's method in the two solvers, SUNDIALS KINSOL and Modelon's OCT-NLESOL. We discuss the differences and the similarities of the two solvers with respect to the Jacobian calculation, the linear solver, globalization and scaling.

1.1 FMUs and Modelica Models

Modelon AB is a provider of model-based simulation tools for technical systems. They use the Modelica language for modeling complex physical system. Modelica is an object oriented equation based language developed by a nonprofit organization called the Modelica association. The modeling process is done through a Modelica editor which provides a text editor that may be combined with a graphical user interface. The graphical user interface facilitates the construction of the model hierarchical components. Then the model equations are edited in text mode. The current version of OCT contains only text editing of Modelica models. An example for an editor that contain both graphical and text editing is Dymola, a product of Dassault Systèmes.

In order to exchange Modelica models between different vendors and suppliers, they should be compiled using a Modelica compiler to binary code that follows an open standard. As we mentioned above, OCT has a Modelica compiler which can be accessed from Matlab or Python through the interfaces FMIT (Function Mock-up Interfaces Toolbox) or PyFMI (Python Function Mock-up Interfaces) respectively. FMI is an open standard which is used to specify industrial models for simulation purposes. It facilitates the exchange of models between different suppliers and original equipment vendors. An FMU, Functional Mock-up Unit, is a model that follows the FMI standard. It is a zip file that consists of two parts, a DLL-file that contains the model implementation and an XML-file that contains the model hierarchical description.

1.2 Thesis Organization

Newton's method is an iterative method for solving a nonlinear system of equations. The idea of Newton's method is to approximate the nonlinear residual function at the current iterate with a linear model, find the solution of this linear model, then take this solution as the next iterate. This process is repeated until a termination criteria is fulfilled. In Chapter 2, we discuss the theoretical background and the implementation of Newton's method. We present the algorithm and some of its details like the Jacobian calculation, the descent property of the Newton direction and the local convergence of Newton's method.

The main advantage of Newton's method is its quadratic local convergence. This means that Newton's method has fast convergence in the neighborhood of the solution. In most cases, the solution of the nonlinear system is unknown. This means finding an initial guess in the neighborhood of the unknown solution is quite impossible. Therefore, Newton's method should be combined with a globalization strategy which extends the range of the initial guesses. In Chapter 3, we discuss the globalization of Newton's method using the line search technique. We discuss different line search methods which are categorized as exact and inexact methods. Moreover, we present the backtracking algorithm. The implementation of the key points in KINSOL line search subroutine is also discussed.

As the iteration variables are represented with different units, they may vary drastically in magnitude. The difference in the variable scales does not affect the Newton iteration. However, it affects the termination criteria as it includes norm calculations which in turn affects the convergence of Newton's method. In Chapter 4, we show that the Newton iteration is affine invariant. We also introduce the idea of constructing scaling invariant termination cirteria and convergence monitors based on Deuflhard's approach. Moreover, We discuss how KINSOL and OCT-NLESOL handle the scaling of iteration variables and residuals.

While we investigate the implementation of Newton's method in KINSOL open source code, we did a few modifications in order to see the effect of the line search globalization on Newton's method convergence. In Chapter 5, we show a number of experiments that visualize the globalization effect based on those modifications. Moreover, we present the implementation and testing of two variations of Newton's method based on Deuflhard's approach. We also test the OCT-NLESOL with several industrial benchmark models to see the effect of the residual scaling on Newton's method convergence. In Chapter 6, we present the conclusions and future work.

Chapter 2

Newton's Method: Background and Software Aspects

Newton's method is one of the most frequently used methods to solve a system of nonlinear algebraic equations. In this chapter, we present the algorithm and we discuss some issues like the idea of Newton's method of solving the nonlinear problem by solving a sequence of linear problems, the descent property of Newton direction, and the Jacobian calculation. As Newton's method is well known by its fast local convergence, we present a convergence theorem of Newton's method and its proof. Moreover, we discuss the implementation of the linear solver and the Jacobian calculation in KINSOL and OCT-NLESOL.

2.1 The Algorithm

In this section, we address how we use Newton's method to solve a system of nonlinear equations. Our problem is to find the vector whose components achieve the nonlinear equation system simultaneously. We define the problem as follows:

$$\text{Solve } F(x) = 0 \text{ where } F : \mathbb{R}^n \to \mathbb{R}^n.$$
$$\text{Find } x^* \in \mathbb{R}^n \text{ such that } F(x^*) = 0.$$

where $F(x) = \{F_1(x), F_2(x), \cdots, F_n(x)\}$ is the residual vector resulted by evaluating each residual equation at the vector x.

Newton's method is an iterative method. It begins with an initial guess $x^{(0)}$ and computes a sequence of points that converges eventually to the solution x^*. The idea of the method is to approximate the function F at the current point, for example $x^{(k)}$, by a linear model . Then we solve to find the zero of this linear model by linear algebra methods for solving linear systems of equations. The zero of this linear model is then taken to be the next iterate $x^{(k+1)}$ and the process is repeated until a termination criteria is met, for example, the Newton step is sufficiently small or the function value approaches zero.

2.1.1 Linear Model Construction

Suppose that we are at iteration number k, we expand F about $x^{(k)}$ in a Taylor series and truncate after the linear term

$$F(x) \approx F(x^{(k)}) + F'(x^{(k)})(x - x^{(k)}) = F_{\text{linear}}(x) \tag{2.1}$$

We call the zero of the linear model $x^{(k+1)}$. Thus , $F_{\text{linear}}(x^{(k+1)}) = 0$ and consequently

$$F(x^{(k)}) + F'(x^{(k)})\ s^{(k)} = 0, \tag{2.2}$$

where $s^{(k)} = \Delta x = (x^{(k+1)} - x^{(k)})$ is the Newton direction or Newton step and $F'(x^{(k)})$ is the Jacobian.

$$F'(x^{(k)})\ s^{(k)} = -F(x^{(k)}) \tag{2.3}$$

The next iterate is computed then as

$$x^{(k+1)} = x^{(k)} + s^{(k)},\ k = 0, 1, \cdots \tag{2.4}$$

$x^{(k+1)}$ is an acceptable next iterate if $x^{(k+1)}$ is nearer to the solution than $x^{(k)}$. This can be checked by calculating the vector norm of the residuals. So if $||F(x^{(k+1)})||$ is less than $||F(x^{(k)})||$ for some norm $||.||$, then $x^{(k+1)}$ is an acceptable next iterate.

In the case of using the 2-norm, the problem can be viewed as finding the minimum of the norm function as follows

$$\min_{x \in \mathbb{R}^n} \{f(x) = \frac{1}{2} F(x)^{\mathrm{T}} F(x)\}. \tag{2.5}$$

One can think that it is a good idea to convert the problem from scratch to a minimization problem and solve for the local minimum of the norm function. However, unfortunately this idea does not work properly because not all the local minima of $f(x)$ are zeros of $F(x)$ [1], see Figure 2.1. As we see in the figure, the minimization function $f(x)$ has 3 local minima A,B, and C. Two of them, A and C, correspond to zeros of the residual function $F(x)$ while B has no corresponding zero. So the better idea is to use the original structure of the problem in all the solution steps and to use only the minimization form to check for convergence.

This means we use the Newton direction for solving nonlinear equation system calculated as

$$s^{(k)} = -J(x^{(k)})^{-1} F(x^{(k)}) \tag{2.6}$$

instead of using the Newton direction for solving a minimization problem which is calculated as

$$s^{(k)} = -H(x^{(k)})^{-1} \nabla f(x^{(k)}), \tag{2.7}$$

where $J(x^{(k)})$ is the first derivative or the Jacobian of the residual function $F(x)$, $\nabla f(x^{(k)})$ and $H(x^{(k)})$ are the gradient and the Hessian of the function $f(x)$ respectively.

2.1.2 Descent Property of Newton's Direction

To find a local minimum of a multivariate function using Newton's method, we use the Newton direction in Equation (2.7). As we solve for points where the gradient is equal to zero, Newton's method can converge to a local minimum, a local maximum, or a saddle point. To guarantee that the method converges only to a local minimum, we must guarantee that the Newton direction has a descent property, i.e. the Hessian must be positive definite.

Our goal is to minimize the norm function in Equation (2.5).As we agreed to keep the original problem structure instead of converting to a minimization problem, we should verify that the Newton direction in Equation (2.6) has a descent property and minimizes the norm function $f(x)$. $s^{(k)}$ is a descent direction, if the directional derivative of f at $x^{(k)}$ in the direction of $s^{(k)}$ is negative i.e.

$${s^{(k)}}^{\mathrm{T}} \nabla f(x^{(k)}) \leq 0. \tag{2.8}$$

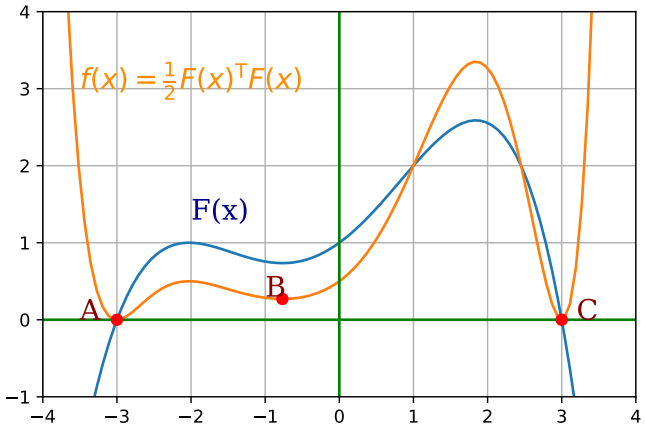


Figure 2.1: A nonlinear equation $F(x)$ and its corresponding minimization function $f(x) = \frac{1}{2}\ F(x)^{\mathrm{T}}\ F(x)$.

This means that the directional derivative makes an obtuse angle with the gradient direction at the point $x^{(k)}$. As the gradient vector points in the direction of the maximal growth of a function, the negative directional derivative in (2.8) means that the Newton direction is pointing in an opposite direction to the gradient and hence it minimizes $f(x)$. The gradient vector for $f(x)$ evaluated at $x^{(k)}$ is derived as follows

$$\nabla f(x^{(k)}) = \frac{1}{2}\frac{\mathrm{d}}{\mathrm{dx}}\sum_{i=1}^{n}(F_i(x^{(k)}))^2 \tag{2.9}$$

$$= \sum_{i=1}^{n} F_i(x^{(k)})\nabla F_i(x^{(k)}) \tag{2.10}$$

$$= \begin{bmatrix}\nabla F_1(x^{(k)}) & \dots & \nabla F_n(x^{(k)})\end{bmatrix}\begin{bmatrix}F_1(x^{(k)})\\ \vdots \\ F_n(x^{(k)})\end{bmatrix} \tag{2.11}$$

$$= J(x^{(k)})^{\mathrm{T}}F(x^{(k)}) \tag{2.12}$$

So the directional derivative is

$$\begin{aligned} \nabla f(x^{(k)})^{\mathrm{T}} s^{(k)} &= -F(x^{(k)})^{\mathrm{T}} J(x^{(k)}) J(x^{(k)})^{-1} F(x^{(k)}) \qquad (2.13) \\ &= -F(x^{(k)})^{\mathrm{T}} F(x^{(k)}) < 0 \qquad (2.14) \end{aligned}$$

as long as $F(x^{(k)}) \neq 0$.

2.1.3 The Calculation of the Jacobian

To calculate the Newton direction at iteration k in Equation (2.6), we need to calculate the first derivative, the Jacobian, of F at $x^{(k)}$. As we mentioned at the beginning of this section, $F : \mathbb{R}^n \to \mathbb{R}^n$ and $F(x) = \{F_1(x), F_2(x), \cdots, F_n(x)\}$. So the Jacobian is

$$J(x^{(k)}) = \begin{pmatrix} \frac{\partial F_1}{\partial x_1} & \frac{\partial F_1}{\partial x_2} & \cdots & \frac{\partial F_1}{\partial x_n} \\ \frac{\partial F_2}{\partial x_1} & \frac{\partial F_2}{\partial x_2} & \cdots & \frac{\partial F_2}{\partial x_n} \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial F_n}{\partial x_1} & \frac{\partial F_n}{\partial x_2} & \cdots & \frac{\partial F_n}{\partial x_n} \end{pmatrix} (x^{(k)}). \qquad (2.15)$$

The Jacobian can be provided analytically as input data to the solver or it can be approximated. Providing the Jacobian analytically can be difficult or sometimes impossible. This might be the case when solving nonlinear equations of some complex models that emerge from industry or physics or when $F(x)$ is not provided analytically, a discrete function for example. In those situation, we have to approximate the Jacobian by finite differences. To approximate the element (i, j) in the matrix by forward differences, we use the following formula

$$J_{i,j} = \frac{F_i(x + h \cdot e_j) - F_i(x)}{h}. \qquad (2.16)$$

We can also use the column-wise formula

$$J_{.j} = \frac{F(x + h_j \cdot e_j) - F(x)}{h_j}, \qquad (2.17)$$

where e_j is the j^{th} unit vector, a vector in $\mathbb{R}^n$ with all components equal zero except the j^{th} element equals 1. h_j is calculated as

$$h_j = \sqrt{\eta} \cdot \max\{|x_j|, \mathrm{typ}_{x_j}\} \cdot \mathrm{sign}(x_j), \qquad (2.18)$$

where η can be the machine precision or it can be given as a user input. typ_{x_j} is the typical magnitude of x_j and

$$\text{sign}(x) = \begin{cases} 1, & \text{if} x \geq 1 \\ \text{-1}, & \text{if} x < 1 \end{cases}.$$

One of the most common problems that affects Newton method is the singularity or the ill-conditioning of the Jacobian at some iteration k which impedes calculating the new search direction. As mentioned in [1] page 89, this problem can be solved by perturbing the Jacobian matrix to make it well-conditioned and proceed with the iteration.

Calculating the Jacobian each iteration may be expensive and impractical. Therefore, some commercial solvers use a modified version of Newton's method which uses outdated Jacobian. This means that the algorithm calculates the Jacobian at the first iteration and keeps using the same matrix for the following iterations and updates the Jacobian every n^{th} iteration where n is a predefined constant.

2.2 Convergence Results

The advantage of Newton's method is its fast local convergence. It has quadratic convergence provided that we start with an initial guess near to the solution. Another advantage, if F is a linear function then the Newton method converges in one step, i.e., the solution is obtained after a single step. In case of F being nonlinear with some linear components, those components converge from the first iteration and the following iterations will modify only the nonlinear components, [1] page 88.

In this subsection, we present a Newton's method convergence theorem and its proof. Before we present the theorem we state the following theorem and lemma that will be needed in the proof of the convergence theorem.

Theorem 1. *Let $\|.\|$ be any norm on $\mathbb{R}^{n\times n}$ that obeys the properties $\|AB\| \leq \|A\|\|B\|$ and $\|\mathrm{I}\| = 1$ and let $E \in \mathbb{R}^{n\times n}$. If $\|E\| < 1$, then $(\mathrm{I} - E)^{-1}$ exists and*

$$\|(\mathrm{I} - E)^{-1}\| \leq \frac{1}{1 - \|E\|}. \tag{2.19}$$

If A is non-singular and $\|A^{-1}(B-A)\| < 1$, then B is non-singular and

$$\|B^{-1}\| \leq \frac{\|A^{-1}\|}{1 - \|A^{-1}(B-A)\|}. \tag{2.20}$$

Remark: All operator norms fulfill the assumptions of Theorem 1.

Lemma 1. *Let $F : \mathbb{R}^n \to \mathbb{R}^n$ be continuously differentiable in the open convex set $D \subset \mathbb{R}^n$, $x \in D$, and the Jacobian matrix J be Lipschitz continuous at x in D, using a vector norm and the induced matrix operator norm and the Lipschitz constant γ. Then, for any $x, p \in D$ also $x + p \in D$ and*

$$\|F(x+p) - F(x) - J(x)p\| \leq \frac{\gamma}{2}\|p\|^2. \tag{2.21}$$

For the proof of Theorem 1 and Lemma 1, see [1]. In the following theorem we will denote the set of all functions which are Lipschitz continuous in a set D with Lipschitz constant γ by $\mathrm{Lip}_\gamma(D)$. Next we state a convergence theorem of Newton's method and its proof.

Theorem 2. *Let $F : \mathbb{R}^n \to \mathbb{R}^n$ be continuously differentiable in an open convex set $D \subset \mathbb{R}^n$. Assume that there exists $x^* \in \mathbb{R}^n$ and constants $r, \beta, \gamma > 0$ such that $N(x^*, r) \subset D$, $F(x^*) = 0$ and $J(x^*)^{-1}$, exists with $\|J(x^*)^{-1}\| \leq \beta$, and $J \in \mathrm{Lip}_\gamma(N(x^*, r))$. Then there exists $\varepsilon > 0$ such that for all $x^{(0)} \in N(x^*, \varepsilon)$, the sequence $x^{(1)}, x^{(2)}, \cdots$ generated by*

$$x^{(k+1)} = x^{(k)} - J(x^{(k)})^{-1}F(x^{(k)}),\ k = 0, 1, 2, \cdots \tag{2.22}$$

is well defined, converges to x^*, and obeys

$$\|x^{(k+1)} - x^*\| \leq \|x^{(k)} - x^*\|^2,\ k = 0, 1, 2, \cdots \tag{2.23}$$

Figure 2.2 sketches the requirements of the theorem and the convergent sequence for better understanding.

Proof. We choose ε such that $J(x)$ is nonsingular for any $x \in N(x^*, \varepsilon)$, so we let

$$\varepsilon = \min\left\{r, \frac{1}{2\gamma\beta}\right\} \tag{2.24}$$

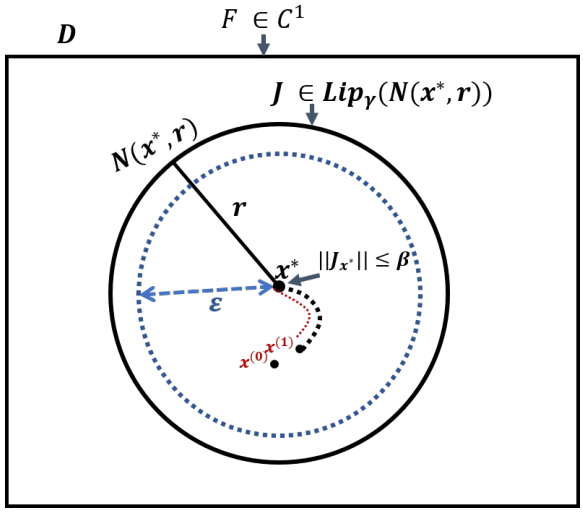


Figure 2.2: Requirements for achieving quadratic convergence

The proof is by induction. For $k = 0$, we choose $x^{(0)} \in N(x^*, \varepsilon)$. Firstly, we want to proof that $x^{(1)}$ is well defined i.e. $J(x^{(0)})$ is nonsingular.
As $\|x^{(0)} - x^*\| < \varepsilon$, J is Lipschitz continuous at x^*, and $\varepsilon = \min\left\{r, \frac{1}{2\gamma\beta}\right\}$ it follows

$$\|J(x^*)^{-1}[J(x^{(0)}) - J(x^*)]\| \leq \|J(x^*)^{-1}\|\|J(x^{(0)}) - J(x^*)\| \tag{2.25}$$
$$\leq \beta\gamma\|x^{(0)} - x^*\| \tag{2.26}$$
$$\leq \beta\gamma \quad \varepsilon = \frac{1}{2} \tag{2.27}$$

From Theorem 1 it follows that $J(x^{(0)})$ is nonsingular and

$$\|J(x^{(0)})^{-1}\| \leq \frac{\|J(x^*)^{-1}\|}{1 - \|J(x^*)^{-1}[J(x^{(0)}) - J(x^*)]\|} \tag{2.28}$$
$$\leq 2\|J(x^*)^{-1}\| \qquad \leq 2\beta. \tag{2.29}$$

Hence, $x^{(1)}$ is well defined.
Now we want to prove that the convergence rate is quadratic.

$$
\begin{aligned}
x^{(1)} - x^* &= x^{(0)} - J(x^{(0)})^{-1}F(x^{(0)}) - x^* && (2.30)\\
&= x^{(0)} - x^* - J(x^{(0)})^{-1}[F(x^{(0)}) - F(x^*)] && (2.31)\\
&= J(x^{(0)})^{-1}[F(x^*) - F(x^{(0)}) - J(x^{(0)})(x^* - x^{(0)})] && (2.32)
\end{aligned}
$$

The term between brackets is the difference between $F(x^*)$ and the approximated linear model around $x^{(0)}$ evaluated at x^*. By taking the norm, we get

$$\|x^{(1)} - x^*\| \leq \|J(x^{(0)})^{-1}\|\|F(x^*) - F(x^{(0)}) - J(x^{(0)})(x^* - x^{(0)})\| \quad (2.33)$$

and by applying Lemma 1

$$\|x^{(1)} - x^*\| \leq 2\beta\frac{\gamma}{2}\|x^* - x^{(0)}\|^2 \leq \beta\gamma\|x^* - x^{(0)}\|^2. \quad (2.34)$$

It remains to prove that $x^{(1)} \in N(x^*, \varepsilon)$.
Since $\|x^* - x^{(0)}\| \leq \varepsilon$, it follows that

$$\|x^{(1)} - x^*\| \leq \frac{1}{2}\|x^{(0)} - x^*\| \quad (2.35)$$

since $x^{(1)}$ has less than the distance to x^* than $x^{(0)}$, then $x^{(1)} \in N(x^*, \varepsilon)$. The proof of the induction step proceeds identically replacing $x^{(0)}$ by $x^{(k)}$ and $x^{(1)}$ by $x^{(k+1)}$. □

2.3 Software

In this section, we discuss some implementation issues of Newton's method in both Kinsol and OCT-NLESOL such as the Jacobian calculations, the linear system solver, and the overall structure of a user program that uses the KINSOL or OCT-NLESOL to solve a nonlinear system of equations.

2.3.1 KINSOL (SUNDIALS)

As we mentioned in Chapter 1, KINSOL is a software package which is based on Newton's method for solving nonlinear systems of equations. It implements the following algorithm:

1. Set $x^{(0)} =$`<initial guess>`.
2. For $k = 0, 1, 2, \cdots$ until termination criteria do:

(a) Solve $J(x^{(k)})s^{(k)} = -F(x^{(k)})$

(b) Set $x^{(k+1)} = x^{(k)} + \lambda^{(k)} s^{(k)}$

(c) Check for convergence.

In this section, we discuss how KINSOL handles Step 2(a). In this step, the Jacobian is computed and a linear system of equations is solved. We present the different types of linear solvers available in KINSOL which include solvers that produce exact solutions as well as solvers that produce approximated solutions.

KINSOL has two variations of Newton's method, modified Newton method and inexact Newton method. Both use the same Newton's algorithm while they differ only in the way of choosing the linear solver. If we choose a linear solver that solves the problem exactly, we then have a modified Newton's method. However, if we choose a linear solver where the solution is computed approximately, then we have an inexact Newton method.

The modified Newton's method is called "modified" in the sense that it uses outdated Jacobian. This means the Jacobian is updated after some specific number of iterations and this is set as default option in KINSOL. If we choose to update the Jacobian each iteration, this results in a classical Newton's method [6].

Jacobian calculation in KINSOL

Calculating the Jacobian in KINSOL has two options. The first option is providing a user subroutine for calculating or approximating the Jacobian. The second option is to use the Jacobian approximation subroutine which is built in KINSOL and which uses the finite differences in Equation (2.17).

In the case of using iterative linear solvers, using the Krylov method, we are interested in approximating the matrix vector product as follows

$$J(x^{(k)})v = \frac{F(x^{(k)} + h\ v) - F(x^{(k)})}{h} \tag{2.36}$$

The default in KINSOL is to use old Jacobian and update the Jacobian every tenth iteration otherwise the Jacobian is updated in the following expected situations:

1. When the problem is initialized, i.e. at the first nonlinear iteration.

2. When the number of the Jacobian reuse is exhausted, the defualt number is 10 iterations.

3. When the linear solver fails with an outdated Jacobian.

4. When the line search fails with an outdated Jacobian.

KINSOL linear solvers

SUNDIALS has three groups of generic linear solvers.

1. Direct linear solvers (dls): like DENSE, BAND.

2. Sparse linear solvers(sls): like KLU, SUPERLUMT.

3. Sparse preconditioned iterative linear solvers(spils): like SPGMR, SPFGMR, SPBCG, SPTQMR.

SUNDIALS direct linear solvers (dls) consist of two solvers DENSE and BAND for solving linear systems with dense and banded matrices respectively. It solves the system using LU factorization. The sparse linear solvers (sls) are designed for linear systems with sparse matrices. KINSOL contains two solvers KLU that is used to solve a linear system where the matrix is given in Compressed Sparse Column (CSC) format. The other solver is SUPERLUMT. It uses threads for efficient matrix factorization. Also, in the sls solvers the system is solved using LU factorization.

The last category of linear solvers is the scaled preconditioned iterative linear solvers (spils). This category is used when the linear system has a huge matrix. It solves the system approximately by using variations of GMRES.

SUNDIALS linear solvers can be used to solve a general system of linear equations or they can be used to solve a linear system as a part of a bigger process in one of SUNDIALS packages. For example, in KINSOL we need to solve a sequence of linear systems of equations as a step of Newton's method nonlinear iteration. In this case, we need to customize the linear solver to be problem specific. This is obtained by building interfaces on top of the generic solvers to include the problem specifications.

The idea of implementing one generic linear solver and several interfaces for it has two advantages. The first one is avoiding code redundancy. The

second advantage is isolating the software packages, e.g. KINSOL, from changes in the generic solver code for example when releasing a new version. KINSOL contains the following linear solver interfaces and Figure 2.3 shows how they connect to the generic SUNDIALS solvers:

1. Direct linear solvers interfaces: KINDENSE, KINBAND.
2. Sparse linear solvers interfaces: KINKLU, KINSUPERLUMT.
3. Sparse preconditioned iterative linear solvers interfaces:KINSPGMR, KINSPFGMR, KINSPBCG, KINSPTQMR.

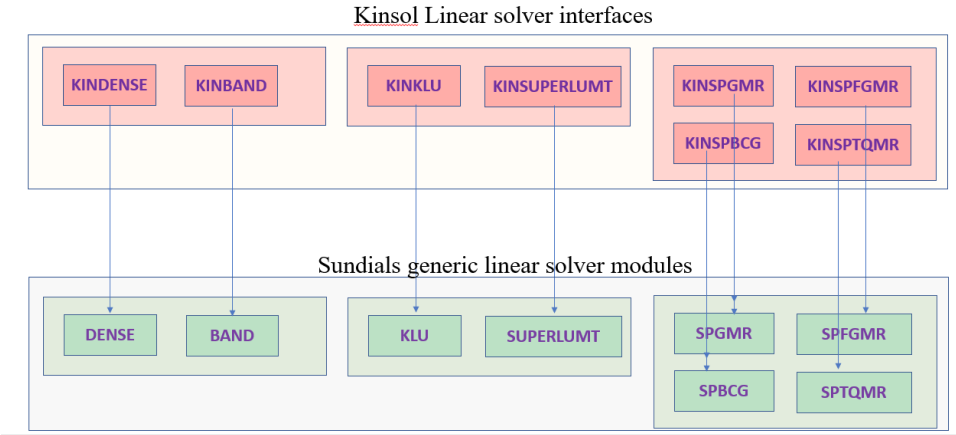


Figure 2.3: Kinsol Linear Solvers and their connection to the generic solvers.

The main structure of a user program that uses KINSOL

The skeleton of a user program that uses KINSOL to solve a system of non-linear equations consists of the following steps:

- **Step1:** Vector definition and initialization.

```
u=N_Vnew_Serial(N); //N is the vector length.
su=N_Vnew_Serial(N);
sf=N_Vnew_Serial(N);
c=N_Vnew_Serial(N);
```

```
SetInitialGuess(u,userdata);

N_Vconst_Serial(const,su); // su is Nvector for scaling.
N_Vconst_Serial(const,sf); // sf is Nvector for scaling .
NV_Ith(c,j)=0;  //c is Nvector for setting the constraint type.
```

KINSOL uses the function `N_Vnew_Serial(N)` that belongs to the NVector header file to create a new Nvector with N components. Here we create four vectors: u which is our iterate vector, `su` and `sf` are the scaling vectors to scale the iteration variables and residuals respectively, and `c` which holds the constraints on `u`.

`N_Vconst_Serial(const,su)` is used to set all the components of the vector with some constant value. We use it if we want to initialize the scaling vectors with the same value for all components. The function `NV_Ith(c,j)` accesses the j^{th} element in the vector c and gives it a value. `NV_Ith(c,j)=0` means that no constraints on the j^{th} iteration variable. To put constraints on a component, we use 1 to express $\geq$, -1 for $\leq$, 2 for $>$, and -2 for $<$

- **Step2:** : Create a memory structure.

```
Kmem=KINCreate();
```

After the initialization is done, we call `KINCreate()` that creates and returns a pointer to a memory structure `Kmem`. This structure encapsulates all the variables and constraints during the process of the problem solution.

- **Step3:** KINSOL settings.

```
flag=KinSetUserData(Kmem, userdata);
flag=KinSetConstraints(Kmem, c);
flag=KinSetFuncNormTol(Kmem, const);
flag=KinSetScaledStepTol(Kmem, const);
```

After creating the `Kmem` structure we start to save in the user data and constraints. `KinSetFuncNormTol` and `KinSetScaledStepTol` are used to set the tolerance of the residual function norm and the scaled Newton step norm respectively.

- **Step4:** KINSOL initialization.

```
flag=KinInit(Kmem, func,u);
```

This step alocates the memory for the structure pointer `Kmem` and does additional memory allocation according to the function to be solved `func` and our iteration vector `u`.

- **Step5:** Link the linear solver.

```
flag=KinDense(Kmem, N)
```

This step is to link the linear solver. Here we choose to use the DENSE linear solver with problem size N.

- **Step6:** Solve.

```
N_Vscale_Serial(Const, u,us); flag=KinSetMaxSetupCalls(Kmem,
    mset);
flag=KinSol(Kmem, u, KIN_NONE, su,sf);
```

Function `KINSetMaxSetupCalls` is used to specify the maximum number of nonlinear iterations where outdated Jacobian is used, the default value is 10. Function `KinSol` contains the actual problem solving where the Newton iteration is performed. `KinSol` takes the following parameter list:

1. A pointer to the structure `Kmem`.
2. The iteration vector u.
3. A predefined constant which indicate the strategy to solve the nonlinear system which has the following choices:
   (a) `KIN_NONE`: Newton's method without line search strategy
   (b) `KIN_LINESEARCH`: Newton's method with line search strategy
   (c) `KIN_FP`: Fixed point iteration.
   (d) `KIN_PICARD`: Solve with Picard iteration.
4. The scaling vector `su` to scale the independent vector.
5. The scaling vector `sf` to scale the residuals.

- **Step7:** Print final statistics (optional step).

```
flag=KINGetNumNonlinearSolvIters(Kmem, &n);
flag=KINGetNumFuncEvals(Kmem, &n);
flag=KINGetNumJacEvals(Kmem, &n);
flag=KINDlsGetNumFuncEvals(Kmem, &n);
```

In this optional step, we show some functions that provide information about the solving process like the number of nonlinear iterations, the number of function evaluations, the number of the Jacobian evaluations , and the number of iteration taken to solve the linear system of equations. This information can be used to compare the quality of the solver with other solvers. One also can use the information to analyze the problem in case the solver fails to converge to a solution.

- **Step8:** Free the memory and destroy vectors.

```
N_Vdestroy_Serial(u); //Free the Nvector u, we repeat this for
    every Nvector we created for example, c, su, sf.
KINFree(Kmem); // Free the created structure Kmem
free(udata);   // Free the specified user data.
```

Intuitively, after the process is finished and the problem is solved, we free the memory assigned to the problem variables.

2.3.2 OCT-NLESOL (Modelon)

OCT-NLESOL is a nonlinear equations solver which is a part of a bigger software package called OCT runtime. OCT runtime contains some other numerical solvers, for example, it contains a fixed point iteration solver and a Brent solver. The nonlinear problem is given to OCT-NLESOL in the form of a model written in Modelica language and stored as FMU (Function Mockup Unit).

Unlike KINSOL, OCT-NLESOL implements only a modified Newton method with line search globalization. This means that the solver does not contain implementation for the inexact Newton's method i.e. the linear system is always solved exactly. The Newton iteration in the OCT-NLESOL is always combined with line search globalization and there is no option to deactivate the line search for using basic Newton's method.

OCT-NLESOL executes two passes of Newton's iteration, refered as the first and the second Newton solve in [7]. This is because in some cases,

the first Newton solve is terminated because the Newton step becomes very small while the current iterate still not converged to the ultimate solution. Therefore, another Newton iteration is executed with different strategy by recalculating the Jacobian, as the solver by default uses outdated Jacobian, and updating the scaling of variables.

OCT-NLESOL linear solvers

The OCT runtime contains two linear solvers. The first is a generic solver that is used to solve generic systems of linear equations. It uses the LAPACK function `dgetrf` that uses the LU factorization to solve the linear system. When the matrix is singular, the solver uses the LAPACK function `dgelss` which computes the minimum norm least squares solution of over-determined and under-determined linear systems. After `dgelss` returns a solution x^*, the solver calculates the $\infty-$norm, $\|Ax^* - b\|_\infty$. If the norm is less than a given tolerance, x^* is then declared as an acceptable solution. Otherwise, a failure status is recorded and the solver terminates its process.

The second solver is customized to be a part of the nonlinear equation solver and it takes care of some additional issues like handling different methods for calculating the Jacobian, managing the singularity of the Jacobian matrix, and handling the Newton step projection as the solution is always required to be bounded $x_{min} \leq x^* \leq x_{max}$.

The Jacobian calculation

OCT-NLESOL implements a modified Newton's method. This means that the solver by default reuses the Jacobian for specific number of iterations which can be modified using the `nle_solver_max_iter_no_jacobian` solver option. The default value is ten iterations like for KINSOL.

OCT-NLESOL has different modes for calculating the Jacobian. We can switch the modes using the solver option `nle_jacobian_calculation_mode`. This option takes values from 0 to 9 and this depends on the Jacobian calculation method and at which Newton pass it is used, the first or the second. These combinations are illustrated in Table 2.1.

OCT-NLESOL uses three formulas to calculate the Jacobian which are

1. **One sided finite difference:**

The j^{th} column in the matrix is approximated as

$$J_{.j} = \frac{F(x + h_j e_j) - F(x)}{h_j} \tag{2.37}$$

Where $h_j = \max(|x_j|, \text{typ}_{x_j})\ \delta_s\ \text{sign}(x_j)$ and δ_s is a user specified constant by using the solver option `nle_jacobian_finite_difference_delta`, the default value is $\sqrt{macheps}$. e_j is the j^{th} unit vector and

$$\text{sign}(x) = \begin{cases} 1, & \text{if} x \geq 1 \\ \text{-1}, & \text{if} x < 1 \end{cases}$$

2. **Central finite difference:**

$$J_{.j} = \frac{F(x + |h_j| e_j) - F(x - |h_j| e_j)}{2|h_j|} \tag{2.38}$$

3. **Two sided finite difference:**

   This formula has the advantage of keeping the derivatives continuous and avoiding evaluating the residuals outside the bounds.

$$J_{.j} = (1-\eta)\frac{F(x + {h_j}^{\text{right}} e_j) - F(x)}{{h_j}^{\text{right}}} + \eta \frac{F(x) - F(x - {h_j}^{\text{left}} e_j)}{{h_j}^{\text{left}}} \tag{2.39}$$

   where

$${h_j}^{\text{right}} = \min((x_{\max j} - x_j), |h_j|) \tag{2.40}$$

   and

$${h_j}^{\text{left}} = \min((x_j - x_{\max j}), |h_j|). \tag{2.41}$$

   η attains values $\in [0, 1]$ as x ranges from $x_{\min}$ to $x_{\max}$. Then we have $\eta = 0$ when x is at the lower bound and $\eta = 1$ when x is at the upper bound.

In the following table, we will use the notation of N_1 and N_2 for both first and second Newton solve. We also use B which means that the equation is used at the bounds and Z which means the equation used at zero.

Mode	Description
0	(2.37) at N_1 and N_2
1	(2.38) at N_1 and N_2
2	(2.39) for B at N_1 and N_2
3	(2.39) for B and z at N_1 and N_2
4	(2.37) at N_1 and (2.38) at N_2
5	(2.37) at N_1 and (2.39) at N_2
6	(2.37) at N_1 and (2.39) at N_2 for B and Z
7	(2.37) when the residual norm $> 10^2$; otherwise (2.38).
8	external calculation of the Jacobian.
9	(2.37) with Jacobian compression.

Table 2.1: Different modes for calculating the Jacobian

The structure of a user program that uses OCT-NLESOL

In this section we give the main structure of a user MATLAB script that uses OCT-NLESOL to solve a nonlinear system of equations specified as a Modelica model.

```
modelName = 'SystemOfEquations';
compiler = 'OCT_Modelica';

fmuName = compileFMU(modelName, compiler, 'modelPath', lib,
    'options', opt);
fmu = FMUModelME1(fmuName);
fmuProblem = FMUProblem(fmu);

fmuProblem.setInitialGuess(x0);
solver = Solver(fmuProblem);

solver.setOptions('max_iter',50);
solver.setOptions('max_iter_no_jacobian',2);
solver.setOptions('log_level',0);
solver.setOptions('residual_equation_scaling',0);
sol = solver.solve();
solver.delete();
```

The variable `modelName` holds the name of the model, in this example we call it `SystemOfEquations`, that contains the nonlinear equations system written in Modelica language and stored in `SystemOfEquations.mo`. As there

are many compilers to compile the Modelica language, we have to specify the compiler name which will be used. Here we use the `OCT_Modelica` compiler. The model name and the compiler chosen are then passed to `compileFMU` method along with two other variables, the model absolute path and some compiler options if any. `compileFMU` then compiles the Modelica model into FMU and save it into a file called `SystemOfEquations.fmu`.

The constructor of the `FMUModelME1` class then takes the FMU file name to create an FMU object of the type ModelExchange. The class `FMUProblem` takes to its constructor the FMU object and creates an object `fmuProblem` which contains the problem's iteration variables and the residuals. We then assign `fmuProblem` an initial guess using the attribute `setInitialGuess`.

Now our problem is ready to be solved. So we create a solver object that takes `fmuProblem` as input to its constructor. Before solving the problem, we can make some settings to the solver like setting the maximum number of iteration, the log level, or the maximum number of reusing the Jacobian. For those settings, we use the solver attribute `solver.setOptions`. `solver.solve()` solves the problem using the modified Newton iteration and returns x^*. Finally, we delete the solver object after we finish the process.

2.4 Summary

In this chapter, we presented Newton's method for solving system of nonlinear equations and presented its classical convergence theorem. The theorem shows that Newton's method has quadratic convergence which is considered a very fast convergence. The problem is that this fast convergence is only local. This means starting with a bad initial guess may affect the convergence rate or even the possibility of convergence. So, for better performance of Newton's method we need to combine it with a globalization technique that extends Newton's method to a wider margin of initial guesses as well as making benefit from the fast convergence of Newton's method.

In the following chapter, we present Newton's method globalization using the line search method. We also investigate the implementation of the line search algorithm in KINSOL.

Chapter 3

Globalization Strategy

As we discussed in the previous chapter, the main advantage of Newton's method is its local quadratic convergence. However, Newton's method is not globally convergent. This means that the method may diverge or oscillate if the initial guess is not sufficiently close to the solution. To solve this problem, we need a globalization strategy that benefits from the quadratic convergence of Newton's method and at the same time increases its radius of convergence.

Starting from a particular initial guess, the globalization strategy directs the Newton iterates towards the zero of the residual. Once the current iterate enter the neighborhood of attraction, Newton's method converges quadratically. There are many globalization techniques for Newton's method, the most popular are line search and trust region methods.

In this chapter, we address Newton's method globalization using line search methods. We present the exact and the inexact line search methods as well as the backtracking algorithm. Later in this chapter, we also present the implementation of the line search subroutine in KINSOL. Moreover, we are going to present our own modification to KINSOL to change the line search parameters and to save the iteration variables.

3.1 The Line Search Algorithm

The idea of line search is that instead of updating the current iterate with the full Newton step, we only apply a factor λ of that step. This can be beneficial when the full Newton step is very big and drifts away from the

solution. Our goal is to minimize the norm function introduced in Subsection 2.1.1, Equation 2.5. So the optimal value of λ, we refer to it as λ^*, in the direction of $s^{(k)}$ from the current iterate $x^{(k)}$ is calculated as the minimum point of the one dimensional function $f(x^{(k)} + \lambda s^{(k)})$. The following is the general structure of the line search algorithm assuming that $F \in C^1$:

- $x^{(0)} =$ `<initial guess>`.
- At iteration k:
  1. Compute the Newton search direction $s^{(k)} = -J(x^{(k)})^{-1}F(x^{(k)})$.
  2. Find λ_k that minimizes $f(x^{(k)}+\lambda_k s^{(k)})$ using a line search method.
  3. Set $x^{(k+1)} = x^{(k)} + \lambda_k s^{(k)}$.

In any line search method, our goal is to find the minimum of $f(x^{(k)}+\lambda_k s^{(k)})$. To find the minimum we differentiate f with respect to λ and we solve to find λ_k such that the derivative equals to zero.

$$\frac{\mathrm{df}}{\mathrm{d}\lambda}(x^{(k)} + \lambda s^{(k)})\bigg|_{\lambda=\lambda_k} = \nabla f((x^{(k)} + \lambda_k\ s^{(k)}))^{\mathrm{T}}\ s^{(k)} = 0 \qquad (3.1)$$

This is equivalent to

$$\nabla f((x^{(k)} + \lambda_k\ s^{(k)})) \perp s^{(k)}. \qquad (3.2)$$

This means that we want to find λ_k where the search direction $s^{(k)}$ is perpendicular to the gradient vector at the point $x^{(k+1)}$.

The line search methods are divided into two types exact and inexact line search which will be discussed in the following two subsections.

3.2 Exact Line Search

The exact line search methods are designed for finding a fine approximation to λ^*. They are divided into two groups [3]:

1. Methods that do not use the derivative, e.g, uniform search, Dichotomous method, the golden section method, and Fibonacci method.
2. Methods that use the derivative like the bisection method and Newton method.

3.2.1 Exact Line Search Methods that do not use Derivatives

The exact line search methods that do not use derivatives have one method that adopts the idea of simultaneous search, the uniform method, whereas the others adopt the idea of sequential search [3]. For both kinds of methods we have to choose a value of λ within an interval $[a, b]$. This interval can be for example, $[0, 1]$, where 1 means that we apply the full Newton step, while 0 means that $x^{(k+1)} = x^k$.
The algorithm of the uniform method is

1. Subdivide the interval $[a, b]$ into N equidistant points $\lambda_1, \lambda_2, \cdots, \lambda_N$.
2. Evaluate the function at those points. i.e. $f(\lambda_1), f(\lambda_2), \cdots, f(\lambda_N)$.
3. Choose among them λ_i with the minimum function value.

This method is very simple. However, the accuracy of finding the optimal λ depends on the quality of the grid subdivision. Moreover, it needs many function evaluations which may be very expensive.

On the other hand, the methods which use the sequential line search have the idea of diminishing the interval $[a, b]$ of possible λ around λ^* until the interval length becomes sufficiently small. We define the interval at iteration k to be $[a_k, b_k]$. After the interval becomes sufficiently small, λ^* can be taken as the mid point of the interval or one of the interval ends. The following are the steps of the methods using sequential search. We assume that we at iteration k.

1. Choose λ, μ where $a_k < \lambda < \mu < b_k$.
2. Compute $f(\lambda), f(\mu)$.
3. One of two cases results:
   - Case 1: If $f(\lambda) > f(\mu)$, the new interval is $[\lambda, b_k]$.
   - Case 2: If $f(\mu) > f(\lambda)$, the new interval is $[a_k, \mu]$

There are several methods that apply the previous algorithm but they differ in the way they define λ and μ. One of those methods is Dichotomous method [3, 4]. It defines λ and μ as follows:

$$\lambda = \frac{a_k + b_k}{2} - \epsilon \tag{3.3}$$

$$\mu = \frac{a_k + b_k}{2} + \epsilon \tag{3.4}$$

where ϵ is a small number.

The golden section method defines λ and μ as

$$\lambda_k = a_k + (1-\alpha)(b_k - a_k) \tag{3.5}$$
$$\mu_k = a_k + \alpha(b_k - a_k), \tag{3.6}$$

where α is the golden section number and it equals ≈ 0.618 [3, 4]. This formulation in equation (3.5) decreases the interval length by a factor α at each iteration. The golden section reuses function evaluations from previous iterations. For example, if the algorithm executes Case 1, then $\lambda_{k+1} = \mu_k$ while if it executes Case 2, then $\lambda_k = \mu_{k+1}$.

The Fibonacci method uses the same idea of reusing the function evaluations like the golden section while defining λ and μ as follows [3, 4]

$$\lambda_k = a_k + \frac{N_{n-k-1}}{N_{n-k+1}}(b_k - a_k) \tag{3.7}$$
$$\mu_k = a_k + \frac{N_{n-k}}{N_{n-k+1}}(b_k - a_k) \tag{3.8}$$

where $\{N_k\}_{k=0}^{\infty}$ are the Fibonacci numbers which are defined by the relation $N_K = N_{k-1} + N_{k-2}, k \geq 2$ and $N_0 = N_1 = 1$, n is an integer represent a fixed number of Fibonacci numbers.

3.2.2 Exact Line Search Methods That Use Derivatives

Two examples of line search methods that use the derivative are the bi-section method and Newton's method. The bi-section method defines $\lambda = \frac{a_k+b_k}{2}$. If $f'(\lambda_k) \leq 0$, the new interval is $[\lambda_k, b_k]$ and if $f'(\lambda_k) > 0$, the new interval is $[a_k, \lambda_k]$.

The idea of Newton method is different than all the previous methods because it does not adopt the idea of decreasing intervals rather it generates a sequence of $\{x_k\}^{\infty}$ that converge to the minimum point x^*. We begin with initial value $\lambda_k = \lambda_0$ and then iterate by computing $\lambda_{k+1} = \lambda_k - \frac{f(\lambda_k)}{f'(\lambda_k)}$ until a predefined convergence criteria is met.

3.3 Inexact Line Search

Finding the exact λ^* is very expensive because it requires many function evaluations which can be time consuming. Moreover, the line search is used as a sub-algorithm in solving a system of nonlinear equation. So, it is not logical to pay such price of processing time to find the exact λ^* in each nonlinear iteration while we aim to find a reasonable approximation to the solution of the nonlinear problem.

The idea of the inexact line search is to find a coarse approximation to λ^* such that $f(x^{(k+1)}) < f(x^{(k)})$ where $x^{(k+1)} = x^{(k)} + \lambda^* s^{(k)}$. However, this condition does not guarantee that the iterates will converge to a local minimum of $f(x)$. In some cases, the decrease obtained in the value of $f(x)$ is very small compared to the the length of the Newton step. This may lead the iterates to stuck in a point which is not a local minimum of $f(x)$. In other situations, after some iterations the Newton steps become too small relative to the initial rate of reduction in f which leads to a stuck position as well, see the examples in [1] page 118. These situations are avoided by imposing two conditions to determine an upper and lower limit of λ^*. Any value of λ between that upper and lower limit is acceptable to be taken as λ^*.

Before we discuss those two conditions, known by the Goldstein conditions [1, 2], we define the one dimensional function $g(\lambda) = f(x^{(k)} + \lambda s^{(k)})$. So we have

$$\begin{aligned} g(0) &= f(x^{(k)}) && (3.9)\\ g'(\lambda) &= \nabla f(x^{(k)} + \lambda s^{(k)})^T s^{(k)} && (3.10)\\ g'(0) &= \nabla f(x^{(k)})^T s^{(k)} && (3.11) \end{aligned}$$

The Goldstein conditions that put the upper and lower limits to λ are defined as

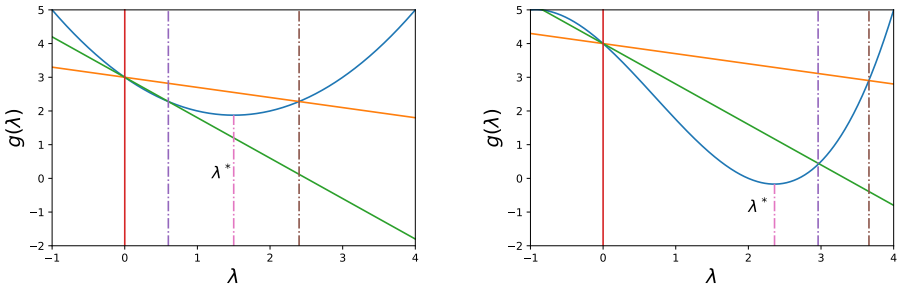
$$f(x^{(k)} + \lambda s^{(k)}) \leq f(x^{(k)}) + \lambda\alpha\nabla f(x^{(k)})^T s^{(k)} \tag{3.12}$$

$$f(x^{(k)} + \lambda s^{(k)}) \geq f(x^{(k)}) + \lambda(1-\alpha)\nabla f(x^{(k)})^T s^{(k)} \tag{3.13}$$

or in terms of the function g as

$$\begin{aligned} g(\lambda) &\leq g(0) + \lambda\alpha g'(0) && (3.14)\\ g(\lambda) &\geq g(0) + \lambda(1-\alpha) g'(0) && (3.15) \end{aligned}$$

where $\alpha \in (0, \frac{1}{2})$. The intersection of the orange and green lines with the curve in Figure 3.1a indicates the range of the acceptable values of λ. The problem with the Goldstein conditions is that in some cases where the function is not quadratic, the lower bound exceeds the minimum point. This means that the interval indicated by the Goldstein conditions doesn't include the optimal λ[2]; look Figure 3.1b.



(a) f is quadratic. (b) f is cubic

Figure 3.1: Goldstein upper and lower bounds for λ^*.

To solve this problem we replace the second condition in (3.13) by the following condition

$$\nabla f(x^{(k)} + \lambda s^{(k)}) s^{(k)} \geq \beta \nabla f(x^{(k)})^T s^{(k)} \tag{3.16}$$

$$\Leftrightarrow g'(\lambda) \geq \beta g'(0) \tag{3.17}$$

where $\beta \in (\alpha, 1)$. Conditions (3.12) and (3.16) are known by Goldstein-Armijo conditions [1]. Condition (3.16) can be interpreted as starting from $\lambda = 0$ and moving along the curve of $g(y)$, we choose the lower bound for λ^* to be first value of λ such that the tangent line at this point is parallel to the line that passes through $\lambda = 0$ and has a slope $\beta g'(0)$. This insures that λ^* is greater than the lower limit, see Figure 3.2.

Condition (3.16) sometimes is replaced in practice by the following condition which puts a tighter two sided condition on λ [2].

$$|g'(\lambda)| \leq -\beta g'(0). \tag{3.18}$$

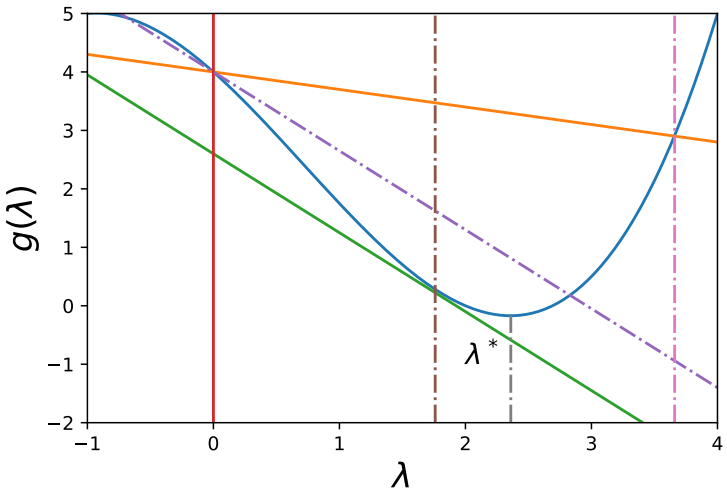


Figure 3.2: Goldstein-Armijo upper and lower bounds for λ^*.

3.3.1 The Backtracking Algorithm

The idea of the backtracking algorithm is to begin with the full Newton step i.e. $\lambda_k = 1$. If $\lambda_k = 1$ does not satisfy conditions (3.12) and (3.16), then a backtracking procedure is applied to decrease λ_k. The idea of the algorithms is as follows:

- Given $\alpha \in (0, \frac{1}{2})$, and two constants $0 < l < u < 1$.
- $\lambda_k = 1$.
- while $f(x^{(k)} + \lambda_k s^{(k)}) > f(x^{(k)}) + \alpha\lambda_k \nabla f(x^{(k)})^T s^{(k)}$,
  - $\lambda_k = \rho\lambda_k$ for some $\rho \in [l, u]$.
- $x^{(k+1)} = x^{(k)} + \lambda_k s^{(k)}$.

Where ρ is a decreasing factor for λ_k. So we want to choose ρ so that the value of the next λ decreases g.

If we are at the first iteration and $\lambda_k = 1$ does not satisfy condition (3.12), then we have three pieces of information about $g(\lambda)$ which are

1. $g(0) = f(x_k)$.
2. $g'(0) = \nabla f(x_k)^T s_k$.
3. $g(1) = f(x_k + s_k)$.

We can use these information to construct a quadratic polynomial that approximates $g(\lambda)$. Then we put the next λ_k equal to the minimum point of this quadratic polynomial.

$$\text{mq}(\lambda) = (g(1) - g'(0) - g(0))\lambda^2 + g'(0)\lambda + g(0) \tag{3.19}$$

To get the minimum point of mq, we differentiate with respect to λ and equate to zero and the minimum point is

$$\lambda_{\text{mqmin}} = \frac{-g'(0)}{2(g(1) - g'(0) - g(0))}. \tag{3.20}$$

As we backtrack because we violated condition (3.12), this means that

$$g(1) > g(0) + \alpha g'(0) > g(0) + g'(0). \tag{3.21}$$

$$\Rightarrow mq''(\lambda_{\text{mqmin}}) = 2(g(1) - g'(0) - g(0)) > 0. \tag{3.22}$$

This confirms that λ_{mqmin} is a minimum point. Also, since we require $g'(0) < 0$, a descent direction, and that $(g(1) - g'(0) - g(0))$ is positive by (3.22) then $\lambda_{\text{mqmin}} > 0$ so we take $\lambda_k = \lambda_{\text{mqmin}}$.

We have $g(1) > g(0) + \alpha g'(0) \Rightarrow \lambda_{\text{mqmin}} < \frac{1}{2(1-\alpha)} \Rightarrow u \approx \frac{1}{2}$ then we have an upper limit of λ equal $\frac{1}{2}$. Also, if $g(1)$ is much larger than $g(0)$, λ_k can be too small. To avoid decreasing λ_k too much we choose a lower bound $l = \frac{1}{10}$. So if $\lambda_{\text{mqmin}} < 0.1$, we set $\lambda_k = 0.1$.

If condition (3.12) is violated for the second time, we have then four pieces of information, instead of three, which are $g(0)$, $g'(0)$, and the last two values of $g(\lambda)$. So we can construct a cubic model instead of a quadratic one. Cubic interpolation has the advantage of giving more accurate representation especially to those areas with negative curvature. For formula of the cubic model and its minimum point see [1]. For the upper and lower bounds, if $\lambda_k > \frac{1}{2}\lambda_{\text{prev}}$, set $\lambda_k = \frac{1}{2}\lambda_{\text{prev}}$ and if $\lambda_k < \frac{1}{10}\lambda_{\text{prev}}$, set $\lambda_k = \frac{1}{10}\lambda_{\text{prev}}$.

The backtracking algorithm can be combined with two additional features. The first features is to define a minimum step length to prevent having extremely small globalized Newton steps. For example, if Condition (3.12) is not satisfied and $||\lambda_k s^{(k)}||_2$ is smaller than a defined constant `minimumStep`, then the algorithm terminates. This has the advantage of preventing the line search from infinite loops if s_k is not a descent direction. The second feature is to indicate a maximum step length. This feature prevents taking too long steps that may exceed our domain of interest.

3.4 Line Search in KINSOL

As we mentioned before, KINSOL implements Newton's method with the possibility of activating the line search globalization. This is done by passing the `KIN_LINESEARCH` flag to the `KinSol` function which executes the Newton nonlinear iteration. In this case, `KinSol` computes the Newton direction and then calls the function `KINLineSearch` which implements the backtracking algorithm presented in Section 3.3.1 to compute λ^*. `KINLineSearch` uses Condition (3.12) for putting an upper bound to λ^* in combination with the following condition that puts a lower bound to λ^*

$$f(x^{(k)} + \lambda s^{(k)}) \geq f(x^{(k)}) + \lambda \beta \nabla f(x^{(k)})^T s^{(k)}. \tag{3.23}$$

Conditions (3.12) and (3.23) are referred in [6] as Wolfe curvature condition. Because the `KINLineSearch` function is quite large, we only discuss those parts of the implementation which corresponds to the steps of the backtracking algorithm.

```
static int KINLineSearch(KINMem kin_mem, realtype *fnormp, realtype
    *f1normp,
                         booleantype *maxStepTaken)
{
  realtype pnorm, ratio, slpi, rlmin, rlength, rl, rlmax, rldiff;
  realtype rltmp, rlprev, pt1trl, f1nprv, rllo, rlinc, alpha, beta;
  realtype alpha_cond, beta_cond, rl_a, tmp1, rl_b, tmp2, disc;
  int ircvr, nbktrk_l, retval;
  booleantype firstBacktrack, fOK;

  /* Initializations */

  nbktrk_l = 0;        /* local backtracking counter */
  ratio    = ONE;      /* step change ratio       */
  alpha    = POINT0001;
```

```
  beta     = POINT9;

  firstBacktrack = TRUE;
  *maxStepTaken = FALSE;

  rlprev = f1nprv = ZERO;
```

KINLineSearch takes four parameters: a pointer to the KINSOL structure, two pointers to the values of the residual norm and the norm of the minimum function, and a pointer to Boolean variable that is assigned true if the maximum step has been taken and false otherwise. The function begins with variable declarations and initialization. In KINSOL, α and β are defined as constants and they are assigned the values 10^{-4} and 0.9 respectively.

```
/* Compute length of Newton step */

  pnorm = N_VWL2Norm(pp, uscale);
  rlmax = mxnewtstep / pnorm;
  stepl = pnorm;

/* If the full Newton step is too large, set it to the maximum
    allowable value */

  if(pnorm > mxnewtstep ) {
    ratio = mxnewtstep / pnorm;
    N_VScale(ratio, pp, pp);
    pnorm = mxnewtstep;
    rlmax = ONE;
    stepl = pnorm;
  }
```

Function N_VWL2Norm computes the 2-norm of the Newton step pp and saves in the variable pnorm after scaling it with the vector uscale. The variable rlmax holds the maximum value of λ which equals

$$\lambda_{\max} = \frac{\text{MaxNewtonStep}}{\|s^{(k)}\|_{D_x}} \tag{3.24}$$

where $\|s^{(k)}\|_{D_x}$ is the norm of the scaled Newton step. Then the function checks if the Newton step is greater than the predefined constant mxnewtstep then we scale the Newton step so that its norm equals to the mxnewtstep. Next we put $\lambda_{\max} = 1$ which means we begin with the full Newton step after truncation.

```
/* Attempt (at most MAX_RECVR times) to evaluate function at the
    new iterate */

 fOK = FALSE;

 for (ircvr = 1; ircvr <= MAX_RECVR; ircvr++) {

   /* compute the iterate unew = uu + pp */
   N_VLinearSum(ONE, uu, ONE, pp, unew);

   /* evaluate func(unew) and its norm, and return */
   retval = func(unew, fval, user_data); nfe++;

   /* if func was successful, accept pp */
   if (retval == 0) {fOK = TRUE; break;}

   /* if func failed unrecoverably, give up */
   else if (retval < 0) return(KIN_SYSFUNC_FAIL);

   /* func failed recoverably; cut step in half and try again */
   N_VScale(HALF, pp, pp);
   ratio *= HALF;
   pnorm *= HALF;
   rlmax = ONE;
   stepl = pnorm;

 }

 /* If func() failed recoverably MAX_RECVR times, give up */

 if (!fOK) return(KIN_REPTD_SYSFUNC_ERR);
```

As we begin with $\lambda = 1$, the function then evaluates $f(x^{(k)} + s^{(k)})$. If the function evaluation fails then we try another time by shrinking the Newton step to its half. This continues until the function evaluates successfully or a maximum number of recovery is met. In case of success the function values are stored in the variable `fval` and in case of failure the process terminates by returning a failure flag `KIN_REPTD_SYSFUNC_ERR`.

```
 /* Evaluate function norms */

 *fnormp = N_VWL2Norm(fval, fscale);
 *f1normp = HALF * (*fnormp) * (*fnormp) ;
```

```
/* Estimate the line search value rl (lambda) to satisfy both
    ALPHA and BETA conditions */

slpi = sFdotJp * ratio;
rlength = KINScSNorm(kin_mem, pp, uu);
rlmin = scsteptol / rlength;
rl = ONE;
```

Here, the function computes the residual norm and the value of the norm function and it stores them in the variables `*fnormp` and `*f1normp` respectively. `sFdotJp` contains the directional derivative at the current iterate which equals to $\nabla f(x^{(k)})^{\mathrm{T}} s^{(k)}$ which equals $F(x^{(k)})^{\mathrm{T}} J(x^{(k)}) s^{(k)}$ by using Equation (2.9). Using the residual scaling `sFdotJp` is computed as `sFdotJp`$= D_F F(x^{(k)})^{\mathrm{T}} J(x^{(k)}) s^{(k)}$. Next the minimum value of λ is stored in `rlmin` and λ is initialized to 1 to begin the backtracking algorithm. Next the function checks for condition (3.12).

```
/* Loop until the ALPHA condition is satisfied. Terminate if rl
    becomes too small */

loop {

  /* Evaluate test quantity */

  alpha_cond = f1norm + (alpha * slpi * rl);

  if (printfl > 2)
    KINPrintInfo(kin_mem, PRNT_ALPHA, "KINSOL", "KINLinesearch",
                 INFO_ALPHA, *fnormp, *f1normp, alpha_cond, rl);

  /* If ALPHA condition is satisfied, break out from loop */

  if ((*f1normp) <= alpha_cond) break;

  /* Backtracking. Use quadratic fit the first time and cubic fit
      afterwards. */

  if (firstBacktrack) {

    rltmp = -slpi / (TWO * ((*f1normp) - f1norm - slpi));
    firstBacktrack = FALSE;

  } else {
```

```
      tmp1 = (*f1normp) - f1norm - (rl * slpi);
      tmp2 = f1nprv - f1norm - (rlprev * slpi);
      rl_a = ((ONE / (rl * rl)) * tmp1) - ((ONE / (rlprev * rlprev))
          * tmp2);
      rl_b = ((-rlprev / (rl * rl)) * tmp1) + ((rl / (rlprev *
          rlprev)) * tmp2);
      tmp1 = ONE / (rl - rlprev);
.
.
.
```

This loop executes until λ satisfies condition (3.12). The left hand side of the α condition is stored in the variable `alpha_cond`. Inside the loop we check if the flag `firstBacktrack` is true, then the function assigns λ the minimum value of the quadratic model that approximate the function. On the other hand, if `firstBacktrack` equals false and this means that there are previous values of λ then we update the value of λ with the minimum of the cubic model.

Next the function checks for condition (3.23). Violating the condition means that λ is smaller than the lower limit of the optimal λ. In this case, the function increases the value of λ until condition (3.23) is satisfied. The function has two strategies for increasing the value of λ. First, If $\lambda = 1$, this means no previous values of λ, `KINLineSearch` sets the new value of $\lambda = \min\{2\lambda, \lambda_{\max}\}$.

```
rlprev = rl;
f1nprv = *f1normp;
rl = SUNMIN((TWO * rl), rlmax);
```

Second, if there were previous backtracking and there where previous values of λ, `KINLineSearch` sets the new value of $\lambda = \min\{\lambda, \lambda_{\text{prev}}\} + (\frac{1}{2})^k |\lambda_{\text{prev}} - \lambda|$ where k is the number of loop iterations.

```
      rllo = SUNMIN(rl, rlprev);
      rldiff = SUNRabs(rlprev - rl);

      do {

        rlinc = HALF * rldiff;
        rl = rllo + rlinc;
        nbktrk_l++;
```

```
    N_VLinearSum(ONE, uu, rl, pp, unew);
    retval = func(unew, fval, user_data); nfe++;
    if (retval != 0) return(KIN_SYSFUNC_FAIL);
    *fnormp = N_VWL2Norm(fval, fscale);
    *f1normp = HALF * (*fnormp) * (*fnormp);

    alpha_cond = f1norm + (alpha * slpi * rl);
    beta_cond = f1norm + (beta * slpi * rl);

    if (printfl > 2)
      KINPrintInfo(kin_mem, PRNT_ALPHABETA, "KINSOL",
          "KINLineSearch",
                  INFO_ALPHABETA, *f1normp, alpha_cond,
                      beta_cond, rl);

    if ((*f1normp) > alpha_cond) rldiff = rlinc;
    else if (*f1normp < beta_cond) {
      rllo = rl;
      rldiff = rldiff - rlinc;
    }

  } while ((*f1normp > alpha_cond) ||
      ((*f1normp < beta_cond) && (rldiff >= rlmin)));
```

3.4.1 Own Modifications to KINSOL

In this subsection, we present our own modification to KINSOL to allow for some line search experiments. As KINSOL is an open source code, we downloaded a copy of the code and we did the modification and then we built the code to be used through Python, this will be presented in Chapter 5. We did two modifications, modifying the line search parameters and saving the iteration variables.

Modifying The Line Search Parameters

As we discussed in the previous subsection, KINSOL defines the values of α and β in Conditions (3.12) and (3.23) as constants. They are always equal to 10^{-4} and 0.9 respectively. And this gives a wide interval of potential values of λ^*.

Our idea is to investigate the convergence of the globalized Newton's method

for different values of α and β. To do this we modified the value of those parameters manually inside the `KINLineSearch` subroutine in the following two lines at the beginning of the subroutine, in the initialization part.

```
alpha   = ONETHIRD; //POINT0001;POINT4;POINT0001;//
beta    =TWOTHIRDS;//HALF; // POINT9; //
```

Where `ONETHIRD, TWOTHIRDS, HALF, POINT4` are predefined constants in KINSOL.
By changing those values we tried to narrow the interval of possible values of λ^*. There where no methodology behind selecting the values rather we tried different combinations to see how this affects the convergence.

Saving the Iteration Variables

We did another experiment to trace a point during the iteration of Newton's method with and without globalization. For this we needed to get the values of the iterate and this option is not provided by KINSOL.
Our modification is to register the value of the current iterate in a text file which is then read by a Python script to make the plotting. For this modification, we added the following code to the Newton iteration in the `Kinsol` function.

```
int noElm;
noElm=N_VGetLength_Serial(unew);
realtype * vecDataunew=N_VGetArrayPointer(unew);
for (int i=0;i<noElm;i++)
{
   fprintf(fp,"%f ", vecDataunew[i]);
}
fprintf(fp,"\n");
```

The `Kinsol` function computes the current iterate and save it in `unew` defined as a vector of the type `NVector`. `N_VGetLength_Serial` is a function that takes the vector as input and return its number of elements. Function `N_VGetArrayPointer` returns an array that contains the vector data. Then the elements of the `vecDataunew` array are written in a text file. To write to a text file we need to define a file pointer, so we add this line to the beginning of the `Kinsol` function.

```
FILE *fp = fopen("/home/amira/Work/IterationVariables.txt", "w");
```

3.5 Summary

In this chapter, we studied the globalization of Newton's method using line search in both literature and software implementation. We also presented our modification to KINSOL line search subroutine for modifying the line search parameters and registering the iteration variables.

Another problem that affects the convergence of Newton's method is the scaling of the iteration variables and the residual equations. This scaling affects the norm computation and which in turn affects the convergence process. In the following chapter, we show that the Newton's iteration is affine-invariance. We also present another affine invariant version of Newton's method convergence theorem based on defining affine-invariant Lipschitz constants.

Chapter 4

Scaling

In Chapter 2 and 3, we discussed the theory and the implementation of Newton's method. We discussed local convergence and how to modify Newton's method to be globally convergent using the line search strategy.

In this chapter, we address the problem of scaling that also affects Newton's method convergence. This problem occurs when the independent and/or the dependent variables vary significantly in magnitude. For instance, when some variables are represented in kilometers and other variables are represented in milliseconds.

The big difference in variable scaling causes numerical problems like ill-conditioned Jacobian approximation. Also, the difference in magnitude of the dependent variables affects the norms of the residuals which in turn affects the convergence process. This might be the case when one component of the dependent vector is relatively small compared with the other components. In this case, the larger components will dominate the norm value and the effect of the small component will be neglected.

Analogically for what we did in the previous chapters, We discuss how KINSOL and OCT-NLESOL handle scaling of iteration variables and residual equations.

4.1 Scaling of Independent Variables and Residuals

If the independent variables have different scales i.e. each component x_j of the independent variable is defined to have a typical magnitude typ_{x_j} which extremely differs in magnitude, the solution to this problem is to scale the variables using a diagonal matrix D_x where $D_{xj,j} = \frac{1}{\text{typ}_{x_j}}$. By this, we convert the problem domain to $\hat{x} = D_x\, x$, where the new variables are almost in the same range.

For example, if $x \in \mathbb{R}^2$ and $x_1 \in [10^2, 10^3]$ and $x_2 \in [10^{-7}, 10^{-6}]$. Then we have $\text{typ}_{x_1} = 10^3$ and $\text{typ}_{x_2} = 10^{-6}$ and the scaling matrix is

$$D_x = \begin{bmatrix} 10^{-3} & 0 \\ 0 & 10^6 \end{bmatrix}. \tag{4.1}$$

This moves the new scaled variables to the range of $[10^{-1}, 1]$.

To solve a nonlinear system of equations using Newton's method and scaling of the independent variables, we do the following steps, illustrated in Figure 4.1, provided that the Newton step is scaling invariant. For the following steps, we need to define $\hat{F}(\hat{x}) = F(D_x^{-1}\hat{x}),\ \hat{x} = D_x\, x$:

1. Transform the independent variable $\hat{x} = D_x\, x$, where $D_x \in \mathbb{R}^{n\times n}$ is a non-singular matrix.

2. Calculate the local step in the domain of the new variable by solving the linear system.
$$\hat{J}(\hat{x}^{(k)})\ \hat{s}^{(k)} = -\hat{F}(\hat{x}^{(k)}), \tag{4.2}$$
where $\hat{J}(\hat{x}^{(k)})$ is the first derivative of $\hat{F}$ evaluated at $\hat{x}^{(k)}$.

3. Calculate the line search factor $\hat{\lambda}$ to calculate the global step.

4. Calculate the new iterate in the scaled domain.
$$\hat{x}^{(k+1)} = \hat{x}^{(k)} + \hat{s}^{(k)} \tag{4.3}$$

5. Transform back to the original problem to get the original iterate.
$$x^{(k+1)} = D_x^{-1}\ \hat{x}^{(k+1)} \tag{4.4}$$

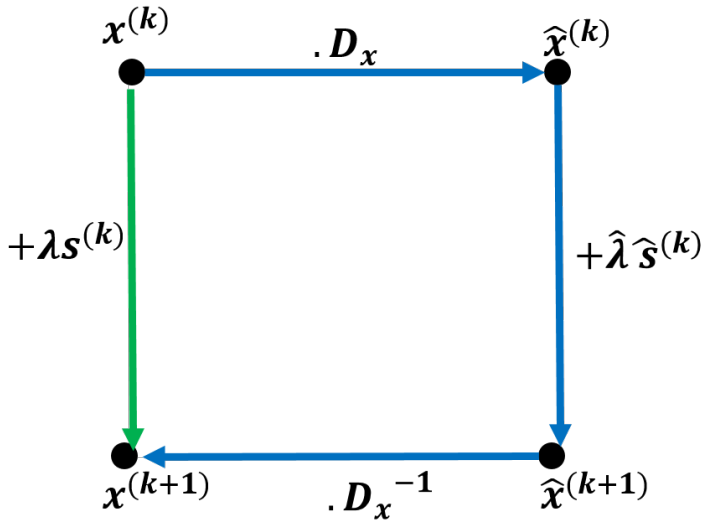


Figure 4.1: The scaled Newton iteration, in blue versus the original one, in green.

Now we need to show that scaling of variables does not affect the Newton step. Otherwise, the previous procedure will not work out. In general, the Newton step is scaling invariant in either case, in the case of nonlinear equation system or the case of unconstrained minimization.

First, we consider the case of an unconstrained minimization problem and suppose that we want to minimize a function of several variables $f(x)$. We then define $\hat{f}(\hat{x}) = f(D_x^{-1}\ \hat{x})$ where $\hat{x} = D_x\ x$. Then, the first and second derivative for $\hat{f}$ evaluated at $\hat{x}$ are

$$\begin{aligned} \nabla \hat{f}(\hat{x}) &= D_x^{-\mathrm{T}} \nabla f(D_x^{-1}\hat{x}) && (4.5) \\ &= D_x^{-\mathrm{T}} \nabla f(x) && (4.6) \end{aligned}$$

$$\begin{aligned} \nabla^2 \hat{f}(\hat{x}) &= D_x^{-\mathrm{T}} \nabla^2 f(D_x^{-1}\hat{x}) D_x^{-1} && (4.7) \\ &= D_x^{-\mathrm{T}} \nabla^2 f(x) D_x^{-1} && (4.8) \end{aligned}$$

The scaled Newton direction is then calculated as

$$\begin{aligned}
\hat{s} &= -\nabla^2 \hat{f}(\hat{x})^{-1} \nabla \hat{f}(\hat{x}) && (4.9)\\
&= -(D_x^{-\mathrm{T}} \nabla^2 f(x) D_x^{-1})^{-1} (D_x^{-\mathrm{T}} \nabla f(x)) && (4.10)\\
&= -D_x \nabla^2 f(x)^{-1} D_x^{\mathrm{T}} D_x^{-\mathrm{T}} \nabla f(x) && (4.11)\\
&= -D_x \nabla^2 f(x)^{-1} \nabla f(x) && (4.12)
\end{aligned}$$

To transform back the original Newton step, we inverse scale $\hat{s}$.

$$D_x^{-1} \hat{s} = -D_x^{-1} D_x \nabla^2 f(x)^{-1} \nabla f(x) = -\nabla^2 f(x)^{-1} \nabla f(x) = s \qquad (4.13)$$

Hence, the Newton direction for solving an unconstrained minimization problem is scaling invariant.

Next we consider the case of solving nonlinear system of equations $F(x) = 0$. The Newton step is defined as $s = -J^{-1}(x)F(x)$. We define the following:

$$\begin{aligned}
\hat{F}(\hat{x}) &:= F(D_x^{-1}\hat{x}) && (4.14)\\
\hat{J}(\hat{x}) &= \hat{F}'(\hat{x}) = F'(D_x^{-1}\hat{x}) D_x^{-1} = J(x) D_x^{-1}. && (4.15)
\end{aligned}$$

Then we compute the Newton step in the scaled domain.

$$\begin{aligned}
\hat{s} &= -\hat{J}(\hat{x})^{-1} \hat{F}(\hat{x}) && (4.16)\\
&= -(J(x) D_x^{-1})^{-1} F(x) && (4.17)\\
&= -D_x J(x)^{-1} F(x) && (4.18)
\end{aligned}$$

This also shows that the Newton step for solving nonlinear equation system is scaling invariant as $D_x^{-1}\hat{s} = -D_x^{-1} D_x J(x)^{-1} F(x) = s$.

Scaling of Residuals

The termination criteria used in Newton's method is that the norm of the residual is less than a given tolerance.

$$\|F(x^{(k)})\| < \text{ FTOL} \qquad (4.19)$$

Where FTOL is a user specified tolerance. As the residual vector also may contain components which have a large difference in magnitude, the components with relatively small magnitude will be neglected. Therefore, we scale the dependent variables with a diagonal matrix D_F before we calculate the residual norm. This changes our termination criteria to

$$\|D_F F(x^{(k)})\| < \text{ FTOL}. \qquad (4.20)$$

4.2 Affine Invariant Lipschitz Constant

Despite that the Newton direction is affine invariant, scaling of iteration variables and residual equations affect calculating both the norm of the step and the norm of the residual which in turn affects the algorithm termination criteria. This means that changing the norm calculation results in changing the criteria of accepting and rejecting new iterates.

An approach to solve the problem of scaling, proposed by Deuflhard [5], is to reformulate the convergence theorems of Newton's method by defining new scaling invariant Lipschitz constants. This leads to constructing a new algorithm which is affine invariant for both, the Newton direction and the termination criteria. Deuflhard in [5], mentions that there is no possibility to construct an affine invariant Newton's method that works with all classes of problems. So his approach divides the class of problems with respect to scaling into four classes:

1. **Affine covariance:** this class considers problems that only contain residual scaling.
2. **Affine contravariance:** this class considers problems that only contain variable scaling.
3. **Affine conjugacy:** this class considers scaling problems in minimization of multivariate functions.
4. **Affine similarity:** this class concerns with scaling problems in the area of differential equations.

In this section, we only focus on the first two classes, affine covariance and affine contravariance, which are relevant to the thesis work. Before we describe the details of Deuflhard's approach, we recall the necessary assumptions in the classical convergence theorem of Newton's method presented in Theorem 2 in Section 2.2. The first assumption is that the derivatives $J(x^k),\ k = 0, 1, \cdots$ exist, are invertible and bounded by $\beta_k < \infty$

$$\|J(x^{(k)})^{-1}\| < \beta_k < \infty, x \in D. \tag{4.21}$$

The second assumption is that the derivative is Lipschitz continuous in the neighborhood of the solution x^*.

$$\|J(x) - J(y)\| \leq \gamma\|x - y\|,\ x,\ y \in N(x^*, r). \tag{4.22}$$

We also recall that the scaled version of the residual function $F(x)$ can be defined as

$$G(y) = D_F F(D_x^{-1} y) = 0, \; x = D_x^{-1} y, \tag{4.23}$$

where D_F and $D_x \in \mathbb{R}^{n \times n}$ are non-singular matrices. We then apply Newton method to $G(y)$

$$\begin{aligned} G'(y^{(k)}) \Delta y^{(k)} &= -G(y^{(k)}) && (4.24) \\ y^{(k+1)} &= y^{(k)} + \Delta y^{(k)} && (4.25) \end{aligned}$$

where $G'(y^{(k)}) = D_F F'(x^{(k)}) D_x^{-1}$, the initial guess is $y^{(0)} = D_x x^{(0)}$ and we obtain $x^{(k)} = D_x^{-1} y^{(k)}, \; k = 1, 2, \cdots$

4.2.1 Affine Covariance

In this class of problems we assume $D_x = \mathrm{I}$. So our problem is to solve

$$G(y) = D_F F(x) = 0. \tag{4.26}$$

We construct an affine invariant Lipschitz constant by combining the two theoretical assumptions in (4.21), for a point z and (4.22), for two points x and $y \in D$.

$$\begin{aligned} \|F'(z)^{-1}(F'(x) - F'(y))\| \leq \omega \|x - y\|, && (4.27) \\ x, y, z \in D. && (4.28) \end{aligned}$$

ω is affine invariant because if we apply this condition to the scaled function $G(y)$ we get

$$\begin{aligned} G'(z)^{-1}(G'(x) - G'(y)) &= (D_F F'(z))^{-1}(D_F(F'(x) - F'(y))) && (4.29) \\ &= F'(z)^{-1} D_F^{-1} D_F (F'(x) - F'(y)) && (4.30) \\ &= F'(z)^{-1}(F'(x) - F'(y)) && (4.31) \end{aligned}$$

This condition contains an operator norm in the left hand side of inequality (4.27) . It can be modified so that it contains only vector norms while retaining the property of affine invariance as follows

$$\begin{aligned} \|F'(z)^{-1}(F'(x) - F'(y))(x - y)\| \leq \omega \|x - y\|^2, && (4.32) \\ x, y, z \in D. && (4.33) \end{aligned}$$

Next, we state the affine covariant Newton-Mysovskikh theorem [5], page 49. This theorem leads to results in terms of the iterates $\{x^{(k)}\}^{\infty}$, and the step norm $\|s^{(k)}\|$ or the error norm $\|x^{(k)} - x^*\|$.

Theorem 3. *Let $F : D \to \mathbb{R}^n$ be a continuously differentiable function where $D \subset \mathbb{R}^n$ is convex. Suppose that $F'(x)$ is invertible for each $x \in D$. Assume that the following affine covariant Lipschitz condition holds:*

$$\|F'(z)^{-1}(F'(y) - F'(x))(y - x)\| \leq \omega\|y - x\|^2 \tag{4.34}$$

for collinear $x, y, z \in D$. For the initial guess $x^{(0)}$ assume that

$$h_0 := \omega\|s^{(0)}\| < 2 \tag{4.35}$$

and that $N(x^{(0)}, \rho) \subset D, \rho = \frac{\|s^{(0)}\|}{1-\frac{1}{2}h_0}$.
Then the sequence $\{x^{(k)}\}^\infty$ of Newton iterates remains in $N(x^{(0)}, \rho)$ and converges to a solution $x^* \in N(x^{(0)}, \rho)$. Moreover, the following error estimates hold

$$\|x^{(k+1)} - x^{(k)}\| \leq \frac{1}{2}\omega\|x^{(k)} - x^{(k-1)}\|^2 \tag{4.36}$$

$$\|x^{(k)} - x^*\| \leq \frac{\|x^{(k)} - x^{(k+1)}\|}{1 - \frac{1}{2}\omega\|x^{(k)} - x^{(k+1)}\|} \tag{4.37}$$

For the proof of Theorem 3, see [5]. Now we use the results from this theorem to define a convergence monitor and a new termination criteria for actual implementation of Newton's method. The idea of defining a convergence monitor is to give as early estimate as possible of the divergence of Newton's method. This prevents executing unnecessary iterations until the maximum number of iterations is exhausted.

Theorem 3 states inequality (4.36) which can be rewritten in terms of $s^{(k)}$ as

$$\|s^{(k)}\| \leq \frac{1}{2}\omega\|s^{(k-1)}\|^2. \tag{4.38}$$

Extending the definition of h_0 in Theorem 3, we now define

$$h_k := \omega\|s^{(k)}\|. \tag{4.39}$$

so by multiplying (4.38) by ω we get

$$h_k \leq \frac{1}{2}h_{k-1}^2. \tag{4.40}$$

We define the convergence monitor as

$$\theta_k := \frac{\|s^{(k+1)}\|}{\|s^{(k)}\|} \tag{4.41}$$

which also can be expressed in terms of h_k and by using (4.40) as

$$\theta_k := \frac{h_{k+1}}{h_k} \leq \frac{1}{2} h_k. \tag{4.42}$$

By using Equation (4.35), we find

$$\theta_k < 1. \tag{4.43}$$

This means that Newton's method diverges when $\theta_k \geq 1$. If we assumed that $h_0 < 1$, then $\theta_k < \frac{1}{2}$ which imposes more restrictive convergence criteria. As θ is a fraction of the norm of the Newton increments s, this quantity is computable during the process. So we do not need additional calculations.

Our ultimate goal in Newton's method is to find an iterate $x^{(k)}$ which is sufficiently near to the solution x^*. So our termination criteria is defined as

$$\|x^{(k)} - x^*\| < \text{XTOL} \tag{4.44}$$

where XTOL is a user defined constant. As x^* is unknown and inequality (4.44) is only for theoretical interest, we can replace this condition by the computationally cheaper right hand side of inequality (4.37). So our new termination criteria is

$$\frac{\|x^{(k)} - x^{(k+1)}\|}{1 - \frac{1}{2}\omega\|x^{(k)} - x^{(k+1)}\|} \leq \text{XTOL} \tag{4.45}$$

We can rewrite the condition in a cheaper form in terms of the previously computed Newton step $\|s^{(k)}\|$ as

$$\frac{\|x^{(k)} - x^{(k+1)}\|}{1 - \frac{1}{2}\omega\|x^{(k)} - x^{(k+1)}\|} = \frac{\|s^{(k)}\|}{1 - \frac{1}{2}\omega\|s^{(k)}\|} \tag{4.46}$$

$$= \frac{\|s^{(k)}\|}{1 - \frac{1}{2}h_k} \leq \text{XTOL} \tag{4.47}$$

As we already computed the quantity θ_k, we want to express the termination criteria in terms of θ_k instead of h_k. From Equation (4.42) we find $2\theta_k \leq h_k$.

We define the quantity $\bar{h}_k := 2\theta_k \leq h_k$. Also from Equation (4.42), we get $h_{k+1} = \theta_k h_k$. By shifting the index we get $h_k = \theta_{k-1} h_{k-1}$. By using $\bar{h}_{k-1}$, we get $h_k = \theta_{k-1}\bar{h}_{k-1} = 2\theta_{k-1}$. Then we can rewrite our termination criteria as

$$\frac{\|s^{(k)}\|}{1-\theta_{k-1}^2} \leq \text{XTOL} \tag{4.48}$$

4.2.2 Affine Contravariance

In this class we consider problems where $D_F = \text{I}$ and we study the problem with only scaled independent variables.

$$G(y) = F(D_x^{-1}y),\ x = D_x^{-1}y \tag{4.49}$$

For this class of problems, another affine invariant Lipschitz constant is defined as

$$\|(F'(x) - F'(\bar{x}))(x-\bar{x})\| \leq \omega\|F'(x)(\bar{x}-x)\|^2,\ x,\ \bar{x} \in D. \tag{4.50}$$

As we see, both sides of the inequality are independent of D_x^{-1} because

$$G'(y)(\bar{y}-y) = F'(x)D_x^{-1}(\bar{y}-y) = F'(x)(\bar{x}-x). \tag{4.51}$$

We will now state the affine contravariance version of the Newton convergence theorem.

Theorem 4. *Let $F : D \to \mathbb{R}^n$ be a continuously differentiable function where $D \subset \mathbb{R}^n$ is open and convex. Suppose that $F'(x)$ is invertible for each $x \in D$. Assume that the following affine contravariant Lipschitz condition holds:*

$$\|(F'(y) - F'(x))(y-x)\| \leq \omega\|F'(x)(y-x)\|^2 \tag{4.52}$$

for $x, y \in D$.
Define the open level set $\mathcal{L}_\omega = \{x \in D|\ \|F(x)\| < \frac{2}{w}\}$ and let $\bar{\mathcal{L}_\omega} \subset D$ be bounded. For a given initial guess $x^{(0)}$ for an unknown solution x^* let

$$h_0 = w\|F(x^{(0)})\| < 2 \quad \textit{i.e. } x^{(0)} \in \mathcal{L}_\omega. \tag{4.53}$$

Then the sequence $\{x^{(k)}\}$ of Newton iterates remains in $\mathcal{L}_\omega$ and converges to some solution $x^* \in \mathcal{L}_\omega$ with $F(x^*) = 0$. The iterative residuals $\{F(x^{(k)})\}$ converge to zero at an estimate rate.

$$\|F(x^{(k+1)})\| \leq \frac{1}{2}\omega\|F(x^{(k)})\|^2 \tag{4.54}$$

For the proof of Theorem 4, see [5].

For this class of problems we define a convergence monitor

$$\theta_k := \frac{\|F(x^{(k+1)})\|}{\|F(x^{(k)})\|} \tag{4.55}$$

We extend the definition of h_0 in Equation (4.53) in Theorem 4 by defining $h_k := \omega\|F(x^{(k)})\|$. And by multiplying inequality (4.54) by ω we get

$$h_{k+1} \leq \frac{1}{2}h_k^2 \tag{4.56}$$

then

$$\theta_k = \frac{h_{k+1}}{h_k} \leq \frac{1}{2}h_k < 1. \tag{4.57}$$

This means that for $\theta_k \geq 1$, Newton's method diverges. The termination criteria for this class of problems is

$$\|F(x)\| < \text{FTOL} \tag{4.58}$$

where FTOL is a user defined constant.

4.3 Software

In this section, we present how KINSOL and OCT-NLESOL handle scaling of iteration variables and residuals as well as the termination criteria.

4.3.1 Scaling and Termination Criteria in KINSOL

KINSOL allows manual scaling for both the solution vector and the residual vector. The scaling factors are provided by the user as input to the KINSOL iteration. The user specify two vectors `u_scale` and `f_scale` that holds the diagonal elements for both scaling matrices D_x and D_F.

KINSOL scales the iteration variables and the residuals in all the solution steps of the nonlinear equation system including the Jacobian calculation, solving the linear system and line search. In case the user needs no scaling, D_x and D_F are the identity matrix by default. KINSOL calculates the scaled Newton step norm and the scaled residual norm as $\|D_x x^{(k)}\|_2$ and $\|D_F F(x^{(k)})\|_2$ which is reflected in the code as follows.

```
pnorm = N_VWL2Norm(pp, uscale);
*fnormp = N_VWL2Norm(fval, fscale);
```

where `uscal` and `fscal` are the user input scaling vectors for iteration variables and the residuals respectively.
KINSOL uses the scaled $\infty-$norm, $\|D_x x^{(k)}\|_\infty$ and $\|D_F F(x^{(k)})\|_\infty$, to check for convergence. KINSOL has two termination criteria

$$\|D_F F(x^{(k)})\|_\infty < \text{FTOL} \tag{4.59}$$

and

$$\|D_x \lambda s^{(k)}\|_\infty < \text{STEPTOL} \tag{4.60}$$

Condition (4.59) means that the process stops when the scaled residual is less than a user given tolerance FTOL with a default value equals $(\text{macheps})^{\frac{1}{3}}$. While condition (4.60) means that the KINSOL iteration stops when the scaled globalized Newton step is less than a user specified step tolerance STEPTOL with a default value equals $(\text{macheps})^{\frac{2}{3}}$.

KINSOL considers the termination of the nonlinear iteration caused by fulfilling condition (4.59) as successful termination to the Newton iteration. Whereas it considers the termination upon condition (4.60) may not be successful because the Newton step may be getting smaller and smaller and gets stuck near a point for which the residual is not sufficiently near zero [6].

The `KINSol` subroutine executes the Newton nonlinear iteration . Upon user specified flag the `KINSol` subroutine directs the computation flow to either `KINFullNewton` to compute the full Newton step or `KINLineSearch` to compute the globalized Newton step. Those subroutines return a flag which expresses the status of the subroutine termination. It can hold values like `KIN_SYSFUNC_FAIL`, `KIN_REPTD_SYSFUNC_ERR`, or `STEP_TOO_SMALL`.
The `KINSol` subroutine calls then `KINStop` which is responsible of checking the termination criteria. It checks first for the scaled step norm as follows

```
/* Check for too small a step */

if (sflag == STEP_TOO_SMALL) {

  if (setupNonNull && !jacCurrent) {
```

```
    /* If the Jacobian is out of date, update it and retry */
    sthrsh = TWO;
    return(RETRY_ITERATION);
  } else {
    /* Give up */
    if (strategy == KIN_NONE) return(KIN_STEP_LT_STPTOL);
    else                      return(KIN_LINESEARCH_NONCONV);
  }

}
```

In this code snippet, the function checks whether the scaled step norm is less than the STEPTOL by checking if the `sflag` mentioned above holds the value `STEP_TOO_SMALL`. If this is the case and the Jacobain is outdated, it updates the Jacobian and returns to the `KINSol` subroutine to retry the iteration with the new Jacobian. If the Jacobian is fresh then a failure is reported by returning `KIN_STEP_LT_STPTOL`, in case of non globalized Newton, or `KIN_LINESEARCH_NONCONV` in case of globalized Newton.

The `KINStop` function then checks for the residual norm and it returns success status if the $\infty-$norm of the residual is less than the given tolerance. Otherwise the function returns a flag to continue the iteration of the `KINSol` subroutine.

```
fmax = KINScFNorm(kin_mem, fval, fscale);
if (fmax <= fnormtol) return(KIN_SUCCESS);
return(CONTINUE_ITERATIONS);
```

4.3.2 Scaling and Termination Criteria in OCT-NLESOL

For the scaling of iteration variables, OCT-NLESOL has an option `iteration_variable_scaling` which can be set to $0, 1$, or 2 with a default value 1.

- 0 means no scaling i.e. $D_x = \mathrm{I}$.
- 1 means scaling based on the unit type declarations or the typical magnitude for each iteration variable typ_{x_i}. If typ_{x_i} is provided to the solver then $D_{xii} = 1/\mathrm{typ}_{x_i}$, otherwise $Dx_{ii} = 1$.
- 2 means heuristic scaling, an algorithm which tries different strategies to set the value of Dx_{ii} based on the typical magnitude typ_{x_j}, the start

value, and the bounds `xmin` and `xmax`. For the details of the heurisitic scaling algorithm, see [7].

For the residual equations scaling, OCT-NLESOL has a solver option called `residual_equation_scaling` which takes values from 0 to 5.

- 0 no scaling i.e. $D_F = \mathrm{I}$.
- 1 automatic scaling which selects the scaling factors based on the Jacobian calculation as

$$D_{Fii} = \frac{1}{\|\hat{J}_{.i}(x^{(0)})\|_\infty} \tag{4.61}$$

  where $\hat{J}(x^{(0)}) = F'(x)D_x^{-1}$
- 2 manual scaling which is based on the typical magnitude of the residuals given as typ_{f_i}. In this case $D_{Fii} = 1/\mathrm{typ}_{f_i}$. If typ_{f_i} is not available, $D_{Fii} = 1$.
- 3 hybrid scaling which mixes automatic and manual scaling approaches.
- 4 automatic scaling with forcing updating the scaling factors every iteration.
- 5 automatic re-scaling after full Jacobian update.

OCT-NLESOL uses the same termination criteria as KINSOL with FTOL and STEPTOL are equal to a default value 10^{-15} or they can be set to a user given value by setting the solver option `nle_solver_default_tol`.

4.4 Summary

In this chapter, we discussed the problem of having iteration variables and residuals of different scales and how this affects the norm calculation and hence the convergence. We discussed the approach of affine invariant Lipschitz constants and how this leads to a new algorithm with different termination criteria. Moreover, we investigated how KINSOL and OCT-NLESOL handle both scaling of variables and residuals and the termination criteria.

In the following chapter, we present some experiments in both line search and scaling. For line search, we show the results of KINSOL modifications

presented in Chapter 3. For scaling, we compare the ordinary Newton algorithm with the algorithm based on the affine invariant Lipschitz constant. Moreover, we show the results of testing the OCT-NLEOL with some industrial bench mark models.

Chapter 5

Experiments

We introduced our modification to KINSOL in Subsection 3.4.1. We did two modifications, changing the line search parameters and saving the intermediate values of the iteration variable during the nonlinear iteration. In this chapter, we present some experiments that depend on those modifications to visualize the difference between the convergence of the globalized and non globalized Newton's method.

In Section 4.2, we introduced Deuflhard's approach of defining convergence monitors and new termination criteria which are affine invariant. We introduced two convergence theorems that depend on a new definition of an affine invariant Lipschitz constant. Those theorems correspond to two classes of problems, affine covariance and affine contravariance. The affine covariance convergence theorem led to defining a convergence monitor and a termination criteria in terms of the iteration variable. For this class of problems, we do not care about the bad scaling of the residuals. The affine contravariance theorem, on the contrary, led to a convergence monitor and a termination criteria in terms of the residual. For this class of problems, we do not care about the bad scaling of the iteration variables.

In this chapter, we present the implementation of two versions of Newton's method, for the affine covarince and affine contravariance class of problems. Moreover, we show the results of testing OCT-NLESOL with some industrial benchmark models. We test the effect of using automatic scaling of the residual equations versus no scaling.

5.1 Line Search Experiments

In this section, we present the results of three experiments about the line search globalization of Newton's method which are:

1. The effect of line search globalization on Newton's method convergence.
2. The effect of changing the line search parameters α and β.
3. Tracing the iterations of an initial guess for both basic and globalized Newton's method.

For testing, we use the following nonlinear functions, $F(x), G(x)$, and $K(x)$.

$$\begin{aligned} F_1(x) &= x_1^3 - 3x_1x_2^2 - 1 = 0 \quad (5.1) \\ F_2(x) &= 3x_2x_1^2 - x_2^3 = 0 \quad (5.2) \end{aligned}$$

$$\begin{aligned} G_1(x) &= x_1^3 - 3x_1x_2^2 - 2x_1 - 2 = 0 \quad (5.3) \\ G_2(x) &= 3x_2x_1^2 - x_2^3 - 2x_2 = 0 \quad (5.4) \end{aligned}$$

$$\begin{aligned} K_1 = x_1^8 - 28x_1^6x_2^2 + 70x_1^4x_2^4 + 15x_1^4 - 28x_1^2x_2^6 - \\ 90x_1^2x_2^2 + x_2^8 + 15x_2^4 - 16 = 0 \end{aligned} \quad (5.5)$$

$$\begin{aligned} K_2 = 8x_1^7x_2 - 56x_1^5x_2^3 + 56x_1^3x_2^5 + 60x_1^3x_2 - 8x_1x_2^7 \\ -60x_1x_2^3 = 0 \end{aligned} \quad (5.6)$$

5.1.1 The Effect of Line Search Globalization on Newton's Method Convergence

To show the effect of the line search globalization on the local convergence of Newton's method, we construct a rectangular grid of n points in each direction and we execute the Newton iteration n^2 times by taking each point in the grid as an input initial guess. If the function has several zeros, we give each zero a color. Then we color each point in the grid corresponding to the zero which is obtained by Newton's method when started with this point. We execute the Newton iteration for the whole grid two times,

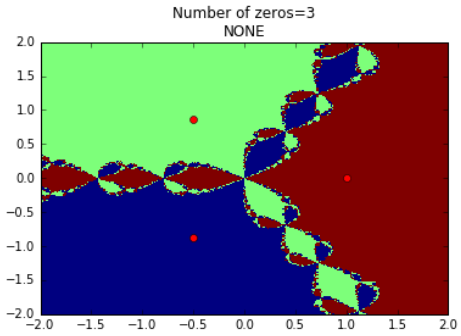
once for non-globalized Newton's method and once for globalized Newton's method.

In the following Python script, we use KINSOL solver through Assimulo, a Cython/Python package for solving ordinary differential equations and differential algebraic equations. Assimulo has interfaces to solvers which are written in C. We define a class called `SolveNonlinearEqSys` which takes the function to be solved, an initial guess and a string indicating the globalization strategy as arguments. The `solve` method defines an object of `KINSOL` which takes an algebraic problem object as an input. Calling `alg_solver.solve()` calls the KINSOL C function KINSol, introduced in Subsection 2.3.1. Figure 5.1 shows the results of testing the `SolveNonlinearEqSys` class for a grid of points from -2 to 2 for the functions $F(x), G(x)$ and $K(x)$.

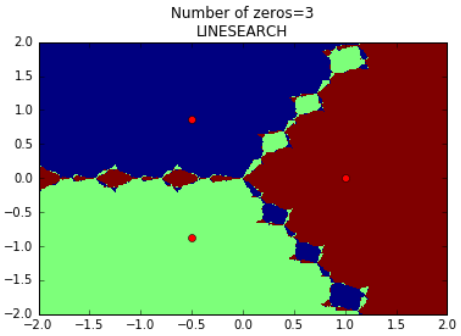
```
from assimulo.solvers import KINSOL
from assimulo.problem import Algebraic_Problem

class SolveNonlinearEqSys:
    def __init__(self, f, initialGuess, globalization):
        self.fcn = f
        self.x=initialGuess
        self.globalization=globalization;

    def solve(self):
        alg_mod = Algebraic_Problem(self.fcn, self.x, name = 'KINSOL
            Example')
        alg_solver = KINSOL(alg_mod)
        alg_solver.globalization_strategy=self.globalization
        alg_solver.max_solves_between_setup_calls=1
        y = alg_solver.solve()
        return y
```

As we see in Figure 5.1, both globalized and non-globalized Newton method produces fractals. Those fractals are created because some points do not converge to the nearest zero. As we notice, all the points in the local neighborhood of a specific zero, inside the radius of convergence, converge to their local zero. Hence, the area around each zero does not contain any fractals. The fractals appear on the separating boundaries between the neighborhood of attraction of each zero. As we notice in Figures 5.1b, 5.1d, and 5.1f, the amount of fractals is reduced remarkably. This indicates that the line search technique modifies the Newton step and tries to guide the search direction

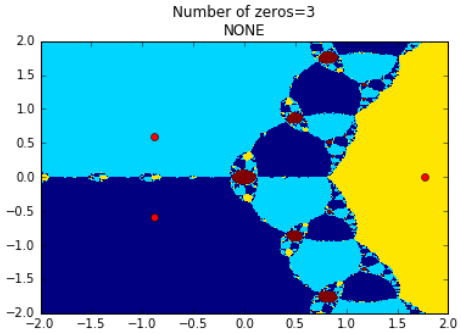
towards the local zero.



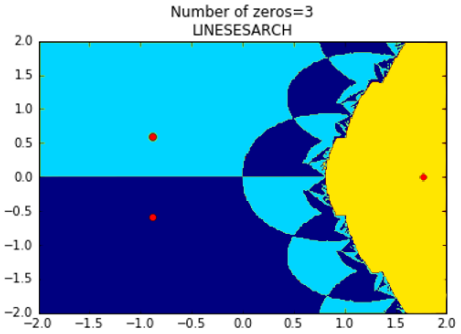
(a) $F(x)$, non-globalized Newton.



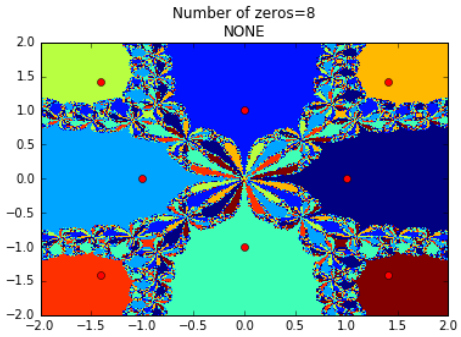
(b) $F(x)$, globalized Newton.



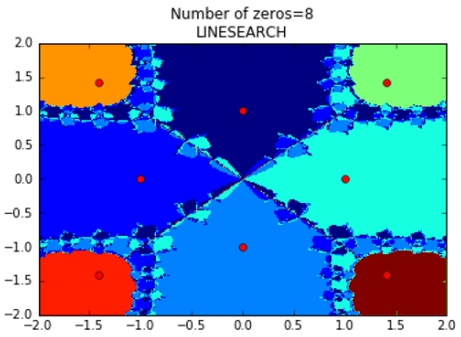
(c) $G(x)$, non-globalized Newton.



(d) $G(x)$, globalized Newton.



(e) $K(x)$, non-globalized Newton.



(f) $K(x)$, globalized Newton.

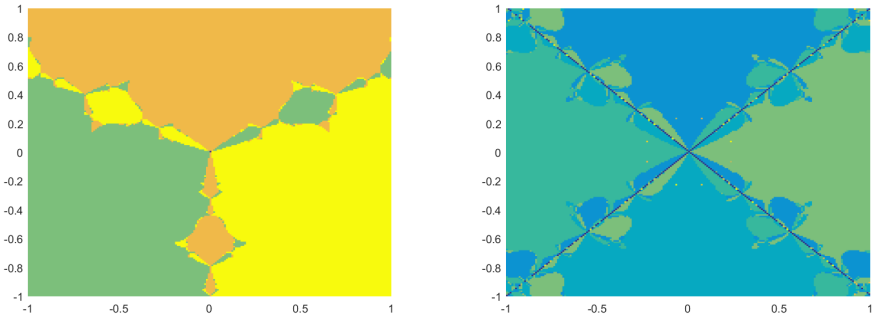
Figure 5.1: Applying Newton's method with and without line search globalizatoin using Assimulo/KINSOL.

Moreover, the line search globalization enhances the possibility of convergence. As we see in Figure 5.1c, the brown areas represent points where the maximum number of nonlinear iterations is exhausted before reaching convergence. While in its corresponding Figure 5.1d, all the points converged to a solution.

Also, we did the same experiment to test OCT-NLESOL for solving the problems $F(x)$ and $K(x)$ on a grid from -1 to 1 by writing the following MATLAB script which uses the FMI toolbox in MATLAB to access the OCT-NLESOL. As the OCT-NLESOL implements only a globalized version of Newton's method and it doesn't contain the option to deactivate the globalization strategy, we only have the results of the globalized Newton which are shown in Figure 5.2. As we notice, the results match with those in Figure 5.1b and 5.1f.

```
fmuName = compileFMU(modelName, compiler, 'modelPath', lib,
    'options', opt);
fmu = FMUModelME1(fmuName);
fmuProblem = FMUProblem(fmu, {'x1'; 'x2'}, {'res_0'; 'res_1'});

solRecord=zeros(1,3);
for i=1:x1dim
    for j=1:x2dim

       solRecord(1,1)=x1grid(1,i);
       solRecord(1,2)=x2grid(1,j);
       fmuProblem.setInitialGuess(solRecord(1,1:2)');

       solver = Solver(fmuProblem);
       solver.setOptions('max_iter',50);
       solver.setOptions('max_iter_no_jacobian',2);
       solver.setOptions('log_level',0);
       solver.setOptions('residual_equation_scaling',0);

       try
           sol = solver.solve();
           [foundZeros,solRecord(1,3)]=
               classifySol_zeroUpdate(foundZeros,sol);
       catch
           solRecord(1,3)=-1;
       end
       pointClassification=[pointClassification;solRecord];
       solver.delete();
```

```
    end
    disp('End of outer for loop number:...')
    disp(i)
end

disp('plotting....')
plotSolutionClasses(pointClassification,foundZeros);
```



(a) Without line search (b) With line search

Figure 5.2: Solving $F(x)$ and $K(x)$ using globalized Newton's method - FMIT/OCT-NLESOL.

5.1.2 Changing the Line Search Parameters

In the previous experiment, we visualized the difference in Newton's method convergence with and without line search globalization. We noticed that using line search reduces the amount of fractals. However, the figures produced by globalized Newton's method still contain noticeable fractals. This might be because KINSOL implements an inexact line search based on the Wolfe curvature conditions (3.12) and (3.23). We mentioned in Section 3.4 that KINSOL defines α and β as constants which equal 10^{-4} and 0.9 respectively. This gives a wide range for a potential λ^*.

This experiment is the result of changing the values of α and β discussed in Subsection 3.4.1. For this we do the following steps:

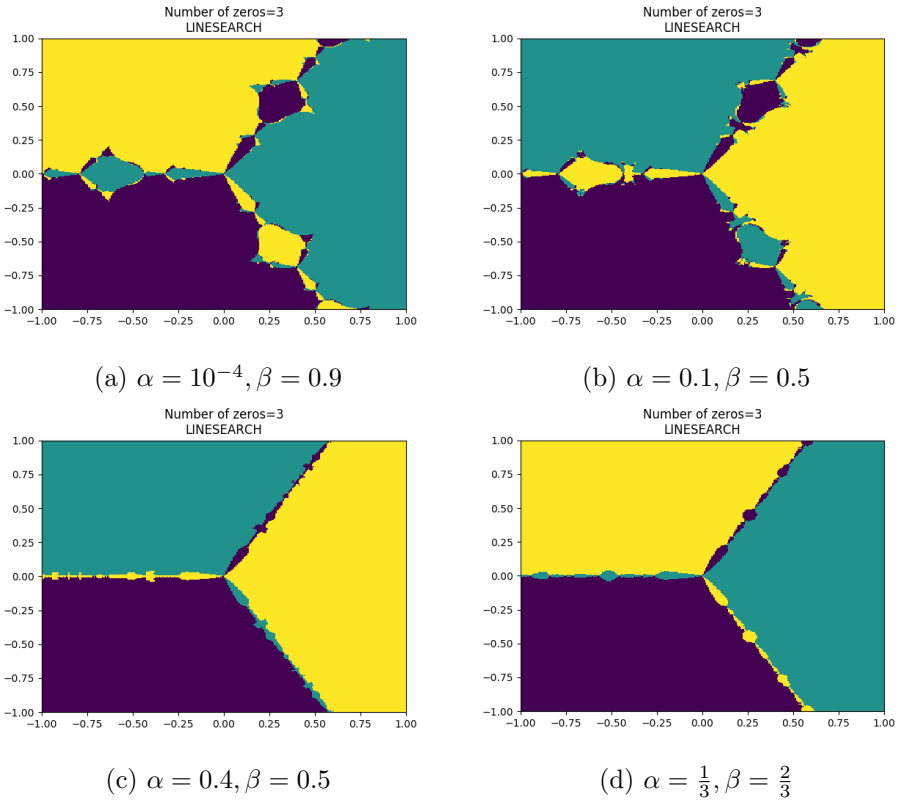
1. We change the value of α and β in the `KINLineSearch` subroutine in the file `kinsol.c`.

2. We build the KINSOL source code using cmake.

3. We build Assimulo so that it links to the newly built KINSOL.

4. We run the Python script in the previous experiment.

In Figure 5.3, we see the effect of giving α and β different values on the fractals produced by testing the solver with $F(x)$. As we notice, narrowing the range of the potential λ^* decreased noticeably the fractals on the boundaries between the function zeros.



(a) $\alpha = 10^{-4}, \beta = 0.9$ (b) $\alpha = 0.1, \beta = 0.5$

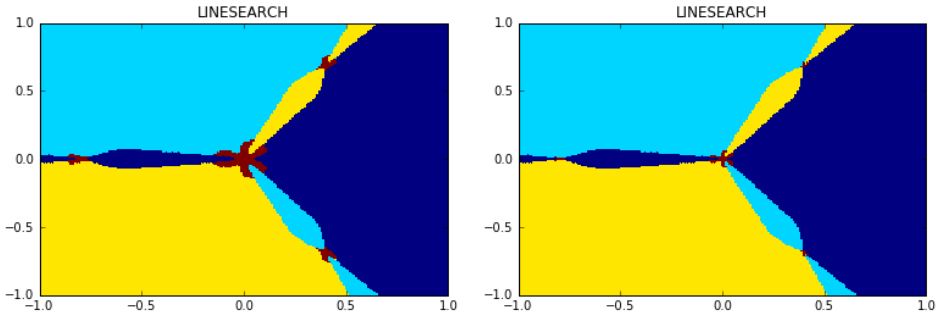
(c) $\alpha = 0.4, \beta = 0.5$ (d) $\alpha = \frac{1}{3}, \beta = \frac{2}{3}$

Figure 5.3: Changing the line search parameters for $F(x)$.

As we mentioned in Chapter 3, the line search methods are divided into exact and inexact methods. To test the effect of the exact line search globalization on Newton's convergence, we added the following two functions to the `SolveNonlinearEqSys` class. Function `solveNewton_LS` implements

the globalized Newton iteration. After it computes the Newton direction, it calls the `find_min_lmda` function which computes λ^* according to the uniform line search method presented in Section 3.2. Figure 5.4 shows the results of solving $F(x)$ with different interval subdivision for λ. In Figure 5.4a we subdivide the λ interval $[0, 1]$ to 100 points while in Figure 5.4b to 1000 points.

```
def find_min_lmda(self,xk,sk):
    lmdaArr=linspace(0,1,50)
    fValArr=zeros(50)

    for k in range(0,50):
        fnorm=norm(self.fcn(xk+lmdaArr[k]*sk),2)
        fValArr[k]=0.5*fnorm**2
    ind=argmin(fValArr)
    return lmdaArr[ind]

def solveNewton_LS(self):
    xk=self.x
    h=1e-6
    dist=1
    while dist>1e-5:
        f_xk=self.fcn(xk)
        if self.Jac==False:
            Jac_xk=self.approxJac(self.fcn,xk,h)
        else:
            Jac_xk=self.Jac(xk)
        sk=solve(Jac_xk,-f_xk)
        minLmda=self.find_min_lmda(xk,sk)
        xkp1=xk+minLmda*sk
        dist=norm(xk-xkp1,2)
        xk=xkp1
    return xk
```



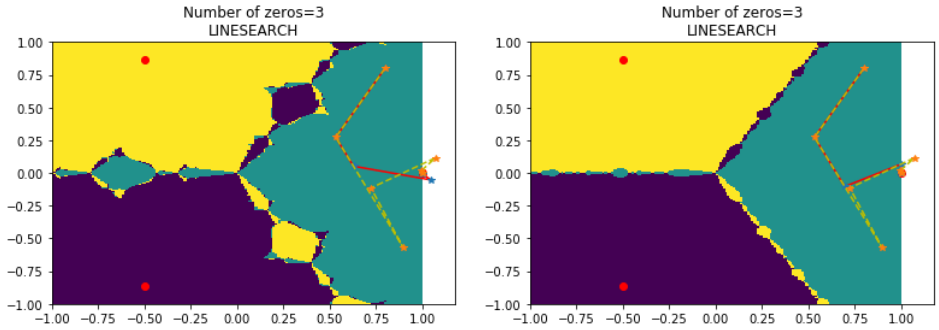
(a) $\Delta\lambda = 0.01$ (b) $\Delta\lambda = 0.001$

Figure 5.4: Newton's method globalization using exact line search.

5.1.3 Tracing a Point With and Without Line Search

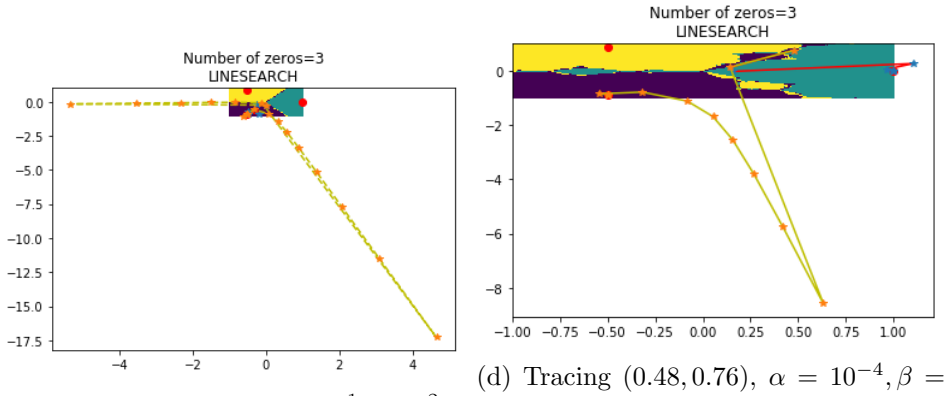
In this experiment, we trace an initial guess during the nonlinear iteration of the basic and globalized Newton. As KINSOL does not save the intermediate values of the solution vector, we modified KINSOL as discussed in Subsection 3.4.1 by saving the iteration variable in a text file. To plot the paths of the traced points, we added a function called `returnIterationVariables` to the `SolveNonlinearEqSys` class which reads the saved intermediate values of the solution vector from the text file, converts them from string to float, and then save them in an array. The function returns the array which can be then plotted.

```
def returnIterationVariables(self):
    file =open("/home/amira/Work/IterationVariables.txt","r")
    iterVar=[]
    iterVar.append(self.x)
    for line in file:
        #print(line)
        lines=line.split(' ')
        lines=lines[0:2]
        linesArr=array(lines)
        linesArrf = linesArr.astype(float)
        #linesArr=array(lines)
        #iterVar.append(float)
        iterVar.append(linesArrf)
    print(array(iterVar))
    return(array(iterVar))
```



(a) Tracing $(0.8, 0.8)$, $\alpha = 10^{-4}, \beta = 0.9$

(b) Tracing $(0.8, 0.8)$, $\alpha = \frac{1}{3}, \beta = \frac{2}{3}$

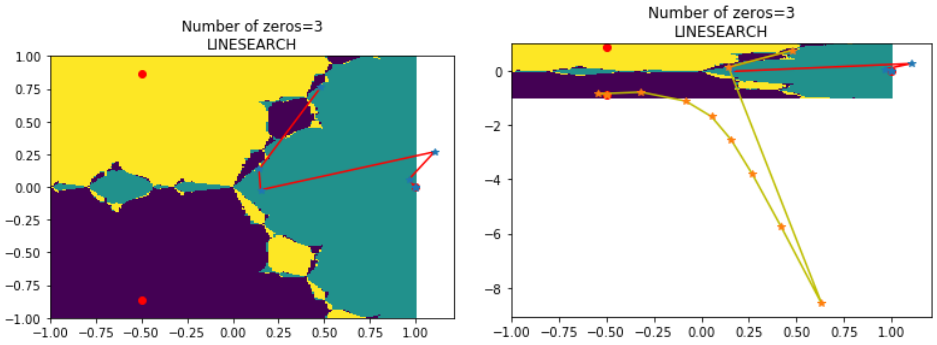


(c) Tracing $(0, -0.25)$, $\alpha = \frac{1}{3}, \beta = \frac{2}{3}$

(d) Tracing $(0.48, 0.76)$, $\alpha = 10^{-4}, \beta = 0.9$

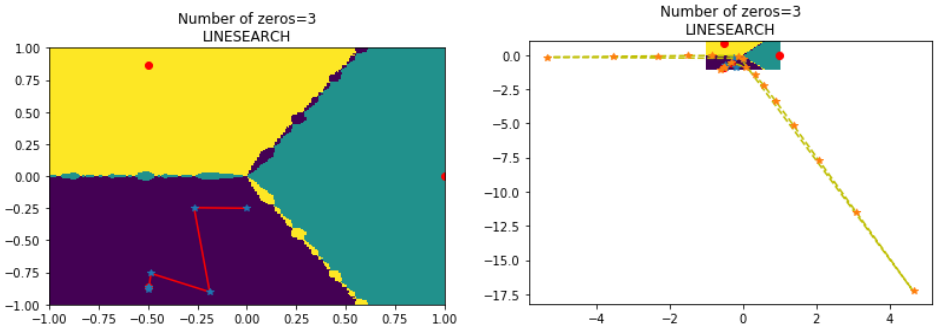
Figure 5.5: Tracing the path of an initial guess to solve $F(x)$ using Newton's method with and without line search globalization with different values for α and β.

Figure 5.5 shows different cases for tracing an initial guess to solve $F(x) = 0$ in Equation (5.1). The yellow path represents the non globalized Newton step while the red path represents the globalized Newton step. Figure 5.5c and 5.5d show the big difference in the method convergence. The figures show that the globalized Newton iteration takes less iterations and it directs the search towards the local zero. To show the big difference between the globalized step and the basic Newton step in Figures 5.5c and 5.5d, we separate the tracing of the point in two different figures one for the classical Newton path and the other for the globalized Newton path, see Figure 5.6 and 5.7.



(a) Iteration path, with line search (b) Iteration path, without line search

Figure 5.6: Tracing $(0.48, 0.76)$ as an initial guess for $F(x) = 0$, $\alpha = 10^{-4}, \beta = 0.9$ with and without Newton globalizaion.



(a) Iteration path, with line search (b) Iteration path, without line search

Figure 5.7: Tracing $(0, -0.25)$ as an initial guess for $F(x) = 0$, $\alpha = \frac{1}{3}, \beta = \frac{2}{3}$ with and without Newton globalizaion.

5.2 Scaling Experiments

In this section, we present some experiments about scaling. It contains two subsections. In the first subsection, we present two subroutines with the

implementation of two versions of Newton's method based on Deuflhard's approach, see Section 4.2. We show the results of testing those two subroutines by modifying $F(x)$, introduced in the beginning of this chapter in Equation (5.1). In the second subsection, we test OCT-NLESOL by some industrial models provided by Modelon AB in order to investigate how scaling affects the convergence of Newton's method implementation.

5.2.1 Experiments Based on Deuflhard's Approach

The following is a Python script of the affine covariant version of Newton's method.

```
def solveDeuflhardCovariant(fnc,x0):
    xk=x0
    f_xk=fnc(xk)
    m=size(f_xk)
    n=size(xk)
    Jac_xk=zeros([m,n])
    Jac_xk=approxJac(fnc,xk,m,n)

    sk=solve(Jac_xk,-f_xk)
    i=0

    for i in range (100):
        xkp1=xk+sk
        f_xkp1=fnc(xkp1)
        Jac_xkp1=approxJac(fnc,xkp1,m,n)

        skp1=solve(Jac_xkp1,-f_xkp1)
        thetak=norm(skp1)/norm(sk)

        convQuant=norm(skp1)/(1-thetak**2)

        if  thetak < 0.5 and convQuant <= 1e-5 :
            return xkp1,i
        xk=xkp1
        sk=skp1
    else:
        raise NewtonException('DH: Maximum iteration attained and no
            convergence')
```

For testing, we modified the function $F(x)$ in Equation (5.1) so that the residual components differ significantly in magnitude. We did this by multiplying the first residual component by 10^{14} and the other component by

10^{-14}. The following are the results of testing the Deuflhard algorithm versus the classical algorithm twice, once using a well scaled $F(x)$ and once using its badly scaled version. We also tested with two different initial guesses, once with $(-0.4, 0.7)$ and once with $(0.5, 0.5)$. The results show that there is no significant difference between the two algorithms in terms of the number of iterations or the possibility of convergence.

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...Deuflhard VS ordinary Newton test for residual scaling...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

--------------------
Well scaled function:
--------------------
Initial guess [-0.4 0.7]
Deuflhard [-0.50000358 0.86602973]
Deuflhard Newton Error [ 3.57588274e-06 4.32671559e-06]
Deuflhard Newton Error Norm 5.6131457438e-06
Deuflhard Newton Number Of Iteration 2

Initial guess [-0.4 0.7]
Original [-0.5      0.8660254]
Original Newton Error [ 2.37732056e-11 2.06226147e-11]
Original Newton Error Norm 3.14715354555e-11
Original Newton Number Of Iteration 4
--------------------
Badly scaled function:
--------------------
Initial guess [-0.4 0.7]
Deuflhard [-0.50000358 0.86602973]
Deuflhard Newton Error [ 3.57586905e-06 4.32675904e-06]
Deuflhard Newton Error Norm 5.61317051757e-06
Deuflhard Newton Number Of Iteration 2

Initial guess [-0.4 0.7]
Original [-0.5      0.8660254]
Original Newton Error [ 2.38582487e-11 2.06954454e-11]
Original Newton Error Norm 3.15835002818e-11
Original Newton Number Of Iteration 4
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...Deuflhard VS ordinary Newton test for residual scaling...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
```

```
---------------------
Well scaled function:
---------------------
Initial guess [ 0.5 0.5]
Deuflhard [-0.49999997 0.86602543]
Deuflhard Newton Error [ 3.13379428e-08 2.75076634e-08]
Deuflhard Newton Error Norm 4.16981798493e-08
Deuflhard Newton Number Of Iteration 7

Initial guess [ 0.5 0.5]
Original [-0.5      0.8660254]
Original Newton Error [ 1.83186799e-15 1.55431223e-15]
Original Newton Error Norm 2.40242104081e-15
Original Newton Number Of Iteration 9
---------------------
Badly scaled function:
---------------------
Initial guess [ 0.5 0.5]
Deuflhard [-0.49999997 0.86602543]
Deuflhard Newton Error [ 3.13391083e-08 2.75064479e-08]
Deuflhard Newton Error Norm 4.16982539861e-08
Deuflhard Newton Number Of Iteration 7

Initial guess [ 0.5 0.5]
Original [-0.5      0.8660254]
Original Newton Error [ 1.55431223e-15 1.44328993e-15]
Original Newton Error Norm 2.12107811032e-15
Original Newton Number Of Iteration 9
```

The following is a Python script of the affine contravariance version of Newton's method.

```
def solveDeuflhardContravariance(fnc,x0):
    xk=x0
    f_xk=fnc(xk)
    m=size(f_xk)
    n=size(xk)
    rhok=norm(f_xk)
    Jac_xk=zeros([m,n])
    Jac_xk=approxJac(fnc,xk,m,n)
    sk=solve(Jac_xk,-f_xk)

    for i in range (100):
        xkp1=xk+sk
```

```
        f_xk=fnc(xkp1)
        rhokp1=norm(f_xk)
        Jac_xkp1=approxJac(fnc,xkp1,m,n)
        sk=solve(Jac_xkp1,-f_xk)
        thetak=rhokp1/rhok

        if  thetak < 1 and rhokp1 <= 1e-5 :
            return xkp1,i
        xk=xkp1
        rhok=rhokp1

    else:
        raise NewtonException('DH: Maximum iteration attained and no
            convergence')
```

Analogically, to test this subroutine, we modified the function $F(x)$ in Equation (5.1) so that the components of the iteration variable differ significantly in magnitude. We did this by multiplying the first component by 10^{10} and the other component by 10^{-10}. The following are the results of testing the affine contravariance Deuflhard algorithm versus the classical algorithm also twice, once using a well scaled iteration variable and once using a badly scaled one. Also, we tested with two different initial guesses, once with $(-0.4, 0.7)$ and once with $(0.5, 0.5)$. As scaling the iteration variable changes the zero of the function, we scale also the initial guess as shown in the results. We also notice from the results that Deuflhard algorithm always shows a minor improvement with respect to the number of iterations.

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...Deuflhard VS ordinary Newton test for Variable scaling...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

--------------------
Well scaled function:
--------------------
Initial guess [-0.4 0.7]
Deuflhard [-0.5      0.8660254]
Deuflhard Newton Error [ 2.37732056e-11 2.06226147e-11]
Deuflhard Newton Error Norm 3.14715354555e-11
Deuflhard Newton Number Of Iteration 3

Initial guess [-0.4 0.7]
Original [-0.5      0.8660254]
Original Newton Error [ 2.37732056e-11 2.06226147e-11]
```

```
Original Newton Error Norm 3.14715354555e-11
Original Newton Number Of Iteration 4
---------------------
Badly scaled function:
---------------------
Initial guess [ -4.00000000e-11 7.00000000e+09]
Deuflhard [ -5.00000000e-11 8.66025404e+09]
Deuflhard Newton Error [ 2.37669670e-21 2.07542419e-01]
Deuflhard Newton Relative Error Norm 2.39649343457e-11
Deuflhard Newton Number Of Iteration 3

Initial guess [ -4.00000000e-11 7.00000000e+09]
Original [ -5.00000000e-11 8.66025404e+09]
Original Newton Error [ 0.00000000e+00 1.90734863e-06]
Original Newton Error Norm 1.90734863281e-06
Original Newton Number Of Iteration 6
```

```
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...Deuflhard VS ordinary Newton test for Variable scaling...
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx

---------------------
Well scaled function:
---------------------
Initial guess [ 0.5 0.5]
Deuflhard [-0.49999997 0.86602543]
Deuflhard Newton Error [ 3.13379428e-08 2.75076634e-08]
Deuflhard Newton Error Norm 4.16981798493e-08
Deuflhard Newton Number Of Iteration 7

Initial guess [ 0.5 0.5]
Original [-0.5      0.8660254]
Original Newton Error [ 1.83186799e-15 1.55431223e-15]
Original Newton Error Norm 2.40242104081e-15
Original Newton Number Of Iteration 9
---------------------
Badly scaled function:
---------------------
Initial guess [ 5.00000000e-15  5.00000000e+13]
Deuflhard [ -4.99999991e-11 8.66025385e+09]
Deuflhard Newton Error [ 9.44222973e-19 1.88446995e+02]
Deuflhard Newton Relative Error Norm 2.1759984633e-08
Deuflhard Newton Number Of Iteration 24
```

```
Initial guess [ 5.00000000e-15 5.00000000e+13]
Original [ -5.00000000e-11 8.66025404e+09]
Original Newton Error [ 6.46234854e-27 1.90734863e-06]
Original Newton Error Norm 1.90734863281e-06
Original Newton Number Of Iteration 27
```

5.2.2 Residual Scaling of Industrial Models

In this subsection, we investigate the effect of residual scaling on solving nonlinear argebraic equations which emerge from some MFUs using OCT-NLESOL. Those FMUs are industrial benchmark models which were used as test cases in [8]. [8] compares and evaluates the robustness of some commercial nonlinear equation solvers which are Minpack, Nleq1 and Nleq2, Tensolve and Dymola Solver. The results lead to the conclusion that the Dymola and Minpack solvers have the highest probability of convergence among the other solvers.

In [8] the robustness of the solver is defined as the ability of a solver to find a solution independently of the quality of the starting point. It compares the robustness of the solvers according to their probability of convergence which depends on the quality of the starting point represented by the distance from the solution. The thesis mentions that the problem of badly scaled state variables may affect the robustness test. So it suggests the normalization of the iteration variables as a solution so that the components of all variables move to the range $[0, 1]$.

By using a MATLAB script provided by Modelon AB, we tested the OCT-NLESOL with the following list of test models given as FMUs to investigate the effect of automatic scaling of the residual equations on the Newton convergence. Table 5.1 records the number of iteration variables in each model as well as the convergence status and the number of iterations taken in both cases of no scaling and automatic scaling.

1. NLA.Melville1993.OA741.VariableGain.Components.Npn.
2. NLA.Melville1993.OA741.VariableGain.Components.Pnp.
3. NLA.Melville1993.OA741.VariableGain.Experiment.
4. NLA.Melville1993.OA741.VariableStimulus.Components.Npn.

5. NLA.Melville1993.OA741.VariableStimulus.Components.Pnp.
6. NLA.Melville1993.OA741.VariableStimulus.Experiment.
7. NLA.AliasDifficult.CounterCurrentReactors2.Experiment.
8. NLA.AliasDifficult.DirectKinematics.Experiment.
9. NLA.AliasDifficult.JermannChair.SubproblemA.Experiment.
10. NLA.AliasDifficult.JermannChair.SubproblemB.Experiment.
11. NLA.AliasDifficult.SynthesisProblem.SubproblemA.Experiment.
12. NLA.Baharev2008.LiquidPhaseSplitC2.LiquidPhaseSplitC2Model.
13. NLA.Lee2001.Experiment.

		no scaling		scaling	
Model No.	No of Iteration Variables	Conv. (Y/N)	No of Iterations	Conv. (Y/N)	No of Iterations
1	3	Y	(1,0)	Y	(1,1)
2	3	Y	(1,0)	Y	(1,1)
3	11	N	(101,0)	N	(2,3)
4	3	Y	(1,0)	Y	(1,1)
5	3	Y	(1,0)	Y	(1,1)
6	11	N	(101,0)	N	(-,-)
7	6	Y	(4,0)	Y	(4,1)
8	11	N	(31,0)	Y	(100,60)
9	14	N	(2,0)	N	(37,2)
10	9	Y	(25,0)	Y	(23,1)
11	9	Y	(75,0)	Y	(100,12)
12	4	Y	(30,0)	Y	(24,1)
13	8	Y	(4,0)	Y	(3,1)

Table 5.1: The result of testing OCT-NLESOL with the test models in case of no scaling and automatic scaling of the residuals.

In Table 5.1, the number of iterations is represented by a pair of two numbers (a, b) where a represents the number of iterations taken by the first Newton solve and b represents the number of iterations of the second Newton solve. The first and seconed Newton solve of OCT-NLESOL are described in Subsection 2.3.2.

As we observe, scaling of the residual equations does not have a noticeable effect in enhancing the convergence on the Newton iteration either with respect to the number of iteration or the possibility of convergence. The only observation is for Model 9 where in the case of 'no scaling', the solver fails to converge after 31 iterations and raises a line search error. The error is that the line search subroutine is unable to find an iterate sufficiently distinct from the current iterate. While in the case of 'automatic scaling' the solver converged after 160 iterations.

5.3 Summary

In this chapter, we showed how the line search globalization modifies the Newton step. As we saw in the figures, the Newton step may be very long and this may lead to a convergence to a zero which is not in the local neighborhood of the initial guess. This was the reason behind the fractals between the zeros boundaries.

We also showed the results of testing two algorithms based on Deuflhard's approach versus the classical algorithm. Our examples showed that there is no significant improvement of Deuflhard algorithms over the classical one. However, Deuflhard algorithms always take negligibly less iterations than the classical algorithm.

As a trial to find some examples where scaling really matters, we used some industrial models to test OCT-NLESOL with respect to scaling of the residuals. The test showed that automatic scaling of the residuals showed almost no difference whether in terms of the number of iterations or the possibility of convergence.

Chapter 6

Conclusion and Future work

In this thesis, we studied the theoretical background of Newton's method globalization using line search as well as the line search subroutine in KINSOL which is built in OCT-NLESOL to answer the question about how the KINSOL line search subroutine works, mentioned in Chapter 1. KINSOL line search subroutine implements the backtracking algorithm that finds the optimal factor of the Newton direction according to Wolfe curvature conditions where their parameters α and β are fixed in KINSOL. We did two modifications to KINSOL code in order to visualize the effect of the line search globalization on Newton's method convergence. The experiments presented in Chapter 5 show that the line search globalization modifies the Newton step attempting to map each initial guess to the nearest solution. This is noticed by the effect of reducing the fractals on the boundary of the convergence ball of each solution.

We mentioned in Chapter 3 that the values of α and β define the upper and lower limits for the inexact λ. The results in Chapter 5 show that changing the values of α and β have the potential to reduce the amount of fractals significantly. A possible future work can be finding a methodology to select the best values of α and β according to the problem itself.

For the question about the effect of scaling on the termination criteria, mentioned in Chapter 1, we showed that the Newton iteration is scaling invariant. We also studied the scaling techniques used in both KINSOL and OCT-NLESOL. We found out that KINSOL scales the iteration variables and residuals manually as a user input. However, OCT-NLESOL uses more options like automatic scaling and heuristic scaling.

From the reason that the Newton step is scaling invariant, Deuflhard's approach adopts the idea of defining new termination criteria and new convergence monitors that do not depend on scaling. Deuflhard found that it is not possible to construct an algorithm that fits all kinds of problems, this is why he divided the class of problems into four categories, mentioned in Section 4.2. We studied and implemented only two categories, the affine covariance and affine contravariance, which are related to the thesis subject. The implementation results in Subsection 5.2.1 did not show significant differences over the classical algorithm.

As an attempt to find a more complicated example where scaling enhances the convergence of Newton's method, we tried to investigate the effect of scaling on some industrial benchmark models provided by Modelon AB. We tested the OCT-NLESOL, Modelon solver, with those models given as FMUs. The results showed that there is almost no difference between the two cases of no scaling and automatic scaling.

As Deuflhard algorithms showed a minor enhancement for our examples by executing negligibly less iterations, a possible future work can be adding the convergence monitor and changing the termination criteria of the OCT-NLESOL and KINSOL based on Deuflhard's approach. In this case, we can test the convergence of Newton's method on more complex industrial models.

Bibliography

[1] J.E. Dennis, JR. , Robert B. Schnabel. *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*. Prentice Hall Series in Computational Mathematics, 1983.

[2] Roger Fletcher. *Practical Methods of Optimization*. Wiley-Interscience, 1990.

[3] Lars-Christer Böiers. *Mathematical Methods of Optimization*. Studentlitteratur AB, 2010.

[4] Andreas Antoniou, Wu-Sheng Lu. *Practical Optimization Algorithms and Engineering Applications*. Springer Science, 2007.

[5] Peter Deuflhard. *Newton Methods for Nonlinear Problems, Affine Invariance and Adaptive Algorithms*. Springer Series in Computational Mathematics, 2004.

[6] Aaron M. Collier, Alan C. Hindmarsh, Radu Serban, Carol S. Woodward. *User Documentation for kinsol v2.9.0*. Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, 2016.

[7] Modelon AB. *OCT Technical Documentation*. 2016.

[8] Michael Sielemann. *Device-Oriented Modeling and Simulation in Aircraft Energy Systems Design*. Technical University of Hamburg, 2012.