A Problem-Solving Environment for the Numerical Solution of Nonlinear Algebraic Equations

A Thesis Submitted to the

College of Graduate Studies and Research

in Partial Fulfillment of the Requirements

for the degree of Master of Science

in the Department of Computer Science

University of Saskatchewan

Saskatoon

By

Thian-Peng Ter

Permission to Use

In presenting this thesis in partial fulfilment of the requirements for a Postgraduate degree from the University of Saskatchewan, I agree that the Libraries of this University may make it freely available for inspection. I further agree that permission for copying of this thesis in any manner, in whole or in part, for scholarly purposes may be granted by the professor or professors who supervised my thesis work or, in their absence, by the Head of the Department or the Dean of the College in which my thesis work was done. It is understood that any copying or publication or use of this thesis or parts thereof for financial gain shall not be allowed without my written permission. It is also understood that due recognition shall be given to me and to the University of Saskatchewan in any scholarly use which may be made of any material in my thesis.

Requests for permission to copy or to make other use of material in this thesis in whole or part should be addressed to:

Head of the Department of Computer Science

176 Thorvaldson Building

110 Science Place

University of Saskatchewan

Saskatoon, Saskatchewan

Canada

S7N 5C9

Abstract

Nonlinear algebraic equations (NAEs) occur in many areas of science and engineering. The process of solving these NAEs is generally difficult, from finding a good initial guess that leads to a desired solution to deciding on convergence criteria for the approximate solution. In practice, Newton's method is the only robust general-purpose method for solving a system of NAEs. Many variants of Newton's method exist. However, it is generally impossible to know *a priori* which variant of Newton's method will be effective for a given problem.

Many high-quality software libraries are available for the numerical solution of NAEs. However, the user usually has little control over many aspects of what the library does. For example, the user may not be able to easily switch between direct and indirect methods for the linear algebra. This thesis describes a problem-solving environment (PSE) called `pythNon`for studying the effects (e.g., performance) of different strategies for solving systems of NAEs. It provides the researcher, teacher, or student with a flexible environment for rapid prototyping and numerical experiments. In `pythNon`, users can directly influence the solution process on many levels, e.g., investigation of the effects of termination criteria and/or globalization strategies. In particular, to show the power, flexibility, and ease of use of the `pythNon` PSE, this thesis also describes the development of a novel forcing-term strategy for approximating the Newton direction efficiently in the `pythNon` PSE.

Acknowledgements

I thank Dr. Raymond J. Spiteri for his generosity, patience, encouragement, and guidance throughout my study as an M.Sc. student. I also thank him for his directions and insights to research. I enjoyed being his student. I thank Dr. David Mould, Dr. Michael Horsch, and Dr. Jacek Szmigielski for their contributions to the final form of this thesis.

I thank the Numerical Simulation Lab for their friendship and many interesting discussions. I also thank the Department of Computer Science for their help and support.

I extend my gratitude to the Saskatoon Chinese Alliance Church for both their spiritual and physical support. I thank my wife, Hsin-ni Lee, for her endless support and love. I thank God for making this thesis possible.

Contents

List of Tables

List of Figures

List of Abbreviations

AD	Automatic Differentiation
CFD	Computational Fluid Dynamics
GUI	Graphical User Interface
HPC	High-Performance Computing
LOF	List of Figures
LOT	List of Tables
NAE	Nonlinear Algebraic Equations
ODE	Ordinary Differential Equation
OZ	Ornstein-Zernike
PDE	Partial Differential Equation
PSE	Problem-Solving Environment
SER	Switched Evolution Relaxation
SVD	Singular Value Decomposition

Chapter 1

Introduction

Many problems of science and engineering can be reduced to a quantifiable form through the process of mathematical modelling. For example, computational fluid dynamics (CFD) is a sophisticated technique based on mathematical modelling to predict fluid (i.e., liquid or gas) flow, heat and mass transfer, chemical reactions, and other related phenomena. CFD allows biologists to study the blood flow in the human body [59], meteorologists to predict weather [75], oceanographers to simulate ocean currents [35], and engineers to study air flow around solid bodies for the design of aircraft and cars [75].

These mathematical models are often described in terms of partial differential equations (PDEs). For example, quasi-linear second-order PDEs appear in many applications. They are of particular interest in CFD [32]. These PDEs can be classified as elliptic, parabolic, and hyperbolic, depending only on the coefficients of the highest-order derivatives, and they represent most of the governing equations in CFD, e.g., Laplace's equation, the heat equation, and the wave equation. Laplace's equation is an elliptic PDE that can be used to describe, for example, the behavior of electric, gravitational, and fluid potentials. It is fundamental to the fields of electromagnetism, astronomy, and fluid dynamics [56]. The (unsteady) heat equation is a parabolic PDE used for example to model the temperature distribution in a given region over time. It is important to the field of thermodynamics [32]. The wave equation is a hyperbolic PDE used to model various types of wave propagation, such as sound waves, light waves, and water waves. It is important to the fields of acoustics, electromagnetics, and fluid dynamics [56].

The approximation of the solution of nonlinear algebraic equations (NAEs) is often required as part of the solution of PDEs. Analytic (closed-form) solutions to PDEs typically do not exist, so

we must approximate the solutions numerically. The method of lines is a popular method for the numerical solution of PDEs. In this method the spatial derivatives are discretized, resulting in a system of ordinary differential equations (ODEs). For parabolic equations in particular, the ODE system is often very large and stiff, thus requiring the use of an implicit time integration method. This leads to a very large system of NAEs to solve at each time step. Figure 1.1 shows an overview of the steps taken from mathematical modelling to systems of NAEs.

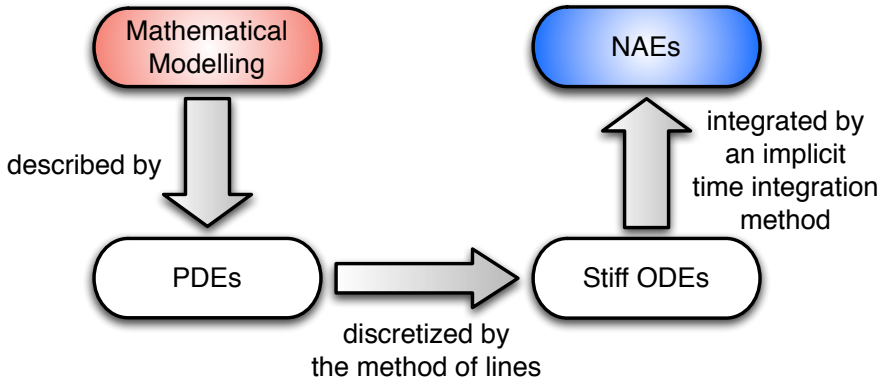

Figure 1.1: An overview of steps taken from mathematical modelling to systems of NAEs.

We denote a system of NAEs by

$$\mathbf{F}(\mathbf{x}) = \mathbf{0}, \tag{1.1}$$

where $\mathbf{F} : \Re^m \to \Re^m$ is the *nonlinear residual function*, $\mathbf{x} \in \Re^m$ is the vector of *unknowns*, and $\mathbf{0}$ is a vector of zeros. We often simply refer to $\mathbf{F}$ as the *residual*. Before attempting to solve (1.1), it is fundamental to analyze the existence and uniqueness of solutions of the system. That is, the system may have a unique solution, multiple solutions, or no solution. Accordingly, we expect to have difficulty computing a solution that does not exist or converging to the desired solution if there is more than one. For example, consider the single NAE

$$f(x) = x^2 + \alpha = 0, \tag{1.2}$$

where x is a real variable, and α is a constant. Depending on the value of α, this equation can have 0, 1, or 2 solutions. If $\alpha > 0$, any numerical method should fail to find a solution because

none exist. If $\alpha < 0$, a method may or may not find the solution, depending on how well we know the properties of the solution desired. For example, if $\alpha = -1$, the roots of (1.2) are 1 and -1. If $\alpha = 0$, (1.2) has one solution. However, in this case the problem is said to be *ill-posed*. In other words, if we perturb the equation by an arbitrarily small amount by adding a constant ϵ, the system has no solution if $\epsilon > 0$ or two solutions if $\epsilon < 0$. This is a dramatic change in outcome (i.e., the number of real roots) from a small change in the problem statement. Figure 1.2 shows the solution plots of $f(x)$ with different values of α. Equation (1.2) is relatively easy to analyze because of its simple form and the fact that it is a one-dimensional problem. As the dimension of the NAEs increases, analysis of existence and convergence of the numerical solution becomes much more difficult. Moreover, there are no foolproof methods that are guaranteed to always converge to a desired solution when there is more than one solution. Thus, systems of NAEs are generally difficult to solve.

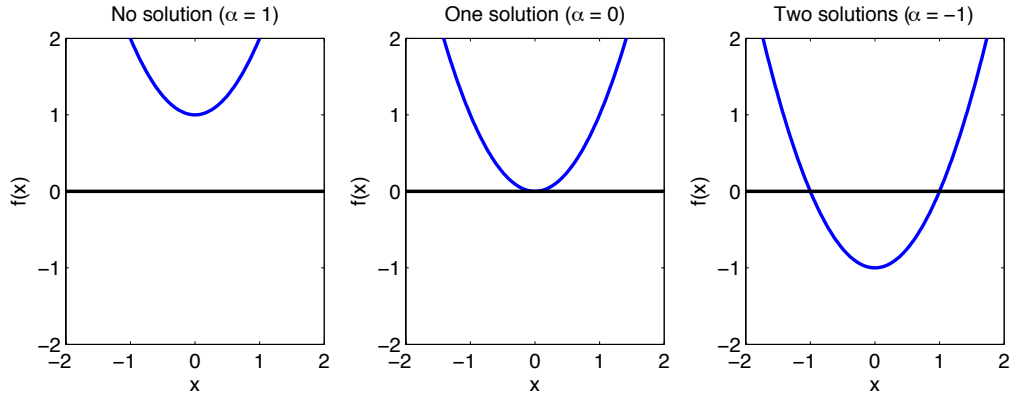

Figure 1.2: The solution plots of $f(x) = x^2 + \alpha$ with different values of α.

In practice, Newton's method is the only mature and efficient method for solving a system of NAEs [63, 5]. Given an initial guess $\mathbf{x}^{(0)}$, the classical version of Newton's method for approximating a desired solution $\mathbf{x}$ to (1.1) is formally defined by the iteration

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} - \mathbf{J}_{\mathbf{F}}^{-1}(\mathbf{x}^{(n)})\mathbf{F}(\mathbf{x}^{(n)}),\ n = 0, 1, 2, \ldots, \tag{1.3}$$

where $\mathbf{x}^{(n)}$ is the nth approximation to the solution of (1.1), and $\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})$ is the *Jacobian matrix*

evaluated at $\mathbf{x}^{(n)}$. The (i,j)-element of the Jacobian matrix $\mathbf{J_F}(\mathbf{x})$ of is defined by

$$[\mathbf{J_F}(\mathbf{x})]_{ij} := \frac{\partial F_i}{\partial x_j}(\mathbf{x}),$$

that is, the partial derivative of the ith component of the residual with respect to the jth unknown. Inversion of the Jacobian matrix is not performed in practice; rather (1.3) is implemented via solution of the following system of linear equations at each iteration:

$$\mathbf{J_F}(\mathbf{x}^{(n)})\mathbf{d}^{(n)} = -\mathbf{F}(\mathbf{x}^{(n)}), \tag{1.4a}$$

followed by the update

$$\mathbf{x}^{(n+1)} = \mathbf{x}^{(n)} + \mathbf{d}^{(n)}, \tag{1.4b}$$

where $\mathbf{d}^{(n)}$ is called the *Newton direction*. The success of Newton's method is based on the following theorem [16] that describes its local convergence to a solution of (1.1):

Theorem 1 (Standard Local Convergence Theorem) *Let $N(\mathbf{x}, r)$ be an open neighbourhood of radius r around $\mathbf{x}$; i.e., $N(\mathbf{x}, r) = \{\tilde{\mathbf{x}} \in \Re^m : \|\tilde{\mathbf{x}} - \mathbf{x}\| < r\}$. Denote the closed line segment connecting $\mathbf{x}, \bar{\mathbf{x}} \in \Re^m$ by $[\mathbf{x}, \tilde{\mathbf{x}}]$. $\mathbf{D} \subset \Re^m$ is called a* convex set *if for every $\mathbf{x}, \tilde{\mathbf{x}} \in \mathbf{D}$, $[\mathbf{x}, \tilde{\mathbf{x}}] \subset \mathbf{D}$. Let $\mathbf{F} : \Re^m \to \Re^m$ be continuously differentiable in an open convex set $\mathbf{D} \subset \Re^m$. Assume that there exists a solution $\mathbf{x}^* \in \Re^m$ to (1.1), and that there exists constants ϵ, β, $L > 0$, where ϵ is sufficiently small, such that $N(\mathbf{x}^*, \epsilon) \subset \mathbf{D}$, $\|\mathbf{J}_\mathbf{F}^{-1}(\mathbf{x}^*)\| \leq \beta$, and $\mathbf{J_F}$ is Lipschitz continuous at $\mathbf{x}^*$; i.e., for all $\mathbf{y} \in N(\mathbf{x}^*, \epsilon)$,*

$$\|\mathbf{J_F}(\mathbf{x}^*) - \mathbf{J_F}(\mathbf{y})\| \leq L\|\mathbf{x}^* - \mathbf{y}\|.$$

Then for all $\mathbf{x}^{(0)} \in N(\mathbf{x}^*, \epsilon)$, the sequence $\mathbf{x}^{(1)}$, $\mathbf{x}^{(2)}$, ... generated by (1.3) is well defined. Moreover, the Newton iteration (1.3) converges to $\mathbf{x}^*$ according to

$$\|\mathbf{x}^{(n+1)} - \mathbf{x}^*\| \leq \beta L\|\mathbf{x}^{(n)} - \mathbf{x}^*\|^2. \tag{1.5}$$

The convergence of the classical Newton iteration described by (1.5) is called q-quadratic; i.e., the error in the approximate solution is squared at each iteration.

We note that Theorem 1 is widely cited in practice because the convergence of Newton's method based on the assumptions of this theorem has an attractive quadratic convergence. Other local

convergence theorems exist for describing the local convergence of Newton's method, such as the Newton-Kantorovich theorem (see, e.g., [16]), which makes no assumption about the existence of $\mathbf{x}^*$ or the nonsingularity of $\mathbf{J_F}(\mathbf{x}^*)$, and the contractive mapping theorem (see, e.g., [16]), which applies to any iterative method of the form $\mathbf{x}^{(n+1)} = \mathbf{G}(\mathbf{x}^{(n)})$, i.e., $\mathbf{G}(\mathbf{x}) := \mathbf{x} - \mathbf{J}_{\mathbf{F}}^{-1}(\mathbf{x})\mathbf{F}(\mathbf{x})$ for Newton's method. However, the assumptions in these theorems result in slower rate of convergence of Newton's method that is not observed in practical computation [16].

Many variants of Newton's method exist for solving systems of NAEs. Solving a system of NAEs requires three major steps: testing for termination of the Newton iteration, computation of the Newton direction, and approximation to the solution based on the Newton direction [38]. Different decisions made in each major step form a variant of Newton's method. For example, computation of the Newton direction requires solving a system of linear equations (1.4a) at each iteration. We can either evaluate and factorize the Jacobian matrix by a direct method, or we can approximate the solution using an indirect method [38]. Figure 1.3 shows some of the choices one can make to form a variant of Newton's method. Many other variants exist as well; see Chapter 2.

Some variants of Newton's method are available in public domain software packages, such as MINPACK [7], NITSOL [55], NKSOL[12], KINSOL [69], and PETSc [8, 9]. MINPACK is a Newton solver that uses direct methods, which are best for solving systems of NAEs that are relatively small because such systems typically have Jacobians that are dense. On the other hand, NITSOL, NKSOL, and KINSOL are Newton solvers that use indirect methods, which are best for solving large systems of NAEs that have Jacobians that are large and sparse. Finally, the SNES library of PETSc is a high-performance Newton solver that contains both direct and indirect methods; it can thus solve systems of NAEs that are small or large and sparse.

Selecting a suitable variant of Newton's method is crucial for solving a system of NAEs efficiently because the nature of a system of NAEs may differ markedly from one problem to another. For example, systems resulting from discretization of PDEs are large; thus storing the Jacobian matrices or their factors is expensive. Indirect methods are better than direct methods to solve such systems

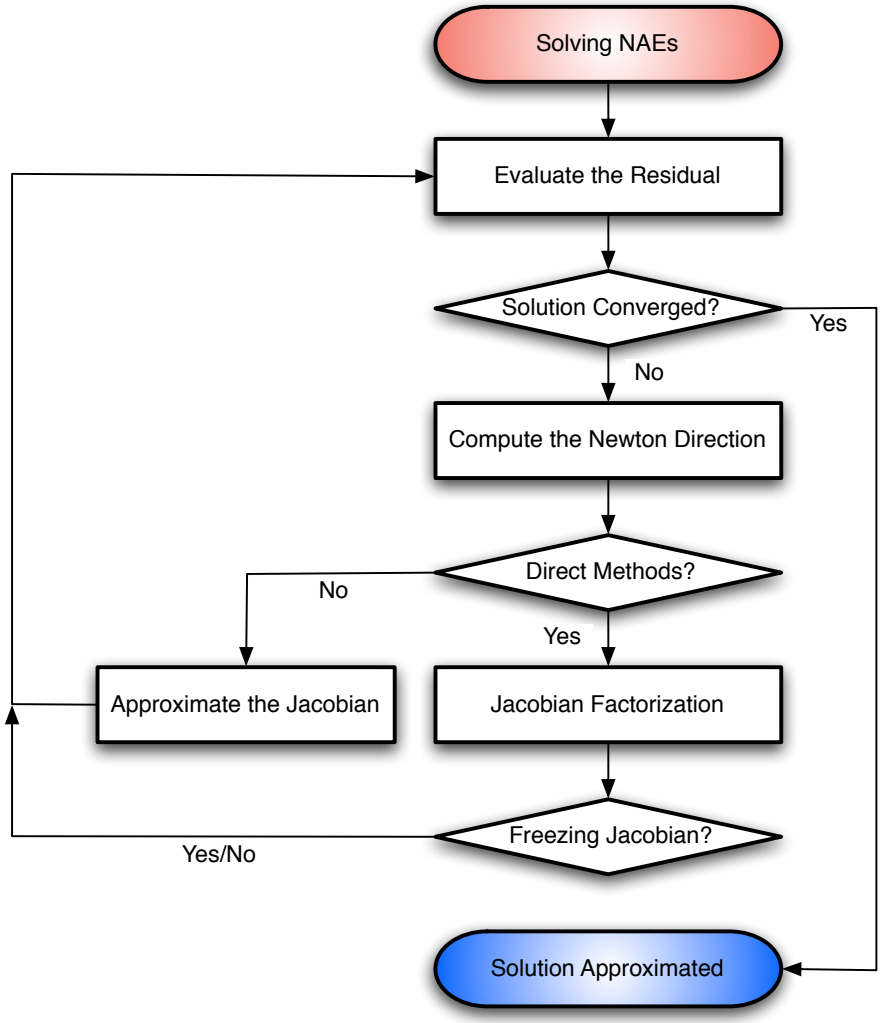

Figure 1.3: Forming a variant of Newton's method.

because they do not require full storage of Jacobians; rather only the effect of a Jacobian-vector product needs to be computed. These methods are therefore also called *matrix-free* methods [38]. Chapter 4 shows an example where a direct method is infeasible to solve a system of NAEs because its Jacobian is too large to store.

However, none of the software packages we have described provides a friendly environment for forming a suitable variant of Newton's method. In other words, we cannot conveniently formulate and experiment with different methods and strategies in the solution process of NAEs. In fact, many of these packages are delivered as highly optimized software libraries for large-scale simulations with massive amounts of data in high-performance computing (HPC) facilities. This means that performance is more important than the convenience and flexibility of switching methods and strategies within the software packages. Moreover, they do not share the same interface or standard file format such as the Matrix Market Exchange Format [48]. Thus, switching from one variant of Newton's method to another generally means switching from one software package to another. It is even more troubling to have to extend the functionality of Newton's method in one software package when such functionality already exists in another.

In this thesis, we describe a problem-solving environment (PSE) called `pythNon` that provides all the computational facilities for studying the effects (e.g., performance) of different strategies for solving systems of NAEs numerically. Other examples of PSEs for solving a more diverse range of problems include MATLAB, Octave, Maple, COMSOL Multiphysics, and Mathematica. Instead of building on an existing PSE, we create a standalone PSE specifically for the numerical solution of NAEs so that the integration of this new PSE as a subsystem of a more sophisticated or specialized PSE is possible. For example, a specialized PSE for the numerical solution of initial value problems in ODEs may use `pythNon` to solve systems of NAEs in an implicit time integration method without the need to include any other PSE. Moreover, `pythNon` is open source; thus it is freely available to the public for use, evaluation, and feedback.

In `pythNon`, the user may find a suitable variant of Newton's method to approximate solutions of NAEs, validate these approximate solutions, and detect numerical difficulties in the solution

process early on. Once the user has found a suitable variant of Newton's method for solving a system of NAEs, in particular for large systems, the user may then easily transfer the variant to a high-performance environment, that is, optimizing and delivering the code as a software library for HPC. Figure 1.4 shows the process of transferring a prototype code to a high-performance software library.

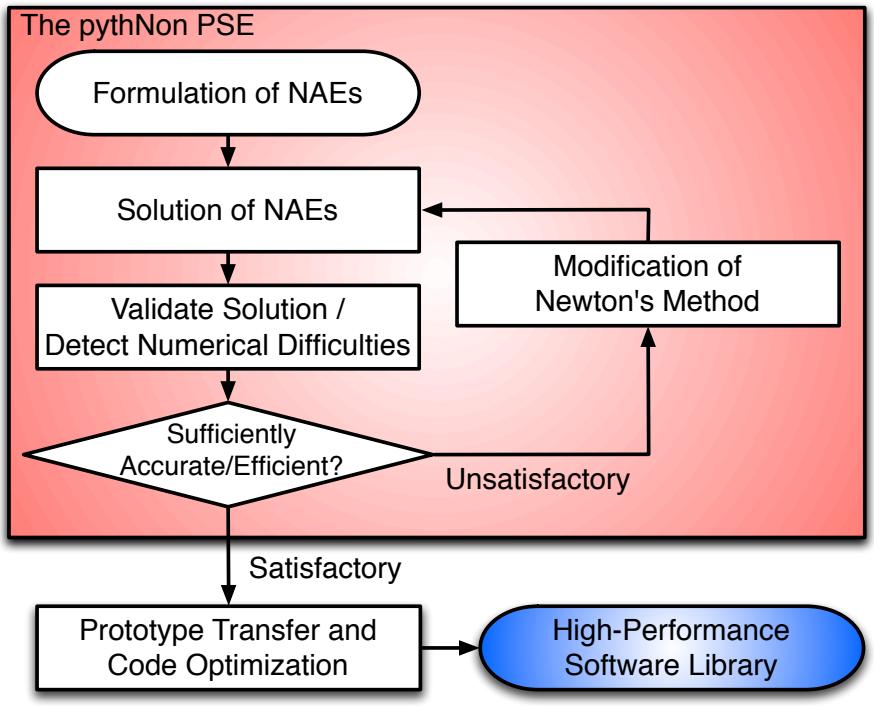

Figure 1.4: NAEs and its solution process.

Users can directly influence the solution process on many levels in `pythNon`. For example, users may specify the termination criterion, the method for solving systems of linear equations (1.4a), and the globalization strategy; see Chapter 2. NAEs and variants of Newton's method may be defined through a text file or a *graphical user interface* (GUI). Users may exploit standard (default) settings without the need to specify each level of the solution process. Moreover, `pythNon` comes with a test suite of benchmark problems for the convenient assessment of new and/or different variants of Newton's method.

To demonstrate the power, flexibility, and ease of use of the `pythNon` PSE, we implement and

evaluate different forcing-term strategies for approximating the Newton direction; see Chapter 4 for details. We have found `pythNon` to be very effective for determining the best forcing-term strategy on a given problem. By solving a number of benchmark problems in `pythNon`, we have found that one of the most popular strategies by Eisenstat and Walker [21] can suffer from *undersolving*; i.e., the strategy fails to approximate a sufficiently accurate $\mathbf{d}^{(n)}$ for the Newton iteration to converge. We have also found that the strategy by An et al. [3] can suffer from *oversolving*; i.e., the strategy imposes too much accuracy on $\mathbf{d}^{(n)}$ when the *local linear model* $\mathbf{F}(\mathbf{x}^{(n)}) + \mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})\mathbf{d}^{(n)}$ poorly approximates the residual $\mathbf{F}(\mathbf{x}^{(n+1)})$; thus Newton's method may fail to converge to a solution. We have proposed a modification to this strategy that ameliorates the effect of oversolving. We have also proposed a new forcing-term strategy that is generally robust and efficient; i.e., it ameliorates the effects of both undersolving and oversolving. Finally, we have found that an adaptive forcing-term strategy should not reduce the forcing term simply based on the agreement of the residual and its local linear model; i.e., the strategy must consider other factors. This unintuitive and hence surprising result brings new insight to the problem of constructing an ideal forcing-term strategy.

Chapter 2 describes a general version of Newton's method and its variants. Chapter 3 describes the design of the `pythNon` PSE. Chapter 4 demonstrates the power, flexibility, and ease of use of the `pythNon` PSE by describing a comprehensive study of forcing-term strategies for Newton's method, including the development of a novel forcing-term strategy for approximating the Newton direction. Finally Chapter 5 discusses our conclusions and future work.

Chapter 2

Newton's Method

Newton's method is by far the most mature and efficient method for solving a system of NAEs in practice, yet the structure among many variants of Newton's method has not been examined formally. In this chapter we first give a brief description of the historical development of Newton's method. We then describe a general algorithm for Newton's method such that any variant of Newton's method is an instance of this algorithm. Finally, we examine some popular variants of Newton's method, in particular Newton direct methods and Newton indirect methods.

2.1 Historical Development of Newton's Method

According to Kollerstrom [42] and Ypma [76], Newton's method is more appropriately referred as the *Newton-Raphson-Simpson method* because of the major contributions by Isaac Newton (1643-1727), Joseph Raphson (1648-1715), and Thomas Simpson (1710-1761). In the following sections, we give a brief description of their contributions and the historical development of Newton's method.

2.1.1 Vieta's Method

Francois Vieta (1540-1603) designed a *perturbation* technique for approximating the solution of the scalar polynomial equations by adding a *correcting term* (see Section 2.1.2 for an example) to the approximate solution of (1.1) [18]. Vieta's method adds one decimal place of accuracy to the approximate solution at each step by calculating successive polynomials of the successive perturbations [76]. We note that the precise origins of this method are not completely clear because certain ancient Greek, Babylonian, and Arabic methods also have this form [76].

2.1.2 Newton's Method

Isaac Newton improved Vieta's method by *linearizing* the successive polynomials, that is, ignoring terms higher than linear at each perturbation. Newton demonstrated his method by solving the cubic equation

$$f(x) = x^3 - 2x - 5 = 0 \tag{2.1}$$

with $x^{(0)} = 2$. Using Vieta's perturbation technique, let the exact solution be $2 + p$, where p is a correcting term with some small value. Substituting $2 + p$ into (2.1) gives the polynomial equation

$$p^3 + 6p^2 + 10p - 1 = 0. \tag{2.2}$$

Newton's method ignores terms higher than linear in (2.2) since p is small. This leads to the approximate solution $10p - 1 = 0$ or $p = 0.1$. Next, the method perturbs the approximate solution p with $0.1 + q$, where q is a correcting term with some smaller value, and forms another polynomial equation

$$q^3 + 6.3q^2 + 11.23q + 0.061 = 0. \tag{2.3}$$

By ignoring terms higher than linear, this leads to the approximate solution $q \approx -0.0054$. Here Newton's method only keeps the first few significant digits of the approximate solution q. Newton observed that the number of significant digits of accuracy of the approximate solution doubled at each perturbation: this is known as *quadratic convergence* [46].

2.1.3 Raphson's Method

Both Vieta's method and Newton's method approximate the solution of polynomial equations by generating intermediate polynomials, such as (2.2) and (2.3), that have the form of the original polynomial at each step. However, these methods become impractical to approximate solutions having higher accuracy because they generate many intermediate polynomials [18]. Joseph Raphson addressed this issue by providing a fully *iterative* scheme that is similar to (1.3). Unlike Newton's method, which uses only the first few significant digits of the approximate solution at each step, Raphson's method retains all the significant digits. Raphson demonstrated his method by also

solving (2.1) with $x^{(0)} = 2$. Thus, the approximate solution to (2.1) is defined by the following iteration:

$$x^{(n+1)} = x^{(n)} - \frac{(x^{(n)})^3 - 2x^{(n)} - 5}{3(x^{(n)})^2 - 2}.$$

This iteration terminates when $x^{(n+1)}$ meets the desired accuracy.

2.1.4 Simpson's Method

Like Vieta's method and Newton's method, Raphson's method forms the equations algebraically rather than using the rules of calculus to form a derivative term. Raphson wrote out the algebraic expressions corresponding to the original function $f(x)$ and its derivative $f'(x)$ in full as polynomials [76]. Thomas Simpson then introduced the use of derivative terms and generalized the iterative method for systems of NAEs (including nonpolynomials) as in (1.3) [76]. Simpson's formulation is now generally referred to as "Newton's Method" [76].

2.2 A General Algorithm

Solving a system of NAEs is sometimes called *root-finding* because solutions $\mathbf{x}^*$ to (1.1) can be called *roots*, and we are interested in finding some or all of them. In Chapter 1 we formally define the classical version of Newton's method shown in (1.4), and we see that it requires the solution of a system of linear equations (1.4a) at each iteration. In practice, the classical Newton iteration (1.4) must be augmented by a *termination criterion* to ensure that the approximate solution is sufficiently accurate, a computationally efficient (or at least feasible) method for solving (1.4a), and a *globalization strategy* to ensure that Newton's method converges to a solution, when one exists, for any initial guess. Algorithm 1 shows the practical version of Newton's method. This algorithm is a template for many existing variants of Newton's method, such as the methods described in Section 2.3 and Section 2.4.

The input arguments of Algorithm 1 are the initial iterate $\mathbf{x}^{(0)}$, the residual function $\mathbf{F}$, the absolute error tolerance τ_a, and the relative error tolerance τ_r.

Algorithm 1 Practical version of Newton's Method.

Input: initial iterate $\mathbf{x}^{(0)}$, residual function $\mathbf{F}$, absolute tolerence τ_a, and relative tolerance τ_r.

Output: the approximate solution $\mathbf{x}$

1: $\mathbf{x} \leftarrow \mathbf{x}^{(0)}$

2: **while** (termination criterion is not met) **do**

3: Choose a forcing term η that determines the appropriate accuracy with which to compute $\mathbf{d}$; see Section 2.2.3

4: Find $\mathbf{d}$ such that $\|\mathbf{F}(\mathbf{x}) + \mathbf{J}_{\mathbf{F}}(\mathbf{x})\mathbf{d}\| \leq \eta\|\mathbf{F}(\mathbf{x})\|$

5: If $\mathbf{d}$ cannot be found, terminate with failure

6: Find a step length λ

7: $\mathbf{x} \leftarrow \mathbf{x} + \lambda\mathbf{d}$

8: **end while**

9: **return** $\mathbf{x}$

2.2.1 Choosing an Initial Iterate

Choosing a good initial iterate is important for finding the desired solution. Recall that in Chapter 1, a system of NAEs may have more than one solution. For example, (1.2) with $\alpha = -1$ has two roots: -1 and 1. Suppose the initial iterate is $x^{(0)} = -1$. This iterate satisfies (2.5) and thus terminates the Newton iteration. The resulting root is then $x^* = -1$. This initial iterate is good (indeed the best possible) if the desired root is $x^* = -1$ because the algorithm terminates in only one iteration. On the other hand, this initial iterate is bad if the desired root is $x^* = 1$. Thus, choosing a good initial iterate allows the algorithm to converge rapidly and avoid undesired solutions. Unfortunately, no general strategy exists for choosing a good initial iterate; in general it is only recommended to choose one that has as many properties of the desired solution as possible; e.g., NAEs that describe a distribution of chemical concentrations require the approximate solution to be nonnegative, and hence a good initial iterate would also likely to be nonnegative [38].

2.2.2 Terminating the Newton Iteration

A termination criterion controls the number of Newton iterations and determines whether a solution is accurate enough. Deuflhard [18] describes three classes of criteria for terminating the Newton iteration, namely by minimizing

- the residual norm $\|\mathbf{F}(\mathbf{x}^{(n)})\|$,
- the error norm $\|\mathbf{e}(\mathbf{x}^{(n)})\|$, where $\mathbf{e}(\mathbf{x}^{(n)}) := \mathbf{x}^* - \mathbf{x}^{(n)}$, or
- the energy norm $\|\mathbf{e}(\mathbf{x}^{(n)})\|_{\mathbf{J}} := \sqrt{\mathbf{e}(\mathbf{x}^{(n)})^T \mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})\mathbf{e}(\mathbf{x}^{(n)})}$ when the Jacobian matrix is symmetric positive definite.

We note that $\mathbf{e}(\mathbf{x}^{(n)})$ cannot be evaluated in practice; however it is often possible to estimate it [38]. Depending on the method used for computing the Newton direction in step 4 of Algorithm 1, each variant of Newton's method must have an appropriate norm for the termination criterion. For example, Newton-GMRES is a Newton variant that computes the Newton direction by minimizing the residual norms over some Krylov subspace (see Section 2.4.1), thus requiring a termination criterion that minimizes the residual norm; see Section 2.4.1. Moreover, depending on the rate of convergence of the Newton variant, one may choose one termination criterion over another that both minimize the same type of norm. For example, the *chord method* or *modified Newton method*, which only uses $\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(0)})$ throughout the Newton iterations, uses a more stringent terminination criterion than a standard implementation of Newton's method would use because its rate of convergence is slower [18]. The termination criterion for the chord method [18] is

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| \leq \sqrt{1-4\theta}\ \text{FTOL},$$

where $0 < \theta < \frac{1}{4}$ is a user-defined constant, and FTOL is a user-defined *residual error tolerance*, is more restrictive than the termination criterion for a standard implementation of Newton's method [18]

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| \leq \text{FTOL}.$$

As previously mentioned, the local convergence theory for Newton's method requires that all approximate solutions $\mathbf{x}^{(n)}$ for $n = 0, 1, 2, \ldots$, be sufficiently near the solution $\mathbf{x}^*$ and that the Jacobian be nonsingular (or more precisely, *well conditioned* [40, page 64]). If so, then the following condition holds [38]:

$$\frac{\|\mathbf{e}(\mathbf{x}^{(n)})\|}{4\kappa(\mathbf{J}_\mathbf{F}(\mathbf{x}^*))\|\mathbf{e}(\mathbf{x}^{(0)})\|} \leq \frac{\|\mathbf{F}(\mathbf{x}^{(n)})\|}{\|\mathbf{F}(\mathbf{x}^{(0)})\|} \leq \frac{4\kappa(\mathbf{J}_\mathbf{F}(\mathbf{x}^*))\|\mathbf{e}(\mathbf{x}^{(n)})\|}{\|\mathbf{e}(\mathbf{x}^{(0)})\|}, \tag{2.4}$$

where

$$\kappa(\mathbf{J}_\mathbf{F}(\mathbf{x}^*)) = \|\mathbf{J}_\mathbf{F}(\mathbf{x}^*)\|\|(\mathbf{J}_\mathbf{F}^{-1}(\mathbf{x}^*))\|$$

is the *condition number* of $\mathbf{J}_\mathbf{F}(\mathbf{x}^*)$. That is, (2.4) compares a relative reduction in the norm of the error with a relative reduction in the norm of the residual [38]. If the Jacobian is nonsingular, that is, $\kappa(\mathbf{J}_\mathbf{F}(\mathbf{x}^*))$ is not very large, then (2.4) leads to the following termination criterion that is well-suited for most variants of Newton's method [38]:

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| \leq \tau_r\|\mathbf{F}(\mathbf{x}^{(0)})\| + \tau_a, \tag{2.5}$$

where $\|\cdot\|$ is a suitable norm. That is, Algorithm 1 terminates the iteration when $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ is relatively small compared to $\|\mathbf{F}(\mathbf{x}^{(0)})\|$. Equation (2.5) requires an absolute error tolerance ($\tau_a > 0$) because an initial iterate that is near the solution may make (2.5) impossible to satisfy. It also requires a relative error tolerance ($\tau_r > 0$) because an initial iterate too far away from the solution may lead to premature termination of the Newton iteration [38].

If Algorithm 1 satisfies the termination criterion in (2.5), $\mathbf{x}$ will be the approximate solution upon output.

2.2.3 Choosing the Forcing Term

Having chosen an initial iterate, we may use an indirect method to compute the Newton direction that satisfies the *inexact Newton condition*

$$\|\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n)})\| \leq \eta^{(n)}\|\mathbf{F}(\mathbf{x}^{(n)})\|, \tag{2.6}$$

where $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n)}) := \mathbf{F}(\mathbf{x}^{(n)}) + \mathbf{J}_\mathbf{F}(\mathbf{x}^{(n)})\mathbf{d}^{(n)}$ is called the *local linear model*. The parameter $\eta^{(n)}$ is called the *forcing term*, which can be varied at each Newton iteration [38]. If we use a direct

method to solve (1.4a) for the Newton direction $\mathbf{d}^{(n)}$ to within roundoff errors, then the forcing term is zero, and (2.6) holds with equality. If we approximate the Newton direction with an indirect method, then the forcing term is chosen based on the properties of the solution and the system. A good choice of forcing term leads to rapid convergence of the iteration. However, no general strategy exists for choosing the forcing term that leads to an optimal convergence of the iteration. In Chapter 4 we investigate the effects of different strategies for choosing the forcing term. We also capitalize on the flexibility offered by `pythNon` to make new observations on well-known forcing-term strategies, propose a modification to improve an existing forcing-term strategy, and ultimately propose a new forcing-term strategy (4.17) that performs well compared to other well-known forcing-term strategies.

2.2.4 Globalization Strategies

According to Theorem 1, if the approximate solution $\mathbf{x}^{(n)}$ is far from $\mathbf{x}^*$, Newton's method may fail to converge to a solution. In particular, it is possible that $\|\mathbf{F}(\mathbf{x}^{(n)})\| > \|\mathbf{F}(\mathbf{x}^{(n-1)})\|$. To achieve an acceptable level of robustness and general applicability, Newton's method must be augmented by a globalization strategy to ensure that for any given initial iterate, the iteration converges to a solution. Methods such as line search methods [17, 20, 37, 53], trust region methods [17, 57, 31], and continuation methods [14, 74] are available to accomplish this.

Line search methods search for a decrease in the residual norm $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ along the line segment $[\mathbf{x}^{(n)}, \mathbf{x}^{(n)} + \mathbf{d}^{(n)}]$. In other words, these methods determine a fraction λ of the *full Newton step* $\mathbf{d}^{(n)}$ so that $\|\mathbf{F}(\mathbf{x}^{(n)} + \lambda\mathbf{d}^{(n)})\| < \|\mathbf{F}(\mathbf{x}^{(n)})\|$. The quantity λ is often called *step length*, and $\lambda\mathbf{d}^{(n)}$ is called the Newton step. For example, Armijo's rule [4] is a line search method that terminates with the smallest $l \geq 0$ such that

$$\|\mathbf{F}(\mathbf{x}^{(n)} + \lambda^{(l)}\mathbf{d}^{(n)})\| \leq (1 - \nu\lambda^{(l)})\|\mathbf{F}(\mathbf{x}^{(n)})\|, \tag{2.7}$$

where $\lambda^{(0)} = 1$, and $\nu \in (0, 1)$ that makes (2.7) easy to satisfy [38]. Following Dennis and Schnabel [16], the default value of ν in `pythNon` is 10^{-4}; thus the line search method may terminate with only a modest reduction in the residual. A common way to determine $\lambda^{(l)}$, $l \geq 1$, is to minimize

the *merit function*

$$\phi(\lambda) = \frac{1}{2}\|\mathbf{F}(\mathbf{x} + \lambda\mathbf{d})\|^2$$

using a quadratic polynomial that interpolates values of $\phi(\lambda)$ [16]. The step length $\lambda^{(l)}$ is then taken to be the minimum of this polynomial; see Section 4.4.5 for an example.

Instead of moving along the original Newton direction $\mathbf{d}^{(n)}$ in a line search method, trust region methods search for a solution inside a "trusted region" described by the linear model

$$\mathbf{F}(\mathbf{x}^{(n)}) + \lambda\mathbf{J}_{\mathbf{F}}(\mathbf{x}^{(n)})\tilde{\mathbf{d}}^{(n)},$$

where $\tilde{\mathbf{d}}^{(n)}$ is typically selected as a linear combination of two or more candidate directions, one of which approximates $\mathbf{d}^{(n)}$. Generally, we choose λ such that it minimizes the norm of this linear model [41].

We note that both line search and trust region methods require a reduction in the norm of the residual at each step. Accordingly, they lead to iterations that either converge to a solution, diverge to infinity, or stagnate at a point where the Jacobian is singular (e.g., at a discontinuity of $\mathbf{F}(\mathbf{x})$) [14]. Continuation methods, on the other hand, allow an increase in the residual. These methods embed the given problem in a family of problems. For example, a continuation method can be defined by solving a sequence of NAEs

$$\tilde{\mathbf{F}}(\mathbf{x}; \mu^{(k+1)}) := \mathbf{F}(\mathbf{x}) + (\mu^{(k+1)} - 1)\tilde{\mathbf{F}}(\mathbf{x}^{(k)}; \mu^{(k)}) = \mathbf{0},$$

where $\mu^{(k)}$ is an artificial parameter, and $\mathbf{x}^{(k)}$ is the approximate solution to $\tilde{\mathbf{F}}(\mathbf{x}; \mu^{(k)}) = \mathbf{0}$. For certain values of $\mu^{(k)}$, these NAEs can be solved more easily. For example, by construction, $\mathbf{x}^{(0)}$ is a solution of $\tilde{\mathbf{F}}(\mathbf{x}; 0) = \mathbf{0}$. The solution of $\tilde{\mathbf{F}}(\mathbf{x}; 1) = \mathbf{0}$ is the solution of (1.1) [65]. The idea is to increment $\mu^{(k)}$ using a step selection scheme and use the solution to $\tilde{\mathbf{F}}(\mathbf{x}; \mu^{(k)}) = \mathbf{0}$ as the initial guess for solving $\tilde{\mathbf{F}}(\mathbf{x}; \mu^{(k+1)}) = \mathbf{0}$. For example, the *switched evolution relaxation* (SER) method increments the step in inverse proportion to residual norm progress [47]:

$$\mu^{(k+1)} = \mu^{(k)} \cdot \frac{\|\tilde{\mathbf{F}}(\mathbf{x}^{(k-1)}; \mu^{(k-1)})\|}{\|\tilde{\mathbf{F}}(\mathbf{x}^{(k)}; \mu^{(k)})\|}.$$

Algorithm 2 shows a typical implementation of Newton's method with continuation methods as globalization strategies. Note that we do not discuss these methods in further detail because we

only include line search methods such as Armijo's rule for computing step length within `pythNon`; however, the functionality exists such that the user may easily specify other line search methods within `pythNon`.

Algorithm 2 Newton's method with continuation method as globalization strategy.

Input: initial iterate $\mathbf{x}^{(0)}$, initial step size $\mu^{(0)}$, residual function $\mathbf{F}$, absolute tolerence τ_a, and relative tolerance τ_r.

Output: the approximate solution $\mathbf{x}$

1: $\mathbf{x} \leftarrow \mathbf{x}^{(0)}$
2: $\mu \leftarrow \mu^{(0)}$
3: **while** $(\mu \leq 1)$ **do**
4: **while** (termination criterion is not met) **do**
5: Choose a forcing term η that determines the appropriate accuracy with which to compute $\mathbf{d}$; see Section 2.2.3
6: Find $\mathbf{d}$ such that $\|\tilde{\mathbf{F}}(\mathbf{x};\mu) + \mathbf{J}_{\tilde{\mathbf{F}}}(\mathbf{x})\mathbf{d}\| \leq \eta\|\tilde{\mathbf{F}}(\mathbf{x};\mu)\|$
7: If $\mathbf{d}$ cannot be found, terminate with failure
8: $\mathbf{x} \leftarrow \mathbf{x} + \mathbf{d}$
9: **end while**
10: Update μ with a step selection scheme
11: **end while**
12: **return** $\mathbf{x}$

2.2.5 Computation of the Newton Direction

Most of the computational cost in Newton's method is the computation of the Newton direction $\mathbf{d}^{(n)}$ in step 4 of Algorithm 1. This requires the storage and factorization of the Jacobian matrix in the case of a direct method or the approximation of the Newton direction in the case of an indirect method. In other words, in order to minimize the overall computational cost of solving a system of NAEs, we should minimize the combined cost of computing the Newton direction and the number

of iterations required for convergence. Kelley [38] points out that some of the most important issues in selecting a variant of Newton's method are the size of the problem, the cost of evaluating the residual and the Jacobian matrix, and the way of solving (1.4a). In the next two sections, we discuss how one can approximate a solution to (1.4a) by rapidly prototyping a computationally efficient (or at least feasible) variant of Newton's method based on these issues.

2.3 Newton Direct Methods

If the size of the problem is small, i.e., it requires a relatively small amount of computer resources (e.g., memory), and the computation of the residual is inexpensive, direct methods are often adequate to solve (1.4a) efficiently. The advantages are that direct methods are generally more robust than indirect methods because they do not have the possible convergence failure of an indirect method [38].

Direct methods require the formation and storage of the Jacobian matrix in Newton's method. A convenient way to approximate the Jacobian matrix is via the use of *finite differences* [38]. Depending on the nature of the problem, the resulting Jacobian matrix can be stored in different forms, e.g., as a dense matrix or a banded matrix. Alternatively, one may provide a code to evaluate the Jacobian matrix or use *automatic differentiation* (AD) [30] to compute an analytical Jacobian matrix. AD is an algorithm that applies the chain rule of differentiation to the floating-point evaluation of a function and its derivatives [66]. It is more robust than finite differences because the resulting derivative values are accurate to within round-off and do not contain discretization and cancellation errors. It also does not have the possible human errors of a user-defined Jacobian.

The **LU** decomposition [27] is a popular method to factorize the Jacobian matrix. Depending on the nature of the problems, other factorizations such as the Cholesky decomposition, the **QR** decomposition, and the Singular Value Decomposition (SVD) [70] exploit the special properties of the Jacobian matrix.

Because the formation and storage of the Jacobian matrix are costly, the chord method or modified Newton method stores and uses only the initial Jacobian $\mathbf{J_F}(\mathbf{x}^{(0)})$ throughout the Newton

iteration. Such a strategy will only work if $\mathbf{J_F}(\mathbf{x}^{(0)})$ closely approximates $\mathbf{J_F}(\mathbf{x}^*)$. Similarly, *Shamanskii's method* [38] updates the Jacobian only if it is inaccurate or the rate of convergence of the residual is too slow. This may require more iterations to approximate the solution, but because the Jacobians and/or their factorizations can be stored from one iteration to the next, each iteration is much less expensive, and the overall cost for solving the problem is often lower [38].

2.4 Newton Indirect Methods

2.4.1 Newton-Krylov Methods

When the problem size is large, i.e., it requires a relatively large amount of computer resources, storing the Jacobian matrices and/or its factors may not be feasible. Newton-Krylov methods are iterative methods that can be used to solve such systems. These methods do not require storage of Jacobians or its factors; rather only the effect of a Jacobian-vector product needs to be computed. These methods are therefore also called *matrix-free* methods [38]. These methods often require *preconditioners* to speed up the convergence of the iterative solution to (1.4a); in fact, the iteration may not converge at all otherwise. That is, we left-multiply (1.4a) by a preconditioner $\mathbf{M}$ so that an indirect method to solve the linear system

$$\mathbf{M}\mathbf{J_F}(\mathbf{x}^{(n)})\mathbf{d}^{(n)} = -\mathbf{M}\mathbf{F}(\mathbf{x}^{(n)})$$

converges rapidly. Section 4.4.4 shows an example where an indirect method requires a preconditioner in order to obtain a solution for a two-dimensional steady-state convection-diffusion equation. A discussion of different preconditioners is beyond the scope of this thesis; see e.g., Trefethen and Bau [70] for further details.

Given an initial iterate $\mathbf{d}^{(0)}$, a Krylov iterative method for approximating the solution to (1.4a) is defined by the iteration

$$\mathbf{d}^{(k)} = \mathbf{d}^{(0)} + \mathbf{K}^{(k)}\mathbf{c}^{(k)},$$

where

$$\mathbf{K}^{(k)} = \left[\begin{array}{c|c|c|c} \mathbf{r}^{(0)} & \mathbf{J_F}(\mathbf{x}^{(n)})\mathbf{r}^{(0)} & \ldots & \mathbf{J_F}^{k-1}(\mathbf{x}^{(n)})\mathbf{r}^{(0)} \end{array} \right],$$

$\mathbf{r}^{(0)} := -\mathbf{F}(\mathbf{x}^{(n)}) - \mathbf{J_F}(\mathbf{x}^{(n)})\mathbf{d}^{(0)}$, and the initial iterate is generally $\mathbf{d}^{(0)} = \mathbf{0}$ [38]. The Jacobian-vector products $\mathbf{J}_\mathbf{F}^k(\mathbf{x}^{(n)})\mathbf{r}^{(0)}$ for $k = 0, 1, 2, \ldots$ form a basis for the *Krylov subspace*

$$\mathcal{K}_k = \text{span}(\mathbf{r}^{(0)}, \mathbf{J_F}(\mathbf{x}^{(n)})\mathbf{r}^{(0)}, \ldots, \mathbf{J}_\mathbf{F}^{k-1}(\mathbf{x}^{(n)})\mathbf{r}^{(0)}).$$

The method computes the coefficients $\mathbf{c}^{(k)} \in \Re^k$ by minimizing $\|\mathbf{K}^{(k)}\mathbf{c}^{(k)} - \mathbf{r}^{(0)}\|$ and terminates with an approximate Newton direction $\mathbf{d}^{(n)}$ from (1.4a) where $\mathbf{d}^{(n)}$ satisfies the inexact Newton condition (2.6).

GMRES [61] is a popular Krylov method for solving linear equations. It minimizes $\|\mathbf{r}^{(k)}\|^2$ over $\mathcal{K}_k$. GMRES requires an accumulation of the history of the linear iteration as an orthonormal basis for the Krylov subspace [38]. In other words, if the number of iterations gets very large, which often happens for large problems, the method may exhaust the available fast memory, such as cache, which is often relatively small in size. Any given implementation of GMRES may arbitrarily limit the number of iterations, but then the approximate solution may be poor. Low-storage Krylov methods are available, such as BiCGSTAB [71] and TFQMR [23], where the overall strategy is modified so that only a fixed number of basis vectors for the Krylov subspace are stored. Discussion of the details of the various iterative solvers is beyond the scope of this thesis. We note that the storage of the linear iterations in GMRES is still much less compared to the storage of the Jacobian matrix in direct methods because it is assumed that the Jacobian matrix is large and sparse.

2.5 Selecting a Newton Variant

Sections 2.2–2.4 show that selecting a suitable variant of Newton's method is crucial for solving a system of NAEs efficiently. Despite its importance, it is generally impossible to know *a priori* which variant of Newton's method will be effective on a given problem. We now describe in Chapter 3 a PSE that provides a flexible environment for studying the effects of different variants of Newton's method on a given problem.

Chapter 3

A PSE for the Numerical Solution of NAEs

Mathematical software libraries represent a classical way to deliver and support the reuse of high-quality software [60]. Public domain software repositories, such as ACM CALGO [1] and Netlib [49], and commercial libraries, such as IMSL [34] and NAG [51], give access to a comprehensive array of mathematical software libraries. The user usually goes through an iterative process to search, download, and familiarize themselves with these software repositories to find a suitable library. The user may use the GAMS on-line catalogue and advisory system [25] that provides a standard framework for indexing and classifying mathematical software to speed up the search process. However, one may be forced to change from one library to another as the computer resources or the problem sizes change [60]. For example, a user would change from LAPACK [43] to ScaLAPACK [62] when solving systems of linear equations on multicomputer systems, or from LAPACK to SuperLU [68] when solving very large and sparse systems of linear equations of size on the order of millions [19]. Unfortunately, the time and cost of acquiring, learning, and configuring a mathematical software library are beyond what the average scientist and engineer would like to invest [60].

Although software libraries are usually well-tested and provide some form of abstraction and code reuse [60], the user usually has little control over what the library does. MINPACK [7], NITSOL [55], NKSOL[12], KINSOL [69], and PETSc [8, 9] are numerical libraries that differ in their factorization and storage of the Jacobian matrix for solving systems of NAEs. However, for example none of these software libraries offers the flexibility to choose a different strategy in each step of Algorithm 1. The user may wish to choose or compare different strategies for computing the forcing term in Newton's method for better performance. In fairness, these libraries often aim

for computing with massive amounts of data, so performance and efficiency of the software package are more important than flexibility and extensibility.

These issues have led to a different concept in software reuse, namely the *problem-solving environment* (PSE). Rice and Boisvert have given the following description of a PSE [60]:

> A PSE is a computer system that provides all the computational facilities necessary to solve a target class of problems efficiently. The facilities include advanced solution methods, automatic or semiautomatic selection of solution methods, and ways to easily incorporate novel solution methods. They also include facilities to check the formulation of the problem posed, to automatically (or semiautomatically) select computing devices, to view or assess the correctness of solutions, and to manage the overall computational process. Moreover, PSEs use the terminology of the target class of problems, so users can solve them without specialized knowledge of the underlying computer hardware, software, or algorithms. In principle, PSEs provide a framework that is all things to all people; they solve simple or complex problems, support both rapid prototyping and detailed analysis, and can be used both in introductory education or at the frontiers of science.

An example of such a PSE is PELLPACK [33]. It is a software system for solving elliptic PDEs on single and multicomputer systems. This PSE comes with a rich set of PDE solvers, a graphical user interface (GUI), and a knowledge-based system to select a solution method for a given problem automatically. Other examples of PSEs for solving a more diverse range of problems include MATLAB, Maple, COMSOL Multiphysics, and Mathematica.

To implement and evaluate the effectiveness of different variants of Newton's method on a given problem, we have built a PSE called `pythNon`. It is a PSE that provides all the computational facilities necessary for studying the performance of different variants of Newton's method for solving systems of NAEs numerically. It provides the researcher, teacher, or student, with a flexible environment for rapid prototyping and numerical experiments.

The `pythNon` PSE is a research tool. In `pythNon`, users can directly influence the process for solving NAEs on many levels including experimentation with different methods of computing the Newton direction $\mathbf{d}^{(n)}$ and investigation of the effects of termination criteria and/or globalization strategies. NAEs and variants of Newton's method may be defined through a text file or an easy-to-use GUI. Standard (default) settings may be exploited without the need for the user to specifically address each step in Algorithm 1. Moreover, `pythNon` comes with a test suite of benchmark problems for convenient testing of new and/or different variants of Newton's method.

The pythNon PSE is a teaching tool. The teacher or student may wish to investigate more well-understood concepts, such as the benefit of storing and manipulating a banded Jacobian over a dense Jacobian or the efficiency of an indirect method over a direct method by experimenting with different choices easily through the GUI in pythNon. Thus, they can focus on appreciating high-level concepts without concerning themselves with the underlying implementation.

In this chapter we briefly describe some popular software packages for solving a system of NAEs and compare their features with pythNon. Then we describe the flexible and extensible architecture in pythNon. Finally, we describe the problem-solving process in pythNon by means of examples.

3.1 pythNon and Public Domain Software Packages

A variety of software packages such as MINPACK [7], NITSOL [55], NKSOL [12], KINSOL [69], and PETSc (the SNES library) [8, 9] are available to solve a system of NAEs. These software packages are mathematical software libraries that come with some predefined variants of Newton's method. However, the user is responsible for choosing a suitable library for a given problem. Moreover, these software packages do not share the same interface or file format; thus switching from one variant of Newton's method means switching from one software package to another. On the other hand, pythNon offers the flexibility to switch or choose a different strategy in each step of Algorithm 1 easily. Table 3.1 compares the variants of Newton's method in pythNon with those in the public domain software packages. We note that the KINSOL library is a successor of the NKSOL library; thus we exclude the NKSOL library in the comparison. This table shows that pythNon not only offers the option to define a new or different strategy for Newton's method, but it also includes the common methods and strategies among the public domain software packages.

Table 3.2 shows a comparison of the software features in the public domain software packages mentioned in relation to pythNon. Each of these features forms an essential part of an easy-to-use PSE [60]. This table shows that the pythNon PSE is more than just a mathematical software library. It is a software environment that provides the user the facilities to solve problems more easily and efficiently. For example, the user may prototype a Newton variant or change from one

Table 3.1: Newton variants in both `pythNon` and the public domain software packages.

Options	`pythNon`	MINPACK	NITSOL	KINSOL	PETSc
Termination Criterion	User-definable	Fixed	Fixed	Fixed	User-definable
Forcing Term Strategy	Cai et al. [13], Dembo and Steihaug [15], Brown and Saad [12], Eisenstat and Walker [20], An et al. [3], Gomes-Ruggiero et al. [28], user-definable	N/A	Fixed	Eisenstat and Walker [20]	Eisenstat and Walker [20]
Globalization strategy	Line search, user-definable	Trust region	Line search	Line search	Line search, trust region
Computing Jacobian	Finite difference, user-definable	Finite difference, user-definable	Finite difference, user-definable	Finite difference, user-definable	Finite difference, AD, user-definable
Jacobian factorization	**LU**, Cholesky, **QR**, SVD, user-definable	**QR**	N/A	**LU**	**LU**, Cholesky, **QR**, SVD, user-definable
Jacobian updating strategy	Modified Newton method, Shamanskii's method, user-definable	Modified Newton method	N/A	Modified Newton method	Modified Newton method
Method for approximating $\mathbf{d}^{(n)}$	GMRES, BiCGSTAB, TFQMR, user-definable	N/A	GMRES, BiCGSTAB, TFQMR	GMRES, BiCGSTAB, TFQMR	GMRES, BiCGSTAB, TFQMR, user-definable.

Table 3.2: A comparison of the software features in the public domain software packages in relation to `pythNon`.

Features	`pythNon`	MINPACK	NITSOL	KINSOL	PETSc
Rapid prototyping of Newton variants	Yes	No	No	No	No
Easy to change problem definition	Yes	No	No	No	No
Easy to change solver settings	Yes	No	No	No	Limited
Choice of predefined or custom settings	Yes	Limited	Limited	Limited	Limited
Load and save settings of problem and solver settings	Yes	No	No	No	No
GUI	Yes	No	No	No	No
Automated test suite	Yes	No	No	No	No
Suite of benchmark problems	Yes	No	Basic	No	No
Easy access to a wide range of numerical libraries	Yes	No	No	No	Limited

Newton variant to another rapidly and conveniently through a text file or a GUI. On the other hand, these public domain software packages (e.g., NITSOL) often require the user to acquire, learn, and understand the source code to form a new or different variant of Newton's method. We note that Section 3.2.1 and Section 3.3 show how users can take advantage of features in `pythNon` such as easily changing problem definition and solver settings for solving a system of NAEs through a GUI.

3.2 Architecture and Design

The `pythNon` PSE is a software environment to implement and evaluate different variants of Newton's method that emphasizes the principles of ease of use and flexibility. The design of `pythNon` addresses the following requirements:

1. It shall allow the user to conveniently construct multiple NAE solvers in order to experiment with different solvers on a given problem. For example, the user can compare the approximate solutions from a direct method with those of an indirect method.

2. It shall allow the user to extend the functionality of the solver. For example, the developer may implement another strategy for choosing a forcing term without having to rewrite code,

such as the termination criterion or globalization strategy; see Chapter 4 for an example.

3. It shall provide facilities to monitor the Newton iterations and the solution accuracy.

4. It shall provide a suite of test problems for convenient assessment of new and/or different variants of Newton's method.

5. It shall provide an easy-to-use user interface (textual and/or graphical) to define or change problem and solver settings.

The overall structure of `pythNon` adopts the layered model of software architecture [26] as shown in Figure 3.1. In the top layer, the `pythNon` GUI provides the user with an interactive interface to define a problem, specify the solution process, and display solution accuracy and CPU time of `pythNon` solvers. In the second layer, the `pythNon` Controller processes the problem and solution specification, generates instances of a `pythNon` solver, and computes solutions to the problems. In the third layer, SciPy [64] provides the underlying linear algebra package for solving linear systems of equations and gives access to a wide variety of numerical libraries, such as BLAS [11] and LAPACK [43] for performing linear algebra operations and FFTW [24] for performing fast Fourier transform (see Section 4.4.4). In the bottom layer, the Python Standard Library [73] contains extensive and well-designed libraries, such as libraries for text processing and system programming.

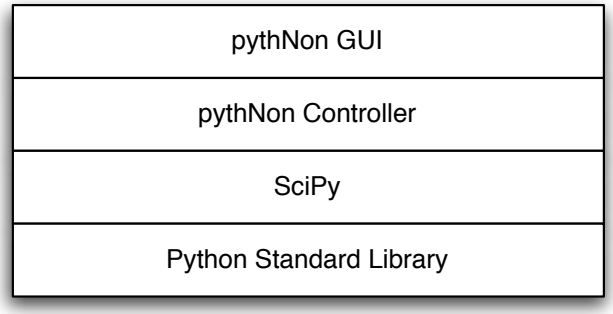

Figure 3.1: System overview of `pythNon`.

The layered approach supports the incremental development of `pythNon`. As a new layer is developed, for example, a platform that supports parallel computing, the services provided by this

layer may be made available to the bottom layer; see Figure 3.1. This architecture is also changeable and portable [67]. For example, the `pythNon` Controller layer can be replaced by another layer, such as a different software package for solving NAEs. As each layer only depends on its lower layer, it is possible to provide multi-platform implementations of `pythNon` [67]. That is, only the bottom layer needs to be changed to take account of the facilities of a different operating system. One caveat of using the layered approach is that performance can suffer because a service request from the top layer may have to be interpreted several times in different layers before it is processed, e.g., loading and saving solver settings from and to a file. To ameliorate this problem, `pythNon` makes some exceptions to communicate directly with the lower layers; for example, the GUI layer calls the Python Standard Library layer directly to load or save solver settings from or to a file.

3.2.1 The `pythNon` GUI

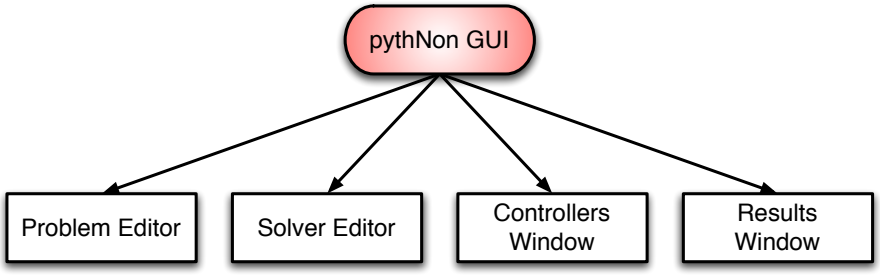

Figure 3.2: The overall structure of the `pythNon` GUI components.

The structure of the `pythNon` GUI in Figure 3.2 includes four major components:

1. The *Problem Editor* is a text window that supports the process of defining the NAEs and the initial iterate. Figure 3.3 shows an instance of the `pythNon` Problem Editor. This editor provides a template with helpful instructions for defining both the NAE and the initial iterate. It also provides some text editing facilities, such as syntax highlighting and code folding, to help the user to define their problem quickly and conveniently. We note that the user may create multiple instances of the Problem Editor for defining different problems.

2. The *Solver Editor* supports the process of defining different variants of Newton's method.

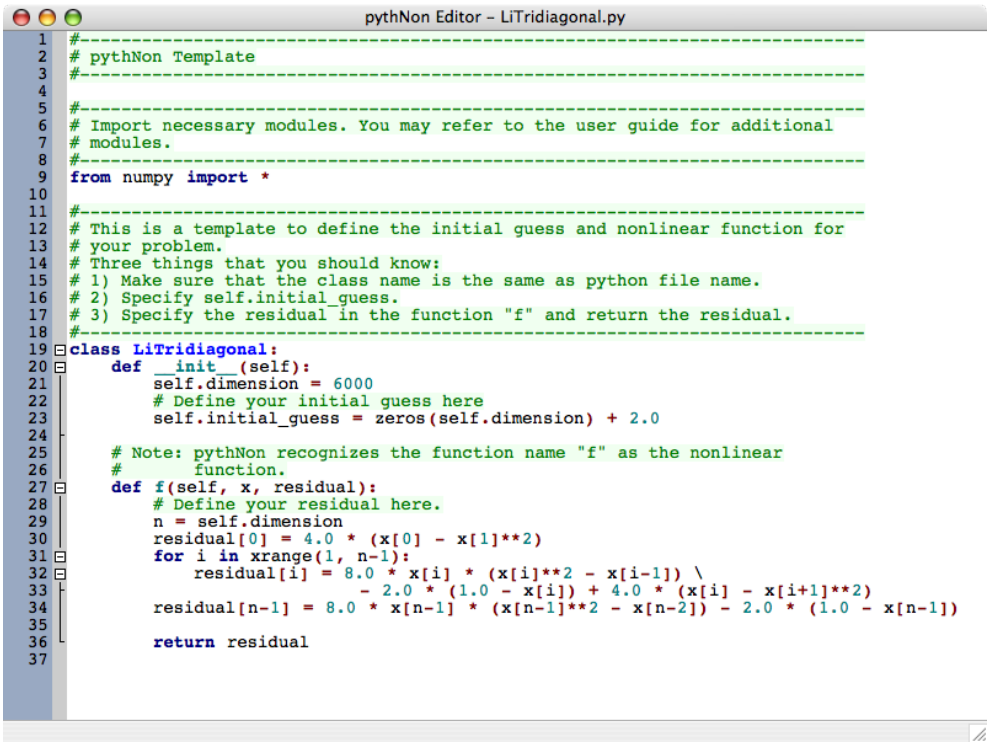

Figure 3.3: An instance of the `pythNon` Problem Editor.

Figure 3.4 shows an instance of the `pythNon` Solver Editor. In this editor, the user specifies a `pythNon` controller that represents a variant of Newton's method. The user may exploit the standard settings or specifically address each step of Algorithm 1 in the *Advanced Settings*. The user may also specify the output of the approximate solution and a reference solution for numerical comparison. Again, these settings may be stored to a file for future use.

3. The *Controllers Window* contains a list of `pythNon` controllers defined by the user. Figure 3.5 shows the Controllers Window. The user may select a `pythNon` controller from the list to view the settings of a Newton variant; e.g., Figure 3.5 shows the settings of a Newton direct method with dense Jacobian. The user may add or remove any number of `pythNon` controllers from the list. After defining and selecting a set of `pythNon` controllers, the user may run the controllers for solving problems or stop the controllers during the solving process. The Controllers Window also shows the current status of each `pythNon` controller, such as the elapsed time of the running `pythNon` controller or whether the `pythNon` controller has terminated successfully.

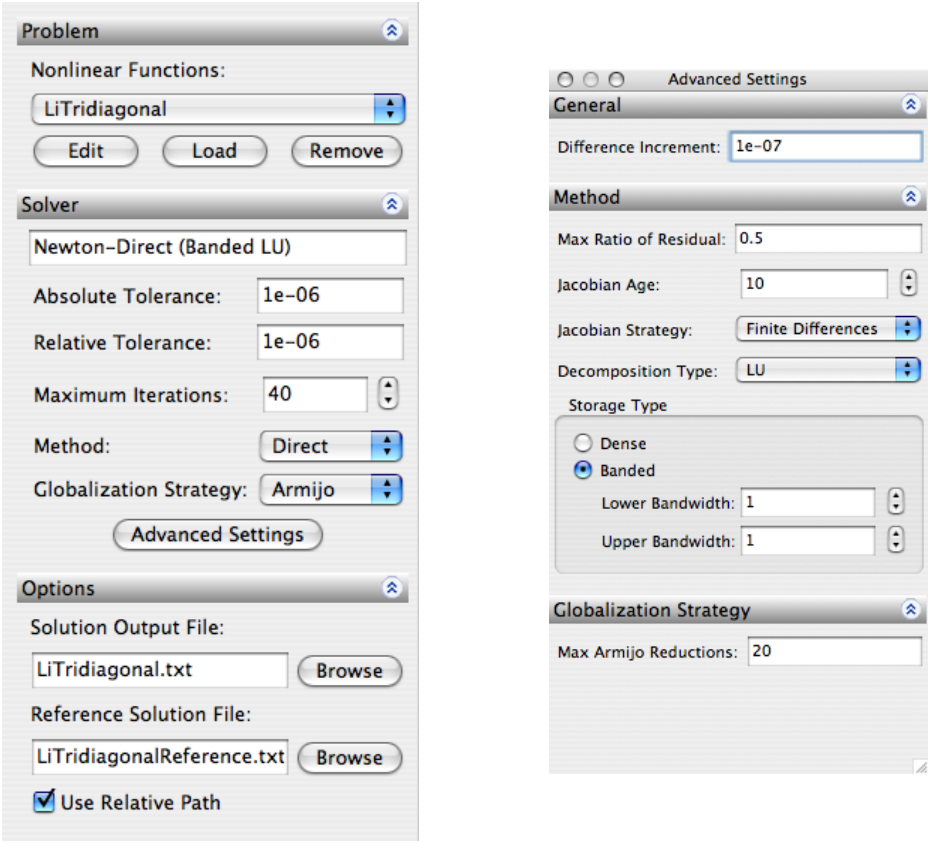

Figure 3.4: An instance of the `pythNon` Solver Editor.

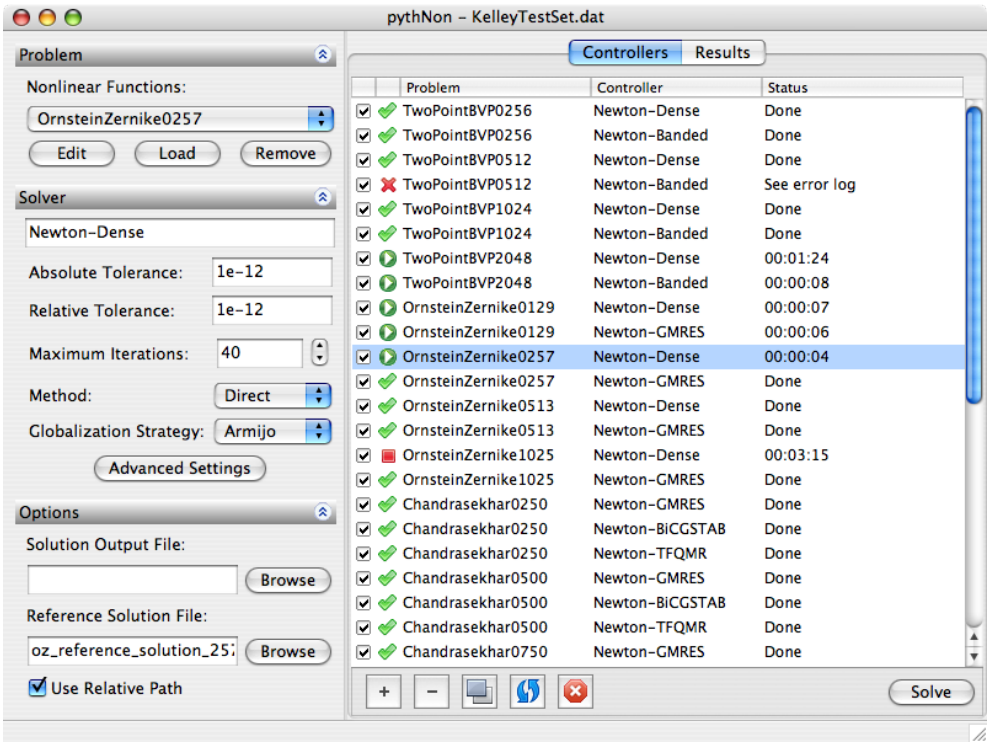

Figure 3.5: The `pythNon` Controllers Window.

4. The *Results Window* shows the solution accuracy obtained and CPU time (in seconds) required of each `pythNon` controller on a given problem. Figure 3.6 shows an instance of the Results Window. Again, the user may select a `pythNon` controller from the list to view the settings of a Newton variant. The user may save these results for further analysis.

3.2.2 The `pythNon` Controller

The architecture of the `pythNon` controller adopts the repository model [26] as shown in Figure 3.7. The main idea is to create a central data structure that stores all information of a `pythNon` controller and define a collection of independent modules. The *Solver Dictionary* module is the central data structure for the `pythNon` controller. It stores the information for a Newton variant and the Newton iteration, such as error tolerances and residuals. On the other hand, each independent module is responsible for acquiring data from the Solver Dictionary module, processing the data, and updating the data to the Solver Dictionary module. These independent modules are:

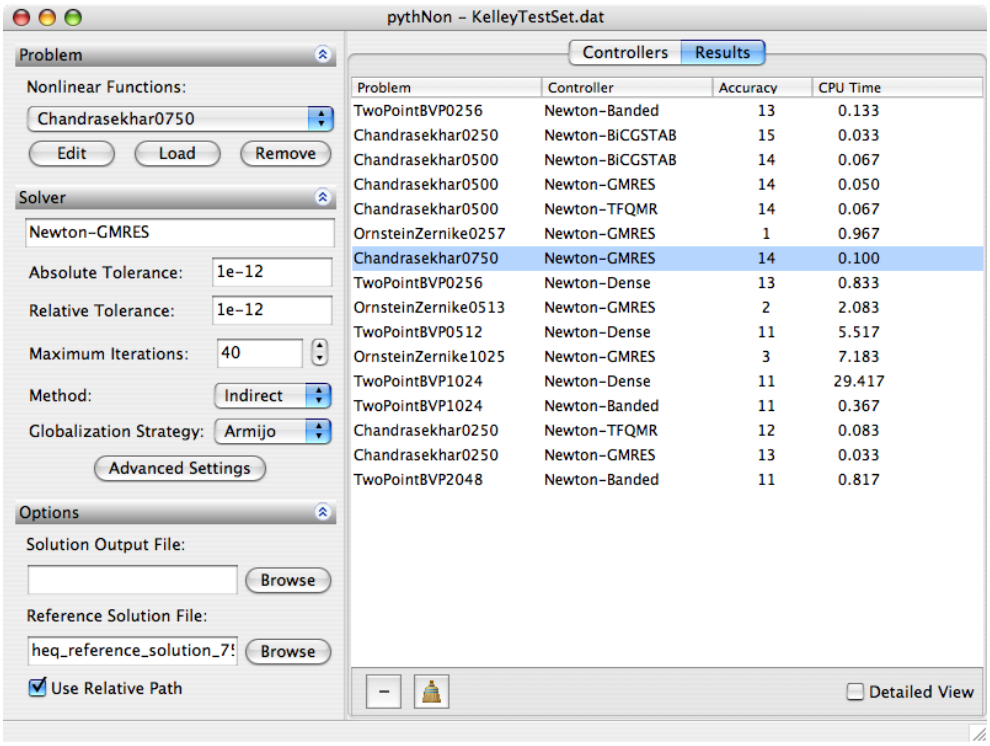

Figure 3.6: The `pythNon` Results Window.

1. The *Termination Criterion* module defines the termination criterion of Newton's method and determines whether to terminate the Newton iteration.

2. The *Jacobian* module provides the facilities to compute or approximate the Jacobian, such as by finite differences.

3. The *Linear Solver* module contains a set of linear equations solvers to compute and approximate the Newton direction $\mathbf{d}^{(n)}$.

4. The *Globalization Strategy* module defines the globalization strategy so that Newton's method converges to a solution from an arbitrary initial guess (or terminates gracefully with an appropriate error message).

5. The *Input Parser* module processes the input settings from a text file or GUI and initializes the Solver Dictionary module.

6. The *Automated Test Suite* module compares the solution from the Solver Dictionary module

with a reference solution for a suite of benchmark problems. The reference solution may be given by the user or included as part of the test suite; see Section 3.4.

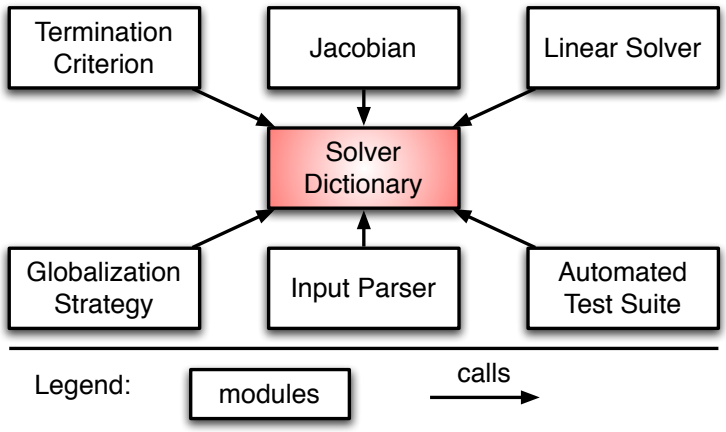

Figure 3.7: Architecture of the `pythNon` Controller.

The repository approach is an efficient way to share large amounts of data. For example, there is no need to pass a large solution vector from one module to another. This architecture is also extensible. For example, one can easily add a preconditioner module to the `pythNon` controller given that it acquires and updates information from and to the Solver Dictionary module. One caveat of this approach is that it may be difficult to distribute the Solver Dictionary module over a number of machines. However, it is possible to distribute a logically centralized repository over a distributed environment [67] using the *distributed memory* paradigm [72], such as pyMPI [58].

3.3 Problem-Solving in `pythNon`

The `pythNon` PSE is written in the Python programming language. Python is becoming increasingly popular in the scientific computing community. Because Python is an interpreted language with a concise syntax, the resulting programs are easy to read and understand. Similar to the MATLAB PSE, `pythNon` can be run in interactive mode using the Python interpreter. This mode allows the user to test small pieces of code easily and learn by trial-and-error [2]. With all these features, `pythNon` enables users to prototype their problems rapidly and conveniently.

Suppose we want to solve a simple NAE in `pythNon`, such as

$$F_1(x) = x_1^2 - x_2 + 0.25, \tag{3.1a}$$

$$F_2(x) = -x_1 + x_2^2 + 0.25, \tag{3.1b}$$

with an initial guess $\mathbf{x}^{(0)} = \mathbf{0}$. To solve this problem using `pythNon`, we would write the following program in the interactive mode or to a text file.

```
# Load pythNon
import pythNon
# Define the Solver Dictionary
info = {}
# Define the nonlinear function
def f(x, residual):
    residual[0] = x[0]**2 - x[1] + 0.25
    residual[1] = -x[0] + x[1]**2 + 0.25
info['function'] = f
# Define the initial guess
info['initial_guess'] = zeros(2)
# Set up a pythNon Controller
controller = pythNon.Controller(info)

# Run the pythNon Controller
controller.Solve()
```

This simple program is concise and easy to read. It first defines the problem by providing both the nonlinear residual function $\mathbf{F}$ and the initial iterate $\mathbf{x}^{(0)}$. Then it passes the problem definition to the Solver Dictionary. Finally, it sets up a `pythNon` controller with the Solver Dictionary and runs the `pythNon` controller to solve the problem. The user may change the problem definition easily by modifying the function `f`.

The simple program above shows that the minimum required input for solving a system of NAEs in `pythNon` consists of the nonlinear function and the initial iterate. In other words, `pythNon` provides a set of predefined (default) settings of a Newton variant, such as termination criterion, method for computing the Newton direction $\mathbf{d}^{(n)}$, and globalization strategy. Figure 3.8 shows the steps of using the `pythNon` PSE to solve a system of NAEs.

The `pythNon` PSE offers flexibility without compromising simplicity. The user may change the solver settings to suit their problem-solving needs. For example, to change the (default) error tolerances and the method for computing the Newton direction $\mathbf{d}^{(n)}$, the user may define a text file

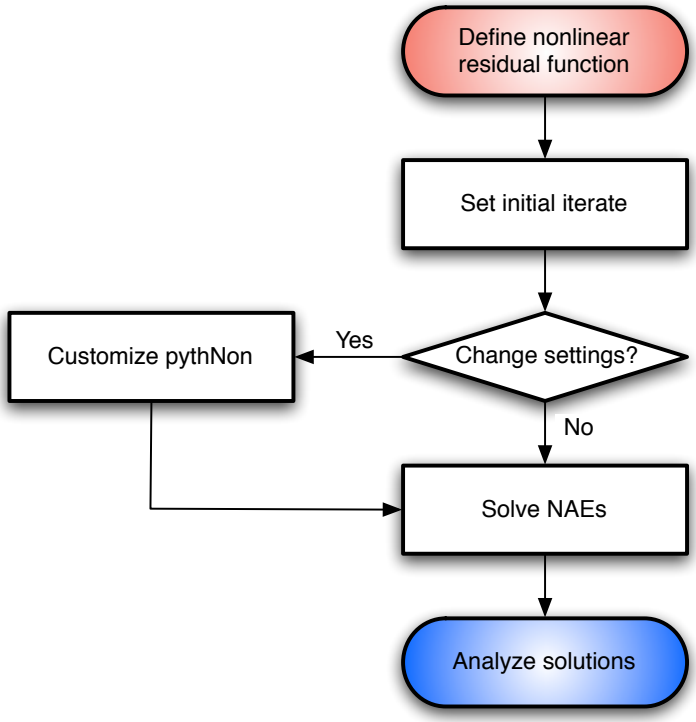

Figure 3.8: Using `pythNon` to solve NAEs.

for `pythNon` to read in:

```
method = indirect
absolute tolerance = 1e-12
relative tolerance = 1e-12
```

This text file changes the method for solving the linear systems (1.4a) from the default direct method to the default indirect method (GMRES) for approximating the Newton direction $\mathbf{d}^{(n)}$ and sets both absolute and relative tolerances to 10^{-12}. Again, `pythNon` provides the standard (default) settings; the user does not need to specifically define each solver setting. Table 3.3 shows some of the predefined settings in `pythNon`.

As mentioned in Section 3.2.1, the user may define the problem, load, save, and change solver settings easily through an easy-to-use GUI. Also included with `pythNon` is an automated test suite so that the user may evaluate different solution methods and analyze the computed solutions by comparing them with the reference solutions. The next section discusses the automated test suite in detail.

3.4 Automated Test Suite

The `pythNon` PSE provides an environment to conveniently formulate and experiment with different variants of Newton's method. The process of forming a variant of Newton's method in `pythNon` directly influences the solutions generated and the manner in which they are obtained. Certain decisions in the solution process such as the initial iterate used, the way of computing the Newton direction, and the globalization strategy, may lead to an undesired solution or no solution at all. For example, Section 4.2 gives an example where a Newton direct method with dense **LU** decomposition solver fails to return a solution when solving a large and sparse system of NAEs. As mentioned in Chapter 1, a system of NAEs may have multiple solutions, and whichever one is obtained by a numerical method will depend on the initial iterate; thus Newton's method may return a mathematically correct but undesired solution with no way to warn the user. This is problematic because the user may not know how to go about obtaining the desired solution.

Another issue when forming a suitable variant of Newton's method is computational efficiency.

Table 3.3: Some predefined settings in `pythNon`.

Settings	Options currently available	Default setting
General		
Absolute tolerance	> 0	10^{-6}
Relative tolerance	> 0	10^{-3}
Maximum number of Newton iterations	≥ 1	40
Method for computing $\mathbf{d}^{(n)}$	Direct, indirect	Direct
Globalization strategy	Armijo's rule, backtracking	Armijo's rule
Direct methods		
Computing Jacobian	Finite difference, user-defined Jacobian	Finite difference
Decomposition	**LU**, Cholesky, **QR**, SVD	**LU**
Storage type	Dense, banded	Dense
Jacobian updating strategy	Modified Newton method, Shamanskii's method	Shamanskii's method
Indirect methods		
Method for approximating $\mathbf{d}^{(n)}$	GMRES, BiCGSTAB, TFQMR	GMRES
Forcing-term strategy	User-defined constant, Cai et al. [13], Dembo and Steihaug [15], Brown and Saad [12], Eisenstat and Walker [20], An et al. [3], Gomes-Ruggiero et al. [28]	Eisenstat and Walker [20]
Maximum number of linear iterations	≥ 1	40
Globalization strategy		
Maximum step length $\lambda_{\max}$	$0 < \lambda_{\max} \leq 1$	1

As mentioned in Chapter 1, the use of an implicit time integration method for solving stiff ODEs requires solving a large system of NAEs at each time step. Thus, the user wants to find a variant of Newton's method that solves the system as efficiently as possible.

Testing is a key process to help the user to identify desired solutions and performance (i.e., CPU time) produced by `pythNon`. It also increases our level of confidence in both the variant of Newton's method used and the solutions produced. For example, by creating a test suite of benchmark problems, the user can formulate new or different variants of Newton's method and verify the solutions with the test suite.

A problem with testing is that the number of test problems and configurations of `pythNon` can be large. Repeating the testing manually can be slow, laborious, and error prone. To alleviate these problems, we build a framework for testing that is automated and thus can be run conveniently and frequently. We describe this framework next.

3.4.1 Automating the Tests

A common practice of testing is to maintain an automated test suite [10]. This test suite in `pythNon` runs some tests and compares the results of the tests with the results from the reference solutions automatically. For example, the test suite reads in a text file called the *configuration file* that contains the settings of a `pythNon` controller, where each `pythNon` controller represents a variant of Newton's method. The test suite then runs the `pythNon` controller and directs its output to a *verification file*. The test suite compares this verification file containing the generated solution to another file containing the *reference solution*, that is, the desired solution provided by the user, if applicable. The test is successful if the differences of the solutions in both files are acceptable. Otherwise, the user should re-configure that `pythNon` controller and run the test again. Figure 3.9 shows the process of testing in `pythNon`.

To compare the solution accuracy obtained and CPU time required of a set of Newton variants on a given problem, the user first defines a set of configuration files or generates them through the GUI. Once the user provides the set of configuration files and the reference solution to the problem,

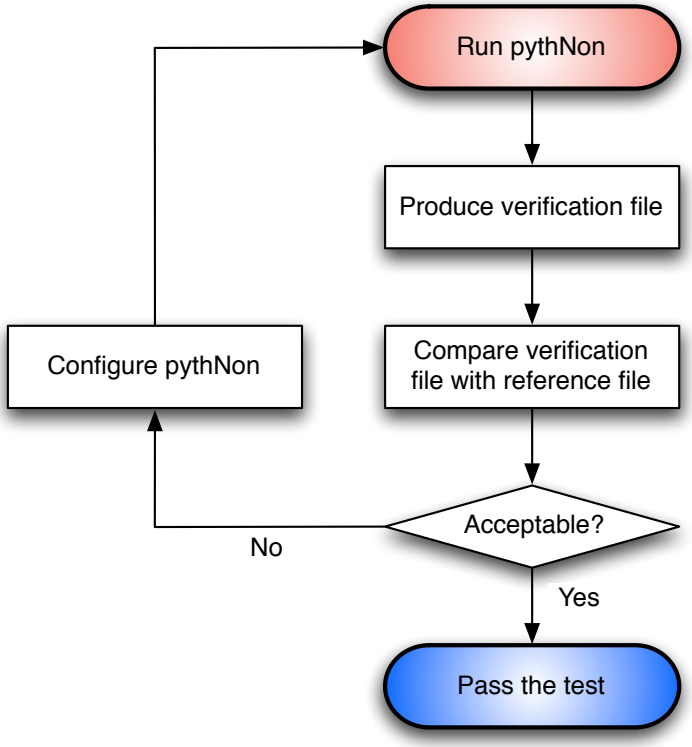

Figure 3.9: Process of testing in `pythNon`.

the test suite in `pythNon` automatically creates a set of `pythNon` controllers, runs them, records their CPU time, and compares the generated solutions with the reference solutions automatically, as shown in Algorithm 3. The output of this algorithm is a text file that allows the user to easily examine the results of the testing. The user may view the solution accuracy and the CPU time of each `pythNon` controller. The next section describes in detail a way to compute the solution accuracy with the reference solution.

Algorithm 3 Steps in the testing.

Input: A set of configuration files and a reference file

Output: A text file that reports the solution accuracy and CPU time for each `pythNon` controller

1: **for** each configuration file **do**
2: Create a `pythNon` controller with the configuration file
3: Run the `pythNon` controller and record its CPU time
4: Compute the solution accuracy of the new solution with respect to the reference solution
5: Output the solution accuracy and the CPU time to the text file
6: **end for**

3.4.2 Verifying the Solutions

The user generally does not expect or even desire the generated solution to be as accurate as the reference solution. To verify that a solution is the desired one, the user may assess the accuracy of the computed solution by comparing the number of matching digits between the generated solution and the reference solution within the test suite in `pythNon`. Algorithm 4 shows a way for computing the number of matching digits between two floating-point numbers. We note that for IEEE double-precision, the maximum number of matching decimal digits $d_{\max}$ between two floating-point numbers is about 16. In practice, due to roundoff errors, this number is less.

Algorithm 4 Comparing two floating-point numbers.

Input: Two floating-point numbers r_1 and r_2, and maximum matching decimal digits $d_{\max}$

Output: Number of matching digits d

$d \leftarrow 0$
while $(d < d_{\max})$ **do**
  $r_1' \leftarrow r_1$ rounded to d decimal digits
  $r_2' \leftarrow r_2$ rounded to d decimal digits
  if r_1' and r_2' are identical **then**
    $d \leftarrow d + 1$
  else
    return d
  end if
end while
return d

Chapter 4

Numerical Experiments

We show the power and flexibility of `pythNon` by means of the following numerical experiments. All the experiments reported in Sections 4.1–4.3 were performed using a 1.33GHz PowerPC G4 processor with 768MB of RAM. The experiments in Section 4.4 were performed using an IBM System x3550 with two 3.00GHz Dual-Core Intel Xeon processors and 2GB of RAM.

4.1 A Comparison of Dense and Banded Jacobians

To illustrate the advantages of storing and manipulating a banded Jacobian versus a dense Jacobian, we wish to compute a nontrivial solution $v(r)$ to the two-point boundary-value problem [6]:

$$v'' + \frac{4}{r}v' + (rv - 1)v = 0, \quad 0 < r < 20, \tag{4.1a}$$

$$v'(0) = 0, \quad v(20) = 0. \tag{4.1b}$$

To obtain a system of NAEs, we discretize (4.1) as follows. We first convert the second-order system (4.1a) to a first-order system by defining

$$\mathbf{y} = \begin{pmatrix} y_1 \\ y_2 \end{pmatrix} = \begin{pmatrix} v \\ v' \end{pmatrix}. \tag{4.2}$$

Then (4.1a) becomes

$$\mathbf{y}'(r) = \begin{pmatrix} y_2(r) \\ -\frac{4}{r}y_2(r) - (ry_1(r) - 1)y_1(r) \end{pmatrix} := \mathbf{g}(r, \mathbf{y}(r)). \tag{4.3}$$

We discretize (4.3) by the trapezoidal rule with an equally spaced mesh $\{r_i\}_{i=0}^{N+1}$; that is, $r_i = i\Delta r$ for $i = 0, 1, \ldots, N+1$, where $\Delta r = \frac{20}{N+1}$. This leads to

$$\frac{\mathbf{y}_{i+1} - \mathbf{y}_i}{\Delta r} = \frac{1}{2}(\mathbf{g}(r_{i+1}, \mathbf{y}_{i+1}) + \mathbf{g}(r_i, \mathbf{y}_i)), \tag{4.4}$$

where $\mathbf{y}_i \approx \mathbf{y}(r_i)$ for $i = 0, 1, \ldots, N$. This yields $2(N+1)$ equations for the $2(N+2)$ unknowns defined in (4.2). The remaining two equations are provided by the boundary conditions:

$$\mathbf{y}_{0,2} = 0 \text{ and } \mathbf{y}_{N+1,1} = 0.$$

The problem can be expressed as a system of NAEs $\mathbf{F}(\mathbf{Y}) = \mathbf{0}$, where $\mathbf{Y} = (\mathbf{y}_0^{\mathrm{T}}, \mathbf{y}_1^{\mathrm{T}}, \ldots, \mathbf{y}_{N+1}^{\mathrm{T}})^{\mathrm{T}}$. From (4.4), it is apparent that the upper and lower bandwidths of the Jacobian are 2.

We consider solving the above system with a Newton direct method. An NAE solver that treats all Jacobians as dense can only solve small systems in practice. As we increase the number of mesh points in the system, the system of NAEs becomes large, and it may quickly become infeasible to solve. In the `pythNon` PSE, the user may invoke a direct method that solves (1.4a) while exploiting the banded structure, so it can solve even very large systems efficiently.

Figure 4.1 shows the run-time statistics for the two-point boundary-value problem (4.1) treated with a dense Jacobian and with a banded Jacobian. Note that the x-axis in Figure 4.1 represents the size of the system, and the y-axis is log-scaled and represents the CPU time for solving the system. As we increase the number of mesh points, a direct method with dense **LU** decomposition takes longer to solve the problem than a direct method with banded **LU** decomposition; it is about 52 times slower with 1800 mesh points. Moreover, the direct method with dense **LU** decomposition fails to solve the system with 2000 mesh points in a reasonable amount of time.

This example shows that the teacher or student may conveniently investigate well-understood concepts in the `pythNon` PSE, such as the benefit of storing and manipulating a banded Jacobian over a dense Jacobian.

4.2 A Comparison of Newton Direct and Indirect Methods

For a given problem, it is sometimes not clear when it is more efficient to use a Newton direct method or a Newton indirect method. In general, the smaller the problem, the more advantageous a Newton direct method is to use. However, if the problem size grows, at some point it will be more advantageous to use a Newton indirect method. To show the break-even point between a

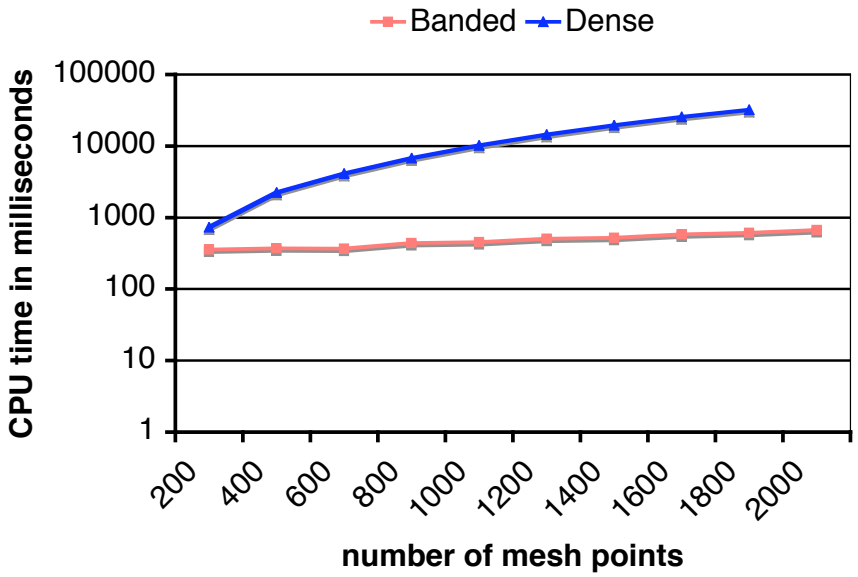

Figure 4.1: Run-time statistics for the two-point boundary-value problem with dense Jacobian and banded Jacobian.

Newton direct method and a Newton indirect method, we consider solving the Ornstein-Zernike (OZ) equations [52]. In their simplest isotropic form, the OZ equations consist of an integral equation coupled with an algebraic constraint for two unknown functions $h = h(r)$ and $c = c(r)$. The integral equation is

$$F(h,c)(r) = h(r) - c(r) - \rho(h * c)(r) = 0, \tag{4.5}$$

where $(h * c)(r) = \int_{\Re^3} c(\|\mathbf{r} - \mathbf{r}'\|)h(\|\mathbf{r}'\|)d\mathbf{r}'$, and ρ is a parameter. In practice, it is standard to truncate the computational domain to be a sphere of radius R. Following Kelley [38], we take $R = 9$. The algebraic constraint is

$$a(h,c)(r) = e^{-\beta u(r)+h(r)-c(r)} - h(r) - 1 = 0, \tag{4.6}$$

for all $0 \leq r \leq R$, where $u(r) = 4\epsilon\left[(\frac{\sigma}{r})^{12} - (\frac{\sigma}{r})^{6}\right]$, and β, ϵ, and σ are parameters. Following Kelley [38], we choose

$$\beta = 10, \ \rho = 0.2, \ \epsilon = 0.1, \ \text{and} \ \sigma = 2.0.$$

The convolution $(h*c)$ can be computed using only one-dimensional numerical quadratures through the spherical-Bessel transform. Assuming h decays sufficiently rapidly, we let

$$\hat{h}(k) = \mathcal{H}(h)(k) = 4\pi \int_0^\infty \frac{\sin(kr)}{kr} h(r) r^2 dr$$

and

$$h(r) = \mathcal{H}^{-1}(\hat{h})(k) = \frac{1}{2\pi^2} \int_0^\infty \frac{\sin(kr)}{kr} \hat{h}(k) k^2 dk.$$

The convolution $(h * c)$ is then approximated by discretizing the equation

$$h * c = \mathcal{H}^{-1}(\hat{h}\hat{c}) \tag{4.7}$$

on a uniform mesh with grid points $\mathbf{r} := \{r_i\}_{i=1}^N$; i.e., $r_i = i\Delta r$ for $i = 1, 2, \ldots, N$, where $\Delta r = \frac{R}{N-1}$, and $\hat{h}\hat{c}(r) := \hat{h}(r)\hat{c}(r)$ is the pointwise product of functions. Defining vectors $\mathbf{h}$ and $\mathbf{c}$ with components $h_i = h(r_i)$ and $c_i = c(r_i)$, respectively, (4.5) and (4.6) lead to

$$\mathbf{F}(\mathbf{x}) = \begin{pmatrix} \mathbf{h} - \mathbf{c} \\ e^{-\beta u(\mathbf{r}) + \mathbf{h} - \mathbf{c}} - \mathbf{h} - \mathbf{1} \end{pmatrix} + \begin{pmatrix} \mathbf{0} \\ \rho(\mathbf{h} * \mathbf{c}) \end{pmatrix} = \mathbf{0},$$

where $\mathbf{x} := (\mathbf{h}^T, \mathbf{c}^T)^T$, $\mathbf{h}, \mathbf{c} \in \Re^N$ are the unknowns, $\mathbf{h} * \mathbf{c}$ is the discretization of (4.7), $\mathbf{1} := (1, 1, \ldots, 1) \in \Re^N$, and the equations are to be interpreted componentwise. We refer to Kelley and Pettitt [39] for further implementational details.

Figure 4.2 shows that in terms of performance the break-even point between the Newton direct method with dense **LU** decomposition and the Newton-GMRES method is at 128 mesh points. That is, the Newton direct method with dense **LU** decomposition is faster than the Newton-GMRES method for solving systems that are small, i.e., with fewer than 128 mesh points. However, as we increase the number of mesh points, the Newton direct method takes much longer to compute, about 121 times longer than a Newton indirect method for a mesh of 512 points. When it reaches 1024 mesh points, the system becomes so large that it is infeasible to solve with a Newton direct method. On the other hand, the Newton indirect method continues to solve the system because it does not require storage of the Jacobian.

This example shows that the user can identify the break-even point between Newton variants and choose the appropriate Newton variant for solving the problem efficiently in the `pythNon` PSE.

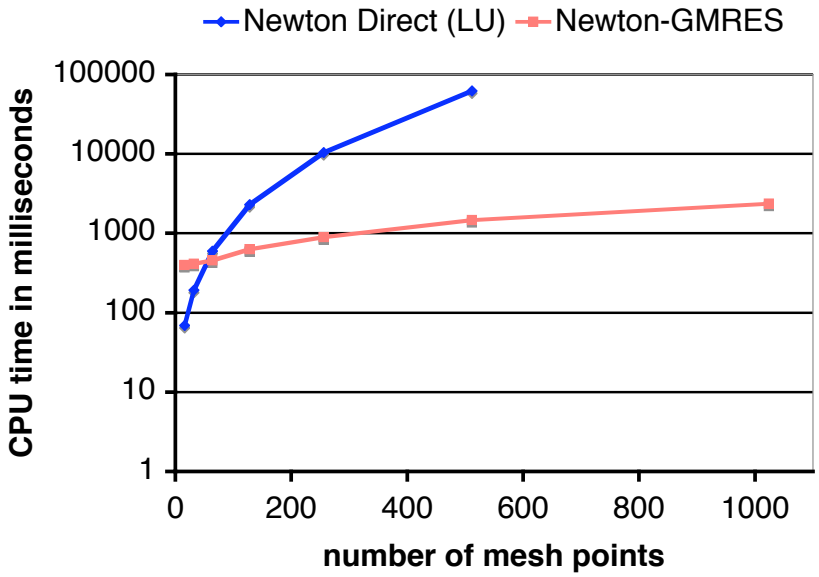

Figure 4.2: Run-time statistics for the Ornstein-Zernike equations with Newton Direct method and Newton-GMRES method.

4.3 A Comparison of Newton Indirect Methods

In order to provide a comparison of Newton indirect methods, we solve the Chandrasekhar H-equation [36], which is an integral equation that arises in radiative transfer theory:

$$H(\mu) - \left(1 - \frac{c}{2}\int_0^1 \frac{\mu H(\nu)}{\mu+\nu} d\nu\right)^{-1} = 0. \tag{4.8}$$

Equation (4.8) has exactly two solutions for $0 < c < 1$ [38]. We discretize this problem by approximating the integral by the composite midpoint rule on a uniform mesh with grid points $\{\mu_i\}_{i=0}^{m+1}$. Thus we take the unknowns to be

$$H_{i+\frac{1}{2}} \approx H(\mu_{i+\frac{1}{2}}),$$

where $\mu_{i+\frac{1}{2}} = \mu_i + \frac{h}{2}$ for $i = 0, 1, \ldots, m$, and $h = \frac{1}{m+1}$, this leads to

$$F_i(\mathbf{H}) = H_{i+\frac{1}{2}} - \left(1 - \frac{c}{2(m+1)} \sum_{j=0}^{m} \frac{\mu_{i+\frac{1}{2}} H_{j+\frac{1}{2}}}{\mu_{i+\frac{1}{2}} + \mu_{j+\frac{1}{2}}}\right)^{-1},$$

where $\mathbf{H} = \left(H_{\frac{1}{2}}, H_{\frac{3}{2}}, \ldots, H_{m+\frac{1}{2}}\right)^{\mathrm{T}}$. We start with an initial iterate $\mathbf{x}^{(0)} = \mathbf{1} := (1, 1, \ldots, 1)^T \in \Re^m$ that tends to converge to the solution of physical interest.

We consider solving the above system with the Newton-GMRES, Newton-BiCGSTAB, and Newton-TFQMR methods. Figure 4.3 shows that the Newton-TFQMR method is the most efficient method overall for solving the above system with different numbers of mesh points. However, all three methods appear to be competitive in terms of CPU time when solving the problem with a large number of mesh points, e.g., when solving (4.8) on a mesh of 1800 points.

This simple example demonstrates that the `pythNon` PSE can help the user to easily identify the most efficient Newton variant on a given problem. In the next section, we provide an in-depth study of how the `pythNon` PSE enables the user to study the effects of different forcing-term strategies in Newton's method.

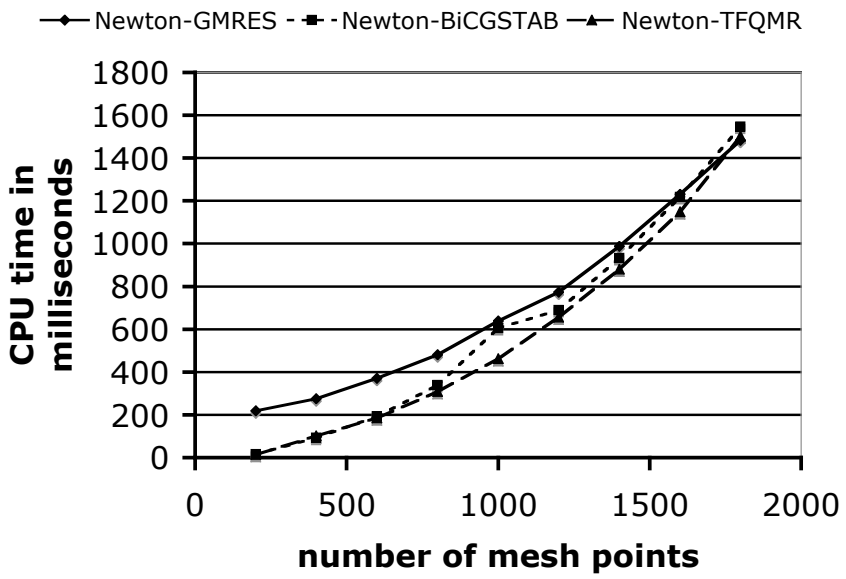

Figure 4.3: Run-time statistics for the Chandrasekhar H-equation with Newton-GMRES, Newton-BiCGSTAB, and Newton-TFQMR.

4.4 An Investigation of Forcing Terms

The inexact Newton condition (2.6) expresses a certain reduction in the local linear model $\mathbf{L_F}(\mathbf{x}^{(n)})$ of $\mathbf{F}(\mathbf{x}^{(n+1)})$ assuming that $\mathbf{L_F}(\mathbf{x}^{(n)})$ closely approximates $\mathbf{F}(\mathbf{x}^{(n+1)})$, and $\|\mathbf{L_F}(\mathbf{x}^{(n)})\|$ is smaller than $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ by a factor of $\eta^{(n)}$.

The forcing term $\eta^{(n)}$ controls the level of accuracy of solving the Newton direction $\mathbf{d}^{(n)}$ in (1.4a) [21]. That is, solving (1.4a) with a small forcing term $\eta^{(n)}$ results in an accurate approximation to the Newton direction $\mathbf{d}^{(n)}$. In the case of direct methods, the forcing term $\eta^{(n)} = 0$ for all Newton iterations because direct methods solve (1.4a) to within roundoff errors. In the case of indirect methods, the forcing term influences the number of linear iterations to be performed. If the forcing term $\eta^{(n)}$ is too small, indirect methods may need to perform a large number of linear iterations (or fail) to obtain a sufficiently accurate Newton direction $\mathbf{d}^{(n)}$ to satisfy the inexact Newton condition (2.6).

The forcing term also influences the rate of convergence and performance of Newton's method. Choosing $\eta^{(n)}$ too large may fail to produce a sufficiently accurate Newton direction $\mathbf{d}^{(n)}$ for the Newton iteration to converge and thus lead to *undersolving* (1.4a) [50]. If the approximate solution $\mathbf{x}^{(n)}$ is far from the solution $\mathbf{x}^*$, $\mathbf{L_F}(\mathbf{x}^{(n)})$ may approximate $\mathbf{F}(\mathbf{x}^{(n+1)})$ poorly [21]. Choosing $\eta^{(n)}$ too small imposes too much accuracy on the Newton direction $\mathbf{d}^{(n)}$, thus leads to *oversolving* (1.4a) [21]. In other words, (1.4a) is solved to far more precision than is really needed [38]. In particular, the additional accuracy desired in solving (1.4a) by an indirect method requires additional linear iterations and is therefore more costly. Moreover, the rate of convergence of Newton's method suffers because oversolving results in little or no reduction in the residual [21]. We note that the effects of undersolving and oversolving may occur when approximating the Newton direction $\mathbf{d}^{(n)}$ in (1.4a) using an indirect method. No precise quantitative definitions of these two phenomena exist.

In the following sections, we review several representative strategies for choosing forcing terms, investigate their effects, and compare their performance on several benchmark problems in the `pythNon` PSE. We also demonstrate the `pythNon` PSE as a flexible and easy-to-use environment in which to develop, implement, and evaluate different forcing-term strategies.

4.4.1 Different Strategies for Choosing a Forcing Term

The main purpose of choosing a good forcing term is to achieve fast convergence and ameliorate the effect of oversolving and undersolving [21]. However, no foolproof strategy is known for choosing good forcing terms throughout the Newton iterations [54]. Several strategies for choosing forcing terms have been suggested, e.g., Cai et al. [13], Dembo and Steihaug [15], Brown and Saad [12], Eisenstat and Walker [21], An et al. [3], and Gomes-Ruggiero et al. [28]. We now summarize these strategies.

Cai et al. [13] propose a small constant, $\eta^{(n)} \equiv 10^{-4}$ for each Newton iteration. It requires the approximation of the Newton direction to be uniformly accurate [21]. Dembo and Steihaug [15] propose the first *adaptive* forcing-term strategy $\eta^{(n)} = \min\{1/(n+2), \|\mathbf{F}(\mathbf{x}^{(n)})\|\}$. It allows the Newton direction $\mathbf{d}^{(n)}$ to be relatively inaccurate for small n, thus requiring fewer linear iterations to satisfy (2.6). The computation of the Newton directions is then less costly. It uses some information about $\mathbf{F}(\mathbf{x}^{(n)})$. However, this strategy depends on the scale of $\mathbf{F}(\mathbf{x}^{(n)})$, and it does not reflect the agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ [21]. Brown and Saad [12] propose the strategy $\eta^{(n)} = 1/2^{n+1}$. Again, this strategy allows the Newton direction $\mathbf{d}^{(n)}$ to be relatively inaccurate for small n. However, it does not use any information about $\mathbf{F}(\mathbf{x}^{(n)})$.

Eisenstat and Walker [21] propose two adaptive forcing-term strategies that are at present the most popular in practice:

1. Given $\eta^{(0)} \in [0, 1)$, choose

$$\eta^{(n)} = \frac{\|\mathbf{F}(\mathbf{x}^{(n)}) - \mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})\|}{\|\mathbf{F}(\mathbf{x}^{(n-1)})\|}, \tag{4.9}$$

   or

$$\eta^{(n)} = \frac{\big|\|\mathbf{F}(\mathbf{x}^{(n)})\| - \|\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})\|\big|}{\|\mathbf{F}(\mathbf{x}^{(n-1)})\|}. \tag{4.10}$$

2. Given $\gamma \in [0, 1]$, $\alpha \in (1, 2]$, and $\eta^{(0)} \in [0, 1)$, choose

$$\eta^{(n)} = \gamma \left(\frac{\|\mathbf{F}(\mathbf{x}^{(n)})\|}{\|\mathbf{F}(\mathbf{x}^{(n-1)})\|} \right)^{\alpha}. \tag{4.11}$$

The first strategy reflects the agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$. That is, this choice allows the Newton direction $\mathbf{d}^{(n)}$ to be relatively inaccurate when both $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree considerably. Equation (4.10) is used more often than (4.9) because indirect methods may keep $\|\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})\|$ when evaluating (2.6); thus (4.10) is more convenient and less costly to evaluate [21]. The second strategy, on the other hand, does not reflect the agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ but depends on the reduction rate of $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ relative to $\|\mathbf{F}(\mathbf{x}^{(n-1)})\|$. The rationale is that if $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ is relatively close to $\|\mathbf{F}(\mathbf{x}^{(n-1)})\|$, the approximate solution $\mathbf{x}^{(n)}$ may still be far from the solution $\mathbf{x}^*$, thus the resulting $\eta^{(n)}$ allows the Newton direction $\mathbf{d}^{(n)}$ to be less accurate. Both γ and α are user-defined parameters that affect the rate of convergence of Newton's method. Eisenstat and Walker show that $\gamma \geq 0.9$ and $\alpha \geq (1+\sqrt{5})/2$ offer an attractive rate of convergence of the solution and have the best performances in practice [21]. However, our experiments in Section 4.4.6 show that smaller values of γ and α are in fact necessary to successfully solve the two-dimensional steady-state convection-diffusion equation.

In practice, these two strategies are combined with safeguards to prevent the forcing terms from becoming too small too quickly [21]. The first strategy uses the safeguard

$$\eta^{(n)} = \max\left\{\eta^{(n)}, \left(\eta^{(n-1)}\right)^{\alpha}\right\}, \tag{4.12}$$

whenever $\left(\eta^{(n-1)}\right)^{\alpha} > 0.1$. The second strategy uses the safeguard

$$\eta^{(n)} = \max\left\{\eta^{(n)}, \gamma\left(\eta^{(n-1)}\right)^{\alpha}\right\},$$

whenever $\gamma\left(\eta^{(n-1)}\right)^{\alpha} > 0.1$. Following An et al. [3], we use $\alpha = (1+\sqrt{5})/2$ for the first strategy and $\alpha = 2$ and $\gamma = 0.9$ for the second strategy.

An et al. [3] propose the strategy

$$\eta^{(n)} = \begin{cases} 1 - 2p_1, & r^{(n-1)} < p_1, \\ s_1\eta^{(n-1)}, & p_1 \leq r^{(n-1)} < p_2, \\ s_2\eta^{(n-1)}, & p_2 \leq r^{(n-1)} < p_3, \\ s_3\eta^{(n-1)}, & r^{(n-1)} \geq p_3, \end{cases} \tag{4.13}$$

where $0 < s_3 < s_2 < s_1 \leq 1$ are shrinking factors, $0 < p_1 < p_2 < p_3 < 1$ are user-defined

parameters, and $p_1 \in (0, 0.5)$. $r^{(n-1)}$ is the ratio of the *actual reduction*

$$\|\mathbf{F}(\mathbf{x}^{(n-1)})\| - \|\mathbf{F}(\mathbf{x}^{(n)})\|$$

to the *predicted reduction*

$$\|\mathbf{F}(\mathbf{x}^{(n-1)})\| - \|\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})\|$$

of $\mathbf{F}(\mathbf{x}^{(n-1)})$ that often appears in trust region methods [3]. That is, if $r^{(n-1)}$ is close to 1, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ agree well. This may indicate some reduction in the residual $\|\mathbf{F}(\mathbf{x}^{(n-1)})\|$, thus the forcing term $\eta^{(n)}$ should be relatively small. If $r^{(n-1)}$ is relatively small, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree. This may indicate that the approximate solution is still far from the solution; thus the forcing term should be relatively large, e.g., $1 - 2p_1$, to allow the approximation of the Newton direction to be less accurate to ameliorate the effect of oversolving. If $r^{(n-1)}$ is negative, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree. This only happens when the globalization strategy allows an increase in the residual. If $r^{(n-1)}$ is relatively large, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ also disagree, but this also indicates some reduction in the residual $\mathbf{F}(\mathbf{x}^{(n)})$, thus a smaller forcing term imposes an accurate approximation of the Newton direction for the Newton iteration to converge. From (4.13), $r^{(n-1)}$ determines the forcing term $\eta^{(n)}$. In some cases, $\eta^{(n)}$ may not change throughout the iteration because $r^{(n-1)} < p_1$ for some sequence of iterates. For example, if $\eta^{(n)}$ remains large for several iterations, i.e., $1-2p_1$, the approximate Newton direction may not be accurate enough for Newton's method to converge to a solution. Thus An et al. introduce the following safeguard:

$$\eta^{(n)} = s_3\eta^{(n-1)} \text{ whenever } \eta^{(n-2)}, \eta^{(n-1)} > t \text{ and } r^{(n-2)}, r^{(n-1)} < p_1. \tag{4.14}$$

An et al. report that $s_1 = 1$, $s_2 = 0.8$, $s_3 = 0.5$, $p_1 = 0.1$, $p_2 = 0.4$, $p_3 = 0.7$, and $t = 0.1$ in (4.13) and (4.14) are the most effective in their experiments [3].

Both the first strategy of Eisenstat and Walker and the strategy of An et al. determine the forcing term based on $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$. If $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree, both strategies choose a relatively large forcing term to ameliorate the effect of oversolving [21]. On the other hand, if $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ agree well, both strategies choose a relatively small forcing term. However, our experiments in Section 4.4.6 show that a good agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$

does not imply that it is always advantageous to choose a smaller forcing term for the next Newton iteration. This result runs counter to the intuition of many conventional forcing-term strategies. In other words, some problems do not require high accuracy on the Newton direction for most of the Newton iterations; thus the forcing term should be relatively large even though $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ agree well; see Section 4.4.7 for further discussion..

Gomes-Ruggiero et al. [28] propose a forcing-term strategy that based on the reduction rate of the residual and the net computational cost of the number of linear iterations and the number of function evaluations:

$$\eta^{(n)} = [1/(n+1)]^{\rho} \cos^2(\theta^{(n)}) \frac{\|\mathbf{F}(\mathbf{x}^{(n)})\|}{\|\mathbf{F}(\mathbf{x}^{(n-1)})\|}, \tag{4.15}$$

where $\rho \in (1, 2]$ is a user-defined constant. The term $\cos(\theta^{(n)})$ is a measure of the trade off between the rate of convergence and computational costs [28]:

$$\begin{aligned} \cos(\theta^{(n)}) &= \frac{b^{(n)}}{\sqrt{(a^{(n)})^2 + (b^{(n)})^2}}, \\ a^{(n)} &:= \log_{10} \|\mathbf{F}(\mathbf{x}^{(n)})\| - \log_{10} \|\mathbf{F}(\mathbf{x}^{(n-1)})\|, \\ b^{(n)} &:= \log_{10}(c^{(n)} - c^{(n-1)}), \end{aligned}$$

where $\theta^{(n)} \in (-\pi/2, \pi/2)$, and $c^{(n)}$ is the sum of the total number of linear iterations and the total number of function evaluations at the nth Newton iteration. That is, if $\cos(\theta^{(n)})$ is negative or close to -1, the residual decreases; thus choosing a smaller forcing term allows the approximation of the Newton direction $\mathbf{d}^{(n)}$ to be more accurate. If $\cos(\theta^{(n)})$ is close to zero, the net computational cost may be high or there is a risk of oversolving; thus choosing a larger forcing term allows the approximation of the Newton direction to be less accurate. If $\cos(\theta^{(n)})$ is positive, the residual increases. This should not occur unless the globalization strategy allows an increase in the residual. We note that (4.15) is similar to (4.10) except that γ in (4.10) is constant. Following Gomes-Ruggiero et al. [28], we use $\rho = 1.1$.

4.4.2 Modification to the strategy of An et al.

The strategy of An et al. (4.13) determines the forcing term by comparing the ratio of the actual reduction to the predicted reduction of the residual, i.e., $r^{(n)}$ to the values of p_1, p_2, and p_3. In

particular, if $r^{(n-1)} \gg 1$, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree greatly. This generally happens when the approximate solution is far from the solution. According to the strategy of An et al. (4.13), when $r^{(n-1)} > 1$, $\eta^{(n)} = s_3\eta^{(n-1)}$, where $s_3 = 0.5$. However, it is possible that this forcing term is too small, thus leading to oversolving and very little or no reduction in the residual. Section 4.4.7 shows an example where the Newton variant with the strategy of An et al. (4.13) fails when $r^{(n-1)} \gg 1$.

To ameliorate the effect of oversolving when $r^{(n-1)} \gg 1$, we augment the strategy of An et al. (4.13) with the following safeguard:

$$\eta^{(n)} = \eta^{(n-1)} \text{ whenever } r^{(n-1)} > 1. \tag{4.16}$$

Section 4.4.7 shows an example where this safeguard is generally more effective.

4.4.3 A New Forcing-Term Strategy

Forcing-term strategies that do not consider the agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$, such as those suggested by Cai et al. [13], Dembo and Steihaug [15], Brown and Saad [12], Eisenstat and Walker (second strategy (4.11)) [21], and Gomes-Ruggiero et al. [28], may suffer from oversolving because these strategies may impose an accuracy on the approximate Newton direction that results in a disagreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ [21].

Both the first strategy of Eisenstat and Walker (4.10) and the strategy of An et al. (4.13), however, consider the agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ and can thus ameliorate the effect of oversolving. To use the strategy of An et al. (4.13) to approximate the Newton direction efficiently, one has to choose a good set of values for s_1, s_2, s_3, p_1, p_2, p_3, and t in (4.13) based on the nature of the problem. Although An et al. provide a set of values that is effective in their experiments, it is generally impossible for the user to know *a priori* what values will be effective for a given problem.

The first strategy of Eisenstat and Walker (4.10) requires only a user-defined parameter for safeguarding such as the value of α in (4.12), but it can suffer from undersolving [50]. That is, if the approximate solution is far from the solution and $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree consid-

erably, the strategy may determine a forcing term that is too large to approximate a sufficiently accurate Newton direction. This results in very little or no reduction in the residual, thus leading to stagnation in the Newton iterations; see Section 4.4.6 for an example.

To ameliorate the effects of oversolving and undersolving, we propose a new strategy that not only considers the agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ but also the forcing term in the previous Newton iteration. In other words, this strategy attempts to prevent the forcing term from becoming too large or too small too quickly by determining a fraction of the forcing term in the previous Newton iteration. This new strategy can be viewed as a modification to the first strategy of Eisenstat and Walker (4.10). Given $\eta^{(0)} \in [0, 1)$, choose

$$\eta^{(n)} = \frac{\left| \|\mathbf{F}(\mathbf{x}^{(n)})\| - \|\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})\| \right|}{\|\mathbf{F}(\mathbf{x}^{(n-1)})\|} \eta^{(n-1)}. \tag{4.17}$$

We note that we have found that it is crucial to select a suitable initial forcing term $\eta^{(0)}$ for all adaptive forcing-term strategies that require one so that oversolving does not occur in the early Newton iterations. Section 4.4.6 shows an example where a poor choice of $\eta^{(0)}$ results in stagnation of the Newton iterations. Unfortunately, no known strategies exist in general for choosing a good initial forcing term. However, suppose that the new strategy (4.17) begins with a good initial forcing term $\eta^{(0)}$. If the approximate solution is far from the solution and $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree considerably, the fraction in (4.17) can be relatively large. To prevent the forcing term from becoming too large too quickly, the resulting forcing term is bounded by the forcing term in the previous Newton iteration, thus ameliorating the effect of undersolving.

This new forcing-term strategy is robust. Unlike the strategy of An et al. (4.13), it does not require the user to define and change parameters in the forcing-term strategy. It also does not require safeguarding because the forcing term is bounded by the forcing term in the previous Newton iteration, and the forcing term is less than $\eta^{(0)}$ throughout the Newton iterations. We leave the proof of its rate of convergence to future work.

In the following sections, we compare and evaluate the effects of this new strategy with other forcing-term strategies on a suite of benchmark problems in the `pythNon` PSE.

4.4.4 The Test Problems

We use 5 benchmark problems to evaluate the effects of different forcing-term strategies. Following An et al. [3], we use the first three problems, in particular the "Tridiagonal" system, to show that the first strategy of Eisenstat and Walker (4.10) can suffer from undersolving. The fourth problem shows that the strategy of An et al. (4.13) can suffer from oversolving. We note that the first four problems come with standard initial guesses $\mathbf{x}_s$ and have the exact solution $\mathbf{x}^* = \mathbf{1}$.

Generalized function of Rosenbrock [3]

$$
\begin{aligned}
F_1(x) &= -4c(x_2 - x_1^2)x_1 - 2(1 - x_1), && \text{(4.18a)}\\
F_i(x) &= 2c(x_i - x_{i-1}^2) - 4c(x_{i+1} - x_i^2)x_i - 2(1 - x_i),\ i = 2, 3, \ldots, m-1, && \text{(4.18b)}\\
F_m(x) &= 2c(x_m - x_{m-1}^2), && \text{(4.18c)}
\end{aligned}
$$

with $c = 2$, $m = 5000$, and $\mathbf{x}_s = (1.2, 1.2, \ldots, 1.2)^T \in \Re^m$.

"Tridiagonal" system [44]

$$
\begin{aligned}
F_1(x) &= 4(x_1 - x_2^2), && \text{(4.19a)}\\
F_i(x) &= 8x_i(x_i^2 - x_{i-1}) - 2(1 - x_i) + 4(x_i - x_{i+1}^2),\ i = 2, 3, \ldots, m-1, && \text{(4.19b)}\\
F_m(x) &= 8x_m(x_m^2 - x_{m-1}) - 2(1 - x_m), && \text{(4.19c)}
\end{aligned}
$$

with $m = 6000$ and $\mathbf{x}_s = (12, 12, \ldots, 12)^T \in \Re^m$.

"Pentadiagonal" system [44]

$$
\begin{aligned}
F_1(x) &= 4(x_1 - x_2^2) + x_2 - x_3^2, && \text{(4.20a)}\\
F_2(x) &= 8x_2(x_2^2 - x_1) - 2(1 - x_2) + 4(x_2 - x_3^2) + x_3 - x_4^2, && \text{(4.20b)}\\
F_i(x) &= 8x_i(x_i^2 - x_{i-1}) - 2(1 - x_i) + 4(x_i - x_{i+1}^2) + x_{i-1}^2 - x_{i-2} && \text{(4.20c)}\\
&\quad + x_{i+1} - x_{i+2}^2,\ i = 3, 4, \ldots, m-2,\\
F_{m-1}(x) &= 8x_{m-1}(x_{m-1}^2 - x_{m-2}) - 2(1 - x_{m-1}) + 4(x_{m-1} - x_m^2) && \text{(4.20d)}\\
&= + x_{m-2}^2 - x_{m-3},\\
F_m(x) &= 8x_m(x_m^2 - x_{m-1}) - 2(1 - x_m) + x_{m-1}^2 - x_{m-2}, && \text{(4.20e)}
\end{aligned}
$$

with $m = 5000$ and $\mathbf{x}_s = (2, 2, \ldots, 2)^T \in \Re^m$.

Extended Rosenbrock Function [45]

$$
\begin{aligned}
F_{2i-1} &= 10(x_{2i} - x_{2i-1}^2), && \text{(4.21a)}\\
F_{2i} &= 1 - x_{2i-1}, && \text{(4.21b)}
\end{aligned}
$$

with $i = 1, 2, \ldots, m/2$, $m = 32768$ (m is a multiple of 2), and $\mathbf{x}_s = (-1.2, 1, \ldots, -1.2, 1)^T \in \Re^m$.

Following An et al. [3], we solve the first three problems with 10 different initial guesses to illustrate the effects of different strategies, i.e., $\mathbf{x}^{(0)} = j\mathbf{x}_s$ for $j = 1, 2, \ldots, 5$, $\mathbf{x}^{(0)} = j\mathbf{1}$ with $\mathbf{1} := (1, 1, \ldots, 1)^T \in \Re^m$ for $j = 2, 3, \ldots, 5$, and $\mathbf{x}^{(0)} = \mathbf{0}$. We note that we only solve the "pentadiagonal" system with 9 different initial guesses because $\mathbf{x}_s$ is equivalent to $\mathbf{2}$, where $\mathbf{2} := (2, 2, \ldots, 2)^T \in \Re^m$. We solve the extended Rosenbrock problem with 5 different initial guesses, i.e., $\mathbf{x}^{(0)} = j\mathbf{x}_s$ for $j = 1, 2, \ldots, 5$.

To show that an ideal forcing-term strategy should not reduce the forcing term simply based on the agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ and that constant forcing-term strategies may sometimes outperform adaptive forcing-term strategies, we consider solving the two-dimensional steady-state *convection-diffusion* equation [38]

$$
-\nabla^2 u + \kappa u(u_x + u_y) = f(x, y) \qquad \text{(4.22)}
$$

with homogeneous Dirichlet boundary conditions on the unit square $(0,1)\times(0,1)$, where ∇^2 is the Laplacian operator

$$\nabla^2 = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2},$$

and $f(x,y)$ has been constructed so that the exact solution u^* is

$$u^* = 10xy(1-x)(1-y)e^{x^{4.5}}.$$

We discretize (4.22) on a 100×100 uniform grid using centered difference to obtain a system of m equations for m unknowns, where $m = 10^4$. The m unknowns of this system are the values of u at these grid points. Each column of the unknowns u is stored in the one-dimensional vector $\mathbf{x}$ of size m. That is, $u(ih, jh) = x_{i+(j-1)\sqrt{m}}$ for $i, j = 1, \ldots, \sqrt{m}$, where $h = 1/(\sqrt{m}+1)$. We note that (4.22) is a *quasi-linear* PDE. That is, it is (generally) a nonlinear PDE; however, its highest-order derivative terms appear linearly. Because of this, it is possible to use a *preconditioner* $M = -\nabla^{-2}$ to aid in obtaining a numerical solution. Such preconditioning is a necessity in order to solve the system of NAEs arising from the discretization of (4.22) in practice. Operating with M on the left and noting $-M\nabla^2 u = u$, we obtain the preconditioned equation

$$u + \kappa M(u(u_x + u_y)) - Mf = 0. \tag{4.23}$$

The discrete version of (4.23) defines the residual for this problem.

In practice, approximation of the effect of M on the discrete version of (4.23) is realized by means of a *fast Poisson solver*, i.e., a numerical routine that uses the fast Fourier transform to solve the Poisson equation $\nabla^2 w(x,y) = g(x,y)$ for $w(x,y)$ with a uniform mesh on the unit square and subject to homogeneous Dirichlet conditions; see, e.g., [38] for more details.

Following Kelley [38], we set both the absolute and relative tolerances to $h^2/10$.

The initial guess for this problem is $\mathbf{x}^{(0)} = \mathbf{0}$. We consider 7 test cases: $\kappa = 100$, 300, 500, 700, 1000, 2000, and 7000. We note that the initial guess is farther from the solution for the larger values of κ [29, 38]. It also becomes more *convection dominated*, making the centered discretization less stable and generally more difficult to solve [54]; see Section 4.4.7 for further discussion.

4.4.5 The Experiments

We use `pythNon` to solve all the problems with the variant of Newton's method shown in Algorithm 5. Algorithm 5 uses the *backtracking* globalization strategy to ensure that the approximate solutions converge to a solution for any given initial guess [21]. Backtracking is a line search method that searches for a reduction in the residual by reducing the step length at each line search iteration. That is, backtracking first determines if a full Newton step satisfies (4.24) below. If the full Newton step is rejected, it determines a step length $\lambda^{(l)} \in [\theta_{\min}\lambda^{(l-1)}, \theta_{\max}\lambda^{(l-1)}]$, where $l \geq 1$ and $\lambda^{(0)} = 1$ by minimizing the merit function

$$\phi(\lambda) = \frac{1}{2}\|\mathbf{F}(\mathbf{x} + \lambda\mathbf{d})\|^2$$

using a quadratic polynomial that interpolates values of $\phi(\lambda)$. The values $\theta_{\min}$ and $\theta_{\max}$ are the minimum and maximum *step-length reduction factors* that prevent backtracking from reducing the step length by too much or too little [38]. The backtracking in our experiments minimizes a three-point quadratic polynomial $p(\lambda)$ that satisfies the following conditions [54]:

$$\begin{aligned} p(0) = \phi(0) &= \frac{1}{2}\|\mathbf{F}(\mathbf{x}^{(n)})\|^2, \\ p(\lambda^{(l-1)}) = \phi(\lambda^{(l-1)}) &= \frac{1}{2}\|\mathbf{F}(\mathbf{x}^{(n)} + \lambda^{(l-1)}\mathbf{d}^{(n)})\|^2, \\ p(\lambda^{(l-2)}) = \phi(\lambda^{(l-2)}) &= \frac{1}{2}\|\mathbf{F}(\mathbf{x}^{(n)} + \lambda^{(l-2)}\mathbf{d}^{(n)})\|^2, \end{aligned}$$

where $l \geq 2$ and $\lambda^{(1)} = \theta_{\max}$. We use the standard choices $\theta_{\min} = 0.1$ and $\theta_{\max} = 0.5$ [16, 21].

Backtracking terminates when the "*sufficient-decrease*" condition

$$\|\mathbf{F}(\mathbf{x}^{(n)} + \lambda^{(l)}\mathbf{d}^{(n)})\| \leq (1 - \nu\lambda^{(l)}(1 - \eta^{(n)}))\|\mathbf{F}(\mathbf{x}^{(n)})\| \tag{4.24}$$

holds. Comparing to (2.6), (4.24) is equivalent to

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| - \|\mathbf{F}(\mathbf{x}^{(n)} + \lambda^{(l)}\mathbf{d}^{(n)})\| \geq \nu\lambda^{(l)}(1 - \eta^{(n)})\|\mathbf{F}(\mathbf{x}^{(n)})\|,$$

whereas (2.6) is equivalent to

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| - \|\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n)})\| \geq (1 - \eta^{(n)})\|\mathbf{F}(\mathbf{x}^{(n)})\|.$$

Algorithm 5 Inexact Newton Backtracking Method.

Input: initial iterate $\mathbf{x}^{(0)}$, residual function $\mathbf{F}$, absolute tolerance τ_a, and relative tolerance τ_r.

Output: the approximate solution $\mathbf{x}$

$\mathbf{x} \leftarrow \mathbf{x}^{(0)}$

while $\|\mathbf{F}(\mathbf{x})\| > \min\{\tau_r\|\mathbf{F}(\mathbf{x}^{(0)})\| + \tau_a,\ \tau_r\sqrt{m} + \tau_a\}$ **do**

  Choose a forcing term η; see Section 4.4.1

  Find $\mathbf{d}$ such that $\|\mathbf{F}(\mathbf{x}) + \mathbf{J}_{\mathbf{F}}(\mathbf{x})\mathbf{d}\| \leq \eta||\mathbf{F}(\mathbf{x})||$

  If $\mathbf{d}$ cannot be found, terminate with failure

  $\lambda \leftarrow 1$

  while $\|\mathbf{F}(\mathbf{x} + \lambda\mathbf{d})\| > (1 - \nu\lambda(1 - \eta))\|\mathbf{F}(\mathbf{x})\|$ **do**

    Determine $\lambda \in [\theta_{\min}\lambda, \theta_{\max}\lambda]$ that approximately minimizes $\frac{1}{2}\|\mathbf{F}(\mathbf{x} + \lambda\mathbf{d})\|^2$

  end while

  $\mathbf{x} \leftarrow \mathbf{x} + \lambda\mathbf{d}$

end while

return $\mathbf{x}$

In other words, an indirect method such as GMRES terminates with an approximate Newton direction when the predicted reduction in $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ is at least $(1-\eta^{(n)})\|\mathbf{F}(\mathbf{x}^{(n)})\|$, and a globalization strategy such as backtracking terminates with a step length when the actual reduction in $\|\mathbf{F}(\mathbf{x}^{(n)})\|$ is at least $\nu\lambda^{(l)}(1-\eta^{(n)})\|\mathbf{F}(\mathbf{x}^{(n)})\|$ [54]. Following An et al. [3], we use $\nu = 0.5$.

Algorithm 5 terminates with an approximate solution to the problem when

$$\|\mathbf{F}(\mathbf{x}^{(n)})\| \leq \min\{\tau_r\|\mathbf{F}(\mathbf{x}^{(0)})\| + \tau_a,\, \tau_r\sqrt{m} + \tau_a\} \tag{4.25}$$

holds. Following An et al. [3], we use $\tau_r = \tau_a = 10^{-6}$ for all problems except $\tau_r = \tau_a = h^2/10$ for the convection-diffusion problem; see Section 4.4.4. Algorithm 5 is deemed to fail if one of the following situations occurs:

- The number of Newton iterations reaches 300 without satisfying (4.25).
- The number of line search iterations reaches 20 without satisfying (4.24).
- The Newton iterations "stagnate", or the convergence of the solution is too slow [22], i.e.,

$$\left|\|\mathbf{F}(\mathbf{x}^{(n-1)})\| - \|\mathbf{F}(\mathbf{x}^{(n)})\|\right| \leq \tau_r\|\mathbf{F}(\mathbf{x}^{(n)})\|.$$

Following Pernice and Walker [55], we impose a final safeguard for all adaptive forcing-term strategies to ameliorate the effect of oversolving when the approximate solution is close to the solution. That is, if $\eta^{(n)} \leq 2\epsilon\|\mathbf{F}(\mathbf{x}^{(n)})\|$, then $\eta^{(n)} = 0.8\epsilon/\|\mathbf{F}(\mathbf{x}^{(n)}\|$, where

$$\epsilon = \min\{\tau_r\|\mathbf{F}(\mathbf{x}^{(0)})\| + \tau_a,\, \tau_r\sqrt{m} + \tau_a\}.$$

4.4.6 Results

We implement 8 variants of Newton's method based on Algorithm 5 with the following forcing-term strategies in the `pythNon` PSE:

- the strategy of Cai et al. (CGKT) [13],
- the strategy of Dembo and Steihaug (DS) [15],
- the strategy of Brown and Saad (BS) [12],

- the first strategy of Eisenstat and Walker (EW1) [21],
- the second strategy of Eisenstat and Walker (EW2) [21],
- the strategy of An et al. (AML) [3],
- the modified strategy of An et al. (mAML) in Section 4.4.2,
- the strategy of Gomes-Ruggiero et al. (GLT) [28], and
- the new strategy (New) in Section 4.4.3.

To provide an initial forcing term $\eta^{(0)}$ for some forcing-term strategies such as EW1, EW2, AML, mAML, GLT, and New, we use $\eta^{(0)} = 0.5$ for the first three benchmark problems, $\eta^{(0)} = 0.9$ for the extended Rosenbrock problem, and $\eta^{(0)} = 0.95$ for the convection-diffusion problem. We also define the following notations for reporting the statistics of each Newton variant from the `pythNon` PSE:

- `NI`: total number of Newton iterations,
- `LI`: total number of linear iterations,
- `FE`: total number of function evaluations,
- `CPU`: CPU time spent in seconds, and
- `*`: a failure in Algorithm 5; see Section 4.4.5.

We note that the CPU time of each Newton variant reported in the experiments is the minimum of the CPU times for solving a given problem with the same Newton variant 3 times. Following An et al. [3], we compute the average of `NI`, `LI`, `FE`, and `CPU` for each forcing-term strategy with different initial guesses or problem parameters on a given problem in order to characterize its performance.

Table 4.1 shows the statistics of each Newton variant with different forcing-term strategies that solves the generalized function of Rosenbrock with 10 different initial guesses. We observe that all Newton variants solve the problem with all initial guesses successfully. In particular, the Newton variants AML and mAML have the least average CPU time spent, i.e., 5.1 seconds. The Newton variants AML and mAML are comparable to the Newton variant BS and less expensive on average

than the Newton variant New by just over 21%. The Newton variants CGKT and EW2 are also competitive for this problem. This table shows that Newton variants with adaptive forcing-term strategies such as DS and EW1 can be more expensive on average than Newton variants with constant forcing-term strategies such as CGKT by as much as 98%.

Table 4.2 shows the statistics of each Newton variant with different forcing-term strategies that solves the "tridiagonal" system with 10 different initial guesses. This table shows that the Newton variants DS, EW1, EW2, and GLT fail to solve the system with initial guesses $\mathbf{x}^{(0)} = 3\mathbf{x}_s$ and $\mathbf{x}^{(0)} = 5\mathbf{x}_s$, and the Newton variants EW1 and EW2 also fail to solve the system with $\mathbf{x}^{(0)} = 4\mathbf{x}_s$. These Newton variants either perform too many Newton iterations or converge to a solution too slowly; see Section 4.4.5. We note that the average values for `NI`, `LI`, `FE`, and `CPU` in this table only include the successful cases. This table shows that the Newton variant New has the least average CPU time, i.e., 9.1 seconds. The Newton variant BS is the only variant that is comparable to the Newton variant New on this problem for all initial guesses.

Table 4.3 shows the statistics of each Newton variant with different forcing-term strategies that solves the "pentadiagonal" system with 9 different initial guesses. We note that $\mathbf{x}_s$ is equivalent to $\mathbf{2}$ in this problem; thus we exclude the results for $\mathbf{x}^{(0)} = \mathbf{2}$ in the table. This table shows that all Newton variants solve the problem successfully for all initial guesses used. In particular, the Newton variants AML and mAML have the least average CPU time, i.e., 7.7 seconds. These Newton variants are on average less expensive than the Newton variant New by just over 22%. This table shows that the Newton variants BS and GLT are also competitive, whereas the Newton variants CGKT, DS, EW1, and EW2 perform poorly.

Table 4.4 shows the statistics of each Newton variant with different forcing-term strategies that solves the extended Rosenbrock function with 5 different initial guesses. This table shows that the Newton variants BS and AML fail to solve the system with initial guesses $\mathbf{x}^{(0)} = 3\mathbf{x}_s$ because they converge to a solution too slowly; see Section 4.4.5. Comparing to the Newton variant AML, the Newton variant mAML solves the problem with all initial guesses successfully. This table shows that the Newton variant with the constant forcing-term strategy (CGKT) has the least average

Table 4.1: Results for generalized function of Rosenbrock.

Choice	$\{\mathbf{x}^{(0)}\} =$	$\mathbf{x}_s$	$2\mathbf{x}_s$	$3\mathbf{x}_s$	$4\mathbf{x}_s$	$5\mathbf{x}_s$	**2**	**3**	**4**	**5**	**0**	Average
CGKT	NI	4	7	8	9	10	6	8	8	9	9	7
	LI	46	83	78	95	98	69	97	81	95	132	87
	FE	51	91	87	105	109	76	106	90	105	151	97
	CPU	3.6	6.4	6.1	7.3	7.7	5.3	7.4	6.3	7.3	10.7	6.8
DS	NI	8	10	15	28	51	11	17	33	24	9	20
	LI	37	50	77	195	478	54	106	245	155	41	143
	FE	47	64	102	276	683	69	143	355	214	55	200
	CPU	3.3	4.4	7.1	19.2	47.4	4.8	9.9	24.6	14.8	3.8	13.9
BS	NI	6	10	9	12	13	8	10	12	12	8	10
	LI	38	64	51	74	87	47	54	82	82	45	62
	FE	46	76	62	89	103	57	67	97	98	58	75
	CPU	3.2	5.3	4.3	6.2	7.2	4.0	4.7	6.8	6.8	4.1	5.3
EW1	NI	7	10	33	29	21	13	19	34	25	10	20
	LI	31	49	279	226	118	76	141	283	173	47	142
	FE	39	63	393	308	156	101	187	399	232	59	193
	CPU	2.7	4.4	27.4	21.4	10.8	7.0	13.0	27.7	16.1	4.1	13.5
EW2	NI	5	9	22	15	32	9	13	23	15	10	15
	LI	34	49	79	59	128	44	54	94	60	46	64
	FE	40	59	117	79	202	55	72	133	79	60	89
	CPU	2.8	4.1	8.1	5.5	13.9	3.8	5.0	9.2	5.5	4.2	6.2
AML	NI	6	11	10	12	13	8	10	12	12	9	10
	LI	34	67	52	79	75	46	50	76	71	43	59
	FE	41	84	64	96	91	55	63	92	86	57	72
	CPU	2.9	5.8	4.5	6.9	6.5	3.8	4.4	6.4	6.0	4.0	5.1
mAML	NI	6	11	10	12	13	8	10	12	12	9	10
	LI	34	67	52	79	73	46	50	76	71	42	59
	FE	41	84	64	96	89	55	63	92	86	56	72
	CPU	2.8	5.8	4.5	6.7	6.2	3.8	4.4	6.4	6.0	3.9	5.1
GLT	NI	5	10	10	17	14	10	13	10	15	8	11
	LI	28	59	49	108	77	52	75	54	88	44	63
	FE	34	71	61	141	95	65	96	65	115	57	80
	CPU	2.4	5.0	4.3	9.8	6.7	4.5	6.7	4.5	8.0	4.1	5.6
New	NI	6	10	9	13	12	9	10	14	12	8	10
	LI	38	74	58	100	87	65	73	115	90	49	74
	FE	45	86	69	115	102	76	85	134	104	62	87
	CPU	3.1	6.0	4.8	8.1	7.2	5.3	6.0	9.4	7.3	4.3	6.2

Table 4.2: Results for "tridiagonal" system.

Choice	$\{\mathbf{x}^{(0)}\} =$	$\mathbf{x}_s$	$2\mathbf{x}_s$	$3\mathbf{x}_s$	$4\mathbf{x}_s$	$5\mathbf{x}_s$	**2**	**3**	**4**	**5**	**0**	Average
CGKT	NI	12	54	56	62	66	6	8	8	11	9	29
	LI	109	763	782	868	927	69	97	80	130	132	395
	FE	123	928	952	1063	1134	76	106	89	142	151	476
	CPU	11.0	81.7	83.3	91.8	99.9	6.7	9.3	7.7	12.4	13.6	41.7
DS	NI	42	284	*	110	*	11	17	34	22	9	66
	LI	342	3706	*	1244	*	54	106	254	135	41	735
	FE	485	5915	*	1833	*	69	143	370	184	55	1131
	CPU	41.8	506.8	*	158.2	*	5.9	12.4	32.1	15.9	4.8	97.2
BS	NI	12	14	22	16	31	8	9	12	12	8	14
	LI	65	70	213	80	375	47	54	82	82	45	111
	FE	80	87	247	99	443	57	65	97	98	58	133
	CPU	7.0	7.7	21.6	8.5	38.8	5.0	5.6	8.4	8.6	5.0	11.6
EW1	NI	44	176	*	*	*	13	19	35	24	10	45
	LI	394	2642	*	*	*	74	139	297	169	46	537
	FE	544	3828	*	*	*	97	185	419	225	58	765
	CPU	47.5	330.4	*	*	*	8.4	16.0	36.2	19.5	5.1	66.2
EW2	NI	92	242	*	*	*	9	11	22	12	10	56
	LI	401	1609	*	*	*	44	43	97	52	46	327
	FE	805	3097	*	*	*	55	56	134	65	60	610
	CPU	68.7	261.7	*	*	*	4.7	4.8	11.4	5.6	5.2	51.7
AML	NI	12	21	15	26	29	8	10	13	12	8	15
	LI	55	176	78	265	331	46	47	80	71	40	118
	FE	69	210	96	310	389	55	59	99	86	50	142
	CPU	6.0	18.3	8.3	27.1	34.5	4.8	5.1	8.7	7.5	4.3	12.5
mAML	NI	12	21	15	26	29	8	10	13	12	8	15
	LI	55	176	78	265	331	46	47	80	71	40	118
	FE	69	210	96	310	389	55	59	99	86	50	142
	CPU	6.0	18.4	8.4	27.1	34.1	4.8	5.2	8.6	7.5	4.3	12.4
GLT	NI	40	35	*	103	*	9	13	10	15	8	29
	LI	334	277	*	1221	*	51	75	54	88	43	267
	FE	466	375	*	1845	*	63	96	65	115	56	385
	CPU	40.5	33.1	*	160.2	*	5.5	8.3	5.6	10.0	4.9	33.5
New	NI	14	15	15	15	16	9	9	14	12	8	12
	LI	116	93	92	96	109	64	62	114	91	48	88
	FE	132	109	109	112	128	75	73	133	105	61	103
	CPU	11.6	9.5	9.5	9.8	11.2	6.7	6.4	11.6	9.2	5.3	9.1

Table 4.3: Results for "pentadiagonal" system.

Choice	$\{\mathbf{x}^{(0)}\} =$	$\mathbf{x}_s$	$2\mathbf{x}_s$	$3\mathbf{x}_s$	$4\mathbf{x}_s$	$5\mathbf{x}_s$	**3**	**4**	**5**	**0**	Average
CGKT	NI	7	10	12	10	18	8	10	10	9	10
	LI	83	124	137	94	194	83	124	106	142	120
	FE	91	136	154	105	221	92	136	117	162	134
	CPU	9.6	14.3	16.2	11.0	23.3	9.7	14.3	12.3	17.2	14.2
DS	NI	12	36	18	32	95	19	36	26	9	31
	LI	54	265	85	221	918	110	265	167	38	235
	FE	72	392	114	312	1425	156	392	237	52	350
	CPU	7.5	40.8	11.9	32.6	148.5	16.4	40.8	24.7	5.4	36.5
BS	NI	8	13	11	15	14	10	13	12	8	11
	LI	44	83	55	97	86	47	83	65	42	66
	FE	54	100	68	116	104	60	100	80	55	81
	CPU	5.7	10.4	7.1	12.2	11.1	6.3	10.4	8.4	5.7	8.6
EW1	NI	10	20	22	35	123	19	20	29	10	32
	LI	43	118	129	267	1588	127	118	202	42	292
	FE	54	163	177	367	2274	173	163	277	57	411
	CPU	5.6	17.1	18.4	38.5	239.4	18.0	17.1	29.0	5.9	43.2
EW2	NI	10	22	27	46	56	14	22	15	10	24
	LI	44	78	108	179	221	52	78	53	45	95
	FE	56	117	159	314	413	71	117	72	59	153
	CPU	5.9	12.2	16.7	32.6	42.8	7.6	12.2	7.5	6.2	15.9
AML	NI	9	11	11	15	14	10	11	12	9	11
	LI	43	54	53	96	81	47	54	67	40	59
	FE	54	66	66	119	99	60	66	82	54	74
	CPU	5.7	6.9	6.9	12.5	10.4	6.3	6.9	8.6	5.7	7.7
mAML	NI	9	11	11	15	14	10	11	12	9	11
	LI	43	54	53	96	81	47	54	67	39	59
	FE	54	66	66	119	99	60	66	82	53	73
	CPU	5.7	6.9	6.9	12.5	10.3	6.3	6.9	8.6	5.6	7.7
GLT	NI	10	10	11	29	14	10	10	18	8	13
	LI	48	48	44	206	56	51	48	105	41	71
	FE	61	59	57	283	73	63	59	143	54	94
	CPU	6.4	6.2	6.0	29.5	7.6	6.6	6.2	14.8	5.6	9.9
New	NI	8	13	14	11	15	9	13	12	8	11
	LI	53	92	94	58	106	52	92	80	45	74
	FE	63	109	111	71	125	62	109	94	58	89
	CPU	6.6	11.4	11.7	7.5	13.3	6.5	11.4	9.9	6.1	9.4

CPU time, i.e., 0.7 seconds. Moreover, it is at least 71% less expensive than all Newton variants with adaptive forcing-term strategies. Apart from the convergence problems just mentioned for BS and AML, the performance of all the adaptive forcing-term strategies are comparable.

Table 4.5 shows the statistics of each Newton variant with different forcing-term strategies that solves the convection-diffusion equation with 7 different values of κ. This table shows that all Newton variants except the Newton variant EW2 ($\gamma = \alpha = 1$) and the Newton variant with the constant forcing-term strategy ($\eta^{(n)} \equiv 0.95$) fail to solve the system when $\kappa > 1000$. These Newton variants either perform too many Newton iterations or converge to a solution too slowly; see Section 4.4.5. We note that the average CPU time is now excluded because it gives a skewed measure, i.e., many Newton variants fail to solve the problem with different values of κ successfully. This table shows that the Newton variants EW1, EW2 ($\gamma = 0.9, \alpha = 2$), and New solve the problem with κ up to 1000. Moreover, the CPU times for these 3 Newton variants with different initial guesses are comparable. The Newton variant EW2 solves the problem with $\kappa = 2000$ with $\gamma = 1$ and $\alpha = 1$. The Newton variant with the constant forcing-term strategy ($\eta^{(n)} \equiv 0.95$) is the only variant that solves the problem with the largest value of κ, i.e., $\kappa = 7000$.

Table 4.4: Results for extended Rosenbrock function.

Choice	$\{\mathbf{x}^{(0)}\} =$	$\mathbf{x}_s$	$2\mathbf{x}_s$	$3\mathbf{x}_s$	$4\mathbf{x}_s$	$5\mathbf{x}_s$	Average
CGKT	NI	10	2	3	2	2	3
	LI	20	4	6	4	4	7
	FE	56	7	10	7	7	17
	CPU	1.8	0.3	0.4	0.6	0.3	0.7
BS	NI	8	9	*	9	11	9
	LI	15	16	*	15	19	16
	FE	33	35	*	31	45	36
	CPU	1.1	1.3	*	1.1	1.4	1.2
DS	NI	8	11	19	13	20	14
	LI	15	19	35	22	36	25
	FE	32	50	118	57	123	76
	CPU	1.5	1.4	4.0	2.3	3.6	2.5
EW1	NI	10	14	13	17	16	14
	LI	18	24	22	29	27	24
	FE	45	66	53	87	74	65
	CPU	1.5	2.2	2.8	2.9	3.5	2.6
EW2	NI	10	11	13	13	17	12
	LI	18	19	22	22	29	22
	FE	44	50	53	57	98	60
	CPU	1.4	1.9	2.0	2.0	2.9	2.0
AML	NI	10	12	*	13	20	13
	LI	18	21	*	22	36	24
	FE	45	54	*	57	123	69
	CPU	1.4	1.6	*	1.9	3.0	2.0
mAML	NI	10	11	19	13	20	14
	LI	18	19	35	22	36	26
	FE	44	50	118	57	123	78
	CPU	1.3	1.6	3.2	2.2	2.9	2.2
GLT	NI	8	11	10	13	11	10
	LI	15	21	18	24	19	19
	FE	32	56	46	63	45	48
	CPU	1.1	1.8	1.9	2.1	1.5	1.7
New	NI	10	14	13	17	16	14
	LI	18	24	22	29	27	24
	FE	45	66	53	87	74	65
	CPU	1.6	2.3	2.4	2.9	2.9	2.4

Table 4.5: Results for the convection-diffusion equation.

Choice	$\{\kappa\} =$	100	300	500	700	1000	2000	7000
CGKT	NI	6	10	16	*	*	*	*
	LI	98	325	560	*	*	*	*
	FE	111	360	638	*	*	*	*
	CPU	4.4	24.0	43.6	*	*	*	*
DS	NI	8	12	*	*	*	*	*
	LI	50	184	*	*	*	*	*
	FE	65	222	*	*	*	*	*
	CPU	2.1	10.2	*	*	*	*	*
BS	NI	6	11	*	*	*	*	*
	LI	49	275	*	*	*	*	*
	FE	62	312	*	*	*	*	*
	CPU	2.2	20.7	*	*	*	*	*
EW1	NI	10	12	14	15	16	*	*
	LI	49	114	186	232	320	*	*
	FE	63	135	214	257	349	*	*
	CPU	2.2	7.3	12.6	15.6	24.0	*	*
EW2	NI	8	10	14	15	17	*	*
	LI	45	108	173	230	312	*	*
	FE	57	126	200	256	343	*	*
	CPU	2.0	5.8	10.2	16.1	22.7	*	*
AML	NI	9	9	13	12	*	*	*
	LI	49	109	287	208	*	*	*
	FE	62	126	333	230	*	*	*
	CPU	2.0	5.9	20.3	14.7	*	*	*
mAML	NI	9	9	26	12	*	*	*
	LI	48	109	812	203	*	*	*
	FE	61	126	868	225	*	*	*
	CPU	1.9	5.0	62.9	13.5	*	*	*
GLT	NI	6	11	*	*	*	*	*
	LI	48	183	*	*	*	*	*
	FE	60	217	*	*	*	*	*
	CPU	1.8	9.4	*	*	*	*	*
New	NI	10	11	12	14	15	*	*
	LI	49	130	162	238	337	*	*
	FE	63	150	187	262	366	*	*
	CPU	1.9	7.1	10.9	16.1	24.4	*	*
EW2	NI	51	76	82	88	92	100	*
($\gamma = \alpha = 1$)	LI	85	196	300	376	493	852	*
	FE	140	280	391	472	595	965	*
	CPU	3.9	7.9	13.0	16.2	20.2	38.1	*
Constant	NI	75	146	151	179	180	193	220
($\eta^{(n)} \equiv 0.95$)	LI	97	286	352	476	639	1079	3375
	FE	176	439	511	663	829	1286	3611
	CPU	5.0	13.3	15.9	21.1	26.5	46.3	180.9

4.4.7 Discussion

Our results show that none of the forcing-term strategies we have considered is uniformly superior, in agreement with the finding of Pawlowski et al. [54]. This suggests that it is beneficial to have several forcing-term strategies available to determine the most effective strategy for solving a given problem. Table 4.6 summarizes the best forcing-term strategy for solving each benchmark problem in the `pythNon` PSE based on average CPU time required over a number of initial guesses. We note that each of these forcing-term strategies in the table successfully solves the given problem with all the different initial guesses or problem parameters.

Table 4.6: The best Newton variant in terms of forcing-term strategy for solving the benchmark problems.

Problem	Best forcing-term strategy
Generalized function of Rosenbrock	AML and mAML
"Tridiagonal" system	New
"Pentadiagonal" system	AML and mAML
Extended Rosenbrock function	CGKT
Convection-diffusion equation	Constant ($\eta^{(n)} \equiv 0.95$)

The new forcing-term strategy (4.17) that we propose is both efficient and robust. It is the only forcing-term strategy that succeeds in the benchmark problems with different guesses and problem parameters, with the exception of the convection-diffusion equation with the largest two values of κ. In fact, if no special care is given, Newton variants with any of the forcing-term strategies in our experiments will eventually fail to solve the convection-diffusion equation when the value of κ becomes very large. That is, when the convection-diffusion equation is *convection dominated*, i.e., $\kappa \gg 1$, and the computational grid is relatively coarse, symmetric spatial discretizations for the convection-diffusion equation such as centered finite-differences become unstable, producing ill-conditioned Jacobian matrices [54]. This ill-conditioning leads to poor convergence of the indirect method for solving (1.4a). Thus, a more stable non-symmetric (or upwinded) discretization such as the *stabilized finite-element* method may be used to produce Jacobian matrices that are better conditioned [54].

In the following sections, we give examples that illustrate the following results:

- The most popular forcing-term strategy EW1 can suffer from undersolving.
- AML can suffer from suffer oversolving whereas the newly proposed modified AML ameliorates this effect.
- Adaptive forcing-term strategies require a good initial forcing term in order to successfully solve a given problem.
- An ideal forcing-term strategy should not reduce the forcing term simply based on the good agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$.
- Although adaptive forcing-term strategies generally improve the performance of Newton's method, they may still be outperformed sometimes by constant forcing-term strategies.

An example of undersolving. To show that the most popular forcing-term strategy EW1 can suffer from undersolving, Table 4.2 shows that the Newton variant EW1 fails to solve the "tridiagonal" system with initial guesses $\mathbf{x}^{(0)} = 3\mathbf{x}_s$, $4\mathbf{x}_s$, and $5\mathbf{x}_s$ because it either performs too many Newton iterations or converges to a solution too slowly. To show the benefits gained by reducing undersolving, we impose a maximum forcing term of 10^{-3} in the first 10 Newton iterations of the Newton variant EW1 for solving the "tridiagonal" system with $\mathbf{x}^{(0)} = 3\mathbf{x}_s$. Figure 4.4 shows the forcing term $\eta^{(n)}$ at each Newton iteration for EW1, the modified EW1, and New. It shows that EW1 has $\eta^{(n)} > 0.1$ in the first 12 iterations. On the other hand, New continues to reduce its forcing term in the first 12 iterations. We note that this figure only show the forcing terms for the first 20 Newton iterations; the Newton variant EW1 ultimately fails after 300 Newton iterations. This figure shows that the the modified EW1 solves the problem in 19 Newton iterations whereas New solves the problem in 15 Newton iterations.

Figure 4.5 shows the ratio of the actual reduction to the predicted reduction of the residual, $r^{(n)}$, by EW1, the modified EW1, and New at each Newton iteration. In other words, it shows the agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ for each Newton variant at each Newton iteration. That

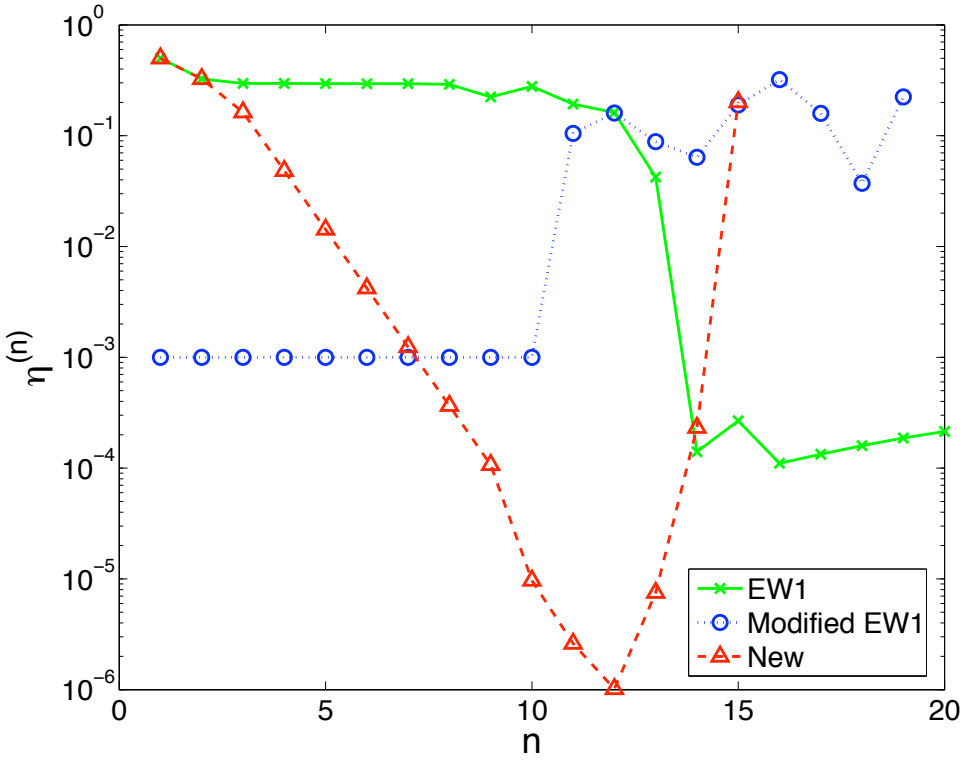

Figure 4.4: The forcing term $\eta^{(n)}$ at each Newton iteration.

is, the closer the ratio is to 1, the better agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$. This figure shows that $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ agree fairly well in the first 12 Newton iterations under EW1. At the 14th Newton iteration, it reduces its forcing term to near 10^{-4} (shown in Figure 4.4). This leads to a great disagreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$. Its ratio is less than 0.6 throughout the rest of the Newton iterations. On the other hand, this figure shows that $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ agree well under New. Compared to EW1 and the modified EW1, the curve of $r^{(n)}$ under New is increasing smoothly to 1; i.e., the forcing terms do not become too large or too small too quickly.

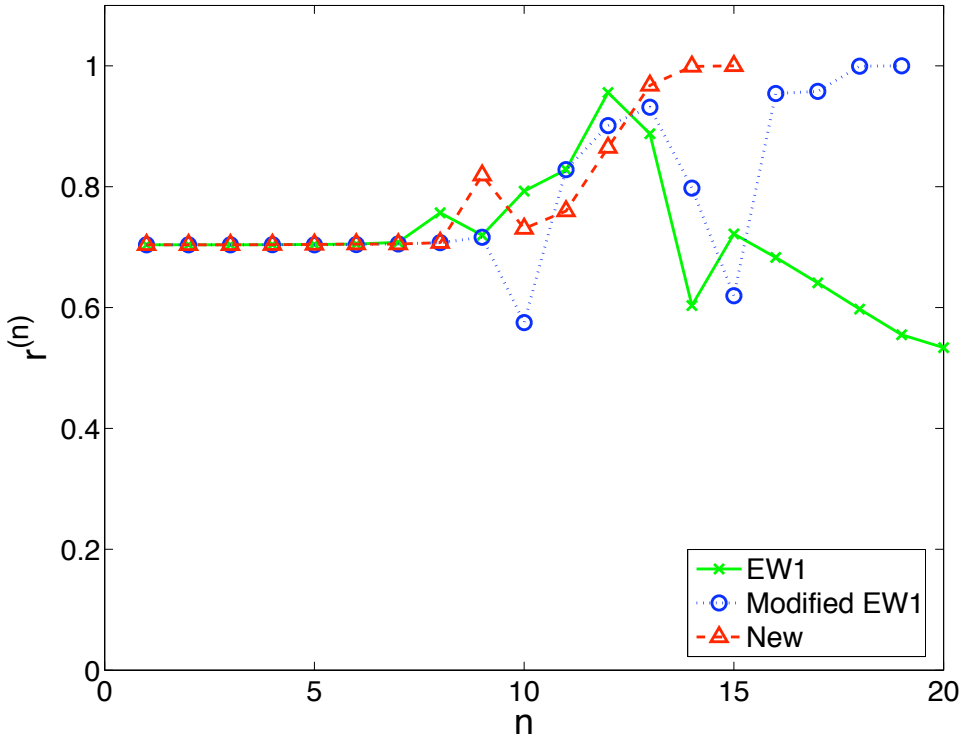

Figure 4.5: The ratio of the actual reduction to the predicted reduction of the residual, $r^{(n)}$, at each Newton iteration.

An example of oversolving in AML. To show that the Newton variant AML can suffer from oversolving (2.6) when $r^{(n)} \gg 1$, Figure 4.6 shows the forcing term $\eta^{(n)}$ of AML and mAML at

each Newton iteration when solving the extended Rosenbrock function with $\mathbf{x}^{(0)} = 3\mathbf{x}_s$. This figure shows that mAML uses the same forcing term between iteration 4 and 8. On the other hand, AML reduces its forcing term throughout the Newton iteration even though $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree greatly, thus leading to oversolving.

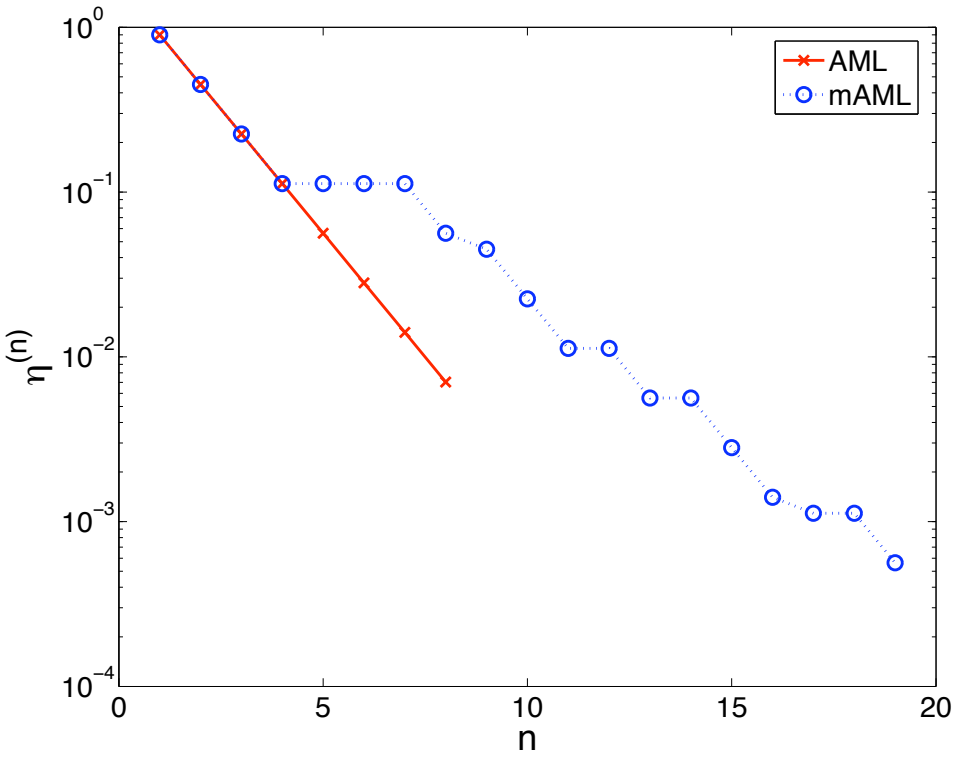

Figure 4.6: The forcing term $\eta^{(n)}$ of AML and mAML at each Newton iteration.

Figure 4.7 shows the ratio of the actual reduction to the predicted reduction of the residual of AML and mAML at each Newton iteration. This figure shows that both strategies have $r^{(4)} \approx 3.2$ and $r^{(5)} \approx 2.2$. That is, $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_{\mathbf{F}}(\mathbf{x}^{(n-1)})$ disagree greatly at $n = 4$ and 5. We note that the Newton variant AML fails after 8 Newton iterations, whereas the Newton variant mAML solves the problem with 19 Newton iterations.

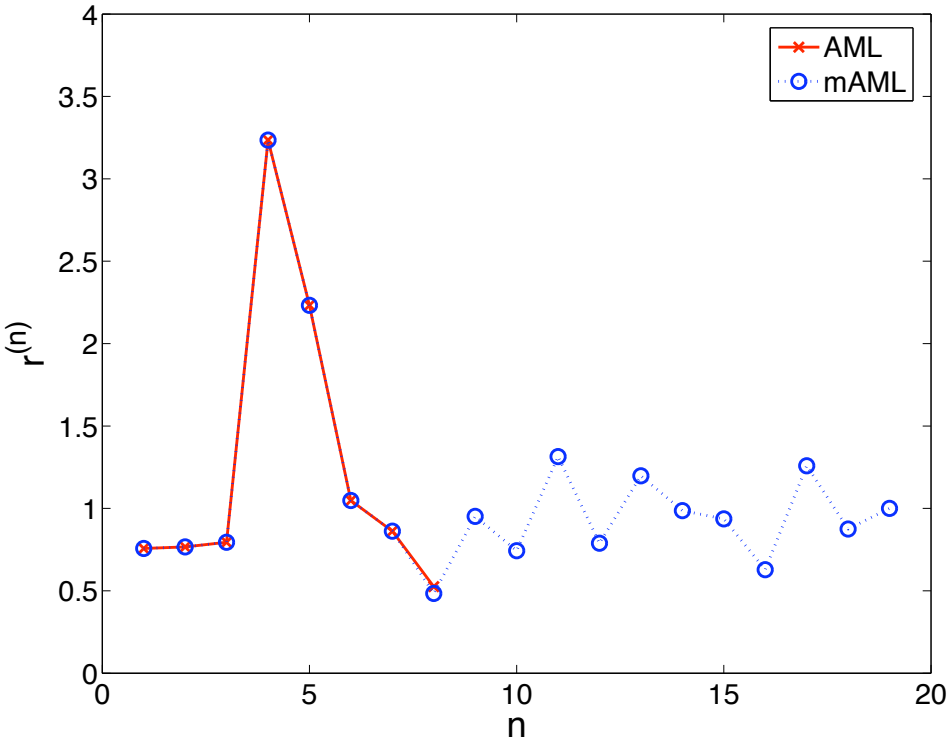

Figure 4.7: The values of $r^{(n)}$ of AML and mAML at each Newton iteration.

Choosing an initial forcing term. Our experiments show that Newton variants with adaptive forcing-term strategies require a good initial forcing term $\eta^{(0)}$ to solve a problem successfully. For example, Newton variants with adaptive forcing-term strategies such as EW1, EW2, AML, GLT, and New fail to solve the extended Rosenbrock function with $\mathbf{x}^{(0)} = 3\mathbf{x}_s$ and $\eta^{(0)} = 0.5$. On the other hand, these Newton variants solve the problem successfully with $\eta^{(0)} = 0.9$. This suggests that these Newton variants suffer from oversolving in the early Newton iterations.

The agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$. As mentioned previously, a relatively large forcing term helps to ameliorate the effect of oversolving when $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ do not agree well. However, our experiments show that the converse is not true. That is, a good agreement between $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ does not imply that a smaller forcing term will be effective. In other words, even though $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ agree well, it is possible that a small forcing term may still lead to oversolving. For example, Table 4.5 shows that EW1, which determines its forcing term based on the agreement of $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$, fails on the convection-diffusion problem when $\kappa \geq 2000$. This is because the strategy uses forcing terms that are too small in the early Newton iterations and causes oversolving. On the other hand, the Newton variant with the constant forcing-term strategy ($\eta^{(n)} \equiv 0.95$) is the only Newton variant that successfully solves the problem with different values of κ. This suggests that the problem requires the forcing terms to be relatively large throughout the Newton iteration.

To show the effect of oversolving in EW1 on the convection-diffusion problem with $\kappa = 2000$, we impose a minimum forcing term of 0.95 in EW1 for the first 20 Newton iterations. Figure 4.8 shows that EW1 reduces its forcing terms throughout the Newton iteration because $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ agree well. However, this strategy suffers from oversolving. On the other, by keeping the forcing terms to 0.95 in the first 20 iterations, the modified EW1 solves the problem successfully.

Figure 4.9 shows that the values of $r^{(n)}$ for EW1 with $n > 1$ fall between 0.5 and 1.5. This indicates that $\mathbf{F}(\mathbf{x}^{(n)})$ and $\mathbf{L}_\mathbf{F}(\mathbf{x}^{(n-1)})$ agree well after the first Newton iteration. However, the strategy fails after 11 iterations. On the other hand, the modified EW1 terminates successfully in 39 Newton iterations.

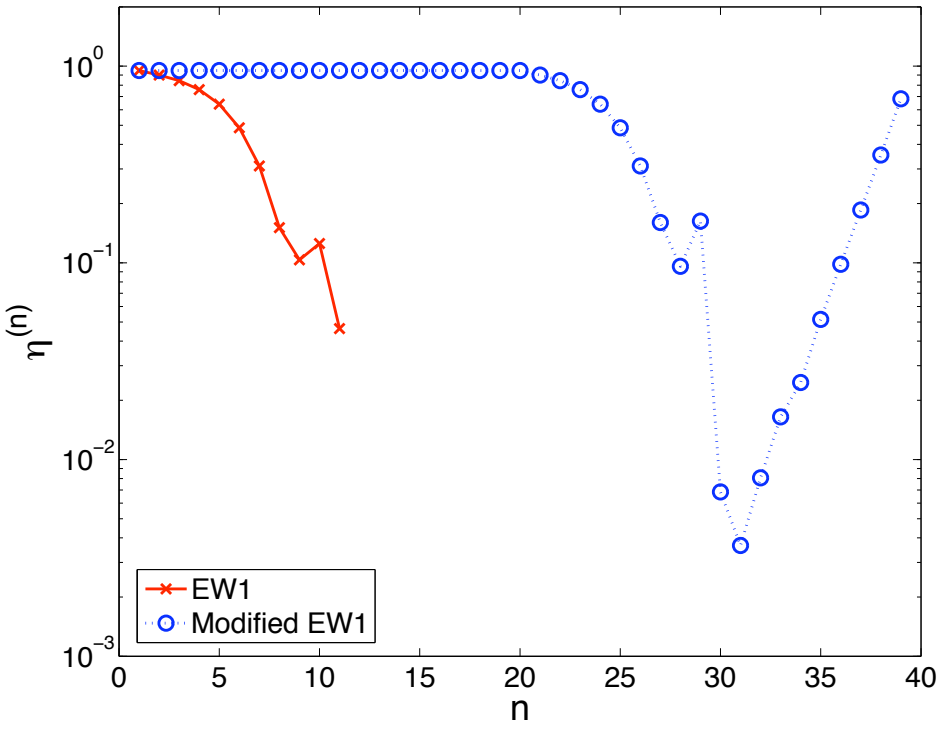

Figure 4.8: The forcing term $\eta^{(n)}$ of EW1 and the modified EW1 at each Newton iteration.

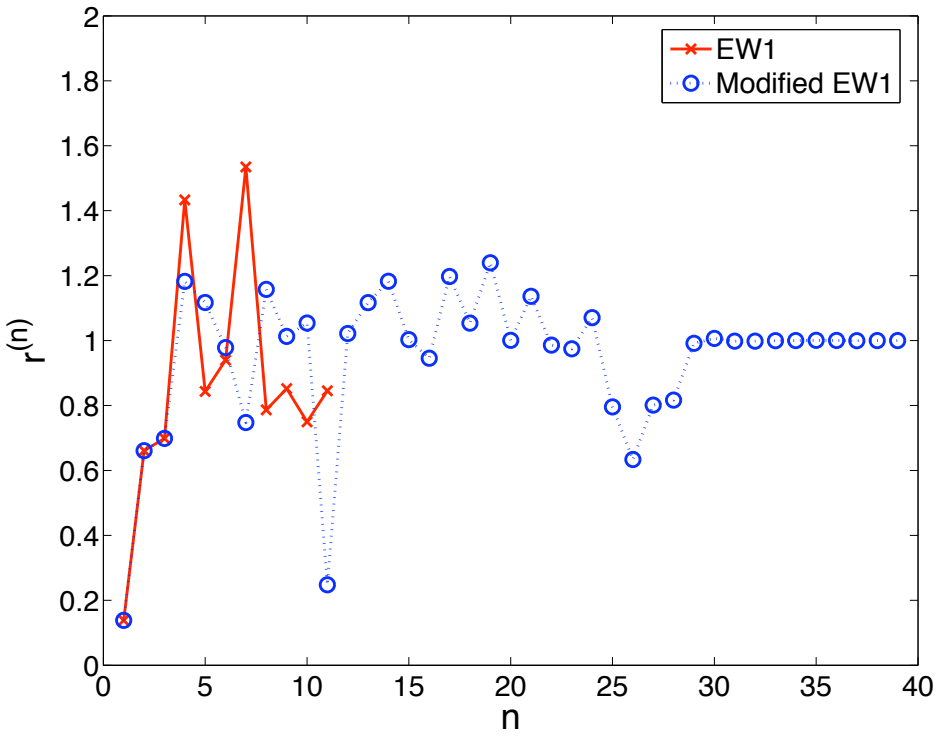

Figure 4.9: The values of $r^{(n)}$ of EW1 and the modified EW1 at each Newton iteration.

Constant forcing term vs. adaptive forcing terms. Tables 4.1–4.5 show that adaptive forcing-term strategies such as the new forcing-term strategy (4.17) generally improve the performance of the Newton variants. However, a constant forcing-term strategy ($\eta^{(n)} \equiv 10^{-4}$) is well-suited for the Newton variants that solve the extended Rosenbrock function because it turns out to be advantageous to always have an accurate computation of the Newton direction. In the case of the convection-diffusion equation, the adaptive forcing-term strategies outperform the constant forcing-term strategies for small values of κ. On the other hand, a constant forcing-term strategy with $\eta^{(n)} \equiv 0.95$ outperforms the adaptive forcing-term strategies for large values of κ; in particular none of these adaptive forcing-term strategies successfully solves the problem with $\kappa = 7000$.

Chapter 5

Conclusions

The process of solving systems of NAEs is generally difficult and complex, from analyzing the existence and uniqueness of solutions of the system to formulating a computationally efficient or at least feasible variant of Newton's method to solve the system. To facilitate this process, we have defined the concept of a PSE for the numerical solution of NAEs and created a PSE called `pythNon` for the implementation and evaluation of different variants of Newton's method. We have demonstrated the effectiveness of `pythNon` as both a teaching and research tool for rapid prototyping and numerical experimentation.

In particular, taking advantage of the power, flexibility, and ease of use of the `pythNon` PSE, we have studied the effects of a number of different forcing-term strategies for approximating the Newton direction. We have found that `pythNon` is very effective for determining the most effective forcing-term strategy on a given problem. Our results indicate that no known forcing-term strategy is uniformly superior. We have also demonstrated that Newton variants with the first forcing-term strategy of Eisenstat and Walker [21] can suffer from undersolving. To ameliorate the effects of undersolving and oversolving, we have developed a novel forcing-term strategy (4.17) that is generally the most efficient and robust in our experiments compared to the two most popular forcing-term strategies, namely the first (4.10) and second (4.11) strategies by Eisenstat and Walker [21]. We have also proposed a modification (4.16) to the strategy of An et al. [3] to ameliorate the effect of oversolving when the ratio of the actual reduction to the predicted reduction of the residual $r^{(n)} \gg 1$. This modification not only enables the Newton variant with the strategy of An et al. to solve the extended Rosenbrock function with $\mathbf{x}^{(0)} = 3\mathbf{x}_s$ successfully, but it also achieves better performance.

The `pythNon` PSE has enabled us to find that Newton variants with adaptive forcing-term strategies require a good initial forcing term to solve a problem successfully. Failing to have a good initial forcing term can lead to the possibility of oversolving, thus resulting in very little or no reduction in norm of the residual. We have also found that a good agreement between the residual and its local linear model does not imply that a smaller forcing term will be effective. This suggests that an adaptive forcing-term strategy should not reduce the forcing term simply based on the agreement of the residual and its local linear model; rather it should consider other factors, but these factors are unknown at present. This unintuitive result brings new insight for constructing an ideal adaptive forcing-term strategy. Finally, we have found that adaptive forcing-term strategies generally improve the performance of the Newton variants in our experiments; however, we have also found that constant forcing-term strategies may sometimes outperform adaptive forcing-term strategies.

The results mentioned lead to the following future work:

1. As mentioned in Chapter 1, solving a very large and stiff ODEs with an implicit time integration method typically requires the solution of a very large system of NAEs at each time step. It is possible to integrate the `pythNon` PSE as a subsystem of a PSE that solves a more complex class of problems, that is, a PSE for the numerical solution of initial value problems in ODEs.

2. We may formulate other variants of Newton's method in `pythNon`, e.g., Broyden's method [38], which approximates the Newton direction by building up an approximation of the Jacobian at each Newton iteration, or other globalization strategies mentioned in Chapter 2. Having a rich set of Newton variants available in `pythNon`, the user may determine the most effective Newton variant for solving a given problem.

3. We plan to quantify and prove the rate of convergence of the solution with the new forcing-term strategy (4.17). We also plan to evaluate the effectiveness of the new forcing-term strategy on a number of NAEs obtained from discretized PDEs because these problems usually require many linear iterations at each Newton iteration, thus providing a more comprehensive

view of the effects of different forcing-term strategies [21].

References

[1] ACM CALGO. The Collected Algorithms of the Association for Computing Machinery, October 2006. http://www.acm.org/pubs/calgo/.

[2] Alonso, J. J., LeGresley, P., van der Weide, E., Martins, J. R. R. A., and Reuther, J. J. pyMDO: A framework for high-fidelity multi-disciplinary optimization. In *Proceedings of the 10th AIAA/ISSMO Multidisciplinary Analysis and Optimization Conference* (Albany, NY, AUG 2004), no. 4480 in AIAA 2004.

[3] An, H.-B., Mo, Z.-Y., and Liu, X.-P. A choice of forcing terms in inexact Newton method. *Journal of Computational and Applied Mathematics 200*, 1 (March 2007), 47–60.

[4] Armijo, L. Minimization of functions having Lipschitz continuous first partial derivatives. *Pacific J. Math. 16* (1966), 1–3.

[5] Ascher, U. M., Mattheij, R. M. M., and Russell, R. D. *Numerical solution of boundary value problems for ordinary differential equations*, vol. 13 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. Corrected reprint of the 1988 original.

[6] Ascher, U. M., and Petzold, L. R. *Computer methods for ordinary differential equations and differential-algebraic equations*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1998.

[7] Averick, B. M., and Moré, J. J. User guide for the MINPACK-2 test problem collection. Tech. Rep. ANL/MCS-TM-157, Argonne National Laboratory, Argonne, IL, USA, 1991.

[8] Balay, S., Buschelman, K., Eijkhout, V., D. Gropp, W., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc Web page, 2001. `http://www.mcs.anl.gov/petsc`.

[9] Balay, S., Buschelman, K., Eijkhout, V., D. Gropp, W., Kaushik, D., Knepley, M. G., McInnes, L. C., Smith, B. F., and Zhang, H. PETSc users manual. Tech. Rep. ANL-95/11 - Revision 2.1.5, Argonne National Laboratory, 2004.

[10] Black, R. *Managing the testing process: practical tools and techniques for managing hardware and software testing*. John Wiley, New York, 2002.

[11] BLAS, October 2006. `http://www.netlib.org/blas/`.

[12] Brown, P. N., and Saad, Y. Hybrid Krylov methods for nonlinear systems of equations. *SIAM J. Sci. Statist. Comput. 11*, 3 (1990), 450–481.

[13] Cai, X.-C., Gropp, W. D., Keyes, D. E., and Tidriri, M. D. Newton-Krylov-Schwarz methods in CFD. In *Proceedings of the International Workshop on the Navier-Stokes Equations, Notes in Numerical Fluid Mechanics* (1994), R. Rannacher, Ed., Vieweg Verlag, Braunschweig.

[14] Coffey, T. S., Kelley, C. T., and Keyes, D. E. Pseudotransient continuation and differential-algebraic equations. *SIAM J. Sci. Comput. 25*, 2 (2003), 553–569 (electronic).

[15] Dembo, R. S., and Steihaug, T. Truncated Newton algorithms for large-scale unconstrained optimization. *Math. Programming 26*, 2 (1983), 190–212.

[16] Dennis, Jr., J. E., and Schnabel, R. B. *Numerical methods for unconstrained optimization and nonlinear equations*. Prentice Hall Series in Computational Mathematics. Prentice Hall Inc., Englewood Cliffs, NJ, 1983.

[17] Dennis, Jr., J. E., and Schnabel, R. B. *Numerical methods for unconstrained optimization and nonlinear equations*, vol. 16 of *Classics in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1996. Corrected reprint of the 1983 original.

[18] Deuflhard, P. *Newton methods for nonlinear problems*, vol. 35 of *Springer Series in Computational Mathematics*. Springer-Verlag, Berlin, 2004. Affine invariance and adaptive algorithms.

[19] Drummond, L. A., and Marques, O. A. An overview of the advanced computational software (acts) collection. *ACM Trans. Math. Softw. 31*, 3 (2005), 282–301.

[20] Eisenstat, S. C., and Walker, H. F. Globally convergent inexact Newton methods. *SIAM J. Optim. 4*, 2 (1994), 393–422.

[21] Eisenstat, S. C., and Walker, H. F. Choosing the forcing terms in an inexact Newton method. *SIAM J. Sci. Comput. 17*, 1 (1996), 16–32. Special issue on iterative methods in numerical linear algebra (Breckenridge, CO, 1994).

[22] Fokkema, D. R., Sleijpen, G. L. G., and Van der Vorst, H. A. Accelerated inexact Newton schemes for large systems of nonlinear equations. *SIAM J. Sci. Comput. 19*, 2 (1998), 657–674 (electronic).

[23] Freund, R. W. A transpose-free quasi-minimal residual algorithm for non-Hermitian linear systems. *SIAM J. Sci. Comput. 14*, 2 (1993), 470–482.

[24] Frigo, M., and Johnson, S. G. Fastest Fourier Transform in the West (FFTW).

[25] GAMS. Guide to Available Mathematical Software, October 2006. http://gams.nist.gov/.

[26] Garlan, D., and Shaw, M. An introduction to software architecture. In *Advances in Software Engineering and Knowledge Engineering*, V.Ambriola and G.Tortora, Eds., vol. I. World Scientific Publishing Company, New Jersey, 1993.

[27] Golub, G. H., and Van Loan, C. F. *Matrix computations*, third ed. Johns Hopkins Studies in the Mathematical Sciences. Johns Hopkins University Press, Baltimore, MD, 1996.

[28] Gomes-Ruggiero, M. A., da Rocha Lopes, V. L., and Benavides, J. V. T. A globally convergent inexact newton method with a new choice for the forcing term. *Annals of Operations Research* (2006). To appear.

[29] Gomes-Ruggiero, M. A., Kozakevich, D. N., and Martinez, J. M. A numerical study on large-scale nonlinear solvers. *Comput. Math. Appl. 32*, 3 (1996), 1–13.

[30] Griewank, A. *Evaluating derivatives*, vol. 19 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2000. Principles and techniques of algorithmic differentiation.

[31] Higham, D. J. Trust region algorithms and timestep selection. *SIAM J. Numer. Anal. 37*, 1 (1999), 194–210 (electronic).

[32] Hoffmann, K. A. *Computational fluid dynamics for engineers*. Engineering Education System, 1989.

[33] Houstis, E. N., Rice, J. R., Weerawarana, S., Catlin, A. C., Papachiou, P., Wang, K.-Y., and Gaitatzes, M. Pellpack: a problem-solving environment for pde-based applications on multicomputer platforms. *ACM Trans. Math. Softw. 24*, 1 (1998), 30–73.

[34] IMSL, October 2006. http://www.vni.com/products/imsl/.

[35] Ji, Z., Wang, B., Liu, Z., and Zeng, Q.-C. Some advances in computational geophysical fluid dynamics. *Progr. Natur. Sci. (English Ed.) 4*, 6 (1994), 688–697.

[36] Kelley, C. T. Solution of the Chandrasekhar H-equation by Newton's method. *J. Math. Phys. 21*, 7 (1980), 1625–1628.

[37] Kelley, C. T. *Iterative methods for linear and nonlinear equations*, vol. 16 of *Frontiers in Applied Mathematics*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1995. With separately available software.

[38] Kelley, C. T. *Solving nonlinear equations with Newton's method*. Fundamentals of Algorithms. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 2003.

[39] Kelley, C. T., and Pettitt, B. M. A fast solver for the Ornstein-Zernike equations. *J. Comput. Phys. 197*, 2 (2004), 491–501.

[40] Kincaid, D., and Cheney, W. *Numerical analysis*, third ed. Brooks/Cole Publishing Co., Pacific Grove, CA, 2002. Mathematics of scientific computing.

[41] Knoll, D. A., and Keyes, D. E. Jacobian-free Newton-Krylov methods: a survey of approaches and applications. *J. Comput. Phys. 193*, 2 (2004), 357–397.

[42] Kollerstrom, N. Thomas Simpson and "Newton's method of approximation": an enduring myth. *British J. Hist. Sci. 25*, 3(86) (1992), 347–354.

[43] LAPACK. Linear Algebra PACKage, October 2006. http://www.netlib.org/lapack/.

[44] Li, G. Y. Successive column correction algorithms for solving sparse nonlinear systems of equations. *Math. Programming 43*, 2, (Ser. A) (1989), 187–207.

[45] Lukšan, L. Inexact trust region method for large sparse systems of nonlinear equations. *J. Optim. Theory Appl. 81*, 3 (1994), 569–590.

[46] Moler, C. B. *Numerical computing with MATLAB*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 2004.

[47] Mulder, W. A., and van Leer, B. Experiments with implicit upwind methods for the Euler equations. *J. Comput. Phys. 59*, 2 (1985), 232–246.

[48] National Institute of Standards. Matrix Market, October 2006. `http://math.nist.gov/MatrixMarket/`.

[49] Netlib, October 2006. http://www.netlib.org/.

[50] Nichols, J. C., and Zingg, D. W. A three-dimensional multi-block Newton-Krylov flow solver for the Euler equations. *American Institute of Aeronautics and Astronautics*, AIAA 2005-5230 (2005).

[51] Numerical Algorithms Group, October 2006. http://www.nag.co.uk/.

[52] Ornstein, L. S., and Zernike, F. Accidental deviations of density and opalescence at the critical point of a single substance. *Proc. K. Ned. Akad. Wet. 17* (1914), 793–806.

[53] Ortega, J. M., and Rheinboldt, W. C. *Iterative solution of nonlinear equations in several variables*. Academic Press, New York, 1970.

[54] Pawlowski, R. P., Shadid, J. N., Simonis, J. P., and Walker, H. F. Globalization techniques for Newton–Krylov methods and applications to the fully coupled solution of the Navier–Stokes equations. *SIAM Review 48*, 4 (2006), 700–721.

[55] Pernice, M., and Walker, H. F. NITSOL: a Newton iterative solver for nonlinear systems. *SIAM J. Sci. Comput. 19*, 1 (1998), 302–318 (electronic). Special issue on iterative methods (Copper Mountain, CO, 1996).

[56] Pinchover, Y., and Rubinstein, J. *An introduction to partial differential equations*. Cambridge University Press, Cambridge, 2005.

[57] Powell, M. J. D. A hybrid method for nonlinear equations. In *Numerical methods for nonlinear algebraic equations (Proc. Conf., Univ. Essex, Colchester, 1969)*. Gordon and Breach, London, 1970, pp. 87–114.

[58] pyMPI, October 2006. `http://pympi.sourceforge.net/`.

[59] Quarteroni, A., and Formaggia, L. Mathematical modelling and numerical simulation of the cardiovascular system. In *Handbook of numerical analysis. Vol. XII*, Handb. Numer. Anal., XII. North-Holland, Amsterdam, 2004, pp. 3–127.

[60] Rice, J. R., and Boisvert, R. F. From scientific software libraries to problem-solving environments. *IEEE Computational Science & Engineering 3*, 3 (Fall 1996), 44–53.

[61] Saad, Y., and Schultz, M. H. GMRES: a generalized minimal residual algorithm for solving nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput. 7*, 3 (1986), 856–869.

[62] ScaLAPACK, October 2006. http://www.netlib.org/scalapack/.

[63] Schatzman, M. *Numerical analysis*. A mathematical introduction. Oxford University Press, 2002.

[64] SciPy, October 2006. `http://www.scipy.org/`.

[65] Shampine, L. F. *Numerical solution of ordinary differential equations*. Chapman & Hall, New York, 1994.

[66] Shampine, L. F., Ketzscher, R., and Forth, S. A. Using AD to solve BVPs in MATLAB. *ACM Trans. Math. Softw. 31*, 1 (2005), 79–94.

[67] Sommerville, I. *Software Engineering*, 7th ed. Pearson Education Limited, Essex, England, 2004.

[68] SuperLU, October 2006. http://acts.nersc.gov/superlu/.

[69] Taylor, A., and Hindmarsh, A. User documentation for KINSOL, a nonlinear solver for sequential and parallel computers. Tech. Rep. Technical Report UCRL-ID-131185, Lawrence Livermore Nat'l Laboratory, July 1998.

[70] Trefethen, L. N., and Bau, III, D. *Numerical linear algebra*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, PA, 1997.

[71] van der Vorst, H. A. Bi-CGSTAB: a fast and smoothly converging variant of Bi-CG for the solution of nonsymmetric linear systems. *SIAM J. Sci. Statist. Comput. 13*, 2 (1992), 631–644.

[72] Van Loan, C. F. *Introduction to Scientific Computing. A Matrix-Vector Approach Using Matlab*, 2nd ed. Prentice-Hall, Upper Saddle River, New Jersey, 07458, 2000.

[73] van Rossum, G. Python library reference, October 2006. `http://docs.python.org/lib/lib.html`.

[74] Watson, L. T., Billups, S. C., and Morgan, A. P. Algorithm 652. HOMPACK: a suite of codes for globally convergent homotopy algorithms. *ACM Trans. Math. Software 13*, 3 (1987), 281–310.

[75] Welty, J., Wicks, C., Wilson, R., and Rorrer, G. *Fundamentals of Momentum, Heat, and Mass Transfer*. John Wiley and Sons Ltd., 2001.

[76] Ypma, T. J. Historical development of the Newton-Raphson method. *SIAM Rev. 37*, 4 (1995), 531–551.