Algorithms and Environments for Complementarity

By

Todd S. Munson

A dissertation submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

2002

Abstract

Complementarity problems arise in a wide variety of disciplines. Prototypical examples include the Wardropian and Walrasian equilibrium models encountered in the engineering and economic disciplines and the first order optimality conditions for nonlinear programs from the optimization community. The main focus of this thesis is algorithms and environments for solving complementarity problems.

Environments, such as AMPL and GAMS, are used by practitioners to easily write large, complex models. Support for these packages is provided by PATH 4.x and SEMI through the customizable solver interface specified in this thesis. The main design feature is the abstraction of core components from the code with implementations tailored to a particular environment supplied either at compile or run time. This solver interface is then used to develop new links to the MATLAB and NEOS tools.

Preprocessing techniques are an integral part of linear and mixed integer programming codes and are primarily used to reduce the size and complexity of a model prior to solving it. For example, wasted computation is avoided when an infeasible model is detected. This thesis documents the new techniques for preprocessing complementarity problems contained in the PATH 4.x and SEMI algorithms.

PATH 4.x is more reliable than prior versions of the code and has been able to find solutions to previously unsolvable models. The reasons for the improvement are discussed in this thesis and include new theoretical developments for a globalization scheme based on the Fischer-Burmeister merit function, and enhancements made to the linear complementarity solver, nonmonotone linesearch, and restart strategy.

SEMI is an alternative to PATH 4.x based on the semismooth algorithm. The main

computation in PATH 4.x is to solve linear complementarity problems, which can be quite expensive. The new SEMI code described in this thesis only solves linear systems of equations and uses iterative methods to process very large models.

The theme of this thesis is “enabling” technologies in complementarity: environments enable practitioners to easily specify models; sophisticated codes enable them to solve the models; and theory assures them that the algorithm is well-defined and efficiently processes models.

Acknowledgements

I would like to thank my family and educators throughout the years who have taught me many important lessons. Without them, I would not have attended university, let alone pursue an advanced degree.

Several people have influenced the work in this thesis. The most important are my advisor, Michael Ferris, who first introduced me to complementarity, and Steven Dirkse and Danny Ralph who paved the way for the current version of PATH. Others include Stephen Billups, Arne Drud, Francisco Facchinei, Andreas Fischer, David Gay, Christian Kanzow, Alex Meeraus, Jorge Moré, and Steve Wright. I have also benefitted immensely from interaction with other graduate students including Paul Bradley, Qun Chen, William Donaldson, Glenn Fung, Yuh-Jye Lee, Jinho Lim, David Musicant, Krung Sinapiromsaran, and Meta Voelker.

Many individuals have tested the software developed in this thesis and have provided feedback. Special thanks to Sherman Robinson, Thomas Rutherford, and Francis Tin-loi who continue to challenge the solvers.

The members of my committee, Thomas Cox, Miron Livny, Olvi Mangasarian, and Stephen Robinson, have made many useful suggestions to improve the presentation of this thesis.

Finally, I want to give many thanks to the secretarial and support staff for the Computer Sciences Department at the University of Wisconsin - Madison. They have kept me on track and provided encouragement throughout my graduate studies. I am especially grateful to Laura Cuccia as she has constantly gone above and beyond the call of duty. When I needed something done on short notice, she always came through.

The work in this thesis was supported by NSF grants CCR-9972372 and CCR-9619765, AFOSR grant F49620-98-1-0417, GAMS Development Corporation, and a fellowship from Cisco Systems.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

A fundamental problem in mathematics is to solve a square system of nonlinear equations. The complementarity problem, a generalization of such systems, is the subject of this thesis. Prototypical examples of complementarity problems include Wardropian [121] and Walrasian equilibrium [3] models encountered in the engineering and economic disciplines [47], and the first order optimality conditions for nonlinear programs [78, 79] from the optimization community. Complementarity also has applications in game theory [94, 95, 80, 82] and options pricing [69, 122]. Examples of complementarity problems arising in a wide variety of disciplines are surveyed in [26, 46].

This chapter contains an overview of complementarity theory and algorithms. We begin with a precise definition of mixed complementarity problems and discuss their relationship to variational inequalities. Section 1.2 then presents two equivalent representations of the complementarity problem as unconstrained systems of nonsmooth equations using the normal map and a semismooth reformulation. Conditions sufficient for the existence of solutions to complementarity problems are developed in Section 1.3. We then discuss algorithms for solving complementarity problems based on the normal map and semismooth reformulations in Section 1.4 and provide regularity conditions and local convergence results. Finally, Section 1.5 summarizes the remainder of this thesis.

Before proceeding, we give a few words about the notation used. If F is a vector valued function, we denote its Jacobian at a point x by $F'(x)$ and let $\nabla F(x)$ signify the transposed Jacobian. The elements of $F'(x)$ and $\nabla F(x)$ are defined componentwise as follows:

$$\begin{aligned} [F'(x)]_{i,j} &:= \tfrac{\partial F_i(x)}{\partial x_j} \\ [\nabla F(x)]_{i,j} &:= \tfrac{\partial F_j(x)}{\partial x_i}. \end{aligned}$$

The gradient of a real valued function f will be denoted by ∇f and will always be viewed as a column vector. Finally, $\langle \cdot, \cdot \rangle$ denotes the inner product.

1.1 Complementarity

Mixed complementarity problems are specified by three pieces of data, namely lower bounds ℓ, upper bounds u, and a function F where we will assume throughout this thesis that F is continuously differentiable.

Definition 1.1.1 (Mixed Complementarity Problem) *Given a continuously differentiable function $F : \mathbf{R}^n \to \mathbf{R}^n$, and lower and upper bounds*

$$\ell \in \{\mathbf{R} \cup \{-\infty\}\}^n$$

$$u \in \{\mathbf{R} \cup \{+\infty\}\}^n$$

with $[\ell, u]$, the Cartesian product of the closed (possibly unbounded) intervals $[\ell_i, u_i]$, nonempty. Then the mixed complementarity problem*, MCP(F, ℓ, u), is to find a $z \in \mathbf{R}^n$ such that* precisely *one of the following holds for each $i \in \{1, \dots, n\}$:*

$$\begin{array}{lll} F_i(z) = 0 & \textit{and} & \ell_i \le z_i \le u_i \\ F_i(z) > 0 & \textit{and} & z_i = \ell_i \\ F_i(z) < 0 & \textit{and} & z_i = u_i. \end{array} \tag{1}$$

Two specializations of this framework are easily recognizable.

Definition 1.1.2 (Nonlinear System of Equations) *The complementarity problem MCP($F, \{-\infty\}^n, \{\infty\}^n$) is equivalent to solving a* square system of nonlinear equations*: find $z \in \mathbf{R}^n$ such that*

$$F(z) = 0.$$

Definition 1.1.3 (Nonlinear Complementarity Problem [19]) *The mixed complementarity problem MCP($F, \{0\}^n, \{\infty\}^n$) is equivalent to finding a $z \in \mathbf{R}^n$ such that*

$$0 \leq z \quad \perp \quad F(z) \geq 0$$

where the $\perp$ notation is used to signify that z and $F(z)$ are orthogonal, $\langle z, F(z) \rangle = 0$. This problem is termed a nonlinear complementarity problem*.*

A complete discussion of nonlinear systems of equations can be found in [96] while surveys of theory, algorithms, and applications for nonlinear complementarity problems can be found in [36, 65]. A special case of the nonlinear complementarity problem that has received much attention is when F is a linear function, the standard linear complementarity problem [21].

For convenience, we will write a mixed complementarity problem as:

$$\ell \leq z \leq u \quad \perp \quad F(z)$$

where the $\perp$ notation is used in a generalized sense to mean that at a solution, z, (1) holds for each z_i and $F_i(z)$ pair.

We can also pose the mixed complementarity problem as a variational inequality.

Definition 1.1.4 (Variational Inequality [68, 83]) *Let $C \subseteq \mathbf{R}^n$ be nonempty, closed and convex, and $F : \mathbf{R}^n \to \mathbf{R}^n$ be continuously differentiable. Then the* variational inequality, *VI(F, C), is to find a* $z \in C$ *such that*

$$\langle F(z), \bar{z} - z \rangle \geq 0$$

for all $\bar{z} \in C$.

The variational inequality can also be written as the generalized equation [105]:

$$0 \in F(z) + N_C(z)$$

where $N_C(z)$ denotes the normal cone to C at z.

Definition 1.1.5 (Normal Cone [110]) *The* normal cone *to a closed convex set* C *at* z *is defined by*

$$N_C(z) := \begin{cases} \{y \mid \langle \bar{z} - z, y \rangle \leq 0,\ \forall \bar{z} \in C\} & \textit{if } z \in C \\ \emptyset & \textit{otherwise.} \end{cases}$$

For example, the normal cone to $[\ell, u]$ can be determined as a Cartesian product, namely

$$N_{[\ell,u]}(z) = \prod_{i=1}^{n} N_{[\ell_i,u_i]}(z_i).$$

Each component in this product depends on z_i, ℓ_i and u_i but is either $\mathbf{R}$, $\mathbf{R}_+$, $\mathbf{R}_-$, $\{0\}$, or $\emptyset$, where $\mathbf{R}_+$ denotes the set of nonnegative real numbers and $\mathbf{R}_-$ denotes the set of nonpositive real numbers. In particular,

$$N_{[\ell_i,u_i]}(z_i) = \begin{cases} \mathbf{R} & \text{if } \ell_i = z_i = u_i \\ \mathbf{R}_- & \text{if } \ell_i = z_i < u_i \\ \{0\} & \text{if } \ell_i < z_i < u_i \\ \mathbf{R}_+ & \text{if } \ell_i < z_i = u_i \\ \emptyset & \text{otherwise.} \end{cases}$$

Theorem 1.1.6 ([77]) *The following are equivalent:*

(a) z *solves MCP*(F, ℓ, u)*.*

(b) z *solves VI*$(F, [\ell, u])$*.*

(c) $0 \in F(z) + N_{[\ell,u]}(z)$*.*

Note that variational inequalities are defined over general closed, convex sets, while the mixed complementarity problem is defined over a box. However, under suitable constraint qualifications [85], variational inequalities can be written as mixed complementarity problems by adding multipliers [65]. One special case we consider in this thesis is when C is a polyhedral set.

Theorem 1.1.7 *(Propositions 1 and 2 of [109]) Let* $F : \mathbf{R}^n \to \mathbf{R}^n$ *and*

$$C = [\ell, u] \cap \{z \mid Az \geq b\}$$

for some $A \in \mathbf{R}^{m \times n}$ *and* $b \in \mathbf{R}^m$*. Then the following hold:*

(a) If $(\bar{z}, \bar{\mu})$ *solves the generalized equation*

$$0 \in \begin{bmatrix} F(z) - A^T\mu \\ Az - b \end{bmatrix} + \begin{bmatrix} N_{[\ell,u]}(z) \\ N_{\mathbf{R}^m_+}(\mu) \end{bmatrix} \tag{2}$$

then $\bar{z}$ *solves VI*(F, C)*.*

(b) If $\bar{z}$ *solves VI*(F, C) *then the linear optimization problem*

$$\begin{array}{ll} \min_{\mu \in \mathbf{R}^m_+} & \langle A\bar{z} - b, \mu \rangle \\ s.t. & 0 \in F(\bar{z}) - A^T\mu + N_{[\ell,u]}(\bar{z}). \end{array} \tag{3}$$

has a nonempty solution set. Further, for any $\bar{\mu}$ *solving (3),* $(\bar{z}, \bar{\mu})$ *solves (2).*

1.2 Nonsmooth Reformulations

The mixed complementarity problem can be reformulated as an unconstrained system of nonsmooth equations. The classical reformulation for nonlinear complementarity problems uses the minimum map

$$\min(z, F(z))$$

where the minimum is defined componentwise. Clearly, $\min(z, F(z)) = 0$ if and only if z solves the nonlinear complementarity problem defined by F.

Two reformulations of the mixed complementarity problem as an unconstrained systems of nonsmooth equation are used in this thesis. The first is as a piecewise smooth system of equations using the normal map, while the second is as a semismooth system of equations using the Fischer-Burmeister function.

1.2.1 Normal Map

The first reformulation of the mixed complementarity problem we consider is as a piecewise smooth system of equations using the normal map [32, 107]. To explain this reformulation, let $[\ell, u]$ be a nonempty subset of $\mathbf{R}^n$. Associated with each face $\mathcal{F}_i$ of $[\ell, u]$ is a (full-dimensional) polyhedral set, $\sigma_i = \mathcal{F}_i + N_{\mathcal{F}_i}$ where $N_{\mathcal{F}_i}$ is the normal cone to $\mathcal{F}_i$ at any point in the relative interior of $\mathcal{F}_i$. Note that $N_{\mathcal{F}_i}$ is constant on the relative interior of $\mathcal{F}_i$ [107]. The collection of these polyhedra comprise a piecewise linear manifold of $\mathbf{R}^n$ called the *normal manifold* and is denoted by $\mathcal{N}_{[\ell,u]}$. Each σ_i is called a *cell* of $\mathcal{N}_{[\ell,u]}$. A full description along with important properties of the normal manifold are given in [107, 104]. For example, the cells of $\mathcal{N}_{\mathbf{R}^n_+}$ are the orthants of $\mathbf{R}^n$.

We denote by $\pi_{[\ell,u]}(\cdot)$ the Euclidean projection mapping onto $[\ell, u]$. We note that

$\pi_{[\ell,u]}(x)$ can be defined componentwise as:

$$\pi_{[\ell_i,u_i]}(x_i) = \begin{cases} l_i & \text{if } x_i \le l_i \\ u_i & \text{if } x_i \ge u_i \\ x_i & \text{otherwise.} \end{cases}$$

The normal map [107] induced by $(F, [\ell, u])$ is the function $F_{[\ell,u]} : \mathbf{R}^n \to \mathbf{R}^n$ given by

$$F_{[\ell,u]}(x) \;=\; F(\pi_{[\ell,u]}(x)) + x - \pi_{[\ell,u]}(x).$$

The projection map $\pi_{[\ell,u]}(x)$ and hence the normal map $F_{[\ell,u]}(x)$ are smooth in the interior of each cell of $\mathcal{N}_{[\ell,u]}$. Furthermore, $F_{[\ell,u]}$ is a continuous map on $\mathbf{R}^n$ and the points of nondifferentiability correspond to the boundaries of the cells of $\mathcal{N}_{[\ell,u]}$.

Theorem 1.2.1 *Let MCP(F, ℓ, u) be given. Then the following hold:*

(a) If z solves MCP(F, ℓ, u) then $F_{[\ell,u]}(z - F(z)) = 0$.

(b) If $F_{[\ell,u]}(x) = 0$ then $\pi_{[\ell,u]}(x)$ solves MCP(F, ℓ, u).

1.2.2 Semismooth Formulation

An alternative reformulation writes the mixed complementarity problem as a system of semismooth equations. Semismooth functions were introduced in [90] and have subsequently been extended to vector valued functions [101, 102]. In order to define the semismooth property, we let $G : \mathbf{R}^n \to \mathbf{R}^n$ be a locally Lipschitzian function and note that by Rademacher's theorem G is differentiable almost everywhere. Let D_G denote the set of points where G is differentiable.

Definition 1.2.2 (B-subdifferential [102]) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be locally Lipschitzian at x. The* B-subdifferential *of G at x is*

$$\partial_B G(x) := \left\{ H \mid \exists \{x^k\}, x^k \in D_G, \;\; \textit{with} \;\; \lim_{x^k \to x} G'(x^k) = H \right\}.$$

Definition 1.2.3 (Clarke subdifferential [17]) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be locally Lipschitzian at x. The* Clarke subdifferential *of G at x is defined as*

$$\partial G(x) := co\ \partial_B G(x),$$

where co denotes the convex hull of a set.

We can now define semismooth functions, which lie between Lipschitz functions and continuously differentiable functions.

Definition 1.2.4 (Semismooth) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be locally Lipschitzian at $x \in \mathbf{R}^n$. We say that G is* semismooth at x *if*

$$\lim_{\substack{H \in \partial G(x+tv') \\ v' \to v, t \downarrow 0}} Hv'$$

exists for all $v \in \mathbf{R}^n$. Furthermore, if G is semismooth at each $x \in \mathbf{R}^n$ then G is said to be a semismooth function.

Definition 1.2.5 (Strongly Semismooth) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be locally Lipschitzian and suppose that G is semismooth at x. We say that G is* strongly semismooth at x *if for any $H \in \partial G(x+d)$, and for any $d \to 0$,*

$$Hd - G'(x; d) = O(\|d\|^2).$$

Furthermore, if G is strongly semismooth at each $x \in \mathbf{R}^n$, then G is said to be a strongly semismooth function.

Note that if G is semismooth at x then it is also directionally differentiable at x.

To describe the semismooth reformulation of a complementarity problem we first define an NCP-function.

Definition 1.2.6 (NCP-function) *We call a mapping* $\phi : \mathbf{R}^2 \to \mathbf{R}$ *an* NCP-function *if it satisfies*

$$\phi(a,b) = 0 \Longleftrightarrow 0 \leq a \ \perp \ b \geq 0.$$

One example of an NCP-function is the Fischer-Burmeister [50] function

$$\phi_{FB}(a,b) := \sqrt{a^2+b^2} - a - b.$$

We then partition the index set $I = \{1, \ldots, n\}$ in the following way:

$$\begin{aligned} I_l &:= \{i \in I \,|\, -\infty < l_i < u_i = +\infty\}, \\ I_u &:= \{i \in I \,|\, -\infty = l_i < u_i < +\infty\}, \\ I_{lu} &:= \{i \in I \,|\, -\infty < l_i < u_i < +\infty\}, \\ I_f &:= \{i \in I \,|\, -\infty = l_i < u_i = +\infty\}. \end{aligned}$$

That is, I_l, I_u, I_{lu} and I_f denote the set of indices $i \in I$ with finite lower bounds only, finite upper bounds only, finite lower and upper bounds, and no finite bounds on the variable x_i, respectively. Hence, the subscripts in the above index sets indicate which bounds are finite, with the only exception of I_f which contains the free variables.

Following the idea in [5], we then define the operator $\Phi : \mathbf{R}^n \to \mathbf{R}^n$ componentwise as:

$$\Phi_i(x) := \begin{cases} \phi_{FB}(x_i - l_i, F_i(x)) & \text{if } i \in I_l, \\ -\phi_{FB}(u_i - x_i, -F_i(x)) & \text{if } i \in I_u, \\ \phi_{FB}(x_i - l_i, \phi_{FB}(u_i - x_i, -F_i(x))) & \text{if } i \in I_{lu}, \\ -F_i(x) & \text{if } i \in I_f. \end{cases}$$

Theorem 1.2.7 ([5]) *The following hold:*

(a) $\Phi(z) = 0$ *if and only if* z *solves MCP*(F, ℓ, u)*.*

(b) Φ *is semismooth.*

(c) If F *has Lipschitz derivatives, then* Φ *is strongly semismooth.*

1.3 Existence of Solutions

Having defined the mixed complementarity problem and presented the reformulations used in this thesis, we now turn our attention toward conditions guaranteeing the existence of a solution to a mixed complementarity problem. We split the existence results into those for linear and nonlinear models.

1.3.1 Linear Problems

Existence and uniqueness results for linear complementarity problems have to do with matrix classes. Throughout this section we will deal with the linear mixed complementarity problem LMCP(M, q, ℓ, u) where $M \in \mathbf{R}^{n\times n}$ and $q \in \mathbf{R}^n$. LMCP(M, q, ℓ, u) is the mixed complementarity problem defined by $F(z) := Mz + q$, ℓ, and u.

We then define the following matrix classes [21]:

Definition 1.3.1 (Positive Semidefinite) *If*

$$x^T M x \geq 0$$

for all $x \in \mathbf{R}^n$*, then* M *is a* positive semidefinite *matrix.*

Definition 1.3.2 (Positive Definite) *If*

$$x^T M x > 0$$

for all $x \in \mathbf{R}^n$ *with* $x \neq 0$*, then* M *is a* positive definite *matrix.*

Definition 1.3.3 (P_0-Matrix) *If all the principal minors of* M *are nonnegative,* M *is called a* P_0-matrix.

Definition 1.3.4 (P-Matrix) *If all the principal minors of* M *are positive, then* M *is called a* P-matrix. *Alternative characterizations of* P*-matrices can be found in [49].*

Using these definitions, we have the following existence results for any linear mixed complementarity problem LMCP(M, q, ℓ, u).

Theorem 1.3.5 ([21]) *The following hold:*

(a) If M *is positive semidefinite then LMCP(*M, q, ℓ, u*) has a convex solution set.*

(b) If M *is positive definite, then LMCP(*M, q, ℓ, u*) has a unique solution for all* q*.*

See [21] for further matrix classes and results for standard linear complementarity problems.

A complete characterization of existence and uniqueness of a solution for the standard linear complementarity problem can be made using the P-matrix property.

Theorem 1.3.6 ([21]) *The following are equivalent:*

(a) M *is a* P*-matrix.*

(b) LMCP($M, q, \{0\}^n, \{\infty\}^n$*) has a unique solution for all* q*.*

A generalization of the P-matrix property used with linear mixed complementarity problems is for $M_{[\ell,u]}$, the normal map associated with M, to be coherently oriented.

Definition 1.3.7 (Coherent Orientation [107]) *$M_{[\ell,u]}$ is* coherently oriented *if the determinants of the affine maps associated with each cell of the normal manifold of $M_{[\ell,u]}$ have the same nonzero sign.*

Theorem 1.3.8 ([107]) *The following are equivalent:*

(a) The normal map $M_{[\ell,u]}$ is a Lipschitzian homeomorphism of $\mathbf{R}^n$ onto $\mathbf{R}^n$.

(b) $M_{[\ell,u]}$ is coherently oriented.

In particular, if $M_{[\ell,u]}$ is coherently oriented, then $M_{[\ell,u]} + q = 0$ has a unique solution for all q.

For standard linear complementarity problems, coherent orientation of the normal map is equivalent to M being a P-matrix since the orthants are precisely the cells of $M_{[\{0\}^n,\{\infty\}^n]}$.

1.3.2 Nonlinear Problems

Existence results for nonlinear mixed complementarity problems are generalizations of those encountered when dealing with linear problems. A basic existence result for nonlinear models based on Brouwer's fixed-point theorem is when $[\ell, u]$ is nonempty and compact.

Theorem 1.3.9 *If $[\ell, u]$ is nonempty and compact then MCP(F, ℓ, u) has a solution for any continuous F.*

Monotonicity plays a key role in existence results for nonlinear models. Let $C \subseteq \mathbf{R}^n$ be nonempty, closed, and convex. We then distinguish between three forms of monotonicity over the set C.

Definition 1.3.10 (Monotone) *If*

$$\langle z - \bar{z}, F(z) - F(\bar{z}) \rangle \geq 0$$

for all $z \in C$ *and* $\bar{z} \in C$ *then* F *is* monotone *on* C*.*

Definition 1.3.11 (Strictly Monotone) *If*

$$\langle z - \bar{z}, F(z) - F(\bar{z}) \rangle > 0$$

for all $z \in C$ *and* $\bar{z} \in C$ *with* $z \neq \bar{z}$*, then* F *is* strictly monotone *on* C*.*

Definition 1.3.12 (Strongly Monotone) *If for some scalar* $\mu > 0$*,*

$$\langle z - \bar{z}, F(z) - F(\bar{z}) \rangle \geq \mu \left\| z - \bar{z} \right\|^2$$

for all $z \in C$ *and* $\bar{z} \in C$ *then* F *is* strongly monotone with modulus μ *on* C*.*

Using these definitions, we have the following existence results for nonlinear complementarity problems.

Theorem 1.3.13 ([65]) *Let* $F : \mathbf{R}^n \to \mathbf{R}^n$ *be continuous on* $\mathbf{R}^n$*. Then the following statements hold:*

(a) If F *is monotone on* $[\ell, u]$*, then MCP(*F, ℓ, u*) has a convex solution set.*

(b) If F *is strictly monotone on* $[\ell, u]$*, then MCP(*F, ℓ, u*) has at most one solution.*

(c) If F *is strongly monotone on* $[\ell, u]$*, then MCP(*F, ℓ, u*) has a unique solution.*

Note that monotone and strictly monotone problems might have *no* solution.

Since we are assuming that F is continuously differentiable on $\mathbf{R}^n$, we can characterize monotonicity using the Jacobian of F.

Theorem 1.3.14 ([96]) *The following hold:*

(a) F is monotone on C if and only if $F'(x)$ is positive semidefinite for all $x \in C$.

(b) If F is strictly monotone on C then $F'(x)$ is positive definite for all $x \in C$.

(c) F is strongly monotone with modulus μ on C if and only if for all $\bar{x} \in C$,

$$x^T F'(\bar{x}) x \geq \mu x^T x$$

for all $x \in C$.

The P-matrix property for linear complementarity problems can be extended to nonlinear mixed complementarity problems using the notion of a P-function.

Definition 1.3.15 (P-function) *If*

$$\max_{1 \leq i \leq n} (F_i(x) - F_i(y))(x_i - y_i) > 0$$

for all $(x, y) \in [\ell, u]$ with $x \neq y$, then F is called a P-function.

Definition 1.3.16 (Uniform P-function) *If there exists a scalar $\mu > 0$ such that*

$$\max_{1 \leq i \leq n} (F_i(x) - F_i(y))(x_i - y_i) \geq \mu \left\| x - y \right\|_2^2$$

for all $(x, y) \in [\ell, u]$ then F is called a uniform P-function.

Theorem 1.3.17 ([65]) *The following hold:*

(a) If F is a P-function then MCP(F, ℓ, u) has at most one solution.

(b) If F is a uniform P-function then MCP(F, ℓ, u) has a unique solution.

Note that a strictly monotone function is a P-function, while a strongly monotone function is a uniform P-function.

Additional conditions using degree theory for the existence of a solution can be found in [61].

1.4 Algorithms and Regularity

We now have conditions that guarantee the existence of a solution to a mixed complementarity problem. This section discusses some algorithms for computing a solution.

Newton's method, perhaps the most famous solution technique, has been extensively used in practice to solve nonlinear systems of equations. The essential idea is to form a linear approximation to the nonlinear function. Solving the resulting linear system of equations produces the next iterate.

$$x^{k+1} = x^k - F'(x^k)^{-1} F(x^k) \tag{4}$$

Theorem 1.4.1 ([96]) *Let $F : \mathbf{R}^n \to \mathbf{R}^n$ be continuously differentiable and x^* be such that $F(x^*) = 0$ and $F'(x^*)$ is nonsingular. Then the following hold:*

(a) The iteration in (4) is well-defined and converges to x^* superlinearly in a neighborhood of x^*.

(b) If in addition F has locally Lipschitz derivatives at x^*, then the rate of convergence is quadratic.

Nonsingularity of the Jacobian is a regularity condition used in conjunction with Newton's method to show the convergence and rate of convergence results. We will be dealing with generalizations of Newton's method for nonsmooth systems of equations and will therefore define generalized regularity conditions.

We consider two algorithms based on Newton's method in this thesis. The first method is the nonsmooth Newton method developed in [71, 72, 73] for generalized equations [105]. Each iteration solves a (linear) generalized equation for z^{k+1}:

$$0 \in F(z^k) + F'(z^k)(z^{k+1} - z^k) + N_{[\ell,u]}(z^{k+1}). \tag{5}$$

This system is a linear complementarity problem which can be solved using Lemke's method [80, 82].

To show the local convergence of this method, we need to use a generalization of the nonsingularity of the Jacobian matrix. The generalization used is the notion of strong regularity at a solution. In order to define strong regularity, we introduce a partitioning of the set $I = \{1, \cdots, n\}$ given z^*, a solution to MCP(F, ℓ, u).

$$\begin{array}{rcl} \alpha(z^*) & := & \{i \,|\, l_i < z_i^* < u_i, F_i(z^*) = 0\}, \\ \beta(z^*) & := & \{i \,|\, z_i^* \in \{l_i, u_i\}, F_i(z^*) = 0\}, \\ \gamma(z^*) & := & \{i \,|\, z_i^* \in \{l_i, u_i\}, F_i(z^*) \neq 0\}. \end{array} \tag{6}$$

Definition 1.4.2 (Strong Regularity [106, 33]) *Let z^* be a solution to the complementarity problem MCP(F, ℓ, u). Then MCP(F, ℓ, u) is* strongly regular *at z^* if the submatrix $F'(z^*)_{\alpha\alpha}$ is nonsingular and the Schur-complement*

$$F'(x^*)_{\alpha\cup\beta,\alpha\cup\beta}(z^*)/F'(z^*)_{\alpha\alpha} := F'(z^*)_{\beta\beta} - F'(z^*)_{\beta\alpha}F'(z^*)_{\alpha\alpha}^{-1}F'(z^*)_{\alpha\beta}$$

is a P-matrix, where the index sets α and β are defined in (6).

If MCP(F, ℓ, u) is strongly regular at a solution z^* and $F'(z^*)$ satisfies a Lipschitz continuity assumption, then the Newton-like method in (5) is locally quadratic convergent. However, it is not globally convergent. An ad-hoc linesearch was proposed for this method [86, 87, 88] in an attempt to obtain global convergence and implemented in MILES [111].

A similar method uses the normal map reformulation of the complementarity problem [108]. Global convergence of this method is achieved by damping the Newton method with a pathsearch [103] on the two-norm of the normal map squared as a merit function. The resulting method is the basis for the implementation in PATH [25, 27, 28]. A nonmonotone pathsearch [63, 64, 38] and a crashing method [29] are used in the code to improve robustness and efficiency.

An alternative algorithm is based on the semismooth reformulation. If $G : \mathbf{R}^n \to \mathbf{R}^n$ is a semismooth function, we can define the Newton iteration as

$$x^{k+1} = x^k - (H^k)^{-1} G(x^k), \tag{7}$$

where H^k is any element in $\partial_B G(x^k)$. The generalization of the nonsingularity of the Jacobian used with this method is the notion of BD-regularity.

Definition 1.4.3 (BD-Regularity) *A semismooth function G is BD-regular at a point x if all the elements in $\partial_B G(x)$ are nonsingular.*

Theorem 1.4.4 ([23]) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be a semismooth function and x^* be such that $G(x^*) = 0$ and G is BD-regular at x^*. Then the following hold:*

(a) The iteration in (7) is well-defined and converges to x^* superlinearly in a neighborhood of x^*.

(b) If in addition G is directionally differentiable in a neighborhood of x^* and strongly semismooth at x^*, then the rate of convergence is quadratic.

We can then apply this method to solve the system $\Phi(x) = 0$. To obtain global convergence, the basic method is damped with an Armijo [2] linesearch on the two-norm of $\Phi(\cdot)$.

Note that strong regularity and BD-regularity are related. As proven in Chapter 6, if F is strongly regular at x^* then Φ is BD-regular at x^*.

1.5 Summary

We have presented a precise definition of the mixed complementarity problem and have investigated several reformulations as nonsmooth systems of equations along with basic existence results. We also mentioned two algorithms for solving the complementarity problem that are globally and locally fast convergent under suitable conditions.

The remainder of this thesis concentrates on algorithms and environments for solving complementarity problems. Many environments are available to practitioner for expressing complementarity problems. Modeling languages [9, 52], such as AMPL [53] and GAMS [11], provide natural facilities for representing mathematical programs. Recently, improvements to these packages for modeling complementarity relationships [31, 35] have been made. Other tools, such as MATLAB [89] and NEOS [22, 40], can also be used to communicate a complementarity problem to a solver. Chapter 2 surveys the mechanisms available in these packages for conveying complementarity problems. In particular, the new links to the MATLAB and NEOS tools that have been developed are documented.

Support for these packages is provided in PATH 4.x and SEMI through the customizable solver interface specified in Chapter 3. The main design feature is the abstraction of core components from the algorithm with implementations tailored to a particular environment supplied either at compile or run time. The components include mechanisms for specifying the function and Jacobian evaluations required in the complementarity problem definition. Other components deal with memory allocation and interrupt handling. Individuals who want to embed the PATH 4.x or SEMI codes into larger applications will benefit from the documentation in Chapter 3.

New techniques for preprocessing complementarity problems are documented in Chapter 4. The benefits of preprocessing have long been known to the linear [1, 10] and mixed integer [115] programming communities, where preprocessing techniques typically are used to reduce the size and complexity of a model prior to solving it. For example, wasted computation is avoided when the preprocessor determines that a model has no solution. In complementarity, we use the variational inequality representation and existence and uniqueness results to eliminate variables from a model. An implementation of the preprocessing techniques developed in Chapter 4 are contained in PATH 4.x and SEMI, which are the first codes for solving complementarity problems with preprocessing technology.

Developing a practical model of a complex situation is a difficult task in which an approximate representation is initially constructed and then iteratively refined until an accurate formulation is obtained. During the intermediate stages, the models generated have a tendency to be ill-defined, poorly conditioned, and/or singular. Information generated by a solver can help the modeler to detect these problems, quickly locate the source, and make appropriate modifications to the model. The diagnostic information reported by PATH 4.x and SEMI is the subject of Chapter 5.

The remaining chapters of the thesis concentrate on the PATH 4.x and SEMI algorithms and implementations. PATH 4.x offers improved reliability over prior versions of the code and has been able to find solutions to previously unsolvable models. PATH is based on a nonsmooth Newton method for the normal map [108, 103]. A key problem encountered by this method is when a solution to the linear complementarity subproblem cannot be found. PATH uses heuristics in this case in an attempt to overcome the difficulty. New theory is developed in Chapter 6 for a globalization scheme based on the Fischer-Burmeister merit function [50] as modified in [5] for mixed complementarity problems and incorporated into PATH 4.x. Using this new theory, we can resort to a projected gradient step on the (differentiable) merit function when a solution to the linear complementarity problem cannot be found. The resulting algorithm is well-defined for arbitrary complementarity problems, has strong global convergence properties, and is locally fast convergent under a strong regularity assumption. Enhancements made to the linear complementarity solver, nonmonotone linesearch, and restart strategies are also partially responsible for the improved robustness of PATH 4.x. These enhancements are documented in Chapter 6.

The main computation per iteration in PATH 4.x is to solve a linear complementarity problem, which can be an expensive operation. A seemingly more attractive approach is to use an algorithm that only solves a linear system of equations at each iteration. SEMI is based on the semismooth algorithm [24] and has this property. The method uses the Fischer-Burmeister [50] and penalized Fischer-Burmeister [13] functions in the implementation and can use iterative methods, such as LSQR [97], GMRES [114, 119], and QMR [54], to solve the linear system of equations generated at each iteration. The iterative methods can be used in the code to solve very large models. A complete description of the new SEMI code is contained in Chapter 7.

The theme of this thesis is "enabling" technologies in complementarity: environments enable practitioners to easily specify models; sophisticated codes enable them to solve the models; and theory assures them that the algorithm is well-defined and efficiently processes models.

Chapter 2

Environments

Environments are used by practitioners to easily specify large, complex models. Modeling languages [9, 52], such as AMPL [53] and GAMS [11], provide natural facilities for representing complementarity relationships [31, 35]. Additional environments, such as MATLAB [89] and NEOS [22, 40], are also available for expressing complementarity problems.

This chapter looks at complementarity from a practitioner's perspective. We first develop two example models illustrating complementarity relationships in Section 2.1. These examples are then used in the subsequent survey of the AMPL, GAMS, MATLAB, and NEOS environments for expressing complementarity problems and communicating them to solvers. The discussion of each environment includes information on the input format, output generated, and unique features of the environment.

2.1 Example Applications

Problems fitting into the complementarity framework occur in a wide variety of disciplines. In this section, we introduce the nonlinear complementarity problem from an economics perspective using the optimality conditions for a transportation model. The more general mixed complementarity problem is motivated using a Walrasian equilibrium [3] model.

2.1.1 Transportation Model

The transportation model is a linear program where demand for a single good must be satisfied by suppliers at a minimal transportation cost. The underlying transportation network is given as a set $\mathcal{A} \subseteq \mathcal{S} \times \mathcal{D}$ of arcs from suppliers, $\mathcal{S}$, to demand centers, $\mathcal{D}$. The problems variables are the amounts $x_{i,j}$ to be shipped over each arc $(i,j) \in \mathcal{A}$. A linear program to find an optimal shipment schedule can be written mathematically as

$$\begin{array}{rl} \min_{x \geq 0} & \sum_{(i,j) \in \mathcal{A}} c_{i,j} x_{i,j} \\ \text{subject to} & \sum_{j:(i,j) \in \mathcal{A}} x_{i,j} \leq s_i, \ \forall i \in \mathcal{S} \\ & \sum_{i:(i,j) \in \mathcal{A}} x_{i,j} \geq d_j, \ \forall j \in \mathcal{D} \end{array} \tag{8}$$

where $c_{i,j}$ is the unit shipment cost on the arc (i,j) connecting supplier i to demand center j, s_i is the amount of supply available at i, and d_j is the demand at j.

The derivation of a complementarity problem for the transportation model begins by associating with each constraint a multiplier, alternatively termed a dual variable or shadow price. These multipliers represent the marginal price on changes to the corresponding constraint. Labeling the prices on the supply constraint p^s and those on the demand constraint p^d, we then intuitively have at each supply node i

$$0 \leq p_i^s, \ s_i \geq \sum_{j:(i,j) \in \mathcal{A}} x_{i,j}.$$

Consider the case when $s_i > \sum_{j:(i,j) \in \mathcal{A}} x_{i,j}$, that is there is excess supply at i. Then, in a competitive marketplace, no rational person is willing to pay for more supply at node i; it is already over-supplied. Therefore, $p_i^s = 0$. Alternatively, when $s_i = \sum_{j:(i,j) \in \mathcal{A}} x_{i,j}$, that is node i clears, we might be willing to pay for additional supply of the good. Therefore, $p_i^s \geq 0$. We write these two conditions succinctly as:

$$0 \leq p_i^s \quad \perp \quad s_i \geq \sum_{j:(i,j) \in \mathcal{A}} x_{i,j}, \quad \forall i$$

where the $\perp$ notation is understood to mean that at least one of the adjacent inequalities must be satisfied as an equality. For example, either $0 = p_i^s$, the first case, or $s_i = \sum_{j:(i,j)\in\mathcal{A}} x_{i,j}$, the second case. This notation is used to represent the complementarity relationship and means that p_i^s and $s_i - \sum_{j:(i,j)\in\mathcal{A}} x_{i,j}$ are orthogonal.

Similarly, at each node j, the demand must be satisfied in any feasible solution, that is

$$\sum_{i:(i,j)\in\mathcal{A}} x_{i,j} \geq d_j.$$

Furthermore, the model assumes all prices are nonnegative, $0 \leq p_j^d$. If there is too much of the commodity supplied, $\sum_{i:(i,j)\in\mathcal{A}} x_{i,j} > d_j$, then, in a competitive marketplace, the price p_j^d will be driven down to 0. Summing these relationships gives the following complementarity condition:

$$0 \leq p_j^d \quad \perp \quad \textstyle\sum_{i:(i,j)\in\mathcal{A}} x_{i,j} \geq d_j, \quad \forall j.$$

The supply price at i plus the transportation cost $c_{i,j}$ from i to j must exceed the market price at j. That is $p_i^s + c_{i,j} \geq p_j^d$. Otherwise, competitors would enter the market and increase the available supply, driving down the market price. This chain would repeat until the inequality is satisfied. Furthermore, if the cost of delivery strictly exceeds the market price, that is $p_i^s + c_{i,j} > p_j^d$, then nothing is shipped from i to j because doing so would incur a loss. In this case, $x_{i,j} = 0$. Therefore,

$$0 \leq x_{i,j} \quad \perp \quad p_i^s + c_{i,j} \geq p_j^d, \quad \forall (i,j) \in \mathcal{A}.$$

Collecting all of the above relationships, we have the linear complementarity problem

$$\begin{array}{lll} 0 \leq p_i^s & \perp & s_i \geq \sum_{j:(i,j)\in\mathcal{A}} x_{i,j}, \quad \forall i \\ 0 \leq p_i^d & \perp & \sum_{i:(i,j)\in\mathcal{A}} x_{i,j} \geq d_j, \quad \forall j \\ 0 \leq x_{i,j} & \perp & p_i^s + c_{i,j} \geq p_j^d, \qquad \forall (i,j) \in \mathcal{A} \end{array} \tag{9}$$

which is easily recognized as the complementary slackness conditions for the linear program (8). For linear programs the complementary slackness conditions are both necessary and sufficient for x to be an optimal solution to (8). Furthermore, the conditions (9) are also the necessary and sufficient optimality conditions for a related problem in the variables (p^s, p^d)

$$\begin{array}{ll} \max_{p^s, p^d \geq 0} & \sum_{j \in \mathcal{D}} d_j p_j^d - \sum_{i \in \mathcal{S}} s_i p_i^s \\ \text{subject to} & c_{i,j} \geq p_j^d - p_i^s, \ \forall (i,j) \in \mathcal{A} \end{array}$$

termed the dual linear program, hence the nomenclature "dual variables".

Looking at these conditions a bit more closely we can gain further insight into complementarity problems. A solution of (9) tells the arcs used to transport goods. A priori we do not need to specify the arcs to use, the solution itself indicates them. This property represents the key contribution of a complementarity problem over a system of equations. If we know what arcs to send flow down, we can just solve a simple system of linear equations. However, the key to the modeling power of complementarity is that it chooses which of the inequalities in (9) to satisfy as equations. In economics we can use this property to generate a model with different regimes and let the solution determine which ones are active. Using the transportation model, we could specify two alternate routes from a supplier to a demand center: one using a four-lane interstate highway and the other a two-lane state highway. A regime shift could occur when the state highway becomes active because of congestion on the interstate highway, which slows traffic down and consequently drives up the transportation cost. Congestion could be modeled, for example by making the cost of traveling along a road a function of the usage:

$$0 \leq x_{i,j} \quad \perp \quad p_i^s + c_{i,j}(x) \geq p_j^d, \quad \forall (i,j) \in \mathcal{A}.$$

for some cost function, $c_{i,j}(\cdot)$.

While many interior point methods for linear programming exploit this complementarity framework (so-called primal-dual methods [123]), the real power of this modeling format is the new problem instances it enables a modeler to create. We now show some examples of how to extend the simple model (9) to investigate other issues and facets of the problem.

Demand in (9) is independent of the prices p. Since the prices p are variables in the complementarity problem (9), we can easily replace the constant demand d with a function $d(p)$. This function can be used to more accurately model demand which typically is higher at a low price than at a high price. Clearly, any function of p can be added to the model. For example, a linear demand function could be expressed using

$$\sum_{i:(i,j)\in\mathcal{A}} x_{i,j} \geq d_j(1 - p_j^d),\ \forall j.$$

Note that the demand is rather strange if p_j^d exceeds 1. Other more reasonable examples for $d(p)$ can be derived. For example, if we assume the demand for the commodity is isoelastic [112], then the demand for the commodity is

$$d_j(p) := \frac{d_j}{(p_j^d)^{\alpha_j}}$$

for fixed d_j and $\alpha_j > 0$. Note that the resulting complementarity problem becomes nonlinear in the variables p and that the function is undefined when $p_j^d = 0$.

Another feature that can be added to this model are tariffs or taxes. In the case where a tax is applied at the supply point, the third general inequality in (9) is replaced by

$$p_i^s(1 + t_i) + c_{i,j} \geq p_j^d,\ \forall (i,j) \in \mathcal{A}.$$

Details about complementarity problems for more general transportation models can be found in [30, 39].

An important observation to make is that with either of these modifications, the resulting complementarity problem is no longer the optimality conditions for a linear program. In many cases, there is no optimization problem corresponding to the complementarity conditions.

We now abstract from the particular example to describe more carefully the complementarity problem in its mathematical form. All the above examples can be cast as nonlinear complementarity problems defined as follows:

(Nonlinear Complementarity Problem [19]) Let $F : \mathbf{R}^n \to \mathbf{R}^n$ be a continuously differentiable function. The nonlinear complementarity problem is to find $z \in \mathbf{R}^n$ such that

$$0 \leq z \ \perp \ F(z) \geq 0.$$

Recall that the $\perp$ sign signifies that one of the inequalities is satisfied as an equality, so that componentwise $z_i F_i(z) = 0$. We frequently refer to this property as z_i is "complementary" to F_i. A special case of the nonlinear complementarity problem that has received much attention is when F is a linear function, the linear complementarity problem [21].

2.1.2 Walrasian Equilibrium

A Walrasian equilibrium [3] can also be formulated as a complementarity problem. In this case, we want to find a price $p \in \mathbf{R}^m$ and an activity level $y \in \mathbf{R}^n$ such that

$$\begin{array}{rcl} 0 \leq y \ \perp \ L(p) & := & -A^T p \geq 0 \\ 0 \leq p \ \perp \ S(p,y) & := & b + Ay - d(p) \geq 0. \end{array} \tag{10}$$

Here, $S(p, y)$ represents the excess supply function, and $L(p)$ represents the loss function. The first complementarity relationship chooses the activities y_i to run (i.e. only those that do not make a loss). The second set of inequalities state that the price of a commodity can only be positive if there is no excess supply. These conditions indeed correspond to the standard exposition of Walras' law [120] which states that supply equals demand if we assume all prices p will be positive at a solution. Formulations of equilibria as systems of equations do not allow the model to choose the activities present, but typically make an a priori assumption on this matter.

Many large scale models of this nature have been developed. An interested modeler could, for example, see how a large scale complementarity problem was used to quantify the effects of the Uruguay round of talks [67].

In many modeling situations, a key tool for clarification is the use of intermediate variables. As an example, the modeler may wish to define a variable corresponding to the demand function $d(p)$ in the Walrasian equilibrium (10). Using the variables d_p to store the value of the demand function referred to in the excess supply equation, we have the following complementarity problem:

$$\begin{array}{rcl} 0 \leq y \quad \perp \quad L(p) & := & -A^T p \geq 0 \\ 0 \leq p \quad \perp \quad S(p, y) & := & b + Ay - d_p \geq 0 \\ & & d_p = d(p). \end{array} \tag{11}$$

The model now contains a mixture of equations and complementarity constraints. Since constructs like the above are prevalent in many practical models, a more general definition of complementarity is needed.

A mixed complementarity problem is specified by three pieces of data, namely the lower bounds ℓ, the upper bounds u, and a function F.

(Mixed Complementarity Problem) Given lower and upper bounds

$$\ell \in \{\mathbf{R} \cup \{-\infty\}\}^n$$
$$u \in \{\mathbf{R} \cup \{+\infty\}\}^n$$

and a continuously differentiable function $F : \mathbf{R}^n \to \mathbf{R}^n$, the mixed complementarity problem is to find $z \in \mathbf{R}^n$ such that *precisely* one of the following holds for each $i \in \{1, \ldots, n\}$:

$$\begin{aligned} F_i(z) = 0 &\quad \text{and} \quad \ell_i \leq z_i \leq u_i \\ F_i(z) > 0 &\quad \text{and} \quad z_i = \ell_i \\ F_i(z) < 0 &\quad \text{and} \quad z_i = u_i. \end{aligned}$$

These relationships define the general complementarity problem needed for (11). We will use the notation

$$\ell \leq z \leq u \quad \perp \quad F(z)$$

to denote the mixed complementarity relationship. Both the nonlinear complementarity problem found in (9) and (10) and nonlinear systems of equations are special cases of mixed complementarity problems. Nonlinear complementarity problems have $\ell = \{0\}^n$ and $u = \{\infty\}^n$ and nonlinear equations have $\ell = \{-\infty\}^n$ and $u = \{\infty\}^n$. A point misunderstood by many experienced modelers is that *the variable bounds determine the relationships satisfied by the function* F. Therefore, modifications to the lower and upper bounds on the variables can drastically change the problem under consideration.

An advantage of the mixed complementarity formulation described above is the pairing between "fixed" variables (ones with equal upper and lower bounds) and a component of F. If a variable z_i is fixed, then $F_i(z)$ is unrestricted since precisely one of the three conditions in the MCP definition automatically holds when $z_i = \ell_i = u_i$. Thus if a variable is fixed, the paired equation can be completely dropped from the model. This convenient trick can be used to remove particular constraints from a model.

In many cases, equations (which are complementary to free variables) can be written in any convenient way since they are internally "substituted out" of the model. In particular, for defining equations, such as those presented in (11), the choice appears to be arbitrary. However, underlying model monotonicity is important. For example, if we have the linear program

$$\begin{array}{rl} \min_x & c^T x \\ \text{subject to} & Ax = b,\ x \geq 0 \end{array}$$

we can write the first order optimality conditions as either

$$\begin{array}{rcl} 0 \leq x & \perp & -A^T\mu + c \geq 0 \\ \mu \text{ free} & \perp & Ax - b \end{array}$$

or, equivalently,

$$\begin{array}{rcl} 0 \leq x & \perp & -A^T\mu + c \geq 0 \\ \mu \text{ free} & \perp & b - Ax \end{array}$$

because we have an equation. The former is a linear (mixed) complementarity problem with a positive semidefinite matrix, while the latter is indefinite. If we need to add a diagonal perturbation to the problem for numerical reasons, the former system becomes positive definite, while the later remains indefinite and unlikely to be solvable.

Modelers typically add bounds to their variables when attempting to solve nonlinear problems in order to restrict the domain of interest. For example, many square nonlinear systems are formulated as

$$\begin{array}{c} F(z) = 0 \\ \ell \leq z \leq u \end{array}$$

where typically, the bounds on z are inactive at the solution. This is *not* a mixed complementarity problem, but is an example of a "constrained nonlinear system." It is important to note the distinction between complementarity problems and constrained systems. The complementarity problem uses the bounds to infer relationships on the function F. If a finite bound is active at a solution, the corresponding component of F is only constrained to be nonnegative or nonpositive in the complementarity problem, whereas in the constrained system F must be zero. Thus there may be many solutions of the complementarity problem that do not satisfy $F(z) = 0$. Only if z^* is a solution of the complementarity problem with $\ell < z^* < u$ is it guaranteed that $F(z^*) = 0$.

Simple bounds on the variables are a convenient modeling tool that translates into efficient mathematical programming tools. For example, specialized codes exist for the bound constrained optimization problem

$$\begin{array}{rl} \min & f(x) \\ \text{subject to} & \ell \leq x \leq u. \end{array}$$

The first order optimality conditions of this problem are precisely $\mathrm{MCP}(\nabla f(x), \ell, u)$. We can easily reason about this condition in the one dimensional setting. If we are at an unconstrained minimizer, then $\nabla f(x) = 0$. Otherwise, if x is a local minimizer at its lower bound, then the function must be increasing as x increases, so $\nabla f(x) \geq 0$. Alternatively, if x is a local minimizer at its upper bound, then the function must be increasing as x decreases, so that $\nabla f(x) \leq 0$. Thus,

$$\ell \leq x \leq u \quad \perp \quad \nabla f(x).$$

The complementarity framework allows such models to be easily and efficiently processed.

Upper bounds can also be used to extend the utility of existing models. For example, in the Walrasian equilibrium it may be necessary to have an upper bound on the activity level y. In this case, we simply add an upper bound to y. The model then becomes:

$$\begin{array}{rcrcl} 0 \le y \le 10 & \perp & L(p) & := & -A^T p \\ 0 \le p & \perp & S(p,y) & := & b + Ay - d(p) \end{array} \tag{12}$$

where we use the $\perp$ notation in the generalized sense of the mixed complementarity problem. *Recall that the bounds on the variables completely determine the relationship satisfied by the equations.* We can interpret the relationships that the above change generates as follows. If $y_j = 0$, the loss function can be positive since we are not producing in the jth sector. If y_j is strictly between its bounds, then the loss function must be zero by complementarity; this is the competitive assumption. However, if y_j is at its upper bound, then the loss function can be negative. Of course, if the market does not allow free entry, some firms may operate at a profit (negative loss).

2.2 AMPL

AMPL [53] is a general purpose language for modeling mathematical programs including linear, nonlinear, and integer programs. The syntax for associating variables and functions in complementarity relationships uses the **complements** keyword [35] in constraint declarations. An implementation of the transportation model from Section 2.1.1 can be found in Figure 1; the actual data for the model is assumed to be given in the file `transmcp.dat`. Example data is found in Figure 2.

Each condition declared using the **complements** keyword must specify exactly two inequalities. In Figure 1, these correspond to one inequality on each variable and function which precisely define the complementarity condition. Any ordering of the inequalities can be used in the pairing. For example, an alternative declaration would be to reorder the `supply` constraint as

```
set S;                        # Suppliers
set D;                        # Markets

param cap {S};                # Capacity of supplier
param dem {D};                # Demand at market
param tra {S,D};              # Transportation cost

var x{S,D};                   # Shipment quantity
var p_s{S};                   # Price at plant
var p_d{D};                   # Price at market

subj to
supply {i in S}:              # Supply limit
    0 <= p_s[i] complements cap[i] >= (sum {j in D} x[i,j]);

subj to
demand {j in D}:              # Demand satisfaction
    0 <= p_d[j] complements (sum {i in S} x[i,j]) >= dem[j];

subj to
rational {i in S, j in D}: # Rational marketplace
    0 <= x[i,j] complements p_s[i] + tra[i,j] >= p_d[j];
```

Figure 1: Transportation model in AMPL, `transmcp.mod`

```
set S := seattle, san-diego;
set D := new-york, chicago, topeka;

param:     cap :=
seattle    325
san-diego  575;

param:     dem :=
new-york   325
chicago    300
topeka     275;

param tra: new-york  chicago   topeka :=
seattle    0.225     0.153     0.162
san-diego  0.225     0.162     0.126;
```

Figure 2: Transportation data in AMPL, `transmcp.dat`

```
subj to
supply {i in S}:           # Supply limit
    cap[i] >= (sum {j in D} x[i,j]) complements 0 <= p_s[i];
```

Recall that mixed complementarity conditions were defined by lower and upper bounds on the variables and an unrestricted function. AMPL can be used to write such a representation by placing both inequalities on the variables.

```
subj to supply {i in S}:
    0 <= p_s[i] <= Infinity complements
        cap[i] - (sum {j in D} x[i,j]);
```

This formulation is particularly useful if a user wants to modify the bounds during an iterative process. If we declare parameters, `p_l` and `p_u`, for the lower and upper bounds on the prices, then we can write the complementarity constraint as:

```
subj to supply {i in S}:
    p_l[i] <= p_s[i] <= p_u[i] complements
        cap[i] - (sum {j in D} x[i,j]);
```

The meaning of the constraint is completely determined by the bound parameters which can be modified at run time. Note that the function must be oriented correctly to preserve the meaning of the complementarity condition. Thus, the following declaration is incorrect:

```
subj to supply {i in S}:
    p_l[i] <= p_s[i] <= p_u[i] complements
        (sum {j in D} x[i,j]) - cap[i];
```

To solve the `transmcp` problem with AMPL, the following statements are issued from the AMPL prompt:

```
ampl: model transmcp.mod;
ampl: data transmcp.dat;
ampl: solve;
```

The first two statements load the model and problem data respectively, while the last passes an internal representation of the problem to a solver. The solver used can be changed with the statement:

```
ampl: option solver path.ampl;
```

where `path.ampl` can be the name of any executable for solving complementarity problems using AMPL input. Here, `path.ampl` refers to the AMPL version of the PATH 4.x code. To add solver specific options, we write the statement:

```
ampl: option path_options "logfile foo.log";
```

The options list and name of the options string, `path_options`, is specific to the algorithm. For example, the SEMI code is called using:

```
ampl: option solver semi.ampl;
ampl: option semi_options "logfile foo.log";
```

Table 1: AMPL Specific Options

Option	Type	Explanation
logfile	string	Specify file where log will be written
optfile	string	Specify file containing solver options
statusfile	string	Specify file where detailed status information will be written
version	boolean	Controls printing of version information
sideineq	integer	Control the handling of side constraints

PATH 4.x and SEMI use a common set of options for AMPL which are detailed in Table 1. Note that an `optfile` can be set which contains a listing of algorithm specific options and the corresponding values to be set.

After the solve statement is encountered, AMPL will generate the model and pass it to the algorithm for solution. Assuming that PATH 4.x is used, the following output for the `transmcp` model will be displayed on exit:

```
Path v4.4a: Solution found.
15 iterations (1 for crash); 18 pivots.
16 function, 16 gradient evaluations.
```

The message can be accessed from the `solve_message` variable within AMPL, and indicates the solution status, "Solution found", and number of iterations taken. The total number of function and gradient evaluations performed are also displayed. Furthermore, a result code is passed back to AMPL that can be checked using the `solve_result_num` variable. A complete listing of the result codes and corresponding messages returned by PATH 4.x and SEMI are listed in Table 2. The result codes are broken into the categories found in Table 3 where the category string for the return code can be accessed with the `solve_result` variable within AMPL.

Once a solution has been found, we are returned to the AMPL prompt. We can then investigate the values for the variables using the normal AMPL syntax. For example, to display the variable values for transportation model, we can issue the command:

Table 2: AMPL Return Codes

Code	String
0	Solution found
200	Infeasible
201	Inconsistent bounds
400	Major iteration limit
401	Cumulative iteration limit
402	Time limit
403	User interrupt
500	Unexpected return code
501	Solver error
503	Domain error
504	Non-square system
505	Model contains side constraints
510	Not enough progress

Table 3: AMPL Return Code Meanings

Interval	String	Meaning
0 – 99	solved	The model has been solved
100–199	solved?	A local solution has been found
200–299	infeasible	The model is has no solution
300–399	unbounded	The objective value is unbounded
400–499	limit	A resource limit was reached by the code
500–599	failure	The algorithm failed to find a solution

```
ampl: display p_s, p_d, x;
:              p_s       p_d      :=
chicago          .      86.9571
new-york         .      87.0291
san-diego   86.8041        .
seattle     86.8041        .
topeka           .      86.9301
;

x :=
san-diego chicago      0
san-diego new-york   300
san-diego topeka     275
seattle   chicago    300
seattle   new-york    25
seattle   topeka       0
;
```

In cases where the algorithm fails to find a solution, it might be useful to specify a `logfile` for the problem and investigate the contents to gain information about the reasons for failure.

The Walrasian equilibrium in Section 2.1.2 can be modeled similarly as presented in Figure 3, where we have used an equation to define the demand function, `d_p`. Equations count as both $\leq$ and $\geq$ inequalities. Thus, the corresponding variable is unrestricted. Note that the demand function is undefined when the prices are zero. If a starting point is not supplied, AMPL uses the zero vector projected onto the simple variable bounds. Therefore, we must initialize the prices to positive values, as done in the variable declaration of Figure 3 in order to start at a well-defined point. Algorithms will typically generate an error if a user supplies a starting point (either directly or indirectly) where the function is undefined. Furthermore, *a good starting point can help an algorithm to rapidly find a solution, especially for nonlinear models.*

In many cases, it is not significant to match free variables explicitly to equations.

```
set I;
set J;

param A {I,J};
param b {I};
param c {I};

var p {I} := 1;   # Price of commodity
var y {J};        # Activity level
var d_p {I};      # Demand

subj to
S {i in I}:       # Excess supply function
    0 <= p[i] complements
        b[i] + (sum {j in J} A[i,j]*y[j]) - d_p[i] >= 0;

subj to
L {j in J}:       # Loss function
    0 <= y[j] complements - (sum {i in I} p[i]*A[i,j]) >= 0;

subj to
D {i in I}:       # Demand function
    d_p[i]    complements
        d_p[i] = c[i]*(sum {k in I} b[k]*p[k]) / p[i];
```

Figure 3: Walrasian equilibrium as an MCP in AMPL, `walrasian.mod`

Therefore, when we have free variables matched to equations, we *can* remove the **complements** from the declaration. For example, we can write the demand function in Figure 3 as:

```
subj to
D {i in I}:                    # Demand function
    d_p(i) = c[i]*(sum {k in I} b[k]*p[k]) / p[i];
```

The solver interface will match any unmatched, unrestricted variables to these equality constraints. The number of free variables and unmatched equations must be the same in order to have a square complementarity problem. Note that this extension allows existing models consisting of a square system of nonlinear equations to be easily recast as a complementarity problem - the model definition is unchanged.

However, equations should be explicitly matched to free variables in order to preserve problem structure, even though not explicitly required. Matrix properties, such as positive semidefiniteness, could be lost with the arbitrary matching performed by the solver interface, leading to poor performance (or even failure) by an algorithm. For example, when constructing the first order optimality conditions for a linear program, the matching of equations to multipliers should be done to preserve problem structure.

Upper bounds can be imposed on the activity levels in the Walrasian equilibrium by declaring the loss function as

```
subj to
L {j in J};                    # Loss function
    0 <= y(j) <= 10 complements -(sum {i in I} p[i]*A[i,j]);
```

The bounds on the variables completely determine the complementarity problem. Hence, for bounded variables, we do not know beforehand if the corresponding function will be satisfied as an equation, less than inequality or greater than inequality, since this determination depends on the values of the solution variables. Therefore, the function is unrestricted.

2.2.1 Generalized Declaration

The notation used in AMPL allows generalized complementarity conditions to be written between two functions. For example we can declare the following:

```
subj to
func {i in I}:
    0 <= (sum {j in J} A[i,j]*x[j]) - a[i] complements
         (sum {j in I} B[i,j]*x[j]) - b[i] >= 0;
```

Internal to the solver interface, the condition is reformulated into a standard mixed complementarity problem by adding variables and constraints. One equivalent definition is:

```
subj to
s_func {i in I}:
         x[i] complements y[i] = (sum {j in I} A[i,j]*x[j]) - a[i];

subj to
func {i in I}:
    0 <= y[i] complements (sum {j in I} B[i,j]*x[j]) - b[i] >= 0;
```

Functions can also be doubly bounded as in the following definition.

```
subj to
func {i in I}:
    x[i] complements
         l[i] <= (sum {j in I} A[i,j]*x[j]) - a[i] <= u[i];
```

A similar reformulation is performed by the interface:

```
subj to
s_func {i in I}:
    x[i] complements y[i] = (sum {j in I} A[i,j]*x[j]) - a[i];
```

```
subj to
func {i in I}:
    l[i] <= y[i] <= u[i] complements x[i];
```

2.2.2 Side Constraints

AMPL provides extensions for complementarity problems with side constraints. For example, variables can be declared with bounds on them. However, the complementarity relationship is defined based upon the bounds present in the constraint declaration. For example, if we define a complementarity problem as:

```
set I;

param A {I,I};
param b {I};

var x {I} >= 1, <= 10;

subj to
func {i in I}:
    0 <= x[i] complements (sum {j in I} A[i,j]*x[j]) - a[i] >= 0;
```

then the x variable will be constrained to be at least between 1 and 10. These bounds are passed along as additional, “side”, constraints to the model. Side constraints can be bounds on the variables or other extraneous equations and inequalities. These are dealt with in the solver interface by adding “dummy” variables to the problem.

However, complementarity problems with side constraints can be difficult for solvers to process. Therefore, the user will be warned when side constraints are present in the model. The behavior can be modified by changing the value of the “sideineq” option which takes on values of:

0 – do not warn

1 – warn about side constraints

2 – exit with return code 503 when side constraints exist

3 – generate a run-time error when side constraints exist

4 – use alternative reformulation (without warnings)

5 – use alternative reformulation (with a warning)

The default is to provide a warning message and reformulate the problem so that the Jacobian matrix does not have empty columns. The alternative reformulation adds variables to the problem, but creates empty columns in the Jacobian matrix.

Since square complementarity problems are typically more amenable to solution than complementarity problems with side constraints, we highly recommend that a user avoid generating a model containing side constraints.

2.2.3 Presolve

AMPL does some preprocessing for all of the mathematical programs that it generates. For the above model, the declared bounds on x are tighter than the bounds in the complementarity condition. Since none of the bounds in the complementarity condition can ever be satisfied, they are redundant and removed. The complementarity condition is then replaced by the single constraint:

```
subj to
func {i in I}:
    x[i] complements (sum {j in I} A[i,j]*x[j]) - b[i] = 0;
```

The declared bounds, $1 \leq x \leq 10$, are added as side constraints to the model. If however, the bounds in the complementarity conditions are tighter, the relevant bounds in the variable declaration are dropped and not passed to the solver.

For complementarity problems, the AMPL presolve phase can create side constraints not otherwise present. Particularly, when the value for a variable becomes known, the corresponding complementary function will be passed along as a side constraint. Therefore, if a side constraint warning is encountered, and the user believes they have a square complementarity problem, the AMPL presolve should be turned off with the command:

```
ampl: option presolve 0;
```

before the solve statement.

Note that the introduction of side constraints by the AMPL presolve can negatively impact an algorithm. In such cases, turning the AMPL presolve off could lead to improved performance by the solver.

2.3 GAMS

GAMS [11] declares complementarity conditions in the model definition statement using the . operator, a surrogate for the generalized $\perp$ notation used when defining a complementarity problem mathematically. An implementation of the transportation model from Section 2.1.1 can be found in Figure 4; the actual data for the model is assumed to be given in the file `transmcp.inc`. Example data is found in Figure 5.

The model corresponds very closely to (9). In GAMS, the $\perp$ sign is replaced in the `model` statement with a ".". The pairing of variables and equations is performed precisely at this point in the GAMS code. Therefore, in Figure 4, the function defined by `rational` is complementary to the variable `x`. The standard GAMS statements on the variables can be used to inform a solver of the bounds, namely (for a declared variable $z(i)$):

```
z.lo(i) = 0;
```

```
sets   i                 Suppliers,
       j                 Markets;

parameter
       cap(i)            Capacity of supplier,
       dem(j)            Demand at market,
       tra(i,j)          Transportation cost;

$include transmcp.inc

positive variables
       x(i,j)            Shipment quantity,
       p_s(i)            Price at plant,
       p_d(j)            Price at market;

equations
       supply(i)         Supply limit,
       demand(j)         Demand satisfaction,
       rational(i,j)     Rational marketplace;

supply(i)..     cap(i) =g= sum(j, x(i,j));

demand(j)..     sum(i, x(i,j)) =g= dem(j);

rational(i,j).. p_s(i) + tra(i,j) =g= p_d(j);

model transport / rational.x, demand.p_d, supply.p_s /;

solve transport using mcp;
```

Figure 4: Transportation model in GAMS, `transmcp.gms`

```
set  i   / seattle, san-diego /,
     j   / new-york, chicago, topeka /;

parameter cap(i) /
     seattle    325,
     san-diego  575
/;

parameter dem(j) /
     new-york   325,
     chicago    300,
     topeka     275
/;

table tra(i,j)
           new-york   chicago    topeka
seattle     0.225      0.153     0.162
san-diego   0.225      0.162     0.126;
```

Figure 5: Transportation data in GAMS, `transmcp.inc`

or

```
     positive variable z;
```

Further information on the GAMS syntax can be found in [112]. Note that positive variables are matched with `=g=` constraints in this model. However, using `=l=` in an equivalent declaration for the `supply` constraint

```
     supply(i)..     sum(j, x(i,j)) =l= cap(i);
```

results in an illegal complementarity problem and GAMS gives a model generation error.

A GAMS implementation of (11) is given in Figure 6, where we have introduced a defining variable, `d_p(i)`, and equation, `D(i)`, for the demand function. Note that free variables must be paired with equations. Thus in Figure 6, `d_p` is a free variable and its

```
$include walras.inc

positive variables p(i), y(j);
variables d_p(i);
equations S(i), L(j), D(i);

S(i)..   b(i) + sum(j, A(i,j)*y(j)) - d_p(i) =g= 0 ;

L(j)..   -sum(i, p(i)*A(i,j)) =g= 0 ;

D(i)..   d_p(i) =e= c(i)*sum(k, b(k)*p(k)) / p(i) ;

model walras / S.p, L.y, D.d_p /;

p.l(i) = 1;
solve walras using mcp;
```

Figure 6: Walrasian equilibrium as an MCP in GAMS, `walrasian.gms`

paired equation `demand` is an equality. Since `p` is nonnegative, its paired relationship `S` is a (greater-than) inequality.

The demand function in Figure 6 is undefined when the prices are zero. Since GAMS uses the projection of zero onto the variable bounds as a default starting point, we must specify an alternate starting point to avoid a GAMS execution error. The statement:

```
p.l(i) = 1;
```

is used to specify an initial point where the function is well-defined. *In general, a starting point close to a solution will typically help the algorithm to quickly solve the model.*

A simplification is allowed to the model statement in Figure 6. In many cases, it is not significant to match free variables explicitly to equations; we only require that there are the same number of free variables as equations. Thus, in the example of Figure 6, the model statement could be replaced by

```
model walras / S.p, L.y, D /;
```

This extension allows existing GAMS models consisting of a square system of nonlinear equations to be easily recast as a complementarity problem - the model statement is unchanged. GAMS generates a list of all variables appearing in the equations found in the model statement, performs explicitly defined pairings and then checks that the number of remaining equations equals the number of remaining free variables. All variables that are not free and all inequalities must be explicitly matched.

Equations should be explicitly matched to free variables to preserve matrix properties, such as positive semidefiniteness, that can be lost when an arbitrary matching is performed. The loss of structure can lead to failure or poor performance by an algorithm. For example, when constructing the first order optimality conditions for a linear program, the matching of equations to multipliers should be done to preserve the skew symmetric structure.

We can place an upper bound on the activity level in the Walrasian equilibrium by simply adding an upper bound to y

```
y.up(j) = 10;
```

and replacing the loss equation with the following definition:

```
L(j)..   -sum(i, p(i)*A(i,j)) =e= 0;
```

For bounded variables, we do not know beforehand if the constraint will be satisfied as an equation, less than inequality or greater than inequality, since this determination depends on the values of the solution variables. We adopt the convention that all bounded variables are paired to equations.

Note that if a variable z_i is fixed, then $F_i(z)$ is unrestricted since precisely one of the three conditions in the MCP definition automatically holds when $z_i = \ell_i = u_i$. Thus if a variable is fixed in a GAMS model, the paired equation is completely dropped from the model. This convenient modeling trick can be used to remove particular constraints from a model at generation time.

Furthermore, we can scale particular equations and variables in GAMS using the `.scale` attribute. For example, we can write the statements

```
D.scale(i) = 5;
d_p.scale(i) = 0.5;
```

to specify scalings for the `D` equation and `d_p` variable. To turn the scaling on for the model we set the `.scaleopt` model attribute.

```
walras.scaleopt = 1;
```

In certain circumstances, scaling can help an algorithm to solve the problem specified.

To solve the problem, the modeler executes the command:

```
gams transmcp
```

where `transmcp` can be replaced by any filename containing a GAMS model. Many command line options for GAMS exist; the reader is referred to [11] for further details as well as the on-line documentation at

```
http://www.gams.com/
```

One useful command line option allows a user to change the solver by issuing the command:

```
gams transmcp mcp=path
```

which sets the MCP solver to PATH 4.x.

The solver can be changed in many different ways, including the following:

1. Add the following line to the `transmcp.gms` file prior to the `solve` statement

```
option mcp = path;
```

where `path` can be any solver for complementarity problems. In this example, PATH 4.x will be used instead of the default solver provided. Note: setting the solver in the GAMS code takes precedence to setting the solver with a command line option.

2. Rerun the `gamsinst` program from the GAMS system directory and choose the desired algorithm as the default solver for MCP.

The SEMI algorithm is used by changing the MCP solver to `semi`.

The default algorithm options should be sufficient for most models; the technique for changing these options are now described. To change the default options on the model `transport`, the modeler is required to write a file `path.opt` for the PATH 4.x algorithm in the working directory and either add a line

```
transport.optfile = 1;
```

before the `solve` statement in the file `transmcp.gms`, or use the command-line option

```
gams transmcp optfile=1
```

Unless the modeler has changed the WORKDIR parameter explicitly, the working directory will be the directory containing the model file. The procedure is the same when using the SEMI algorithm. However, the options file will be called `semi.opt`.

GAMS controls the total number iterations allowed via the `iterlim` option. If more iterations are needed for a particular model then either of the following lines should be added to the `transmcp.gms` file before the solve statement

```
option iterlim = 2000;
transport.iterlim = 2000;
```

An iteration in the PATH 4.x and SEMI codes consists of finding a solution to a linear system of equations $Ax = b$ for some matrix A and right hand side b.

Similarly, if the solver runs out of memory, then the workspace allocated can be changed using

```
transport.workspace = 20;
```

The above example would allocate 20MB of workspace for solving the model.

After a `solve` statement is encountered, control is handed over to the solver which creates a log providing information on what the solver is doing as time elapses. After the solver terminates, a listing file is generated containing the problem solution. We now describe the output in the listing file specifically related to the complementarity problem.

2.3.1 Listing File

The listing file is the standard GAMS mechanism for reporting model results. This file contains information regarding the compilation process, the form of the generated equations in the model, and a report from the solver regarding the solution process.

We now detail the last part of this output for the PATH 4.x solver, an example of which is given in Figure 7. We use "..." to indicate where we have omitted continuing similar output.

After a summary line indicating the model name and type and the solver name, the listing file shows a solver status and a model status. Table 4 and Table 5 display the relevant codes that are returned under different circumstances. A modeler can access these codes within the `transmcp.gms` file using `transport.solstat` and `transport.modelstat` respectively. An objective value is also reported, which can be accessed as `transport.objval`. This objective measures the complementarity error, which

```
          S O L V E      S U M M A R Y

     MODEL   transport
     TYPE    MCP
     SOLVER  PATH                FROM LINE  50

**** SOLVER STATUS     1 NORMAL COMPLETION
**** MODEL STATUS      1 OPTIMAL

 RESOURCE USAGE, LIMIT          0.145     1000.000
 ITERATION COUNT, LIMIT        18        10000
 EVALUATION ERRORS              0            0

 Work space allocated           --    0.06 Mb

---- EQU RATIONAL

                      LOWER     LEVEL     UPPER    MARGINAL

seattle  .new-york    -0.225    -0.225     +INF     50.000
seattle  .chicago     -0.153    -0.153     +INF    300.000
seattle  .topeka      -0.162    -0.126     +INF       .

...

---- VAR X           shipment quantities in cases

                      LOWER     LEVEL     UPPER    MARGINAL

seattle  .new-york      .       50.000     +INF       .
seattle  .chicago       .      300.000     +INF       .

...

**** REPORT SUMMARY :        0     NONOPT
                             0 INFEASIBLE
                             0  UNBOUNDED
                             0  REDEFINED
                             0     ERRORS
```

Figure 7: Listing file solving `transmcp.gms` in GAMS

Table 4: Solution Status Codes for GAMS

Code	String	Meaning
1	Normal completion	Solver returned to GAMS without an error
2	Iteration interrupt	Solver used too many iterations
3	Resource interrupt	Solver took too much time
4	Terminated by solver	Solver encountered difficulty and was unable to continue
8	User interrupt	The user interrupted the solution process

Table 5: Model Status Codes for GAMS

Code	String	Meaning
1	Optimal	Solver found a solution of the problem
6	Intermediate infeasible	Solver failed to solve the problem

is zero at a solution to the problem and positive elsewhere.

After the solver and model status is reported, a listing of the time and iterations used by the algorithm is given, along with a count of the number of evaluation errors encountered. If the number of evaluation errors is greater than zero, further information can typically be found later in the listing file prefaced by the "****" string. Information provided by the solver is then displayed.

Next comes the solution listing starting with each of the equations in the model. For each equation passed to the solver, four columns are reported, namely the lower bound, level, upper bound and marginal. GAMS moves all parts of a constraint involving variables to the left hand side and accumulates the constants on the right hand side. The lower and upper bounds correspond to the constants that GAMS generates. For equations, these should be equal, whereas for inequalities one of them should be infinite. The level value of the equation (an evaluation of the left hand side of the constraint at the current point) should be between these bounds, otherwise the solution is infeasible and the equation is marked as follows:

```
seattle  .chicago     -0.153    -2.000     +INF    300.000 INFES
```

The marginal column for an equation returns the level value of the variable that was matched to the equation. If the modeler did not pair a particular equation with a variable, the value returned is that of the variable that the solver interface paired with the equation.

For the variable listing, the lower, level, and upper columns indicate the lower and upper bounds on the variables and the solution value. The level value returned by PATH 4.x and SEMI will always be between these bounds. The marginal column contains the value of the slack on the equation that was paired to the variable. If a variable appears in one of the constraints in the model statement but is not explicitly paired to a constraint, the slack reported here contains the internally matched constraint slack. The definition of this slack is the minimum of `equ.l - equ.lower` and `equ.upper - equ.l`, where `equ` is the paired equation.

Finally, a summary report is given that indicates how many errors were found. Figure 7 is a good case; when the model has infeasibilities, these can be found by searching for the string "INFES" as described above.

2.3.2 Redefined Equations

Unfortunately, the above description is incomplete because some of the equations may have the following form:

```
                 LOWER      LEVEL       UPPER     MARGINAL
    new-york   325.000    350.000     325.000    0.225 REDEF
```

This notation should be construed as a warning from GAMS, as opposed to an error. In principle, the `REDEF` should only occur if the paired variable to the constraint had a finite lower and upper bound and the variable is at one of those bounds. In this case, at the solution of the complementarity problem the "equation (`=e=`)" may not be satisfied. The problem occurs because of a limitation in the GAMS syntax for complementarity problems. The GAMS equations are used to define the function F. The bounds on the function F are derived from the bounds on the associated variables. Before solving the problem, for finite bounded variables, we do not know if the associated function will be positive, negative or zero at the solution. Thus, we do not know whether to define the equation as "`=e=`", "`=l=`" or "`=g=`". GAMS therefore allows any of these, and informs the modeler via the "REDEF" label that internally GAMS has redefined the bounds so that the solver processes the correct problem, but that the solution given by the solver does not satisfy the original bounds. However, in practice, a `REDEF` can also occur when the equation is defined using "`=e=`" and the variable has a single finite bound. This matching is allowed by GAMS, and as above, if the variable is at its bound at a solution to the complementarity problem, then the function F may not satisfy the "`=e=`" relationship.

Note that this is not an error, just a warning. The solver has solved the complementarity problem specified by this equation. GAMS gives this report to ensure that the modeler understands that the complementarity problem derives the relationships on the equations from the variable bounds, not from the equation definition.

2.3.3 Pitfalls

The ordering of an equation is important in the specification of an MCP. Since the data of the MCP is the function F and the bounds ℓ and u, it is important for the modeler to pass the function F and not $-F$ to the solver.

For example, if we have the optimization problem,

$$\min_{x \in [0,2]} (x-1)^2$$

```
variables x;
equations d_f;

x.lo = 0;
x.up = 2;

d_f.. 2*(x - 1) =e= 0;

model first / d_f.x /;
solve first using mcp;
```

Figure 8: First order conditions as an MCP in GAMS, `first.gms`

then the first order optimality conditions are

$$0 \leq x \leq 2 \quad \perp \quad 2(x-1)$$

which has a unique solution, $x = 1$. Figure 8 provides correct GAMS code for this problem. However, if we accidentally write the valid equation

```
d_f.. 0 =e= 2*(x - 1);
```

the problem given to the solver is

$$0 \leq x \leq 2 \quad \perp \quad -2(x-1)$$

which has three solutions, $x = 0$, $x = 1$, and $x = 2$. This problem in fact formulates the stationary conditions for the non-convex quadratic problem,

$$\max_{x \in [0,2]} (x-1)^2,$$

not the problem we intended to solve.

2.4 MATLAB

MATLAB [89] offers a convenient environment for exploration and visualization. Both linear and nonlinear complementarity problems can be written in MATLAB and conveyed to the PATH 4.x and SEMI solvers through provided m-functions.

Furthermore, the MATLAB interfaces are unique in that polyhedrally constrained variational inequalities can be specified and solved. Mathematically, the polyhedrally constrained variational inequality is given a set $C := \{z|Az \geq b\}$, and a function, F. We then want to find a point, $z \in C$ where:

$$\langle F(z), \bar{z} - z \rangle \geq 0 \text{ for all } \bar{z} \in C.$$

As mentioned in Section 1.1 when C is the Cartesian product of closed (not necessarily bounded) intervals, the (box constrained) variational inequality is equivalent to the mixed complementarity problem. The general polyhedral constraints are handled by adding multipliers, resulting in a standard mixed complementarity problem.

2.4.1 Linear Complementarity Problems

The routines tailored for linear complementarity problems offer improved performance over the nonlinear counterparts, as they do not make any call-backs to MATLAB for function evaluations. Furthermore, the preprocessor for complementarity problems developed in Chapter 4 is enabled when using the MATLAB interface for linear models.

The m-function for solving standard linear complementarity problems,

$$0 \leq z \quad \perp \quad Mz + q \geq 0,$$

is defined in `pathlcp.m` for PATH 4.x and `semilcp.m` for SEMI. To solve a standard linear complementarity problem using PATH 4.x, we write the MATLAB command:

```
>> z = pathlcp(M, q);
```

An implementation of the transportation model from Section 2.1.1 can be found in Figure 9. Note that the matrix M passed to the `pathlcp` routine will be automatically converted into a sparse matrix. The conversion can be avoided if the matrix passed into the function is already stored as a sparse matrix.

Linear mixed complementarity problems can be specified by adding lower and upper bounds:

```
>> z = pathlcp(M, q, l, u);
```

which solves the problem

$$l \leq z \leq u \quad \perp \quad Mz + q.$$

The numerical value used by the MATLAB interfaces for ∞ is 10^{20}.

Many times, a good starting point can help an algorithm to quickly find a solution. Therefore, a starting point can be passed as an additional argument:

```
>> z = pathlcp(M, q, l, u, z);
```

Note that a call of the form `pathlcp(M, q, [], [], z)` is handled by using $\ell = \{0\}^n$ and $u = \{\infty\}^n$ as default values. Therefore, this statement solves the standard linear complementarity problem using a provided starting point.

Additional arguments to `pathlcp` are supplied for conveying polyhedrally constrained variational inequalities. Internally, multipliers on the constraints are added resulting in a linear mixed complementarity problem. The constraints $Az \geq b$ are passed to the solver with additional arguments for A and b:

```
>> z = pathlcp(M, q, l, u, z, A, b);
```

```
>> M = [
   0   0   0   0   0   0  -1   0   0   1   0;
   0   0   0   0   0   0   0  -1   0   1   0;
   0   0   0   0   0   0   0   0  -1   1   0;
   0   0   0   0   0   0  -1   0   0   0   1;
   0   0   0   0   0   0   0  -1   0   0   1;
   0   0   0   0   0   0   0   0  -1   0   1;
   1   0   0   1   0   0   0   0   0   0   0;
   0   1   0   0   1   0   0   0   0   0   0;
   0   0   1   0   0   1   0   0   0   0   0;
  -1  -1  -1   0   0   0   0   0   0   0   0;
   0   0   0  -1  -1  -1   0   0   0   0   0
];
>> q = [
   0.225;
   0.153;
   0.162;
   0.225;
   0.162;
   0.126;
-325.000;
-300.000;
-275.000;
 325.000;
 575.000
];
>> z = pathlcp(M, q)

z =

   25.0000
  300.0000
         0
  300.0000
         0
  275.0000
   87.0291
   86.9571
   86.9301
   86.8041
   86.8041
```

Figure 9: Transportation problem in MATLAB

The inequality type can be passed in an extra vector; a negative value indicates that we have a less than or equal to inequality, a positive value means greater than or equal, and zero means an equation. The multipliers can be passed to and returned from the `pathlcp` function call.

```
>> [z, mu] = pathlcp(M, q, l, u, z, A, b, t, mu);
```

The transportation model can be more succinctly posed as a polyhedrally constrained variational inequality as given in Figure 10.

The polyhedrally constrained variational inequality can be used to easily write the optimality conditions for linear and convex quadratic programs. For example, if we have the quadratic program:

$$\begin{array}{rc} \min_x & \frac{1}{2}x^T Qx + c^T x \\ \text{subject to} & Ax \geq b \\ & l \leq x \leq u \end{array}$$

with Q a symmetric, positive semidefinite matrix, then the necessary and sufficient optimality conditions can be formulated and passed with the command:

```
>> [x, mu] = pathlcp(Q, c, l, u, [], A, b)
```

For convenience, two m-functions for solving linear programs and convex quadratic programs are provided:

```
>> [x, mu] = lp_lcp(c, A, b, l, u, x, t);
>> [x, mu] = qp_lcp(Q, c, A, b, l, u, x, t);
```

These functions pass along the above optimality conditions to the `pathlcp` function.

The default algorithm options should be sufficient for most models. However, solver specific options can be modified by writing a `path.opt` file for PATH 4.x or a `semi.opt`

```
>> A = [
   1   0   0   1   0   0;
   0   1   0   0   1   0;
   0   0   1   0   0   1;
  -1  -1  -1   0   0   0;
   0   0   0  -1  -1  -1
];
>> b = [
   325;
   300;
   275;
  -325;
  -575
];
>> q = [
    0.225;
    0.153;
    0.162;
    0.225;
    0.162;
    0.126
];
>> [z, mu] = pathlcp(sparse(6,6), q, [], [], [], A, b)

z =

   25.0000
  300.0000
         0
  300.0000
         0
  275.0000

mu =

   87.0291
   86.9571
   86.9301
   86.8041
   86.8041
```

Figure 10: Transportation model as a variational inequality in MATLAB

file for SEMI. In cases where the algorithm fails to find a solution, the logfile written in `logfile.tmp` can be investigated to gain information about the reasons for failure. Note that if the algorithm fails to solve the problem, a MATLAB error is generated. This behavior is part of the m-function, so a user could modify the termination behavior if desired.

2.4.2 Nonlinear Complementarity Problems

Nonlinear mixed complementarity problems require that a user provide an m-function that evaluates the nonlinear function and Jacobian at a specified point. The calling syntax is:

```
z = pathmcp(z, l, u, 'mcp_funjac');
```

which uses PATH 4.x where `mcp_funjac` is the name of the nonlinear function evaluation routine. The `z` vector must be explicitly provided in order to determine the size of the complementarity problem. For nonlinear models the algorithm may perform much better by choosing an appropriate starting point close to a solution. $\ell = \{0\}^n$ and $u = \{\infty\}^n$ are the default values used if the lower and upper bounds are not present. That is, we solve a standard nonlinear complementarity problem.

The corresponding m-function `mcp_funjac.m` contains the definition

```
function [F, J, domerr] = mcp_funjac(z, jacflag)
```

that computes the function, F, and, if `jacflag=1`, the sparse Jacobian J at the point z. The `domerr` variable returns the number of domain violations encountered during the evaluation. A domain violation occurs for example when we have division by zero. *The Jacobian must be passed to the solver as a sparse matrix.* As an example, a function for the transportation model from Section 2.1.1 with isoelastic demand is provided in Figure 11.

Providing the derivative explicitly can be time consuming and prone to error. Therefore, we provide a convenience routine that uses the `ADMAT` software [18] to perform automatic differentiation of a supplied m-function. Assuming that `ADMAT` has been installed according to the user guide, we can call PATH 4.x with the commands:

```
>> mcp_set_adfunc('mcp_fun');
>> z = pathmcp(z, l, u, 'mcp_adfunjac');
```

The first line of the code sets a global variable for the user defined function with syntax

```
function F = mcp_fun(z, extra)
```

Note that the `extra` argument is needed by the `ADMAT` software, but should not be used in the function evaluation code. The second line of code tells the interface to use the provided `mcp_adfunjac` function to perform the function and Jacobian evaluations. The implementation of `mcp_adfunjac` is:

```
function [F,J,domerr] = mcp_adfunjac(z,jacflag)

global mcp_adfun;

if jacflag == 1
  [F, J] = evalJ(mcp_adfun, z);
else
  F = eval(mcp_adfun, z);
  J = [];
end;

domerr = 0;
return;
```

The global variable `mcp_adfun` is used to keep track of the m-function that the user specified for their function. Note that the `evalJ` uses `ADMAT` to evaluate the function and

```
function [F, J, domerr] = mcp_funjac(z, jacflag)

% initialize
z = z(:);
F = [];
J = [];
domerr = 0;

% obtain variable values
x = z(1:6);     % quantities
p_d = z(7:9);   % demand price
p_s = z(10:11); % supply price

% check for domain violation
if (p_d <= 0)
  domerr = 1;
  return;
end

cost   = [ 0.225; 0.153; 0.162; 0.225; 0.162; 0.126; ];
demand = [ -325; -300; -275; ] ./ (p_d.^0.5);
supply = [ 325; 575; ];

A = sparse( [
   1   0   0   1   0   0;
   0   1   0   0   1   0;
   0   0   1   0   0   1;
  -1  -1  -1   0   0   0;
   0   0   0  -1  -1  -1;
] );

F = [ (-A'*[p_d; p_s] + cost); A*x + [demand; supply] ];

if (jacflag)
  J = [ sparse(6,6) -A';
        A diag( [ -0.5 * [-325; -300; -275;] ./ (p_d.^1.5); 0; 0] ) ];
end

return
```

Figure 11: Transportation model with isoelastic demand in MATLAB

Jacobian. Currently, domain violations are not reported by the automatic differentiator. Therefore, we always assume that no domain violations occurred.

The function and Jacobian at the returned solution can be obtained by adding additional output parameters:

```
[z, F, J] = pathmcp(z, l, u, 'mcp_funjac');
```

We also allow extensions for polyhedrally constrained variational inequalities with the syntax:

```
[z, F, J, mu] = pathmcp(z, l, u, 'mcp_funjac', A, b, t, mu);
```

where `A`, `b`, and `t` are used to convey the polyhedral set as in Section 2.4.1. These are internally reformulated by adding multipliers so that a mixed complementarity problem is solved. The values for the multipliers can be obtained with the `mu` arguments.

Algorithm options can be changed using the same technique as presented in the section on linear complementarity problems. Specifically, a `path.opt` or `semi.opt` file is written with option, value pairs. A log file is also written in `logfile.tmp` that can be examined. As with the linear interface, if the algorithm fails to solve the problem, a MATLAB error is generated. This behavior can be changed by modifying the m-function.

2.4.3 Mex Functions

Most users will call the PATH 4.x or SEMI codes through the provided m-functions for linear and nonlinear complementarity problems. However, a user could bypass these m-function and directly invoke the appropriate mex-function. Essentially, a mex-function contains a compiled version of a code that can be accessed within MATLAB. To call the mex-function explicitly, we write:

```
status = lcppath(n, nnz, z, l, u, M, q);
```

for linear complementarity problems and

```
[status, F, J] = mcppath(n, nnz, z, l, u, func);
```

for nonlinear models. Note that `lcpsemi` and `mcpsemi` can be used for the SEMI solver with the same arguments. A status of 1 is returned if the model is solved and 0 is returned otherwise. z is used as the initial starting point as well as the solution vector. The F and J output arguments to `mcppath` are not necessary. However, if present, they will contain the function and Jacobian evaluation at the z vector returned by the algorithm.

The number of nonzeros passed into the mex-routine should be an **over-estimate** of the exact number of nonzeros when using a nonlinear function. The reason for this is that a fixed amount of space is initially allocated by the solver for storing the Jacobian. Therefore, in situations where the number of nonzeros in the Jacobian changes, we could run into serious problems. For example, the (sparse) Jacobian of the function x^2 at zero is empty. A heuristic suggested by Jorge Moré is employed in the `pathmcp` m-function that perturbs the initial starting point slightly and evaluates the Jacobian at this perturbed point to calculate the over-estimate of the number of nonzeros in the Jacobian. This technique is not foolproof, but has worked well in practice.

2.4.4 Pitfalls

Interrupting the solver in the middle of a computation can lead to segmentation violations on subsequent calls. Therefore, if the solver has been interrupted, a user should issue the command

```
>> clear mex;
```

before attempting to solve another problem. This statement unloads the mex-functions; subsequent calls will reload them.

2.5 NEOS

NEOS [22] enables users to submit optimization problems across the Internet for solution. While complementarity problems can be solved by sending models written in the AMPL and GAMS modeling languages to the NEOS server, an additional interface is provided for users wanting to supply a function evaluation routine written in FORTRAN.

The essential routine the user provides evaluates the function, F, at a specified point. The function header is as follows:

```
subroutine fcn(n,x,F)
```

where n is the number of variables in the problem, x is a vector of length n containing the specified point, and F is a vector of length n which returns the user defined function evaluation at the point x. For example, in the transportation model, we have 11 variables and the function would be implemented as provided in Figure 12. By default, the complementarity problem uses lower bounds of zero and upper bounds of infinity, a standard nonlinear complementarity problem, and an initial starting point of zero.

Having written the function, we then submit the problem to the NEOS server. There are several different methods available for submission as documented on the NEOS homepage:

```
http://www-neos.mcs.anl.gov/
```

We concentrate on e-mail submissions. In order to solve the transportation problem through e-mail, we send a message to `neos@mcs.anl.gov` containing the following:

```
      subroutine fcn(n,x,F)
      implicit none

      integer n
      double precision x(n), F(n)

C     Rational marketplace constraints
      F(1)  = x(10) - x(7) + 0.225d+0
      F(2)  = x(10) - x(8) + 0.153d+0
      F(3)  = x(10) - x(9) + 0.162d+0
      F(4)  = x(11) - x(7) + 0.225d+0
      F(5)  = x(11) - x(8) + 0.162d+0
      F(6)  = x(11) - x(9) + 0.126d+0

C     Capacity constraints
      F(7)  = x(1) + x(4) - 325.0d+0
      F(8)  = x(2) + x(5) - 300.0d+0
      F(9)  = x(3) + x(6) - 275.0d+0

C     Satisfaction of demand constraints
      F(10) = 325.0d+0 - x(1) - x(2) - x(3)
      F(11) = 575.0d+0 - x(4) - x(5) - x(6)

      return
      end
```

Figure 12: Transportation function for NEOS, `fcn.f`

```
TYPE = CP
SOLVER = PATH

N = 11

BEGIN.FCN
    subroutine fcn(n,x,F)
    implicit none

    integer n
    double precision x(n), F(n)

C   Actual function definition here

    return
    end
END.FCN

END-SERVER-INPUT
```

The results will then be mailed back to the person sending the mail.

Once the mail has been sent to NEOS, a remote machine is contacted which compiles the function, calculates the derivative with ADIFOR [8], links the code with appropriate libraries, and runs the executable. The mail reported to the user appears as below for the transportation model, where we have used ... to denote that some information has been deleted to conserve space.

```
...

***PATH Output***
Path v4.4a: Neos Link
11 row/cols, 24 non-zeros, 19.83% dense.

Could not open options file: path.opt
Using defaults.
Path 4.4a (Sun Jun 18 04:24:05 2000)

SOLUTION FOUND.
```

Table 6: Return messages for NEOS

Message	Meaning
INFEASIBLE	The model is infeasible
ITERATION LIMIT	The iteration limit was reached without finding a solution
OTHER ERROR	An error occurred within the algorithm
SOLUTION FOUND	A solution was found by the algorithm
TIME LIMIT	The time limit expired without finding a solution

```
Major Iterations. . . . 14
Minor Iterations. . . . 18
Restarts. . . . . . . . 0
Crash Iterations. . . . 1
Gradient Steps. . . . . 0
Function Evaluations. . 16
Gradient Evaluations. . 16
Total Time. . . . . . . 0.060000
Residual. . . . . . . . 2.980724e-10


SOLUTION VECTORS:
x(1   ): +2.5000000000e+01    F(1   ): -2.5010549187e-13
x(2   ): +3.0000000000e+02    F(2   ): +1.7894019599e-13
x(3   ): +0.0000000000e+00    F(3   ): +3.5999999999e-02
x(4   ): +3.0000000000e+02    F(4   ): +2.3306356844e-13
x(5   ): +0.0000000000e+00    F(5   ): +9.0000000007e-03
x(6   ): +2.7500000000e+02    F(6   ): -1.1846079673e-13
x(7   ): +8.7029079521e+01    F(7   ): -2.2907897801e-11
x(8   ): +8.6957079521e+01    F(8   ): +2.0571633286e-10
x(9   ): +8.6930079521e+01    F(9   ): -1.3722001313e-10
x(10  ): +8.6804079521e+01    F(10  ): +9.1517904366e-11
x(11  ): +8.6804079521e+01    F(11  ): -1.3710632629e-10

...
```

The last part of the listing contains the log file generate by the algorithm. The solution vectors list x and $F(x)$ for the point returned by the algorithm. A complete listing of the return messages can be found in Table 6.

Other keywords can be sent in the e-mail message, as given in Table 7. The keywords

Table 7: Keywords for NEOS e-mail submissions

Keyword	Type	Meaning
TYPE	model type	The type of model to solve
SOLVER	solver name	The name of the solver to use
N	integer	Number of variables in the problem
FCN	source code	Source code for the function evaluation routine
INITPT	text	Source code for the initial point routine
XBOUND	text	Source code for the bounds subroutine
OPTIONS	text	Algorithm specific options to be set
COMMENTS	text	Comments about the model

specifying multiple lines of input (`FCN,INITPT,XBOUND,COMMENTS`) use `BEGIN.keyword` and `END.keyword` to denote the start and end of the text block. In particular, algorithm specific options can be set with the `OPTIONS` keyword.

Optional routines for specifying a starting point and general bounds can be provided by a user with specifications:

```
subroutine xbound(n,l,u)
subroutine initpt(n,x)
```

The user places the lower bounds in `l`, the upper bounds in `u`, and the initial starting point in `x` respectively. In all cases, `n` is the number of variables specified for the problem. On input to the `xbound` routine, the lower and upper bounds are set to plus and minus infinity respectively. An initial starting point of zero will be used. If the routines are not supplied, the default implementation ($\ell = \{0\}^n$, $u = \{\infty\}^n$, and $x = \{0\}^n$) will be used.

2.5.1 Pitfalls

Evaluation errors in the function and Jacobian cannot currently be reported to the algorithm. Therefore, the results can be undefined when there is a domain violation, such as division by zero, in the function or Jacobian.

The automatic differentiator used, ADIFOR, will process non-differentiable functions and even those containing discontinuities. However, the algorithm is likely to not work well in these situations. Therefore, we encourage the user to specify models utilizing smooth functions.

The Jacobian is passed to the solver as a sparse matrix. A fixed amount of space is initially allocated by the solver. Therefore, in situations where the number of nonzeros in the Jacobian changes, we could run into serious problems. For example, the (sparse) Jacobian of the function x^2 at zero is empty. A heuristic suggested by Jorge Moré is employed in the routines that perturbs the initial starting point slightly and evaluates the Jacobian at this perturbed point to calculate an over-estimate of the number of nonzeros in the Jacobian. This technique is not foolproof, but has worked well in practice.

2.6 Summary

This chapter developed two example complementarity problems, a transportation model and a Walrasian equilibrium, and demonstrated how they are communicated to a solver in the AMPL, GAMS, MATLAB, and NEOS environments. These environments are heavily used in practical applications because of the ease with which a model can be specified and conveyed to different solvers. The solver interface to the PATH 4.x and SEMI codes used to support all of these environments is documented in the next chapter.

We note that an earlier version of Section 2.1 and Section 2.3 appeared in [44]. The original AMPL interface to PATH was developed by David Gay, while the original GAMS interface to PATH was developed by Steven Dirkse and Thomas Rutherford. Both have been revised to support both PATH 4.x and SEMI. The AMPL interface to PATH 4.x can use the preprocessor for complementarity problems developed in Chapter 4. An alternate version of the GAMS interface was written that enables the use of the preprocessor for complementarity problems, but does not support MPSGE models [113]. This new interface is available in GAMS by using the PATHC and SEMIC solvers. (The 'C' at the end of the solver name is used to indicate that support is provided for constrained systems of nonlinear equations, which are internally reformulated as mixed complementarity problems.) Finally, we remark that the MATLAB and NEOS interfaces are new to this thesis.

Chapter 3

Interfaces

The previous chapter discussed complementarity from a modeling perspective and demonstrated how to use the AMPL, GAMS, MATLAB, and NEOS tools to write a complementarity problem and convey it to a solver. This chapter looks at the solver from an application programmer's point of view and documents the mechanisms available for integrating PATH 4.x and SEMI into other environments or applications.

The solver interface needs to be able to support the requirements for each individual environment. For example, GAMS provides a heap for memory allocation purposes, while MATLAB provides a special function for reporting errors. Therefore, the solver interface abstracts core components, output and memory allocation in particular, from the algorithm so that implementations tailored to a particular environment can be supplied either at compile or run time. In addition to specifying a problem and invoking a solver, the solver interface can then be used to control output, memory allocation, and interrupt handling.

Section 3.1 discusses a simplified interface to the PATH 4.x and SEMI codes meant to eliminate most of the errors that may occur while coding a particular application. The remaining sections in this chapter detail the solver interface used to implement the links to the environments from Chapter 2. An application programmer could use this documentation to write interfaces for other uses. In particular, an example interface for solving convex quadratic programs is developed in Section 3.5.

3.1 Subroutines

A simplified subroutine interface has been developed that hides many of the details associated with invoking PATH 4.x or SEMI directly, eliminating most of the errors that may occur while coding a particular application. To use the subroutine library, a user supplies function and Jacobian evaluation routines, and calls the PATH 4.x or SEMI algorithms within their main program.

The subroutine called by the user's program is declared as

```
void pathMain(int n, int nnz, int *status,
              double *z, double *F, double *l, double *u);
```

for PATH 4.x. The inputs to this routine are n and nnz, the number of variables in the problem and an **over-estimate** of the number of nonzeros in the Jacobian of F respectively, and vectors containing the starting point and lower and upper bounds. The numerical value used for infinity in the subroutine library interface is 10^{20}. The subroutine returns the solver status, the solution vector, z, and function evaluation, $F(z)$. The status codes are fully described in Table 8. SEMI can be used by calling the `semiMain` subroutine which has the same input and output arguments as `pathMain`.

The remaining components of the simplified interface are the function and Jacobian evaluation routines with declarations:

```
int funcEval(int n, double *z, double *F);
int jacEval(int n, int nnz, double *z, int *col,
            int *len, int *row, double *data);
```

The `funcEval` routine is given the problem size, n, and a vector containing a point, z. The user is then requested to place the function evaluation at z into F. The `jacEval` routine is also given the problem size, n, the number of nonzeros allocated for the Jacobian, nnz,

Table 8: Status codes for Subroutine Interfaces

Code	String	Meaning
1	Solver	A solution to the problem was found.
2	No Progress	Algorithm could not improve upon the current iterate.
3	Major Iteration Limit	An iteration limit was reached.
4	Cumulative Iteration Limit	The minor iteration limit was reached.
5	Time Limit	Time limit was exceeded.
6	User Interrupt	The user requested that the solver stop execution.
7	Bound Error	The problem is infeasible because $\ell_l > u_i$ for some components.
8	Domain Error	A starting point where the function is defined could not be found.
9	Infeasible	The preprocessor determined the problem is infeasible.
10	Error	An internal error occurred in the algorithm.

and a given point, z. The Jacobian of F at z should be places into `col`, `len`, `row`, and `data`, where the Jacobian is in a compressed sparse column format. The indices in `col` and `len` **must** take on values between 1 and n. That is, the indices should be written in the FORTRAN style. Note that the algorithm will not change the structure passed for the Jacobian. Therefore, if the structure of the Jacobian does not change, `col`, `len`, and `row` only need to be filled once. The function and Jacobian evaluation routines should return the number of domain violations encountered. For example, a domain violation occurs when we have division by zero.

An implementation for the transportation model from Section 2.1 with isoelastic demand is given in Figure 13 and Figure 14. Note that an initial starting point is specified where the function is well-defined and that the structure of the Jacobian matrix uses the correct FORTRAN style indices.

Algorithm specific options can be placed in a file called `path.opt` for PATH 4.x or `semi.opt` for SEMI. Furthermore, if the solver fails to find a complementary solution,

```c
#include "Standalone.h"
#include <math.h>

int main()
{
     int n = 11, nnz = 27, status, i;
     double z[11], F[11], l[11], u[11];

     for (i = 0; i < n; i++)
     {
          z[i] = 1.0; l[i] = 0; u[i] = 1e20;
     }

     pathMain(n, nnz, &status, z, F, l, u);
     return 0;
}

int funcEval(int n, double *z, double *F)
{
     if ((z[6] <= 0) || (z[7] <= 0) || (z[8] <= 0))
          return 1;

     /* Rational marketplace constraints */
     F[0]  = z[9]  - z[6] + 0.225;
     F[1]  = z[9]  - z[7] + 0.153;
     F[2]  = z[9]  - z[8] + 0.162;
     F[3]  = z[10] - z[6] + 0.225;
     F[4]  = z[10] - z[7] + 0.162;
     F[5]  = z[10] - z[8] + 0.126;

     /* Capacity constraints */
     F[6]  = z[0] + z[3] - 325.0 / pow(z[6], 0.5);
     F[7]  = z[1] + z[4] - 300.0 / pow(z[7], 0.5);
     F[8]  = z[2] + z[5] - 275.0 / pow(z[8], 0.5);

     /* Satisfaction of demand constraints */
     F[9]  = 325.0 - z[0] - z[1] - z[2];
     F[10] = 575.0 - z[3] - z[4] - z[5];
     return 0;
}
```

Figure 13: Transportation model using Simplified Interface

```
int jacEval(int n, int nnz, double *z, int *col,
            int *len, int *row, double *data)
{
     static int filled = 0;

     if (!filled)
     {
          /* Fill in the structure of the Jacobian */

          col[0] = 1;  len[0] = 2; row[0] = 7; row[1] = 10;
          col[1] = 3;  len[1] = 2; row[2] = 8; row[3] = 10;
          col[2] = 5;  len[2] = 2; row[4] = 9; row[5] = 10;
          col[3] = 7;  len[3] = 2; row[6] = 7; row[7] = 11;
          col[4] = 9;  len[4] = 2; row[8] = 8; row[9] = 11;
          col[5] = 11; len[5] = 2; row[10]= 9; row[11]= 11;
          col[6] = 13; len[6] = 3; row[12]= 1; row[13]=  4; row[14]= 7;
          col[7] = 16; len[7] = 3; row[15]= 2; row[16]=  5; row[17]= 8;
          col[8] = 19; len[8] = 3; row[18]= 3; row[19]=  6; row[20]= 9;
          col[9] = 22; len[9] = 3; row[21]= 1; row[22]=  2; row[23]= 3;
          col[10]= 25; len[10]= 3; row[24]= 4; row[25]=  5; row[26]= 6;
          filled = 1;
     }

     if ((z[6] <= 0) || (z[7] <= 0) || (z[8] <= 0))
          return 1;

     data[0] =  1; data[1] = -1;
     data[2] =  1; data[3] = -1;
     data[4] =  1; data[5] = -1;
     data[6] =  1; data[7] = -1;
     data[8] =  1; data[9] = -1;
     data[10]=  1; data[11]= -1;
     data[12]= -1; data[13]= -1; data[14]=  0.5*325 / pow(z[6], 1.5);
     data[15]= -1; data[16]= -1; data[17]=  0.5*300 / pow(z[7], 1.5);
     data[18]= -1; data[19]= -1; data[20]=  0.5*275 / pow(z[8], 1.5);
     data[21]=  1; data[22]=  1; data[23]=  1;
     data[24]=  1; data[25]=  1; data[26]=  1;
     return 0;
}
```

Figure 14: Transportation model using Simplified Interface (cont.)

the user might find it helpful to look at the log file. The user can specify a file where the log will be written by calling the function:

```
void Output_SetLog(FILE *fp);
```

before the call to `pathMain` or `semiMain`. The argument is an opened file where the log will be written.

3.2 Problem Structures

We now look at the complete solver interface used to implement the simplified subroutine library and the links to the environments in Chapter 2. The core component of the interface is the notion of a mixed complementarity problem. The `MCP` structure serves as a repository for information about the particular complementarity problem a user wants to solve. Associated with each `MCP` is an `MCP_Interface`, which contains pointers to user specified functions for obtaining bounds, and function and Jacobian evaluations. An optional `Presolve_Interface` provides additional information about the problem for use in the preprocessor developed in Chapter 4. The header file, `MCP.h`, is found in Figure 15 and contains all of the function declarations used in conjunction with the `MCP`, `MCP_Interface` and `Presolve_Interface` structures.

A fundamental operation is to allocate and deallocate a problem instance. Table 9 lists the functions used for this purpose. The `MCP_Create` function has arguments for the problem size and number of nonzeros, n and nnz respectively, which can take on any values. However, for best performance, they should be **over-estimates** of the actual size and number of nonzeros in the problem. Otherwise, memory reallocation will be performed within the solver. The `MCP_Destroy` function is used to deallocate a previously allocated problem instance, while `MCP_Size` can be used to increase the number of

```
struct _MCP;
typedef struct _MCP MCP;

/* Allocation and deallocation functions. */
MCP *MCP_Create(int maxModSize, int maxModNNZ);
void MCP_Destroy(MCP *m);
void MCP_Size(MCP *m, int maxModSize, int maxModNNZ);

/* Jacobian flag functions. */
void MCP_Jacobian_Structure_Constant(MCP *m, int b);
void MCP_Jacobian_Data_Contiguous(MCP *m, int b);
void MCP_Jacobian_Diagonal(MCP *m, int b);
void MCP_Jacobian_Diagonal_First(MCP *m, int b);

/* Interface functions. */
void MCP_GetInterface(MCP *m, MCP_Interface *i);
void MCP_SetInterface(MCP *m, MCP_Interface *i);

void MCP_GetPresolveInterface(MCP *m, Presolve_Interface *i);
void MCP_SetPresolveInterface(MCP *m, Presolve_Interface *i);

/* Access routines. */
double *MCP_GetX(MCP *m);
double *MCP_GetL(MCP *m);
double *MCP_GetU(MCP *m);
int *   MCP_GetB(MCP *m);

double *MCP_GetF(MCP *m);
void    MCP_GetJ(MCP *m, int *nz, int **col, int **len,
                 int **row, double **data);
```

Figure 15: Header file, `MCP.h`

Table 9: Initialization routines

Function	Description
MCP_Create	Allocate an MCP structure with n variables and nnz elements in the Jacobian
MCP_Destroy	Deallocate an MCP
MCP_Size	Resize the MCP structure to have n variables and nnz elements in the Jacobian

variables or nonzeros in a particular problem instance.

The set of functions described in Table 10 provide information about the Jacobian matrix, which is then used to improve the performance of operations internal to the solver. The most important of these functions is `MCP_Jacobian_Structure_Constant` which tells the library whether the structure of the Jacobian remains constant or changes depending on the input vector. If set to a nonzero value, `col`, `len`, and `row` are assumed to be a constant compressed sparse column representation of the Jacobian. *In order to use the preprocessor developed in Chapter 4, a constant Jacobian structure is required.* The remaining flags indicate other features of the Jacobian. Setting `MCP_Jacobian_Data_Contiguous` to a nonzero value means that the compressed sparse column representation of the Jacobian is in a contiguous block. That is, there are no unused elements in the `row` and `data` vectors. The two remaining functions, `MCP_Jacobian_Diagonal` and `MCP_Jacobian_Diagonal_First`, should be set if each column has an entry on the diagonal and if the diagonal element appears as the first entry in each column respectively.

The access routines listed in Table 11 are used to obtain information about the specified model. For example, after the model is solved, a user would typically like to know the answer. The solution can be obtained by using the `MCP_GetX` routine, which returns a vector containing the current iterate, and the `MCP_GetF` routine, which returns

Table 10: Jacobian information routines

Function	Description
MCP_Jacobian_Data_Contiguous	Inform algorithm that the compressed sparse column representation of the Jacobian is in a contiguous block.
MCP_Jacobian_Diagonal	Inform algorithm that each column has an entry on the diagonal.
MCP_Jacobian_Diagonal_First	Inform algorithm that the first element in each column is the diagonal entry if it exists.
MCP_Jacobian_Structure_Constant	Inform algorithm that the structure of the Jacobian matrix remains constant

Table 11: Access functions

Function	Description
MCP_GetB	Obtain the current basis information
MCP_GetF	Obtain the function evaluation at the current iterate
MCP_GetJ	Obtain the Jacobian evaluation at the current iterate
MCP_GetL	Obtain the lower bounds
MCP_GetU	Obtain the upper bounds
MCP_GetX	Obtain the current iterate

a vector containing the function evaluation at the current iterate. The other routines provide the Jacobian evaluation at the current iterate in a compressed sparse column format, `MCP_GetJ`, and the lower and upper bounds, `MCP_GetL` and `MCP_GetU` respectively. `MCP_GetB` returns basis information when using an active-set method. For each variable, the vector indicates the active-set with the flags described in Table 12. These indicators can then be used to "warm" start the algorithm when solving related problem instances.

Table 12: Basis information codes

Identifier	Meaning
Basis_Basic	Variable is between lower and upper bounds and basic.
Basis_Fixed	Variable was fixed by the preprocessor and removed from the model.
Basis_Lower	Variable is fixed at its lower bound.
Basis_Upper	Variable is fixed at its upper bound.
Basis_Superbasic	Variable is between lower and upper bounds, but non-basic. Typically, this type is caused by singularity in the Jacobian matrix.
Basis_Unknown	Algorithm does not return basis information.

Table 13: Interface functions

Function	Description
MCP_SetInterface	Set the `MCP_Interface` for the problem.
MCP_SetPresolveInterface	Set the `Presolve_Interface` for the problem.
MCP_GetInterface	Obtain the `MCP_Interface` for the problem.
MCP_GetPresolveInterface	Obtain the `Presolve_Interface` for the problem.

Finally, the functions used to bind interface structures to the complementarity problem are given in Table 13. The key functions are `MCP_SetInterface`, which is used to specify the problem instance, and `MCP_SetPresolveInterface`, which is used to supply the additional information needed for the preprocessor. The contents of these two structures are discussed in the sequel.

3.2.1 MCP_Interface

All of the required problem information is conveyed in the `MCP_Interface` structure declared in Figure 16. The user allocates an interface structure and fills in the function pointers for their particular application. These functions are then used by the algorithm to obtain problem information. The `interface_data` can be any user allocated data and is passed as the first argument to all of the functions.

Some of the functions alluded to in the declaration of `MCP_Interface` are required while others are optional. The user must provide implementations for the `problem_size`, `bounds`, `function_evaluation`, and `jacobian_evaluation` functions described in Table 14. The number of domain violations encountered during the evaluation should be returned by the `function_evaluation` and `jacobian_evaluation` routines. A domain violation occurs, for example, when we attempt to take the log of a negative number or we encounter division by zero. The Jacobian matrix is stored in a compressed sparse column format with the indices written in the FORTRAN style. That is, each index element

```
typedef struct
{
     void *interface_data;

     void (*problem_size)(void *id, int *size, int *nnz);
     void (*bounds)(void *id, int size,
                    double *x, double *l, double *u);

     int (*function_evaluation)(void *id, int n, double *x,
                                double *f);
     int (*jacobian_evaluation)(void *id, int n, double *x,
                                int wantf, double *f,
                                int *nnz, int *col, int *len,
                                int *row, double *data);

     void (*start)(void *id);
     void (*finish)(void *id, double *x);
     void (*variable_name)(void *id, int variable,
                           char *buffer, int buf_size);
     void (*constraint_name)(void *id, int constr,
                             char *buffer, int buf_size);
     void (*basis)(void *id, int size, int *basX);
} MCP_Interface;
```

Figure 16: MCP_Interface declaration

Table 14: Required functions for `MCP_Interface`

Function	Description
problem_size	Give the size of the problem and number of nonzeros in the Jacobian.
bounds	Give the lower and upper bounds and a starting point for the problem.
function_evaluation	Evaluate the function for a point z. Return the number of domain violations.
jacobian_evaluation	Evaluate the function and Jacobian for a point z. Return the number of domain violations. The Jacobian is stored in a compressed column format. Initially nnz is the allocated size for the row and data arrays. The actual number of nonzeros in the Jacobian should be supplied before returning.

takes on a value between one and the number of indices. Furthermore, the structure of the sparse matrix need only be determined and allocated once. We guarantee that the algorithms will not alter the structure.

The user can optionally provide the functions documented in Table 15. The `start` and `finish` routines are called exactly once at the beginning and the end of the computation respectively. For example, an implementation of the `start` function can be used to perform allocation and initialization at the beginning of the solve, while the `finish` function can deallocate. The `variable_name` and `constraint_name` are used to provide names for the variables and constraints. The indices passed into the functions are in FORTRAN style. For example, `var` will take on a value between 1 and the number of variables in the problem. If not provided, the defaults names are "x" and "f" for the variables and constraints. Finally, the `basis` function can be used to specify basis information at the start of the computation. This information can help an active-set type method to identify the correct active-set at the beginning of the solve, leading to improved performance. The indicators used for this purpose are given in Table 12.

Table 15: Optional functions for `MCP_Interface`

Function	Description
start	Function called at the start of the solve.
finish	Function called at the end of the solve with the final iterate.
variable_name	Provide a name for the specified variable.
constraint_name	Provide a name for the specified constraint.
basis	Used to provide basis information at the start of the computation.

```
typedef struct
{
     void *presolve_data;

     void (*start_pre)(void *pd);
     void (*start_post)(void *pd);

     void (*finish_pre)(void *pd);
     void (*finish_post)(void *pd);

     void (*jac_typ)(void *pd, int nnz, int *typ);
     void (*con_typ)(void *pd, int n, int *typ);
} Presolve_Interface;
```

Figure 17: `Presolve_Interface` declaration

3.2.2 Preprocessing_Interface

The final component of the `MCP` specification is the `Presolve_Interface`, which is used to provide additional information about the model needed for preprocessing. *Note that use of the preprocessor requires that the Jacobian structure remain constant, which is communicated with the `MCP_Jacobian_Structure_Constant` function.* The declaration for the `Presolve_Interface` is found in Figure 17. The `presolve_data` component can be any user allocated data and is passed as the first argument to all of the presolve interface functions.

The only required function, `jac_typ`, fills in the type of each element in the Jacobian.

The defined symbols, PRESOLVE_LINEAR and PRESOLVE_NONLINEAR, are used to indicate linear and nonlinear elements respectively. The elements in the typ vector correspond to the compressed sparse column representation of the Jacobian. For example, in Figure 18 we give an implementation of the presolve interface for linear complementarity problems. Note that in this example, we have assumed that the Jacobian elements are stored in a contiguous block, leading to the simple loop indicating that each element is linear.

Additionally, if your problem comes from the Karush-Kuhn-Tucker conditions for a variational inequality or nonlinear program, the primal and dual constraints can be specified using the con_typ function. PRESOLVE_PRIMAL and PRESOLVE_DUAL, are the defined symbols used to indicate which of the constraints are primal and dual. If a constraint is not a primal or dual constraint, the PRESOLVE_UNKNOWN symbol can be used. The con_typ function is optional and does not need to be specified for the presolve to work.

The four remaining routines are called when the presolve phase starts and finishes, start_pre and finish_pre, and when the postsolve phase starts and finishes, start_post and finish_post.

3.3 Solver Structures

The next ingredients to the interface are the structures and functions associated with the particular algorithms. The entry routine for solving an MCP with PATH 4.x is declared as:

```
MCP_Termination Path_Solve(MCP *m, Information *info);
```

The corresponding declaration for the semismooth algorithm has identical arguments but is called Semi_Solve. MCP_Termination, is an enumerated type that indicates the

```
static void mcp_typ(void *d, int nnz, int *typ)
{
     int i;

     for (i = 0; i < nnz; i++)
     {
          typ[i] = PRESOLVE_LINEAR;
     }
     return;
}

static Presolve_Interface mcp_presolve =
{
     NULL,
     NULL, NULL,
     NULL, NULL,
     mcp_typ,
     NULL
};
```

Figure 18: Presolve for linear complementary problems

Table 16: `MCP_Termination` codes

Identifier	Meaning
MCP_Solved	A solution to the problem was found.
MCP_NoProgress	The algorithm stopped because it could not improve upon the current iterate.
MCP_MajorIterationLimit	An iteration limit was reached.
MCP_MinorIterationLimit	The minor iteration limit was reached.
MCP_TimeLimit	Time limit was exceeded.
MCP_UserInterrupt	The user requested that the solver stop execution.
MCP_BoundError	The problem is infeasible because $\ell_l > u_i$ for some components.
MCP_DomainError	A starting point where the function is defined could not be found.
MCP_Infeasible	The preprocessor determined the problem is infeasible.
MCP_Error	An internal error occurred in the algorithm.

Table 17: Key fields in `Information` structure

Field	Meaning
residual	Value of residual at final point.
major_iterations	Major iterations taken.
minor_iterations	Minor iterations taken.
crash_iterations	Crash iterations taken.
function_evaluations	Function evaluations performed.
jacobian_evaluations	Jacobian evaluations performed.
gradient_steps	Gradient steps taken.
restarts	Restarts used.

termination status for the algorithm. The identifiers are documented in Table 16. The `Information` structure contains statistics about the solution process. The key fields are documented in Table 17.

In addition, each algorithm defines a function used to install their specific options and check the license. For PATH 4.x, these function declarations are:

```
void Path_AddOptions(Options_Interface *i);
int  Path_CheckLicense(int n, int nnz);
```

The corresponding declaration for the semismooth algorithm has identical arguments but is called `Semi_AddOptions`. The next section describes how the options are used, while

```
Options_Interface *Options_Create();
void Options_Destroy(Options_Interface *i);
void Options_Add(Options_Interface *i, Option_Set *opt);
void Options_Default(Options_Interface *i);

void Options_Display(Options_Interface *i);
void Options_Read(Options_Interface *i, char *filename);

void Options_Set(Options_Interface *i, char *opt);
void Options_SetBoolean(Options_Interface *i, char *opt, int val);
void Options_SetInt(Options_Interface *i, char *opt, int val);
void Options_SetDouble(Options_Interface *i, char *opt, double val);
void Options_SetOther(Options_Interface *i, char *opt, int val);

int    Options_GetBoolean(Options_Interface *i, char *opt);
int    Options_GetInt(Options_Interface *i, char *opt);
double Options_GetDouble(Options_Interface *i, char *opt);
int    Options_GetOther(Options_Interface *i, char *opt);
```

Figure 19: Header file `Options.h`

the following section describes the license checking functionality.

3.3.1 Options

Associated with each algorithm is a set of options that can be used to modify the algorithm's behavior. Figure 19 contains the declaration for the header file associated with the options and Table 18 discusses the functions for setting and obtaining option values.

Each algorithm has a particular set of options, contained in an `Options_Interface`, obtained using the `Options_Create` function. The correct sequence for obtaining an algorithm specific option set is:

```
Options_Interface i;

i = Options_Create();
Path_AddOptions(i);
```

Table 18: Functions for dealing with options

Function	Description
Options_Add	Add user defined options to the option set.
Options_Create	Create a new option set.
Options_Default	Set the options to their default values.
Options_Destroy	Deallocate an option set.
Options_Display	Display the values of the options.
Options_GetBoolean	Obtain the value for the named boolean option.
Options_GetInt	Obtain the value for the named integer option.
Options_GetDouble	Obtain the value for the named double precision option.
Options_GetOther	Obtain the value for the name option that takes on a different value.
Options_Read	Read the options from the given file.
Options_Set	Use the string to set an option value. The string has the form "<option> <value>".
Options_SetBoolean	Set the named boolean options to the value specified.
Options_SetInt	Set the names integer option to the value specified.
Options_SetDouble	Set the named double precision option to the value specified.
Options_SetOther	Set the named other option to the value specified.

```
...
Options_Destroy(i);
```

where `Path` can be replaced with any of the other available solvers. The interface created is the first argument to all of the option functions. Components in the option names are separated by underscore characters (_). Only the first three characters of each component are relevant. Therefore, to change the `convergence_tolerance` option to 10^{-4}, we can issue the command:

```
Options_SetDouble(i, "con_tol", 1e-4);
```

Complete lists of the options for PATH 4.x and SEMI can be found in the available user documentation.

3.3.2 License Management

Users will need to specify licensing information prior to using PATH 4.x or SEMI to solve any complementarity problems. Licensing information is specified by a user either through the `License_SetString` function call or by setting the `PATH_LICENSE_STRING` environment variable. The format for the license string is the same for both methods. For example,

```
setenv PATH_LICENSE_STRING \
"2866407532&Evaluation&&&USR&&10_12_2001&1000&PATH&GEN&1_3_2002&1_3_2002&0&0_0"
```

will set the `PATH_LICENSE_STRING` environment variable on a Solaris or Linux system to an evaluation license. Alternatively, in the interface code, the same license can be set with with the following function call

```
License_SetString("2866407532&Evaluation&&&USR&&"
                  "10_12_2001&1000&PATH&GEN&"
                  "1_3_2002&1_3_2002&0&0_0");
```

Additional functions are found in Figure 20 which contains the declaration for the header file associated with license management. Table 19 discusses the functions for setting and obtaining particular values. All of these functions return a nonzero value on successful completion and a zero value on failure.

Each algorithm has a method for explicitly checking the license. The declaration is

```
int Path_CheckLicense(int n, int nnz)
```

where `Path` can be replaced with any of the other available solvers. A nonzero return value indicates that the license check succeeded, while a zero value indicates that the license check failed.

```
int License_SetString(const char *str);

int License_GetAlgorithms(char *algs, int len);
int License_GetArchitectures(char *archs, int len);
int License_GetExpiration(int *day, int *month, int *year);
int License_GetMaxVersion(int *ver);
int License_GetMaxBuild(int *day, int *month, int *year);
int License_GetRestrictions(int *n, int *nnz);

int License_GetDemoRestrictions(int *n, int *nnz);

License_Termination License_GetTermination(void);
License_Termination License_GetDemoTermination(void);
```

Figure 20: Header file `License.h`

Table 19: Functions for dealing with license management

Function	Description
License_GetAlgorithms	Obtain the algorithms licensed. Each algorithm is specified by a sequence of four characters. All of the licensed algorithms are concatenated together in the string returned.
License_GetArchitectures	Obtain the architectures licensed. Each architecture is specified by a sequence of three characters. All of the licensed architectures are conctenated together in the string returned.
License_GetDemoRestrictions	Obtain the size and number of nonzero restrictions for demonstration licenses.
License_GetDemoTermination	Obtain the termination status for the last time the demonstration license was checked.
License_GetExpiration	Obtain the expiration date for the license.
License_GetMaxBuild	Obtain the maximum build date licensed for the library.
License_GetMaxVersion	Obtain the maximum version number licensed for the library.
License_GetRestrictions	Obtain the size and number of nonzero restrictions for the license.
License_GetTermination	Obtain the termination status for the last time the license was checked.
License_SetString	Set the string containing the license information.

3.3.3 Workspace Allocation

A workspace can be allocated for each algorithm which will be used on all subsequent calls to the algorithm. The functions needed are:

```
void Path_Create(int maxSize, int maxNNZ);
void Path_Destroy();
```

which will allocate and deallocate the workspaces for the PATH 4.x algorithm. For the semismooth algorithm, we can use the `Semi_Create` and `Semi_Destroy` functions. These workspace allocation routines are particularly useful when solving multiple complementarity problems in the same session because the workspaces will be reused. That is, memory allocation will only be performed once.

3.4 Control Structures

The control structures are used to change the error reporting, interrupt handling, memory allocation, output, and timing mechanisms. We discuss each of these control structures in the sequel and provide examples of their use.

3.4.1 Error

The functions contained in the `Error_Interface` structure are used to provide information concerning warnings and errors. Figure 21 contains the declaration for the component and Table 20 describes each of the functions. A warning tells the user about difficulties or nonstandard events encountered. At the end of the warning, control should be returned back to the algorithm. An error on the other hand is fatal. Execution stops at the end of the error routine.

```
typedef struct
{
    void *error_data;

    void (*error)(void *ed, char *msg);
    void (*warning)(void *ed, char *msg);
} Error_Interface;

void Error_SetInterface(Error_Interface *i);
```

Figure 21: `Error_Interface` declaration

Table 20: Error Object Functions

Function	Description
error	(Optional) An error has been generated. Do something with the information and exit from the program.
warning	(Optional) A warning has been generated. Do something with the information and return to the code.

An interesting contrast in the error function implementation can be found by comparing a standard implementation, which uses `exit`, to the MATLAB interface error routine, which uses `mexErrMsgTxt`. The code for the MATLAB `Error_Interface` can be found in Figure 22. When `exit` is used, the operating system frees all previously allocated memory and the program terminates. In the case of MATLAB, we need to relinquish all allocated memory before returning control back to MATLAB by using the `mexErrMsgTxt` routine. The use of `exit` is entirely inappropriate within a MATLAB session. Note that warnings are ignored in this implementation.

3.4.2 Interrupts

Interrupts can be used to terminate an algorithm before it has finished computing a solution. Figure 23 contains the declaration for the `Interrupt_Interface` structure and

```
#include <stdio.h>
#include "Error_Interface.h"
#include "mex.h"

static void mat_error(void *ed, char *msg)
{
     mexErrMsgTxt(msg);
     return;
}

static Error_Interface mat_error_interface =
{
     NULL,
     mat_error,
     NULL
};
```

Figure 22: MATLAB implementation of the `Error_Interface`

Table 21: Interrupt Object Functions

Function	Description
set	(Required) Turn the interrupt handler on.
restore	(Required) Turn the interrupt handler off and restore the default handler.
check	(Required) Check to see whether an interrupt has been issued.

Table 21 describes each of the functions. The default implementation uses the `signal` standard function and issues an `Error` if a user repeatedly types CTRL-C.

3.4.3 Memory

The memory structure contains all of the necessary functions to allocate and relinquish memory. Figure 24 contains the declaration for the `Memory_Interface` structure and Table 22 describes each of the functions. We distinguish between two different types of memory allocation: general memory allocation using `allocate` and the allocation necessary for the basis factors, `allocate_factors`. The general memory allocation routine

```
typedef struct
{
    void *interrupt_data;

    void (*set)(void *id);
    void (*restore)(void *id);
    int  (*check)(void *id);
} Interrupt_Interface;

void Interrupt_SetInterface(Interrupt_Interface *i);
```

Figure 23: `Interrupt_Interface` declaration

```
typedef struct
{
     void *memory_data;

     void * (*allocate)(void *data, long int n);
     void * (*allocate_factors)(void *data, long int n);
     void   (*free)(void *data, void *v);
     void   (*free_factors)(void *data, void *v);
} Memory_Interface;

void Memory_SetInterface(Memory_Interface *i);
```

Figure 24: `Memory_Interface` declaration

is frequently called upon to allocate relatively small pieces of memory. The factor allocation requires a single large section of memory to be obtained. Mechanisms optimized for these differing types of memory request patterns can be written. We guarantee that within the algorithm only one set of factors will be allocated at a time. However, the sequence `allocate_factors`, `free_factors` may be repeated within the code.

A standard implementation for the memory component uses the routines `malloc` and `free`. More sophisticated implementations are possible. For example, the GAMS

Table 22: Memory Object Functions

Function	Description
allocate	(Required) Allocate the specified number of bytes from memory and return a pointer to the allocated memory.
allocate_factors	(Required) Allocate the specified number of bytes of memory and return a pointer to the allocated memory. The amount of memory requested in `allocate_factors` is typically much greater than that requested in `allocate`.
free	(Required) Free the indicated memory allocated with `allocate`.
free_factors	(Required) Deallocate the indicated memory previously allocated by `allocate_factors`.

implementation of these routines places the factors on the GAMS heap, a portion of memory previously allocated as workspace for the MPSGE [113] preprocessor.

The MATLAB interface uses the `mxMalloc` routines for memory allocation. The implementation of the memory subsystem for MATLAB can be found in Figure 25. Note that in MATLAB, both the `allocate` and `allocate_factors` routines are the same. This implementation guarantees that MATLAB will free all of the memory allocated within the mex function on termination.

3.4.4 Output

The output interface is key in order to support both C and FORTRAN output. Figure 26 contains the declaration for the `Output_Interface` structure and Table 23 describes each of the functions. The default implementation uses the `fprintf` command to write the messages to the indicated files. The mode passed into the function contains the descriptors:

Output_Log The log is used to demonstrate that the algorithm is making progress.

```
#include <stdlib.h>
#include "Memory_Interface.h"
#include "mex.h"

static void *mat_allocate(void *md, long int n)
{
     void *v;

     v = (void *)mxMalloc(n);
     return v;
}

static void mat_free(void *md, void *v)
{
     mxFree(v);
     return;
}

static Memory_Interface mat_memory_interface =
{
     NULL,
     mat_allocate,
     mat_allocate,
     mat_free,
     mat_free
};
```

Figure 25: Memory subsystem implementation for MATLAB

```
#define Output_Log 1
#define Output_Status 2
#define Output_Listing 4

typedef struct
{
     void *output_data;

     void (*print)(void *od, int mode, char *msg);
     void (*flush)(void *od, int mode);
} Output_Interface;

void Output_SetLog(FILE *f);
void Output_SetStatus(FILE *f);
void Output_SetListing(FILE *f);

void Output_SetInterface(Output_Interface *i);
```

Figure 26: `Output_Interface` declaration

Output_Status The status file is used for debugging purposes and records the essential information for this task.

Output_Listing Output specified for listing file.

These descriptors are combined together with the 'or' operator to indicate the locations for the output. That is

```
Output_Log | Output_Status
```

should send the message to both the log and status files. Note that the `Output_SetLog`, `Output_SetStatus`, and `Output_SetListing` are convenience routines for use with the default implementation.

Table 23: Output Object Functions

Function	Description
flush	(Optional) Flush the output for each of the specified files.
print	(Required) Output the indicated message to each of the specified files.

```
typedef struct
{
    void *timer_data;

    void * (*create)(void *td);
    void   (*destroy)(void *td, void *timer);
    void   (*start)(void *td, void *timer);
    double (*query)(void *td, void *timer);
} Timer_Interface;

void Timer_SetInterface(Timer_Interface *p);
```

Figure 27: `Timer_Interface` declaration

3.4.5 Timer

The final control structure is used to determine the amount of time spent in particular sections of the code. Figure 27 contains the declaration for the `Timer_Interface` structure and Table 24 describes each of the functions. The default implementation uses the `clock` standard function.

Table 24: Timer Object Functions

Function	Description
create	(Required) Allocate and return a time structure.
destroy	(Required) Free the indicated time structure.
start	(Required) Place the correct value for the current time in the indicated structure.
query	(Required) Return the number of seconds elapsed since the indicated structure has been started.

3.5 Driver

Once the interfaces have been written and a solver has been chosen, a main driver routine must be specified. Pseudocode for such a routine follows:

```
1. Initialize control components.
2. Create options set and default.
3. Initialize user defined components.
4. Work with the options.
5. Create and setup an MCP structure.
6. Solve the problem.
7. Do something with the results.
8. Destroy the MCP structure.
9. Destroy the option set.
```

If special setups need to be performed for the system-dependent parts of the code, they are done at the beginning.

As a concrete illustration, we have written a simple interface for solving a convex quadratic program using PATH 4.x or SEMI. The quadratic program is to solve the minimization problem:

$$\begin{array}{rc} \min_x & \frac{1}{2}x^TQx + c^Tx \\ \text{subject to} & Ax \geq b \\ & \ell \leq x \leq u \end{array} \tag{13}$$

where $Q \in \mathbf{R}^{n\times n}$ is a symmetric positive semidefinite matrix. In order to find a solution, we write the Karush-Kuhn-Tucker optimality conditions [78, 79] for (13) as the complementarity problem:

$$\begin{array}{lcl} \ell \leq x \leq u & \perp & Qx - A^T\mu + c \\ 0 \leq \mu & \perp & Ax - b \geq 0 \end{array}$$

A `QPMain` routine has been written which takes as inputs Q and A in (row,col,val) format and the vectors, c, b, ℓ, u. An additional vector is passed along, `c_type`, telling the orientation of the constraints. Note that the `QP_Create` routine used in the following implementation simply forms the KKT conditions.

```
void QPMain(int variables, int constraints,
            int q_nz, int *q_i, int *q_j, double *q_ij, double *c,
            int a_nz, int *a_i, int *a_j, double *a_ij, double *b,
            int *c_type, double *lb, double *ub,
            QP_Termination *status,
            double *z, double *mu, double *obj)
{
     Options_Interface *o;
     MCP *m;
     MCP_Termination t;
     Information info;

     double *x;

     /* 1.  Initialize control components */
     /* 2.  Create options set and default */

     o = Options_Create();
     Path_AddOptions(o);
     Options_Default(o);

     /* 3. Initialize user defined components */

     QP_Create(variables, constraints,
               q_nnz, q_i, q_j, q_ij, c,
               a_nnz, a_i, a_j, a_ij, b, c_type,
               z, mu, lb, ub);


     /* 4.  Work with the options */

     Options_Read(o, "path.opt");
     Options_Display(o);

     /* 5.  Create and setup an MCP structure */

     m = MCP_Create(problem.n, problem.nnz+1);
     MCP_Jacobian_Structure_Constant(m, 1);
     MCP_SetInterface(m, &mcp_interface);
```

```
    MCP_SetPresolveInterface(m, &mcp_presolve);

    /* 6.  Solve the problem */

    t = Path_Solve(m, &info);

    /* 7.  Do something with the results */

    if (t == MCP_Solved)
    {
        *status = QP_Solved;
    } else
    {
        /* code omitted */
    }

    x = MCP_GetX(m);

    for (i = 0; i < variables; i++)
    {
        z[i] = x[i];
    }

    for (i = 0; i < constraints; i++)
    {
        mu[i] = x[i + variables];
    }

    /* code omitted */

    /* 8.  Destroy the MCP structure */

    MCP_Destroy(m);
    QP_Destroy();

    /* 9.  Destroy the option set */

    Options_Destroy(o);
    return;
}
```

3.6 Summary

This chapter documented the solver interface to PATH 4.x and SEMI used to support the environments in Chapter 2 as well as simplified subroutine calls and a specialized convex quadratic programming interface. The main documentation for the interface was split into three parts: the problem structures used to specify a problem and its characteristics, the solver structures used to invoke a particular algorithm, and the control structures used to control output, memory allocation, and interrupt handling. We then demonstrated the usage of the interface with a quadratic programming interface. In particular, the solver interface described can be used by applications programmers to make PATH 4.x and SEMI available in other environments.

We remark that an earlier specification of the solver interface appeared in [42]. The current solver interface specification, simplified subroutine call, and convex quadratic programming interface are new to this thesis. A FORTRAN equivalent of the subroutine libraries is also available where the main program and function and Jacobian evaluation routines are written in FORTRAN. Finally, to support loadable library versions of the PATH 4.x and SEMI codes, function pointers were used in the solver interface specification so that implementations of the control structures can be set at run time.

Chapter 4

Preprocessing

The benefits of preprocessing have long been known to the linear [1, 10] and mixed integer [115] programming communities, yet have not previously been studied from a complementarity perspective. In many cases, the preprocessor simply uncovers structure from a model that may not have been explicitly communicated. The reasons for this are varied. The modeler may be completely unaware of the structure, or unwilling or unable to pass the known structure to a solver. In other cases, the structure arises from simplifications due to particular data values. For example, if a general inequality constraint can be reduced to use only one variable, the solver might prefer to treat it as a bound constraint.

In general, the purpose of a preprocessor is to reduce the size and complexity of a model to achieve improved performance by the main algorithm. For example, by fixing variables, some nonlinear constraints may become linear, and smaller linear systems are typically solved with less memory and in less time. Another benefit of the analysis performed is the detection of model infeasibilities. Most codes continue attempting to solve a model until a time or iteration limit is reached. In cases where the model has no solution, this is obviously wasteful. A preprocessor can sometimes process the model enough to determine (and report) the source of the difficulty.

The preprocessor for complementarity problems developed in this chapter uses the box constrained variational inequality formulation [77, 105] of the mixed complementarity problem:

$$0 \in F(x) + N_{[\ell,u]}(x), \tag{14}$$

where $F : \mathbf{R}^n \to \mathbf{R}^n$ is continuously differentiable and

$$\ell \in \{\mathbf{R} \cup \{-\infty\}\}^n$$
$$u \in \{\mathbf{R} \cup \{+\infty\}\}^n$$

with $[\ell, u]$, the Cartesian product of the closed, not necessarily compact, intervals $[\ell_i, u_i]$. Recall that the normal cone [110] to the box $[\ell, u]$ can be determined as a Cartesian product, namely

$$N_{[\ell,u]}(x) = \prod_{i=1}^{n} N_{[\ell_i,u_i]}(x_i).$$

Each component in this product depends on x_i, ℓ_i and u_i but is either $\mathbf{R}$, $\mathbf{R}_+$, $\mathbf{R}_-$, $\{0\}$, or $\emptyset$. In particular,

$$N_{[\ell_i,u_i]}(x_i) = \begin{cases} \mathbf{R} & \text{if } \ell_i = x_i = u_i \\ \mathbf{R}_- & \text{if } \ell_i = x_i < u_i \\ \{0\} & \text{if } \ell_i < x_i < u_i \\ \mathbf{R}_+ & \text{if } \ell_i < x_i = u_i \\ \emptyset & \text{otherwise.} \end{cases}$$

See Definition 1.1.5 for a specification of the normal cone for a general closed convex set.

The preprocessor works upon two equivalent representations of the same complementarity problem. To understand the basic methodology developed, consider a standard convex quadratic programming problem:

$$\begin{array}{ll} \min & \frac{1}{2}x^TQx + c^Tx \\ \text{s.t.} & Ax \geq b \\ & x \geq 0, \end{array} \tag{15}$$

where $Q \in \mathbf{R}^{n\times n}$ is a symmetric positive semidefinite matrix, $A \in \mathbf{R}^{m\times n}$, $b \in \mathbf{R}^m$, and $c \in \mathbf{R}^n$. The quadratic program in (15) can be posed as a variational inequality in one of two ways. First, when dual variables, λ, are introduced, the complementary slackness conditions for quadratic programs form the box constrained variational inequality:

$$0 \in \begin{bmatrix} Q & -A^T \\ A & 0 \end{bmatrix} \begin{bmatrix} x \\ \lambda \end{bmatrix} + \begin{bmatrix} c \\ -b \end{bmatrix} + \begin{bmatrix} N_{\mathbf{R}^n_+}(x) \\ N_{\mathbf{R}^m_+}(\lambda) \end{bmatrix}. \tag{16}$$

Alternatively, the first order conditions can be succinctly written as a polyhedrally constrained variational inequality:

$$0 \in Qx + c + N_C(x), \tag{17}$$

where $C = \{x \mid Ax \geq b, x \geq 0\}$. Since C is a geometric object, a computationally attractive algebraic representation can be chosen for C. Exploiting two representations of the mixed complementarity problem (analogous to (16) and (17) for quadratic programs) is a key concept in the preprocessor.

The complementarity problem is communicated as the box constrained variational inequality (14). Therefore, any polyhedral structure will need to be recovered from the problem description. Once recovered, the preprocessor automatically rewrites the problem in the form

$$0 \in H(x) + N_C(x) \tag{18}$$

for a polyhedral set C. Both (14) and (18) are used in distinct phases of the preprocessor. For example, the general inequalities in the set C can be used to modify the bounds ℓ_i and u_i on a variable x_i. In particular, if $x_i = \ell_i = u_i$ then $N_{[\ell_i,u_i]}(x_i) = \mathbf{R}$. Hence, fixing a variable x_i means that the corresponding constraint

$$0 \in F_i(x) + N_{[\ell_i,u_i]}(x_i) \iff 0 \in F_i(x) + \mathbf{R}$$

is trivially satisfiable. Thus, preprocessing in the complementarity case attempts to fix variables and thus remove constraints.

Section 4.1 begins by detailing the process used to uncover and exploit polyhedral structure in an MCP. The general idea is to reformulate (14) in a form similar to (17), with a general polyhedral set C replacing $[\ell, u]$. The representation of the set C can then be modified by either removing constraints or bounding variables. When converted back to a mixed complementarity problem a reduction in the number of variables is realized. Note that the process developed in this section recovers most checks done by traditional linear programming codes [1] when given the complementary slackness necessary and sufficient conditions for linear programs, but is applicable to a larger class of problems.

Further reductions to the MCP can be made by utilizing information about F and its Jacobian, F', as developed in Section 4.2. In particular, the range of F is used to eliminate variables from the model, row and column duplicates can also be removed, and by detecting special block structure, a sequence of smaller problems can be solved to find an answer to the original problem.

Both phases are incorporated into a complete preprocessor for mixed complementarity problems in Section 4.3. Computational results for some test problems are presented indicating the success of the procedure outlined.

More information about the problem must be provided to the preprocessor than is necessary to solve it. The basic requirement is a listing of the linear and nonlinear elements in the Jacobian of F, which are conveyed to the solver using the `Presolve_Interface` from Chapter 3. This knowledge is sufficient to find and utilize special structures. The AMPL [53] and GAMS [11] environments, and the `pathlcp` and `semilcp` functions in MATLAB already provide this information. Users of other interfaces, such as NEOS [40] and the `pathmcp` and `semimcp` functions in MATLAB, will need to develop the appropriate routines. Some checks in Section 4.2 based on the nonlinear functions need to know the range of F over $[\ell, u]$. Routines to calculate these intervals are not currently provided by any of the interfaces.

4.1 Polyhedral Constraints

The first stage of the preprocessor detects polyhedral structure in a mixed complementarity problem and exploits it by transforming the source problem into a model of lower dimension where C is the intersection of a closed product of intervals and a polyhedral set. The representation of C is then modified by removing constraints and changing bounds with the resultant MCP having fewer variables. After the preprocessed model has been solved, a solution to the original MCP is recovered with a postsolve step.

4.1.1 Presolve

Polyhedral structure can be exploited when given a special type of complementarity problem. Suppose the variables can be split into (x, y) and (14) has the form:

$$0 \in \begin{bmatrix} F(x) - A^T y \\ Ax - b \end{bmatrix} + \begin{bmatrix} N_X(x) \\ N_Y(y) \end{bmatrix}, \tag{19}$$

where $F : \mathbf{R}^n \to \mathbf{R}^n$ is continuously differentiable, $A \in \mathbf{R}^{m\times n}$, $b \in \mathbf{R}^m$, $X \subseteq \mathbf{R}^n$ is a Cartesian product of closed intervals, and $Y \subseteq \mathbf{R}^m$ is a Cartesian product of $\mathbf{R}$, $\mathbf{R}_+$, or $\mathbf{R}_-$. Note that if $Y_i = \{0\}$ then y_i can be fixed at zero and the corresponding $A_{i,\cdot}x - b_i$ removed. Further, if $Y_i = [\ell_i, \infty)$ or $Y_i = (-\infty, u_i]$ for some finite ℓ_i or u_i then an appropriate change of variables, possibly adding constant vectors to $F(x)$ and b, replaces Y_i with $\mathbf{R}_+$ and $\mathbf{R}_-$ respectively.

Theorem 4.1.1 *(Propositions 1 and 2 of [109]) Under the assumptions placed on X, Y, and the structure of the problem given above the following hold:*

(a) If $(\bar{x}, \bar{y})$ solves (19) then $\bar{x}$ is a solution to

$$0 \in F(x) + N_{X \cap \{x | b - Ax \in Y^o\}}(x), \tag{20}$$

where Y^o denotes the polar cone of Y which is defined as

$$Y^o := \{y \mid \langle y, \bar{y} \rangle \leq 0, \ \forall \bar{y} \in Y\}.$$

(b) If $\bar{x}$ solves (20) then the linear optimization problem

$$\begin{array}{rc} \min_{y \in Y} & \langle A\bar{x} - b, y \rangle \\ s.t. & 0 \in F(\bar{x}) - A^T y + N_X(\bar{x}). \end{array} \tag{21}$$

has a nonempty solution set. Further, for any $\bar{y}$ solving (21), $(\bar{x}, \bar{y})$ solves (19).

Theorem 4.1.1 provides the machinery used by the first stage of the preprocessor. The Jacobian matrix, F', is stored in both row- and column-oriented data structures. Utilizing the information provided about the types of the elements in the Jacobian, a row and column possessing the necessary skew symmetric structure of (19) can be quickly identified. Theorem 4.1.1 is then applied to this single row and column to create a problem of the form (20). The polyhedral set, $X \cap \{x \mid b - Ax \in Y^o\}$, is then checked for possible reductions, that is whether the general constraint, $b - Ax \in Y^o$, can be moved into the bound constraint, X. The new set $\tilde{X} \cap \{x \mid b - Ax \in \tilde{Y}^o\}$ is identical to $X \cap \{x \mid b - Ax \in Y^o\}$ but the MCP recovered using Theorem 4.1.1 on the reduced model is typically simpler. The identification and modification continues until no further simplifications are made. Note that Theorem 4.1.1 is only applied to a small number of constraints at a time during preprocessing while [109] uses the machinery to ready a problem for solution by a polyhedrally constrained variational inequality solver [116]. Automatically finding a set of polyhedral constraints with maximum size from (14) is a harder problem and is not considered here.

Information about any modifications performed are pushed onto a stack. The stack is a convenient data structure with two basic operations: pushing an element onto the top and popping an element from the top. Changes are pushed onto the stack in the order performed and are popped off the stack in the reverse order during the postsolve.

The complete algorithm for the first phase of the preprocessor is as follows:

Algorithm Presolve

(S.1) Mark all rows and columns with the skew symmetric structure as eligible *candidates, excluding any rows complementary to a variable with finite lower and upper bounds.*

(S.2) Using some ordering, pick one of the candidates and transform the problem into a polyhedrally constrained equation using Theorem 4.1.1.

(S.3) Analyze the polyhedral set and modify the representation as detailed below. Push any changes on top of the stack.

(S.4) Transform the modified problem back to box constrained form.

(S.5) Repeat steps (S.2)–(S.4) until there are no reductions possible.

The implementation performs all simple reductions (Section 4.1.1.1 and Section 4.1.1.2) first. Once all of these are completed, forcing constraints (Section 4.1.1.3) and redundant rows (Section 4.1.1.4) are checked. In a nonlinear model, additional rows and columns can become linear when variables are fixed. Therefore, after all tests are completed on the current list of eligible candidates, another pass is made through the Jacobian to mark new eligible rows and columns which are checked using (S.2)–(S.5) of Algorithm Presolve. When no new eligible rows and columns are created the process stops.

4.1.1.1 Simple Reductions

The simplest reduction made is when an eligible row contains zero elements. This situation corresponds to the case where the polyhedral set in the transformed problem is:

$$X \cap \{x \mid b \in Y^o\}. \tag{22}$$

If (22) is empty, then the original problem has no solution. Otherwise, the polyhedral component is irrelevant, because it does not impose any constraints on x, and (22) can be replaced with:

$$X \cap \{x \mid b \in \{0\}^o\}.$$

Note that $\{0\}^o = \mathbf{R}$, ensuring the constraint $b \in \{0\}^o$ is always satisfied. When transformed back to the original space, the corresponding multiplier is fixed at 0 and removed from the problem, resulting in a reduction of one variable and the corresponding constraint.

Another simple reduction occurs when the eligible row contains a single element. The polyhedral set in this case is:

$$X \cap \{x \mid b - ax_i \in Y^o\}. \tag{23}$$

Since Y is $\mathbf{R}_+$, $\mathbf{R}_-$, or $\mathbf{R}$, the constraint will be either $b - ax_i \leq 0$, $b - ax_i \geq 0$, or $b - ax_i = 0$ respectively. Each of these imply simple bounds on x_i, which can be explicitly incorporated in X. Therefore, (23) is replaced by

$$\tilde{X} \cap \{x \mid b - ax_i \in \{0\}^o\},$$

where $\tilde{X}$ includes the tightened bounds on x_i. This modification results in a reduction of at least one variable.

4.1.1.2 Doubleton Rows

Doubleton rows having the skew symmetric property can also be preprocessed. Consider a row, i, which is an equation of the form

$$ax_j + bx_k = c$$

where either x_j or x_k is a column singleton. Assume x_k is the column singleton. If row k also has the skew symmetric property, then X can be modified by changing the bounds on x_j to make x_k free. Immediately following this change, row k (which must be a singleton row complementary to the free variable x_k) is preprocessed out of the model using the singleton check described above.

4.1.1.3 Forcing Constraints

Forcing constraints are constraints for which, given the bounds on the variables, there is exactly one feasible point. Once it is known that only one solution is possible, all variables appearing in the constraint can be fixed, potentially leading to a large reduction in problem size.

Let the polyhedral constraint be written in the form:

$$X \cap \{x \mid b - a^T x \in Y^o\}. \tag{24}$$

Without loss of generality, assume that $Y = \mathbf{R}_+$ which means that $Y^o = \mathbf{R}_-$. Then (24) can then be explicitly stated as:

$$X \cap \{x \mid b \leq a^T x\}.$$

Using X, bounds, $\underline{a}$ and $\bar{a}$, can be implied such that $\underline{a} \leq a^T x \leq \bar{a}$ for all $x \in X$. The ranges are determined as follows:

$$\underline{a} = \sum_{\{i|a_i>0\}} a_i \ell_i + \sum_{\{i|a_i<0\}} a_i u_i$$
$$\bar{a} = \sum_{\{i|a_i>0\}} a_i u_i + \sum_{\{i|a_i<0\}} a_i \ell_i,$$

where ℓ_i and u_i are the lower and upper bounds on variable i respectively. If $\bar{a} < b$ the problem is infeasible. Otherwise, if $\bar{a} = b$, there is only one feasible point and the values of the variables are fixed at u_i for all i with $a_i > 0$ and ℓ_i for all i with $a_i < 0$. Set (24) is then replaced with:

$$\tilde{X} \cap \{x \mid b - a^T x \in \{0\}^o\}, \tag{25}$$

where $\tilde{X}$ contains the fixed variables. The net result is that the forcing constraint and a number of variables are removed from the original problem.

4.1.1.4 Redundant Rows

Redundancy in the Jacobian matrix can cause difficulty for many algorithms. Therefore, it is advantageous to remove as much redundancy as possible. The algorithm given in [117] is used to identify duplicate rows. All eligible constraints are checked simultaneously with the algorithm. After finding two duplicate rows, any inconsistencies are uncovered (meaning that the model is unsolvable) and one of the constraints is removed wherever possible. Without loss of generality, let the constraint set be written as

$$X \cap \left\{ x \,\middle|\, \begin{array}{l} b - ax \in Y_1^o \\ c - ax \in Y_2^o \end{array} \right\}. \tag{26}$$

Table 25: Redundant Rows Cases

Case	Action
$Y_1 = \mathbf{R}$ and $Y_2 = \mathbf{R}$	If $b = c$ remove one of the constraints. Otherwise the problem is infeasible.
$Y_1 = \mathbf{R}$ and $Y_2 = \mathbf{R}_+$	If $b \geq c$ remove the inequality constraint. Otherwise the problem is infeasible.
$Y_1 = \mathbf{R}_+$ and $Y_2 = \mathbf{R}_+$	If $b \geq c$ remove the constraint associated with Y_2. Otherwise remove the Y_1 constraint.
$Y_1 = \mathbf{R}_-$ and $Y_2 = \mathbf{R}_+$	If $b < c$, the problem is infeasible. Otherwise if $b = c$ make one of the constraints an equation and remove the other. Otherwise, it is a range constraint; nothing is done by the preprocessor.

Several cases are presented in Table 25 along with the associated actions taken. The other cases are symmetric to those given in the table.

4.1.1.5 Extensions

The requirements in Theorem 4.1.1 can be slightly weakened. Let $D \in \mathbf{R}^{n \times n}$ be a positive diagonal matrix. Then the following form will suffice instead of (19):

$$0 \in \begin{bmatrix} F(x) - DA^T y \\ Ax - b \end{bmatrix} + \begin{bmatrix} N_X(x) \\ N_Y(y) \end{bmatrix} \tag{27}$$

because (27) can be reduced to (19) by applying a diagonal row scaling of $\begin{bmatrix} D & 0 \\ 0 & 1 \end{bmatrix}^{-1}$ and recalling that the normal cone does not change under multiplication by a positive diagonal matrix.

Furthermore, free variables imply that the corresponding function is an equation because $N_{\mathbf{R}}(\cdot) \equiv \{0\}$. Therefore, the equation can be negated to obtain the required structure.

4.1.2 Postsolve

Once the algorithm has solved the preprocessed model, all of the presolve steps must be undone in the reverse order to recover a solution to the original model. The stack of presolve records is used for this purpose. The following algorithm is performed:

Algorithm Postsolve

(S.1) Remove a presolve record from the top of the stack.

(S.2) Transform the problem into the polyhedrally constrained setting.

(S.3) Undo the changes made to the model.

(S.4) Solve the optimization problem (21) using $\bar{x}$ to obtain $\bar{y}$. The generated $(\bar{x}, \bar{y})$ solves the model before the presolve step was performed.

(S.5) Repeat until the presolve stack is empty.

The optimization problem (21) is typically only in one dimension and is trivial to solve. Care must be taken when calculating $N_X(\bar{x})$ because the algorithm may only find a solution to within a prespecified tolerance. Therefore, variables within some tolerance of their bounds should be treated as if they are exactly on their bounds when constructing the normal cone. The current default used in the implementation is the convergence tolerance as specified by the algorithm.

The only case where two variables are involved in the optimization problem is when two inequalities are replaced by one equation. The optimization problem in this case has an objective function equal to zero because at the solution $A\bar{x} - b = 0$. Therefore, a feasible point need only be generated. In the presolved model, $0 \in F(x) - a\hat{y} + N_X(\bar{x})$ for the solution $(\bar{x}, \hat{y})$ given. Without loss of generality, assume $Y_1 = \mathbf{R}_+$ and $Y_2 = \mathbf{R}_-$.

Select $y_1 \in Y_1 = \mathbf{R}_+$ and $y_2 \in Y_2 = \mathbf{R}_-$ such that $y_1 + y_2 = \hat{y}$. These conditions can always be trivially met. Because the inclusion holds at $\hat{y}$, it also holds for the y_1 and y_2 selected, which is then a feasible point as required.

In the unfortunate case that the algorithm fails to solve the preprocessed model, the optimization problem may have no solution either because it is infeasible or unbounded. In this case, a value for the multiplier is chosen in Y such that the norm of the error in the box constrained representation is minimized given $\bar{x}$. This greedy heuristic will lead to the best possible value in terms of the residual at each stage in the unrolling of the preprocessing steps, but not necessarily the least residual solution overall.

4.2 Structural Implications

The second phase of the preprocessor utilizes complementarity theory to eliminate variables from the MCP. The reductions documented are based on the rows and columns of the Jacobian, F'. The main ingredient for the row-based rules is uniqueness. If the value for a variable can be uniquely determined prior to solving the remainder of the problem, it is fixed and removed. The column-based rules rely upon existence arguments. Once a solution to the reduced model is known, a solution to the original problem always exists and can be calculated. Mechanisms developed include using interval evaluations, uncovering duplicate rows and columns, and exploiting special structure. Note that when a row with zero elements and corresponding zero column are present in a model, the variable can always be fixed at an appropriate value and removed.

4.2.1 Intervals

An interval evaluator determines the tightest possible $\underline{F}$ and $\bar{F}$ such that for all $x \in X$, $\underline{F} \leq F(x) \leq \bar{F}$. For example, with a linear constraint, $F_i(x) = a^T x - b$, the bounds

$$\begin{aligned}
\underline{F}_i &= \sum_{\{j|a_j>0\}} a_j \ell_j + \sum_{\{j|a_j<0\}} a_j u_j - b \\
\bar{F}_i &= \sum_{\{j|a_j>0\}} a_j u_j + \sum_{\{j|a_j<0\}} a_j \ell_j - b
\end{aligned}$$

can be used where ℓ_j and u_j are the lower and upper bounds on variable j respectively. The range of a nonlinear function is dependent upon the model and the bounds must be computed by a user supplied routine.

Using the ranges, variables in the model can be fixed. If $\underline{F}_i > 0$, then x_i must be fixed at its finite lower bound or the problem is infeasible. Furthermore, if $\bar{F}_i < 0$, then x_i must be fixed at its finite upper bound or the problem is infeasible.

Some of the constraints in the model will imply tighter bounds on the variables; i.e. a linear constraint. These can be used by the interval evaluator to strengthen the ranges on other constraints, potentially leading to more variables being fixed. The tightened bounds are only used when calculating the intervals because modifying the bounds on the complementarity problem using this information could change the solution set of the problem.

4.2.2 Duplicates

Duplicate rows and columns can be very problematic for a solver. By applying the same algorithm used in the polyhedrally constrained case (Section 4.1.1.4), two such linear rows or columns can be identified. First consider the case of two duplicate rows in the

Table 26: Duplicate Rows Cases

Case	Action
$Y = \mathbf{R}$ and $Z = \mathbf{R}$	If $b \neq c$ the problem is infeasible. Otherwise, nothing is done.
$Y = \mathbf{R}$ and $Z = \mathbf{R}_+$	If $b > c$ fix z at its lower bound. Otherwise, if $b < c$, the problem is infeasible.
$Y = \mathbf{R}_+$ and $Z = \mathbf{R}_+$	If $b > c$ fix z at its lower bound. If $b < c$ fix y at its lower bound.
$Y = \mathbf{R}_-$ and $Z = \mathbf{R}_+$	If $b < c$, the problem is infeasible. Otherwise, nothing is done.

problem. Without loss of generality, the model can be written as:

$$0 \in \begin{bmatrix} F(x,y,z) \\ a^T(x,y,z) + b \\ a^T(x,y,z) + c \end{bmatrix} + \begin{bmatrix} N_X(x) \\ N_Y(y) \\ N_Z(z) \end{bmatrix}.$$

Table 26 discusses the reductions that can be made.

To remove column duplicates, one of the variables needs to be free and the other must have two finite bounds. The problem in this case is:

$$0 \in F(x) + ay + az + N_{X \times \mathbf{R} \times [\ell,u]}(x,y,z),$$

where ℓ and u are the finite lower and upper bounds on z. The reduction removes the z variable from the problem and solves the reduced system to obtain $(\bar{x}, \bar{y})$. If $F_z(\bar{x}) + a_z\bar{y} > 0$, then $\hat{z} = \ell$. Otherwise, $\hat{z} = u$. Set $\hat{y} = \bar{y} - \hat{z}$. Then $(\bar{x}, \hat{y}, \hat{z})$ solves the original problem as can be easily verified.

4.2.3 Special Structure

For a system of nonlinear equations, if the problem has the form:

$$\begin{bmatrix} F(x) \\ G(x,y) \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}$$

with $F : \mathbf{R}^k \to \mathbf{R}^k$, then $F(x) = 0$ can be solved giving $\bar{x}$ and then a $\bar{y}$ solving $G(\bar{x}, y) = 0$ can be found. If $F(x) = 0$ has multiple solutions, this procedure may fail by finding an $\bar{x}$ for which $G(\bar{x}, y) = 0$ has no solution. For example, consider $F(x) = x^2 - 1$ and $G(x, y) = x + y^2$. $F(x) = 0$ has two solutions $\bar{x} = 1$ and $\bar{x} = -1$. Choosing $\bar{x} = 1$ leads to the case where $G(\bar{x}, y)$ has no solution. If $F(x) = 0$ has at most one solution, this case is precluded provided the original problem has a solution. Similarly, the difficulty is alleviated if $G(\bar{x}, y)$ has a solution $\bar{y}$ for all $\bar{x}$, since whatever $\bar{x}$ is found, the system $G(\bar{x}, y) = 0$ is solvable. This section applies similar block reduction schemes to the mixed complementarity problem.

Consider a problem of the form:

$$0 \in \begin{bmatrix} F(x) \\ G(x, y) \end{bmatrix} + \begin{bmatrix} N_X(x) \\ N_Y(y) \end{bmatrix},$$

where, as usual, X and Y are Cartesian products of intervals. There are two sets of reductions that can be made. If $0 \in F(x) + N_X(x)$ has a unique solution, $\bar{x}$, then x can be fixed and the algorithm will only work on the reduced problem $0 \in G(\bar{x}, y) + N_Y(y)$. If $F(x)$ is an affine function, i.e. $F(x) = Ax - b$, then it is known that $0 \in F(x) + N_X(x)$ has a unique solution if A_X, the normal map associated with this variational inequality, is coherently oriented [107]. For example, when $X = \mathbf{R}^k_+$, this condition is equivalent to A being a P-matrix. For simple cases, coherent orientation can be checked; e.g. when $k = 1$ or 2. In particular, when $F(x) = ax - b$ is a row singleton with a linear element on the diagonal, then coherent orientation is $a \neq 0$ when $X = \mathbf{R}$ and $a > 0$ in all other cases. Satisfaction of this condition guarantees uniqueness of the solution. When $k = 2$, the condition is again that A is a P-matrix, unless one or more of the intervals defining X is $\mathbf{R}$. In these later cases, the conditions are weaker. Table 27 summarizes all of

Table 27: Coherent Orientation Conditions
X is assumed to be a Cartesian product of intervals with ℓ and u being finite.

Case	Coherent Orientation Condition
$X = \mathbf{R}$	$A_{1,1} \neq 0$
$X = [\ell, \infty)$	$A_{1,1} > 0$
$X = (-\infty, u]$	$A_{1,1} > 0$
$X = [\ell, u]$	$A_{1,1} > 0$
$X = \mathbf{R} \times \mathbf{R}$	$det(A) \neq 0$
$X = \mathbf{R} \times [\ell, \infty)$	$det(A) \neq 0$ and $sign(det(A)) = sign(A_{1,1})$
$X = \mathbf{R} \times (-\infty, u]$	$det(A) \neq 0$ and $sign(det(A)) = sign(A_{1,1})$
$X = \mathbf{R} \times [\ell, u]$	$det(A) \neq 0$ and $sign(det(A)) = sign(A_{1,1})$
All other cases	$det(A) > 0$ and $sign(A_{1,1}) = sign(A_{2,2}) = 1$

the checks for coherent orientation. The preprocessor identifies double blocks by finding a linear row with two elements, one of which is on the diagonal. A check of the row corresponding to the other variable is performed to see if there is a doubleton block.

The other reduction to consider is where $0 \in G(x, y) + N_Y(y)$ has a solution for all $x \in X$. In this case, $0 \in F(x) + N_X(x)$ is solved to find $\bar{x}$ and then a $\bar{y}$ satisfying $0 \in G(\bar{x}, y) + N_Y(y)$ is found. Assume that $G(x, y)$ is linear in y, i.e. $G(x, y) = H(x) + By$. The coherent orientation conditions outlined above applied to B suffice in this case as well, since they guarantee existence as well as uniqueness. However, to guarantee only existence, weaker conditions are sufficient. For $k = 1$ it is necessary and sufficient to have coherent orientation or Y compact. When $k = 2$, the conditions are outlined in Table 28 and are derived from Theorem 2 in [60] and [59]. The conditions given for the cases where there is at least one free variable are necessary and sufficient to guarantee existence for all $x \in X$.

In the nonlinear setting, intervals on the Jacobian elements can be used to verify conditions related to uniqueness and existence of a solution. For example in the single element case, if the value of the Jacobian element is positive and uniformly bounded away

Table 28: Existence Conditions

Y is assumed to be a Cartesian product of intervals with ℓ, $\tilde{\ell}$, u, and $\tilde{u}$ being finite.

Case	Condition
$Y = \mathbf{R}$	$B_{1,1} \neq 0$
$Y = [\ell, \infty)$	$B_{1,1} > 0$
$Y = (-\infty, u]$	$B_{1,1} > 0$
$Y = [\ell, u]$	nothing
$Y = \mathbf{R} \times \mathbf{R}$	$det(B) \neq 0$
$Y = \mathbf{R} \times [\ell, \infty)$	$det(B) \neq 0$ and $sign(det(B)) = sign(B_{1,1})$
$Y = \mathbf{R} \times (-\infty, u]$	$det(B) \neq 0$ and $sign(det(B)) = sign(B_{1,1})$
$Y = \mathbf{R} \times [\ell, u]$	$B_{1,1} \neq 0$
$Y = [\ell, \infty) \times [\tilde{\ell}, \infty)$	$B > 0$ or
	$(det(B) > 0$ and $sign(B_{1,1}) = sign(B_{2,2}) = 1)$
$Y = [\ell, \infty) \times (-\infty, \tilde{u}]$	$B > 0$ or
	$(det(B) > 0$ and $sign(B_{1,1}) = sign(B_{2,2}) = 1)$
$Y = [\ell, \infty) \times [\tilde{\ell}, \tilde{u}]$	$B_{1,1} \neq 0$
$Y = (-\infty, u] \times (-\infty, \tilde{u}]$	$B > 0$ or
	$(det(B) > 0$ and $sign(B_{1,1}) = sign(B_{2,2}) = 1)$
$Y = (-\infty, u] \times [\tilde{\ell}, \tilde{u}]$	$B_{1,1} \neq 0$
$Y = [\ell, u] \times [\tilde{\ell}, \tilde{u}]$	nothing

from zero, i.e. it is a uniform P-function, then the existence and uniqueness is always guaranteed and the same substitutions can be performed. Finding the solution becomes more difficult, as it involves solving a nonlinear problem.

4.3 Computational Results

The preprocessing algorithm implemented alternates between exploiting the polyhedral structure and the functions. Initially all possible reductions based on the polyhedral constraints are made. Then all reductions based on the functional implications are made. These two steps are repeated until no changes are made to the model.

The potential for preprocessing is mainly limited to finding and exploiting linear parts of the problem because interval evaluators are not available at present in the modeling language environments. The majority of the reductions made come from exploiting polyhedral structure. However, the reductions from the second stage can also be significant to the success of the algorithm.

The preprocessor was tested on three different sets of problems. The first test compares the performance of the MCP preprocessor to the one used by the commercial CPLEX code [70]. Using the linear programs contained in NETLIB [55], the first order conditions from linear programming were constructed and given to the MCP preprocessor. CPLEX was given the original linear program. Reported in Tables 29 and 30 are the sizes of the preprocessed models. CPLEX is capable of performing aggregations, while the MCP preprocessor currently does not. Therefore, in the tables, the size of the model produced by CPLEX both with aggregations (With) and without aggregations (Without) are stated. As evidenced by the table, the MCP preprocessor is competitive with CPLEX on linear programs when aggregations are not allowed. One interesting point to note is that the problems `fit*p` and `fit*d` are primal-dual pairs - the MCP preprocessor generates an identical system in both cases. Exploiting dual information in the `fit*p` problems significantly reduces the size of the preprocessed models.

A second test was performed using quadratic programming problems reformulated using the complementary slackness conditions (16). Some artificial quadratic programs were created for testing purposes from the NETLIB collection. A term of $\frac{1}{2}x^T x$ was added to the objective function and the resulting complementary slackness conditions were given to the preprocessor and the PATH 4.x code. Table 31 reports the size reductions and compares the solution times for PATH 4.x on the original and preprocessed models. On the problems successfully preprocessed, the reductions in time are significant. Some quadratic programs from other sources were also tested. In one of the models, `hwayoung`,

Table 29: Comparison of CPLEX and MCP preprocessor on NETLIB problems

Model	Size	CPLEX With	CPLEX Without	MCP Preprocessor
adlittle	153	147	147	146
afiro	59	48	52	56
agg	651	271	275	433
agg2	818	530	538	743
agg3	818	531	541	743
bandm	777	398	483	467
beaconfd	435	55	220	205
blend	157	108	140	149
bnl1	1807	1443	1668	1670
bnl2	5769	3031	4226	4341
boeing1	909	711	713	720
boeing2	320	281	281	292
bore3d	547	105	182	191
brandy	431	265	311	311
capri	608	383	547	547
cycle	4743	2700	3416	3884
czprob	4221	2904	3349	3430
d2q06c	7338	6450	6871	6286
d6cube	6588	5844	5867	6423
degen2	978	855	977	974
degen3	3321	3125	3321	3310
dfl001	18301	13062	17091	16915
e226	505	397	411	414
etamacro	1006	754	850	821
fffff800	1378	933	983	1284
finnis	1066	739	786	808
fit1d	1050	1048	1048	1050
fit1p	2304	2054	2054	1050
fit2d	10525	10388	10388	10525
fit2p	16525	16525	16525	10525
forplan	554	466	476	483
ganges	2990	1202	2177	2466
gfrd-pnc	1708	1116	1656	1656
greenbea	7691	4055	5900	5763
greenbeb	7679	4044	5892	5738
grow15	945	945	945	945
grow22	1386	1386	1386	1386
grow7	441	441	441	441
israel	316	304	304	304

Table 30: Comparison of CPLEX and MCP preprocessor on NETLIB problems (cont.)

		CPLEX		
Model	Size	With	Without	MCP Preprocessor
kb2	84	67	79	82
lotfi	461	399	399	408
nesm	3585	3325	3373	3440
perold	1937	1571	1769	1757
pilot4	1380	1111	1200	1210
sc105	207	117	207	207
sc205	407	231	405	405
sc50a	97	57	97	97
sc50b	96	56	96	96
scagr25	971	591	841	734
scagr7	269	159	229	194
scfxm1	787	612	694	698
scfxm2	1574	1228	1388	1396
scfxm3	2361	1844	2082	2094
scorpion	746	172	590	453
scrs8	1659	913	1429	1438
scsd1	837	837	837	817
scsd6	1497	1497	1497	1481
scsd8	3147	3147	3147	3135
sctap1	780	608	608	660
sctap2	2970	2303	2303	2500
sctap3	3960	3111	3111	3340
share1b	342	297	315	310
share2b	175	168	172	172
shell	2061	1427	1935	1935
ship04l	2478	2174	2182	2208
ship04s	1818	1426	1482	1500
ship08l	4995	3569	3569	3611
ship08s	3099	1760	1858	1890
ship12l	6469	4756	4756	4790
ship12s	3805	2114	2258	2288
stair	741	512	740	740
stocfor1	228	113	190	188
stocfor2	4188	2474	3822	3825
tuff	878	514	738	788
wood1p	2838	1898	1898	1971
Total	181745	137202	155734	150753

Table 31: Comparison of PATH 4.x with and without preprocessing on QP models

	Original		Preprocessed	
Model	Size	Solution Time	Size	Solution Time
agg	651	6.6	454	1.0
beaconfd	435	1.1	283	0.7
finnis	1066	9.1	918	1.5
lotfi	461	5.9	434	0.9
nesm	3585	57.6	3481	53.0
scorpion	746	1.1	617	0.8
ship08s	3099	6.3	1966	3.3
tuff	878	4.3	849	3.9

over 70% of the variables were removed by the preprocessor reducing the size from 46123 variables to 13655, leading to a significant reduction in the total time needed to solve the problem.

Finally, the models in MCPLIB [26] were given to the preprocessor. The results on these models are less encouraging than the other two tests because of a lack of linear problems in the test set and the inability to obtain interval evaluations for the nonlinear functions. Many of the models did not benefit from preprocessing. However, some successes are reported in Table 32. Note that the `explcp` model that is supposed to display exponential behavior for Lemke's algorithm is completely solved in the preprocessor. The preprocessor for the `golanmcp` model removes 18 redundant rows. The remaining problem solves without any proximal perturbation, leading to the substantial reduction in solution time.

In our initial testing, some of the preprocessing performed was detrimental. For example, the `force*` models became harder to solve after preprocessing even though they were significantly reduced in size. We added an option that reports extra information to the modeler regarding the orientation of equality constraints. This uncovered four models in MCPLIB that have equality constraints not formulated in a skew symmetric

Table 32: Comparison of PATH 4.x with and without preprocessing on MCP models

	Original		Preprocessed	
Model	Size	Solution Time	Size	Solution Time
electric	158	1.3	149(143)	0.5 (0.1)
explcp	16	0.0	0	0.0
forcebsm	184	0.1	72	0.2 (0.1)
forcedsa	186	0.1	70	0.1 (0.1)
golanmcp	4321	80.9	4303	25.0
merge	9536	2254.6	8417	1954.2

sense, namely the `electric`, `force*` and `lincont` models. Rewriting to change this orientation gave the improved timings noted in parentheses in the table.

4.4 Summary

This chapter has developed a complete preprocessor for complementarity problems and demonstrated the effectiveness of the procedure in reducing the time spent by PATH 4.x to solve available test problems. The preprocessor is currently available with both PATH 4.x and SEMI, which are the first codes for solving complementarity problems with preprocessing technology.

We remark that an earlier version of this chapter appeared in [43].

Chapter 5

Diagnostic Information

Developing a practical model of a complex situation is a difficult task in which an approximate representation is initially constructed and then iteratively refined until an accurate formulation is obtained. During the intermediate stages, the models generated have a tendency to be ill-defined, poorly conditioned, and/or singular. Information generated by a solver can help the modeler to detect these problems, quickly locate the source, and make appropriate modifications to the model. By preventing the propagation of errors to successive models, the development cycle shortens and the final product becomes easier to solve with a more meaningful solution.

This chapter discusses statistics provided that can be used to find potential difficulties. We begin by looking at the merit functions used to indicate if an iterate is close to the solution set of the given MCP in Section 5.1. We then utilize the merit function information in Section 5.2 when we present the first of our problem areas, ill-defined models. Section 5.3 addresses poorly-scaled models and the difficulties encountered in such circumstances. Information provided to detect scaling problems are mentioned. Section 5.4 concludes the trilogy of problems areas by looking at singular models.

We remark that the statistics generated are not a replacement for the modeler's knowledge about the application, rather it augments the available information to help them rapidly identify potential problem areas.

5.1 Merit Functions

A solver for complementarity problems typically employs a merit function to indicate the closeness of the current iterate to the solution set. The merit function is zero at a solution to the original problem and strictly positive otherwise. Numerically, an algorithm terminates when the merit function is approximately equal to zero, thus possibly introducing spurious "solutions".

The modeler needs to be able to determine with some reasonable degree of accuracy whether the algorithm terminated at solution or if it simply obtained a point satisfying the desired tolerances that is not close to the solution set. For complementarity problems, we can provide several indicators with different characteristics to help make such a determination. If one of the indicators is not close to zero, then there is some evidence that the algorithm has not found a solution. We note that if all of the indicators are close to zero, we are reasonably sure we have found a solution. However, the modeler has the final responsibility to evaluate the "solution" and check that it makes sense for their application.

For the NCP, a standard merit function is

$$\|(-x)_+, (-F(x))_+, [(x_i)_+(F_i(x))_+]_i\|$$

with the first two terms measuring the infeasibility of the current point and the last term indicating the complementarity error. In this expression, we use $(\cdot)_+$ to represent the Euclidean projection of x onto the nonnegative orthant, that is $(x)_+ = \max(x, 0)$. For the more general MCP, we can define a similar function:

$$\left\| x - \pi(x), \left[\left(\frac{x_i - \ell_i}{\|\ell_i\| + 1} \right)_+ (F_i(x))_+ \right]_i, \left[\left(\frac{u_i - x_i}{\|u_i\| + 1} \right)_+ (-F_i(x))_+ \right]_i \right\|$$

where $\pi(x)$ represents the Euclidean projection of x onto the set $[\ell, u]$. We can see that if we have an NCP, the function is exactly the one previously given and for nonlinear systems of equations, this becomes $\|F(x)\|$.

There are several reformulations of the MCP as systems of nonlinear (nonsmooth) equations for which the corresponding residual is a natural merit function. Some of these are as follows:

- Generalized Minimum Map: $x - \pi(x - F(x))$
- Normal Map: $F(\pi(y)) + y - \pi(y)$
- Fischer Function: $\Phi(x)$, where $\Phi_i(x) := \phi(x_i, F_i(x))$ with

$$\phi(a, b) := \sqrt{a + b} - a - b.$$

  Note that $\phi(a, b) = 0$ if and only if $0 \leq a \perp b \geq 0$.

In the context of nonlinear complementarity problems the generalized minimum map corresponds to the classic minimum map $\min(x, F(x))$. Furthermore, for NCPs the minimum map and the Fischer function are both local error bounds and were shown to be equivalent in [118]. Figure 29 in the subsequent section plots all of these merit functions for the ill-defined example discussed therein and highlights the differences between them.

The squared norm of Φ, namely $\Psi(x) := \frac{1}{2} \sum \phi(x_i, F_i)^2$, is continuously differentiable on $\mathbf{R}^n$ provided F itself is. Therefore, the first order optimality conditions for the unconstrained minimization of $\Psi(x)$, namely $\nabla\Psi(x) = 0$ give another indication as to whether the point under consideration is a solution of MCP.

5.2 Ill-Defined Models

A problem can be ill-defined for several different reasons. We concentrate on the following particular cases. We will call F well-defined at $\bar{x} \in [\ell, u]$ if $\bar{x} \in \operatorname{dom} F$, the domain of F, and ill-defined at $\bar{x}$ otherwise. Furthermore, we define F to be well-defined near $\bar{x} \in [\ell, u]$ if there exists an open neighbourhood of $\bar{x}$, $\mathcal{N}(\bar{x})$, such that $[\ell, u] \cap \mathcal{N}(\bar{x}) \subseteq \operatorname{dom} F$. By saying the function is well-defined near $\bar{x}$, we are simply stating that F is defined for all $x \in [\ell, u]$ sufficiently close to $\bar{x}$. A function not well-defined near $\bar{x}$ is termed ill-defined near $\bar{x}$.

We will say that F has a well-defined Jacobian at $\bar{x} \in [\ell, u]$ if there exists an open neighbourhood of $\bar{x}$, $\mathcal{N}(\bar{x})$, such that $\mathcal{N}(\bar{x}) \subseteq \operatorname{dom} F$ and F is continuously differentiable on $\mathcal{N}(\bar{x})$. Otherwise the function has an ill-defined Jacobian at $\bar{x}$. We note that a well-defined Jacobian at $\bar{x}$ implies that the MCP has a well-defined function near $\bar{x}$, but the converse is not true.

Many solvers use both function and Jacobian information when attempting to solve an MCP. Therefore, both of these definitions are relevant. We discuss cases where the function and Jacobian are ill-defined in the next two subsections. We illustrate uses for the merit function information and final point statistics within the context of these problems.

5.2.1 Function Undefined

We begin with a one-dimensional problem for which F is ill-defined at $x = 0$ as follows:

$$0 \leq x \quad \perp \quad \tfrac{1}{x} \geq 0.$$

Here x must be strictly positive because $\frac{1}{x}$ is undefined at $x = 0$. This condition implies that $F(x)$ must be equal to zero. Since $F(x)$ is strictly positive for all x strictly positive, this problem has no solution.

We are able to perform this analysis because the dimension of the problem is small. In general, a modeler might not know a priori that a problem has no solution and might attempt to formulate and solve it. For this particular example, we must specify an initial value for x, as the function is undefined at the default starting point, $x = 0$. The AMPL and GAMS interfaces would notice that F is undefined at this initial point, and terminate with an error message. Note that such an error messages only indicates that the function $\frac{1}{x}$ is undefined as $x = 0$. The complementarity problem might have a solution.

After setting a starting point and passing the model along to the algorithm, we proceed to "solve" the model. A portion of the output with relevant statistics about the solution is given in Figure 28. At the end of the solve, all of the merit functions given in Section 5.1 are evaluated at the final point. The Normal Map merit function, and to a lesser extent, the complementarity error, indicate that the "solution" found does not necessarily solve the MCP.

To indicate the difference between the merit functions, Figure 29 plots them all for the simple example. We note that as x approaches positive infinity, numerically, we are at a solution to the problem with respect to all of the merit functions except for the complementarity error, which remains equal to one. As x approaches zero, the merit functions diverge, also indicating that $x = 0$ is not a solution.

The natural residual and Fischer function tend toward 0 as $x \downarrow 0$. From these measures, we might think $x = 0$ is the solution. However, as previously remarked F is ill-defined at $x = 0$. F and F' become very large, indicating that the function (and

```
FINAL STATISTICS
Inf-Norm of Complementarity . .  1.0000e+00 eqn: (F)
Inf-Norm of Normal Map. . . . .  1.1181e+16 eqn: (F)
Inf-Norm of Minimum Map . . . .  8.9441e-17 eqn: (F)
Inf-Norm of Fischer Function. .  8.9441e-17 eqn: (F)
Inf-Norm of Grad Fischer Fcn. .  8.9441e-17 eqn: (F)

FINAL POINT STATISTICS
Maximum of X. . . . . . . . . .  8.9441e-17 var: (X)
Maximum of F. . . . . . . . . .  1.1181e+16 eqn: (F)
Maximum of Grad F . . . . . . .  1.2501e+32 eqn: (F)
                                            var: (X)
```

Figure 28: Output for Ill-Defined Function

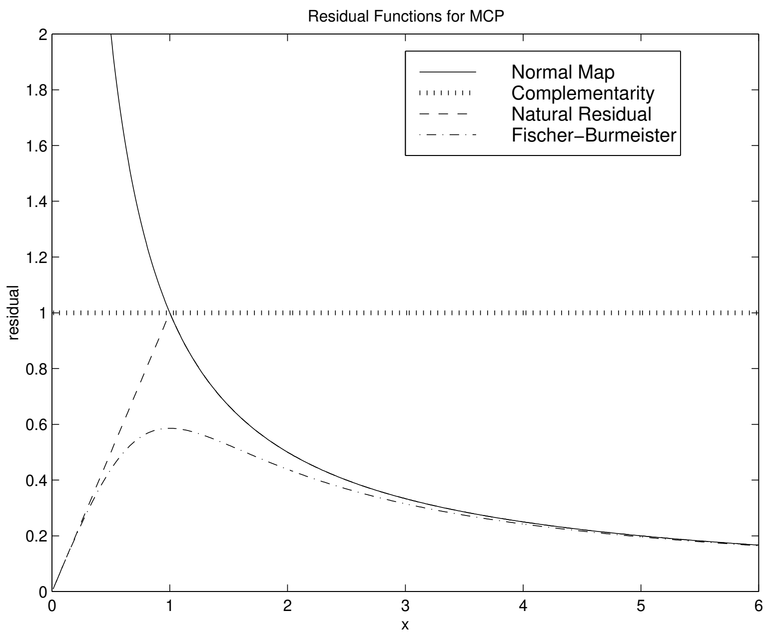

Figure 29: Merit Function Plot

```
FINAL STATISTICS
Inf-Norm of Complementarity . .  1.0000e-14 eqn: (G)
Inf-Norm of Normal Map. . . . .  1.0000e+06 eqn: (G)
Inf-Norm of Minimum Map . . . .  1.0000e-20 eqn: (G)
Inf-Norm of Fischer Function. .  1.0000e-20 eqn: (G)
Inf-Norm of Grad Fischer Fcn. .  1.0000e-20 eqn: (G)

FINAL POINT STATISTICS
Maximum of X. . . . . . . . . .  1.0000e-20 var: (X)
Maximum of F. . . . . . . . . .  1.0000e+06 eqn: (G)
Maximum of Grad F . . . . . . .  1.0000e+12 eqn: (G)
                                            var: (X)
```

Figure 30: Output for Well-Defined Function

Jacobian) might not be well-defined. We might be tempted to conclude that if one of the merit function indicators is not close to zero, then we have not found a solution. This conclusion is not always the case. When one of the indicators is non-zero, we have reservations about the solution, but we cannot eliminate the possibility that we are actually close to a solution. If we slightly perturb the original problem to

$$0 \leq x \quad \perp \quad \tfrac{1}{x+\epsilon} \geq 0$$

for a fixed $\epsilon > 0$, the function is well-defined over $\mathbf{R}^n_+$ and has a unique solution at $x = 0$. In this case, by starting at $x > 0$ and sufficiently small, all of the merit functions, with the exception of the Normal Map, indicate that we have solved the problem as is shown by the output in Figure 30 for $\epsilon = 1 * 10^{-6}$ and $x = 1 * 10^{-20}$. In this case, the Normal Map is quite large and we might think that the function and Jacobian are undefined. This example illustrates the point that all of these tests are not infallible. The modeler still needs to do some detective work to determine if they have found a solution or if the algorithm is converging to a point where the function is ill-defined.

```
FINAL STATISTICS
Inf-Norm of Complementarity . .  1.0000e-07 eqn: (F)
Inf-Norm of Normal Map. . . . .  1.0000e-07 eqn: (F)
Inf-Norm of Minimum Map . . . .  1.0000e-07 eqn: (F)
Inf-Norm of Fischer Function. .  2.0000e-07 eqn: (F)
Inf-Norm of Grad FB Function. .  2.0000e+00 eqn: (F)

FINAL POINT STATISTICS
Maximum of X. . . . . . . . . .  1.0000e-14 var: (X)
Maximum of F. . . . . . . . . .  1.0000e-07 eqn: (F)
Maximum of Grad F . . . . . . .  5.0000e+06 eqn: (F)
                                            var: (X)
```

Figure 31: Output for Ill-Defined Jacobian

5.2.2 Jacobian Undefined

Since the algorithms use Newton-like method to solve the problems, they also need the Jacobian of F to be well-defined. One model for which the function is well-defined over $[\ell, u]$, but for which the Jacobian is undefined at the solution is: $0 \leq x \perp -\sqrt{x} \geq 0$. This model has a unique solution at $x = 0$.

Using the PATH 4.x algorithm and starting from the point $x = 1 * 10^{-14}$, the output given in Figure 31 is generated. We can see the that gradient of the Fischer Function is nonzero and the Jacobian is beginning to become large. These conditions indicate that the Jacobian is undefined at the solution. It is therefore important for a modeler to inspect the given output to guard against such problems.

If we start from $x = 0$, the algorithms correctly report that we are at the solution. Even though the entries in the Jacobian are undefined at this point, some of the interfaces will not return a domain violation. This problem with the Jacobian is therefore undetectable by the algorithm.

```
INITIAL POINT STATISTICS
Maximum of X. . . . . . . . . .  4.1279e+06 var: (w.29)
Maximum of F. . . . . . . . . .  2.2516e+00 eqn: (a1.33)
Maximum of Grad F . . . . . . .  6.7753e+06 eqn: (a1.29)
                                            var: (x1.29)

INITIAL JACOBIAN NORM STATISTICS
Maximum Row Norm. . . . . . . .  9.4504e+06 eqn: (a2.29)
Minimum Row Norm. . . . . . . .  2.7680e-03 eqn: (g.10)
Maximum Column Norm . . . . . .  9.4504e+06 var: (x2.29)
Minimum Column Norm . . . . . .  1.3840e-03 var: (w.10)
```

Figure 32: Output - Poorly Scaled Model

5.3 Poorly Scaled Models

Problems which are well-defined can have various numerical problems that can impede the algorithm's convergence. One particular problem is a badly scaled Jacobian. In such cases, we can obtain a poor "Newton" direction because of numerical problems introduced in the linear algebra performed. This problem can also lead the code to a point from which it cannot recover.

The final model given to the solver should be scaled such that we avoid numerical difficulties in the linear algebra. The statistics provided can be used to iteratively refine the model so that we eventually end up with a well-scaled problem. We note that we only calculate our scaling statistics at the starting point provided. For nonlinear problems these statistics may not be indicative of the overall scaling of the model. Model specific knowledge is very important when we have a nonlinear problem because it can be used to appropriately scale the model to achieve a desired result.

We look at the `titan` model in MCPLIB, that has some scaling problems. The relevant output for the original code is given in Figure 32. The maximum row norm is

```
INITIAL POINT STATISTICS
Maximum of X. . . . . . . . . .  1.0750e+03 var: (x1.49)
Maximum of F. . . . . . . . . .  3.9829e-01 eqn: (g.10)
Maximum of Grad F . . . . . . .  6.7753e+03 eqn: (a1.29)
                                            var: (x1.29)

INITIAL JACOBIAN NORM STATISTICS
Maximum Row Norm. . . . . . . .  9.4524e+03 eqn: (a2.29)
Minimum Row Norm. . . . . . . .  2.7680e+00 eqn: (g.10)
Maximum Column Norm . . . . . .  9.4904e+03 var: (x2.29)
Minimum Column Norm . . . . . .  1.3840e-01 var: (w.10)
```

Figure 33: Output - Well-Scaled Model

defined as

$$\max_{1\le i\le n} \sum_{1\le j\le n} \mid [F'(x)]_{ij} \mid$$

and the minimum row norm is

$$\min_{1\le i\le n} \sum_{1\le j\le n} \mid [F'(x)]_{ij} \mid .$$

Similar definitions are used for the column norm. The norm numbers for this particular example are not extremely large, but we can nevertheless improve the scaling. We first decided to reduce the magnitude of the `a2` block of equations as indicated by the output. After scaling the `a2` block, we re-ran the code found additional blocks of equations and variables that also needed scaling. After scaling all of these blocks of equations in the model, we have improved the scaling statistics which are given in Figure 33 for the new model. For this particular problem the PATH algorithm could not solve the unscaled model, while it can find a solution to the scaled model.

Not all of the numerical problems are directly attributable to poorly scaled models. Problems for which the Jacobian of the active constraints is singular or nearly singular can also cause numerical difficulty as illustrated next.

```
INITIAL POINT STATISTICS
Zero column of order. . . . . .  0.0000e+00 var: (X)
Zero row of order . . . . . . .  0.0000e+00 eqn: (F)
Total zero columns. . . . . . .  1
Total zero rows . . . . . . . .  1
Maximum of F. . . . . . . . . .  1.0000e+00 eqn: (F)
Maximum of Grad F . . . . . . .  0.0000e+00 eqn: (F)
                                            var: (X)
```

Figure 34: Output - Zero Rows and Columns

5.4 Singular Models

Assuming that the problem is well-defined and properly scaled, we can still have a Jacobian for which the active constraints are singular or nearly singular (i.e. it is ill-conditioned). When problems are singular or nearly singular, we are also likely to have numerical problems. As a result the "Newton" direction obtained from the linear problem solver can be very bad. Various heuristics are used by algorithms that attempt to remove the singularity problems from the model. However, it is most often beneficial for solver robustness to remove singularities if possible.

The easiest problems to detect are those for which the Jacobian has zero rows and columns. A simple problem for which we have zero rows and columns is:

$$-2 \leq x \leq 2 \quad \perp \quad -x^2 + 1.$$

Note that the Jacobian, $-2x$, is non-singular at all three solutions, but singular at the point $x = 0$. Output for this model starting at $x = 0$ is given in Figure 34. We display in the code the variables and equations for which the row/column in the Jacobian is close to zero. These situations are problematic and for nonlinear problems likely stem from the modeler providing an inappropriate starting point or fixing some variables resulting in some equations becoming constant. We note that the solver may perform well in the presence of zero rows and/or columns, but the modeler should make sure that these are what was intended.

Singularities in the model can also be detected by the linear solver. This in itself is a hard problem and prone to error. For matrices which are poorly scaled, we can incorrectly identify "linearly dependent" rows because of numerical problems. Typically, singularity does not cause a lot of problems and the algorithm can handle the situation appropriately. However, an excessive number of singularities are cause for concern. A further indication of possible singularities at the solution is the lack of quadratic convergence to the solution.

5.5 Summary

This chapter provided documentation on diagnostic information used to identify ill-defined, poorly conditioned, and/or singular models. We remark that an earlier version of this chapter appeared in [41].

Chapter 6

PATH 4.x

Several algorithms for solving mixed complementarity problems, such as MILES [111] and PATH [25, 27, 28], are based upon successive linearizations: at each iteration, a linear complementarity problem is solved to find the next iterate. One difficulty frequently encountered by these methods in practice is what to do when the linear subproblem has no solution, a situation that the theory associated with these methods [103] assumes does not occur. This chapter proposes a theoretically justifiable escape mechanism that uses a differentiable merit function in conjunction with the direction generated by a linearization method. The resulting framework forms the basis for PATH 4.x.

Merit functions are used extensively in the development of globalization theory and the implementation of robust algorithms. Broadly speaking, a merit function summarizes how close the current iterate is to a solution of the problem under consideration with a single number. In complementarity problems and nonlinear systems of equations, the merit functions are normally nonnegative, and zero precisely at a solution to the original problem. Each merit function is typically used in a globalization strategy that involves searching between the current iterate and the Newton point (the solution of the linearization).

The classical example of a merit function in nonlinear equation solving is the square of the two-norm residual that measures the sum of squares of the errors in satisfying the equations. This merit function has one additional property to those listed above:

namely, it is everywhere differentiable provided that the equation itself is everywhere differentiable. When the linear subproblem cannot be solved, a gradient-based descent direction for the differentiable merit function can be searched, guaranteeing that progress is made toward a stationary point of the merit function.

In complementarity, the two classical merit functions are based on the natural residual [84] and the normal map [107]. Both the natural residual and the normal map provide reformulations of the complementarity problem as a system of equations; unfortunately, the systems and corresponding residual merit functions are nonsmooth. Even with this drawback, [103] showed how to construct an extension of the line search procedure for smooth nonlinear equations that enables fast local convergence of linearization methods based on the normal map [108] under conditions that are the exact generalizations of those required for smooth systems. PATH 2.9 [25, 27, 28] implements this method and uses heuristics when the linear subproblem cannot be solved. While these heuristics are quite successful in practice, this situation is nonetheless unsatisfactory and prone to failure.

Section 6.1 develops the theoretical framework for the globalization of successive linearization methods with a differentiable merit function and shows that the resulting algorithm is well-defined for arbitrary complementarity problems, has strong global convergence properties, and is locally fast convergent under a strong regularity assumption. Section 6.2 then discusses the PATH 4.x implementation of this framework which uses the successive linearization method of [103, 108] to generate the directions. Section 6.3 compares the reliability of PATH 4.x to PATH 2.9 on several test sets, demonstrating that PATH 4.x has better numerical behavior. Finally, Section 6.4 documents the PATH 4.x code from the perspective of a user.

6.1 Algorithm and Theory

The first component of the globalization strategy developed is a reformulation of the mixed complementarity problem using the Fischer-Burmeister function [50]. Recall from Section 1.2 that the Fischer-Burmeister function $\phi_{FB}(a,b) := \sqrt{a^2+b^2} - a - b$ is an NCP-function. To reformulate the mixed complementarity problem using this function we introduced a partitioning of the index set $I = \{1, \ldots, n\}$:

$$\begin{aligned} I_l &:= \{i \in I \mid -\infty < l_i < u_i = +\infty\}, \\ I_u &:= \{i \in I \mid -\infty = l_i < u_i < +\infty\}, \\ I_{lu} &:= \{i \in I \mid -\infty < l_i < u_i < +\infty\}, \\ I_f &:= \{i \in I \mid -\infty = l_i < u_i = +\infty\}, \end{aligned}$$

where I_l, I_u, I_{lu} and I_f denote the set of indices $i \in I$ where there are finite lower bounds only, finite upper bounds only, finite lower and upper bounds and no finite bounds on the variable x_i, respectively. Following the idea in [5], we then defined the operator $\Phi : \mathbf{R}^n \to \mathbf{R}^n$ componentwise as:

$$\Phi_i(x) := \begin{cases} \phi_{FB}(x_i - l_i, F_i(x)) & \text{if } i \in I_l, \\ -\phi_{FB}(u_i - x_i, -F_i(x)) & \text{if } i \in I_u, \\ \phi_{FB}(x_i - l_i, \phi_{FB}(u_i - x_i, -F_i(x))) & \text{if } i \in I_{lu}, \\ -F_i(x) & \text{if } i \in I_f. \end{cases}$$

We then have the following characterization of solutions to the mixed complementarity problem:

Proposition 6.1.1 ([5]) *Let $x^* \in \mathbf{R}^n$ be given. Then the following are equivalent:*

(a) x^* is a solution of the mixed complementarity problem.

(b) $\Phi(x^*) = 0$.

Section 6.1.1 contains information related to Φ and its subdifferential and in particular shows that a strongly regular solution to the complementarity problem implies that Φ is BD-regular at that point. This fact will be needed to establish local convergence of the method proposed.

The corresponding merit function for this nonsmooth system of equations:

$$\Psi(x) := \frac{1}{2}\Phi(x)^T\Phi(x) = \frac{1}{2}\|\Phi(x)\|^2$$

is continuously differentiable. In the special case of nonlinear complementarity problems, both Φ and Ψ are used to design unconstrained algorithms for the solution of the problem [23, 34]. Unfortunately, in many practical situations, the imposed bounds, l and u, on the variables are important not only for the problem definition but also because the complementarity function F may not be defined outside of these bounds. For example, applications that include fractional powers can cause severe difficulties if the function is evaluated outside the feasible region.

Therefore, the basic algorithmic framework developed does not consider Φ directly, but instead attempts to solve the bound constrained optimization reformulation

$$\min \Psi(x) \quad \text{s.t.} \quad x \in [\ell, u].$$

Section 6.1.2 shows that a constrained stationary point of this problem is a solution of the mixed complementarity problem under exactly the same assumptions used to prove a similar result for unconstrained stationary points of Ψ.

Section 6.1.3 then presents the algorithmic framework which only assumes we have a pre-existing feasible method that is locally well-defined and superlinearly convergent. The algorithmic framework generates iterates that lie within the bounds, resorting to a projected gradient step for the bound constrained problem whenever the pre-existing method fails to provide a direction satisfying a sufficient decrease condition. We prove that this method is well-defined as well as globally and locally fast convergent.

6.1.1 Equation Properties

The function Φ is not differentiable everywhere. However, it is locally Lipschitzian and therefore has a nonempty generalized Jacobian in the sense of Clarke [17]. We will use the following overestimation of this generalized Jacobian:

Proposition 6.1.2 ([5]) *We have*

$$\partial\Phi(x)^T \subseteq \{D_a(x) + \nabla F(x) D_b(x)\},$$

where $D_a(x) \in \mathbf{R}^{n\times n}$ and $D_b(x) \in \mathbf{R}^{n\times n}$ are diagonal matrices whose diagonal elements are defined as follows:

(a) If $i \in I_l$, then if $(x_i - l_i, F_i(x)) \neq (0,0)$,

$$\begin{aligned} (D_a)_{ii}(x) &= \frac{x_i - l_i}{\|(x_i - l_i, F_i(x))\|} - 1, \\ (D_b)_{ii}(x) &= \frac{F_i(x)}{\|(x_i - l_i, F_i(x))\|} - 1 \end{aligned}$$

but if $(x_i - l_i, F_i(x)) = (0,0)$,

$$((D_a)_{ii}(x), (D_b)_{ii}(x)) \in \{(\xi - 1, \rho - 1) \in \mathbb{R}^2 | \, \|(\xi, \rho)\| \leq 1\}.$$

(b) If $i \in I_u$, then if $(u_i - x_i, -F_i(x)) \neq (0,0)$,

$$\begin{aligned} (D_a)_{ii}(x) &= \frac{u_i - x_i}{\|(u_i - x_i, -F_i(x))\|} - 1, \\ (D_b)_{ii}(x) &= \frac{-F_i(x)}{\|(u_i - x_i, -F_i(x))\|} - 1 \end{aligned}$$

but if $(u_i - x_i, -F_i(x)) = (0,0)$,

$$((D_a)_{ii}(x), (D_b)_{ii}(x)) \in \{(\xi - 1, \rho - 1) \in \mathbb{R}^2 | \, \|(\xi, \rho)\| \leq 1\}.$$

(c) If $i \in I_{lu}$*, then*

$$(D_a)_{ii}(x) = a_i(x) + b_i(x)c_i(x), \qquad (D_b)_{ii}(x) = b_i(x)d_i(x).$$

Here, if $(x_i - l_i, \phi(u_i - x_i, -F_i(x))) \neq (0,0)$,

$$\begin{aligned} a_i(x) &= \frac{x_i - l_i}{\|(x_i - l_i, \phi(u_i - x_i, -F_i(x)))\|} - 1, \\ b_i(x) &= \frac{\phi(u_i - x_i, -F_i(x))}{\|(x_i - l_i, \phi(u_i - x_i, -F_i(x)))\|} - 1 \end{aligned}$$

but if $(x_i - l_i, \phi(u_i - x_i, -F_i(x))) = (0,0)$,

$$(a_i(x), b_i(x)) \in \{(\xi - 1, \rho - 1) \in \mathbb{R}^2 | \, \|(\xi, \rho)\| \leq 1\}.$$

Further, if $(u_i - x_i, -F_i(x)) \neq (0,0)$*, then*

$$c_i(x) = \frac{x_i - u_i}{\|(u_i - x_i, -F_i(x))\|} + 1, \quad d_i(x) = \frac{F_i(x)}{\|(u_i - x_i, -F_i(x))\|} + 1$$

but if $(u_i - x_i, -F_i(x)) = (0,0)$,

$$(c_i(x), d_i(x)) \in \{(\xi + 1, \rho + 1) \in \mathbb{R}^2 | \, \|(\xi, \rho)\| \leq 1\}.$$

(d) If $i \in I_f$*, then* $(D_a)_{ii}(x) = 0$, $(D_b)_{ii}(x) = -1$.

The statement of Proposition 6.1.2 is rather lengthy because we have to take into account the definition of Φ using the four different index sets I_l, I_u, I_{lu} and I_f. However, Proposition 6.1.2 is extremely important in the subsequent analysis and will be used several times within the proofs of some important results established in this and the next section.

The remainder of this section is devoted to proving Theorem 6.1.6 which says that if x^* is a strongly regular solution to the mixed complementarity problem, then Φ is BD-regular at x^*. The motivation for this result is to establish the local convergence of the algorithm given in Section 6.1.3. To that end, let $x^* \in \mathbf{R}^n$ be a solution of the mixed complementarity problem and partition I as follows:

$$\begin{aligned} \alpha(x^*) &:= \{i \,|\, l_i < x_i^* < u_i, F_i(x^*) = 0\}, \\ \beta(x^*) &:= \{i \,|\, x_i^* \in \{l_i, u_i\}, F_i(x^*) = 0\}, \\ \gamma(X^*) &:= \{i \,|\, x_i^* \in \{l_i, u_i\}, F_i(x^*) \neq 0\}. \end{aligned}$$

The following result is then a simple consequence of Proposition 6.1.2.

Lemma 6.1.3 *Let $x^* \in \mathbf{R}^n$ be a solution of the mixed complementarity problem. Furthermore, let $H \in \partial\Phi(x^*)$ be any fixed matrix, $H = D_a(x^*) + D_b(x^*)F'(x^*)$ with diagonal matrices $D_a(x^*)$ and $D_b(x^*)$ as specified in Proposition 6.1.2. Then these diagonal matrices have the following properties:*

(a) $(D_a)_{ii}(x^*) = 0$ and $(D_b)_{ii}(x^*) = -1$ for all $i \in \alpha(x^*)$.

(b) $(D_a)_{ii}(x^*) \leq 0, (D_b)_{ii}(x^*) \leq 0,$ and $(D_a)_{ii}(x^*) + (D_b)_{ii}(x^*) < 0$ for all $i \in \beta(x^*)$.

(c) $(D_a)_{ii}(x^*) = -1$ and $(D_b)_{ii}(x^*) = 0$ for all $i \in \gamma(x^*)$.

Proof. If $i \in \alpha(x^*)$, then we immediately obtain statement (a) from Proposition 6.1.2 by considering the four possible cases $i \in I_l, i \in I_u, i \in I_{lu}$ and $i \in I_f$ separately.

Next consider statement (c), i.e., assume that $i \in \gamma(x^*)$. Then we either have $x_i^* = l_i$ and $F_i(x^*) > 0$ or we have $x_i^* = u_i$ and $F_i(x^*) < 0$.

First assume that $x_i^* = l_i$ and $F_i(x^*) > 0$. Then the index i necessarily belongs to I_l or to I_{lu}. If $i \in I_l$, we obtain from Proposition 6.1.2 that $(D_a)_{ii}(x^*) = -1$ and

$(D_b)_{ii}(x^*) = 0$. On the other hand, if $i \in I_{lu}$, we get from Proposition 6.1.2, together with the observation that $\phi(a, b) > 0$ outside the nonnegative orthant, that

$$(D_a)_{ii}(x^*) = a_i(x^*) + b_i(x^*)c_i(x^*) = -1 + 0 \cdot c_i(x^*) = -1$$

and

$$(D_b)_{ii}(x^*) = b_i(x^*)d_i(x^*) = 0 \cdot d_i(x^*) = 0.$$

The case where $x_i^* = u_i$ and $F_i(x^*) < 0$ can be proven in a similar manner. Furthermore, statement (b) also follows using an identical argument. $\square$

We next restate a useful characterization of the strong regularity condition [106] in the context of mixed complementarity problems. The strong regularity condition is essentially a generalization of the nonsingularity of the Jacobian assumption used for nonlinear equations. In the case of a nonlinear complementarity problem, $\mathrm{MCP}(F, \{0\}^n, \{\infty\}^n)$, this characterization reduces to the standard characterization in [106].

Proposition 6.1.4 ([33]) *The following are equivalent:*

(a) x^* is a strongly regular solution of the mixed complementarity problem.

(b) The submatrix $F'(x^*)_{\alpha\alpha}$ is nonsingular, and the Schur-complement

$$F'(x^*)_{\alpha\cup\beta,\alpha\cup\beta}(x^*)/F'(x^*)_{\alpha\alpha} := F'(x^*)_{\beta\beta} - F'(x^*)_{\beta\alpha}F'(x^*)_{\alpha\alpha}^{-1}F'(x^*)_{\alpha\beta}$$

is a P-matrix.

In order to establish a nonsingularity result for the generalized Jacobian $\partial\Phi(x^*)$ at a strongly regular solution of the mixed complementarity problem, we also need the following result; see [62] for several extensions.

Proposition 6.1.5 ([75]) *Let $M \in \mathbf{R}^{n\times n}$ be given. The following are equivalent:*

(a) M is a P-matrix.

(b) For all negative semidefinite diagonal matrices $D_a, D_b \in \mathbb{R}^{n\times n}$ with $D_a + D_b$ is negative definite:

$$D_a + D_b M$$

is nonsingular.

We are now able to prove the main result of this section.

Theorem 6.1.6 *If x^* is a strongly regular solution of the mixed complementarity problem, then all elements $H \in \partial\Phi(x^*)$ are nonsingular. In particular, Φ is BD-regular at x^*.*

Proof. Let $H \in \partial\Phi(x^*)$. Then, by Proposition 6.1.2, there exist diagonal matrices $D_a(x^*), D_b(x^*) \in \mathbf{R}^{n\times n}$ such that

$$H = D_a(x^*) + D_b(x^*)F'(x^*). \tag{28}$$

Hence, if we write

$$D_a(x^*) = \begin{pmatrix} (D_a)_{\alpha\alpha}(x^*) & 0 & 0 \\ 0 & (D_a)_{\beta\beta}(x^*) & 0 \\ 0 & 0 & (D_a)_{\gamma\gamma}(x^*) \end{pmatrix},$$

$$D_b(x^*) = \begin{pmatrix} (D_b)_{\alpha\alpha}(x^*) & 0 & 0 \\ 0 & (D_b)_{\beta\beta}(x^*) & 0 \\ 0 & 0 & (D_b)_{\gamma\gamma}(x^*) \end{pmatrix}$$

and

$$F'(x^*) = \begin{pmatrix} F'(x^*)_{\alpha\alpha} & F'(x^*)_{\alpha\beta} & F'(x^*)_{\alpha\gamma} \\ F'(x^*)_{\beta\alpha} & F'(x^*)_{\beta\beta} & F'(x^*)_{\beta\gamma} \\ F'(x^*)_{\gamma\alpha} & F'(x^*)_{\gamma\beta} & F'(x^*)_{\gamma\gamma} \end{pmatrix}$$

and if we take into account Lemma 6.1.3, the homogeneous linear system $Hd = 0$ can be rewritten as

$$\begin{aligned} F'(x^*)_{\alpha\alpha}d_\alpha + F'(x^*)_{\alpha\beta}d_\beta + F'(x^*)_{\alpha\gamma}d_\gamma &= 0_\alpha, & (29) \\ (D_a)_{\beta\beta}(x^*)d_\beta + (D_b)_{\beta\beta}(x^*)\left[F'(x^*)_{\beta\alpha}d_\alpha + F'(x^*)_{\beta\beta}d_\beta + F'(x^*)_{\beta\gamma}d_\gamma\right] &= 0_\beta, & (30) \\ -d_\gamma &= 0_\gamma. & (31) \end{aligned}$$

Since $d_\gamma = 0$ by (31) and $F'(x^*)_{\alpha\alpha}$ is nonsingular by assumption and Proposition 6.1.4, we obtain from (29):

$$d_\alpha = -F'(x^*)^{-1}_{\alpha\alpha}F'(x^*)_{\alpha\beta}d_\beta. \tag{32}$$

Substituting (31) and (32) into (30), we obtain after some rearrangements:

$$\left[(D_a)_{\beta\beta}(x^*) + (D_b)_{\beta\beta}(x^*)(F'(x^*)_{\alpha\cup\beta,\alpha\cup\beta}/F'(x^*)_{\alpha\alpha})\right] d_\beta = 0_\beta. \tag{33}$$

Since the Schur complement $F'(x^*)_{\alpha\cup\beta,\alpha\cup\beta}/F'(x^*)_{\alpha\alpha}$ is a P-matrix by assumption and Proposition 6.1.4 and since, by Lemma 6.1.3 (b), the diagonal matrices $(D_a)_{\beta\beta}(x^*)$ and $(D_b)_{\beta\beta}(x^*)$ are negative semidefinite with a negative definite sum, it follows from Proposition 6.1.5 that the coefficient matrix in (33) is nonsingular. Hence we obtain $d_\beta = 0_\beta$. This, in turn, implies $d_\alpha = 0_\alpha$ by (32). Reference to (31) shows that $d = 0$, so that H is nonsingular. Since $\partial_B\Phi(x^*) \subseteq \partial\Phi(x^*)$ it follows that Φ is BD-regular at x^*. $\square$

6.1.2 Merit Function Properties

We now investigate properties of the residual merit function

$$\Psi(x) = \frac{1}{2}\Phi(x)^T\Phi(x)$$

associated with the equation operator Φ. Despite the fact that Φ is nondifferentiable in general, the merit function Ψ is continuously differentiable everywhere.

Proposition 6.1.7 ([5]) *The function Ψ is continuously differentiable with gradient $\nabla\Psi(x) = H^T\Phi(x)$ for an arbitrary $H \in \partial\Phi(x)$.*

We next provide a stationary point result for the unconstrained reformulation

$$\min\ \Psi(x), \qquad x \in \mathbf{R}^n,$$

of the mixed complementarity problem. To this end, we need the following characterization of P_0-matrices. Generalizations of this result can be found in [14, 62].

Proposition 6.1.8 *Let $M \in \mathbf{R}^{n\times n}$ be given. The following are equivalent:*

(a) M is a P_0-matrix.

(b) For all negative definite diagonal matrices $D_a, D_b \in \mathbb{R}^{n\times n}$:

$$D_a + D_b M$$

is nonsingular.

We will also need the following technical result.

Lemma 6.1.9 *Let $x \in \mathbf{R}^n$ be arbitrary and $H \in \partial\Phi(x), H = D_a(x) + D_b(x)F'(x)$ with diagonal matrices $D_a(x), D_b(x) \in \mathbf{R}^{n\times n}$ as defined in Proposition 6.1.2. Then the following two statements hold:*

(a) For all $i \in I$*,* $[D_a(x)\Phi(x)]_i[D_b(x)\Phi(x)]_i \geq 0$ *.*

(b) For each $i \notin I_f$*,* $[D_a(x)\Phi(x)]_i = 0 \Longleftrightarrow [D_b(x)\Phi(x)]_i = 0 \Longleftrightarrow \Phi_i(x) = 0$ *.*

Proof. (a) By considering the four possible cases $i \in I_l, i \in I_u, i \in I_{lu}$ and $i \in I_f$, it is easy to see that $(D_a)_{ii}(x) \leq 0$ and $(D_b)_{ii}(x) \leq 0$ for all $i \in I$. Hence

$$[D_a(x)\Phi(x)]_i[D_b(x)\Phi(x)]_i = (D_a)_{ii}(x)(D_b)_{ii}(x)\Phi_i(x)^2 \geq 0$$

for all $i \in I$.

(b) If $\Phi_i(x) = 0$, we immediately have

$$[D_a(x)\Phi(x)]_i = 0 \quad \text{and} \quad [D_b(x)\Phi(x)]_i = 0.$$

Conversely, assume that $[D_a(x)\Phi(x)]_i = 0$ for some index $i \notin I_f$ (the proof is analogous if $[D_b(x)\Phi(x)]_i = 0$). Then

$$(D_a)_{ii}(x) = 0 \quad \text{or} \quad \Phi_i(x) = 0.$$

In the latter case, there is nothing to show. So suppose that $(D_a)_{ii}(x) = 0$. Due to the definition of $D_a(x)$, we distinguish three cases.

Case 1: $i \in I_l$.

If $(x_i - l_i, F_i(x)) = (0, 0)$, then $\Phi_i(x) = 0$ follows immediately from the definition of the operator Φ. Otherwise Proposition 6.1.2 (a) gives

$$0 = (D_a)_{ii}(x) = \frac{x_i - l_i}{\|(x_i - l_i, F_i(x))\|} - 1.$$

This implies $x_i - l_i > 0$ and $F_i(x) = 0$, so that $\Phi_i(x) = 0$ in view of the very definition of Φ and the NCP-property of the function ϕ.

Case 2: $i \in I_u$.

The proof of this case is very similar to the one given for Case 1 and we therefore omit the details.

Case 3: $i \in I_{lu}$.

If $(x_i - l_i, \phi(u_i - x_i, -F_i(x)) = (0,0)$, we are done. So assume that $(x_i - l_i, \phi(u_i - x_i, -F_i(x)) \neq (0,0)$. Then Proposition 6.1.2 (c) gives

$$0 = (D_a)_{ii}(x) = a_i(x) + b_i(x)c_i(x) \tag{34}$$

with certain numbers $a_i(x), b_i(x)$ and $c_i(x)$ specified in Proposition 6.1.2 (c). Since it follows immediately from this Proposition that

$$a_i(x) \leq 0, \;\; b_i(x) \leq 0 \quad \text{and} \quad c_i(x) \geq 0,$$

the right-hand side of (34) is the sum of two nonpositive expressions which can therefore be equal to zero only if $a_i(x) = 0$. This, however, implies $x_i - l_i > 0$ and $\phi(u_i - x_i, -F_i(x)) = 0$ in view of the definition of $a_i(x)$ given in Proposition 6.1.2 (c). Hence $\Phi_i(x) = 0$ by the NCP-property of the function ϕ. $\square$

Note that Lemma 6.1.9 (b) does not hold for indices $i \in I_f$ since $(D_b)_{ii}(x) = -1$ for all $i \in I_f$ in view of Proposition 6.1.2 (d).

Proposition 6.1.8 and Lemma 6.1.9 are used to prove the first major result of this section. In this result and in the remainder of this section, we use the short-hand notation $\nabla F(x^*)_{ff}$ to denote the submatrix $\nabla F(x^*)_{I_f I_f}$. A similar notation is used for submatrices and subvectors defined by other index sets.

Theorem 6.1.10 *Let $x^* \in \mathbf{R}^n$ be a stationary point of Ψ. Assume that* *(a) the principal submatrix* $\nabla F(x^*)_{ff}$ *is nonsingular, and*

(b) the Schur complement $\nabla F(x^*)/\nabla F(x^*)_{ff}$ *is a* P_0*-matrix.*

Then x^* *is a solution of the mixed complementarity problem.*

Proof. Let x^* be a stationary point of Ψ. Then, by Proposition 6.1.7, we have

$$H^T\Phi(x^*) = \nabla\Psi(x^*) = 0 \tag{35}$$

for an arbitrary $H \in \partial\Phi(x^*)$. By Proposition 6.1.2, there exist diagonal matrices $D_a(x^*), D_b(x^*) \in \mathbf{R}^{n\times n}$ such that

$$H = D_a(x^*) + D_b(x^*)F'(x^*).$$

Therefore, (35) can be written as

$$\left[D_a(x^*) + \nabla F(x^*)D_b(x^*)\right]\Phi(x^*) = 0. \tag{36}$$

Writing

$$D_a(x^*) = \begin{pmatrix} (D_a)_{ff}(x^*) & 0 \\ 0 & (D_a)_{\bar f\bar f}(x^*) \end{pmatrix},$$

$$D_b(x^*) = \begin{pmatrix} (D_b)_{ff}(x^*) & 0 \\ 0 & (D_b)_{\bar f\bar f}(x^*) \end{pmatrix}$$

and

$$\nabla F(x^*) = \begin{pmatrix} \nabla F(x^*)_{ff} & \nabla F(x^*)_{f\bar f} \\ \nabla F(x^*)_{\bar f f} & \nabla F(x^*)_{\bar f\bar f} \end{pmatrix},$$

where $I_{\bar f} := I \setminus I_f$, and taking into account that

$$\begin{aligned} (D_a)_{ii}(x^*) &= 0 \qquad \forall i \in I_f, \\ (D_b)_{ii}(x^*) &= -1 \qquad \forall i \in I_f \end{aligned}$$

by Proposition 6.1.2, we can rewrite (36) as

$$\begin{aligned} -\nabla F(x^*)_{ff}\Phi(x^*)_f + \nabla F(x^*)_{f\bar{f}}(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}} &= 0_f, &(37)\\ (D_a)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}} - \nabla F(x^*)_{\bar{f}f}\Phi(x^*)_f + \nabla F(x^*)_{\bar{f}\bar{f}}(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}} &= 0_{\bar{f}}. &(38)\end{aligned}$$

Due to the assumed nonsingularity of $\nabla F(x^*)_{ff}$, we obtain from (37):

$$\Phi(x^*)_f = \nabla F(x^*)_{ff}^{-1}\nabla F(x^*)_{f\bar{f}}(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}. \tag{39}$$

Substituting this expression into (38) and rearranging terms gives

$$\left[(D_a)_{\bar{f}\bar{f}}(x^*) + (\nabla F(x^*)/\nabla F(x^*)_{ff})(D_b)_{\bar{f}\bar{f}}(x^*)\right]\Phi(x^*)_{\bar{f}} = 0_{\bar{f}}. \tag{40}$$

Define the index sets:

$$\begin{aligned} f_0 &:= \{i \mid [(D_a)_{\bar{f}\bar{f}}(x^*)]_{ii} = 0 \text{ or } [(D_b)_{\bar{f}\bar{f}}(x^*)]_{ii} = 0\}\\ \bar{f}_0 &:= \{i \mid [(D_a)_{\bar{f}\bar{f}}(x^*)]_{ii} < 0 \text{ and } [(D_b)_{\bar{f}\bar{f}}(x^*)]_{ii} < 0\}.\end{aligned}$$

Using Proposition 6.1.2, we have that $f_0 \cup \bar{f}_0 = \{1, \ldots, k\}$ where k is the cardinality of $\bar{f}$. Then, by Lemma 6.1.9, $[\Phi(x^*)_{\bar{f}}]_{f_0} = 0$. Consequently, (40) reduces to the system:

$$\left[[(D_a)_{\bar{f}\bar{f}}(x^*)]_{\bar{f}_0\bar{f}_0} + [\nabla F(x^*)/\nabla F(x^*)_{ff}]_{\bar{f}_0\bar{f}_0}[(D_b)_{\bar{f}\bar{f}}(x^*)]_{\bar{f}_0\bar{f}_0}\right]\Phi(x^*)_{\bar{f}_0} = 0_{\bar{f}_0}. \tag{41}$$

Since $(\nabla F(x^*)/\nabla F(x^*)_{ff})$ is a P_0-matrix by assumption, it follows that the principle submatrix, $[\nabla F(x^*)/\nabla F(x^*)_{ff}]_{\bar{f}_0\bar{f}_0}$, is also P_0-matrix. Then by Proposition 6.1.8 the coefficient matrix in (41) is nonsingular. Hence, $[\Phi(x^*)_{\bar{f}}]_{\bar{f}_0} = 0$. Therefore, $\Phi(x^*)_{\bar{f}} = 0_{\bar{f}}$. But then (39) implies $\Phi(x^*)_f = 0_f$. Hence, $\Phi(x^*) = 0$, i.e., x^* is a solution of the mixed complementarity problem by Proposition 6.1.1. □

We now provide a sufficient condition for a stationary point of the constrained reformulation of the mixed complementarity problem

$$\min\ \Psi(x) \quad \text{s.t.} \quad x \in [\ell, u] \tag{42}$$

to be a global minimum. In fact, this result is of much more importance than the unconstrained stationary point result given in Theorem 6.1.10. However, Theorem 6.1.10 will be used to establish the constrained stationary point result.

Theorem 6.1.11 *Let $x^* \in \mathbf{R}^n$ be a stationary point of the constrained reformulation (42) of the mixed complementarity problem. Assume that*

(a) the principal submatrix $\nabla F(x^*)_{ff}$ is nonsingular, and

(b) the Schur complement $\nabla F(x^*)/\nabla F(x^*)_{ff}$ is a P_0-matrix.

Then x^* is a solution of the mixed complementarity problem.

Proof. Since x^* is a stationary point of the reformulation (42), it satisfies the following conditions (which themselves form a mixed complementarity problem):

$$\begin{aligned} x_i^* = l_i &\implies [\nabla\Psi(x^*)]_i \geq 0, \\ x_i^* = u_i &\implies [\nabla\Psi(x^*)]_i \leq 0, \\ x_i^* \in (l_i, u_i) &\implies [\nabla\Psi(x^*)]_i = 0. \end{aligned} \tag{43}$$

The main part in proving that x^* is already a solution of the mixed complementarity problem consists in showing that we actually have $[\nabla\Psi(x^*)]_i = 0$ for all $i \in I$.

The proof is by contradiction, so assume that $\nabla\Psi(x^*) \neq 0$. Since $\nabla\Psi(x^*)$ can be written as

$$\nabla\Psi(x^*) = H^T\Phi(x^*) = D_a(x^*)\Phi(x^*) + \nabla F(x^*)D_b(x^*)\Phi(x^*)$$

for a matrix $H \in \partial\Phi(x^*)$ and certain diagonal matrices $D_a(x^*), D_b(x^*) \in \mathbf{R}^{n\times n}$ by Propositions 6.1.2 and 6.1.7, and since we necessarily have

$$[\nabla\Psi(x^*)]_f = 0_f$$

because of (43), we can follow the argument used in the proof of Theorem 6.1.10 in order to show that

$$\Phi(x^*)_f = \nabla F(x^*)_{ff}^{-1} \nabla F(x^*)_{f\bar{f}} (D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}.$$

Substituting this into the expression for $[\nabla\Psi(x^*)]_{\bar{f}}$ and rearranging terms, we obtain

$$[\nabla\Psi(x^*)]_{\bar{f}} = (D_a)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}} + (\nabla F(x^*)/\nabla F(x^*)_{ff})(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}. \tag{44}$$

Since $\nabla\Psi(x^*) \neq 0$ by assumption, there exists an index $i \in I_{\bar{f}}$ such that either

$$x_i^* = l_i \quad \text{and} \quad [\nabla\Psi(x^*)]_i > 0 \tag{45}$$

or

$$x_i^* = u_i \quad \text{and} \quad [\nabla\Psi(x^*)]_i < 0. \tag{46}$$

Now it follows easily from Proposition 6.1.2 that if $x_i^* = l_i$, then

$$[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i \leq 0$$

and that if $x_i^* = u_i$,

$$[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i \geq 0.$$

Therefore, if we premultiply $[\nabla\Psi(x^*)]_i$ in (45) and (46) by $[(D_b)_{\bar{f}\bar{f}}\Phi(x^*)_{\bar{f}}]_i$ and substitute the ith component from the expression (44) for $[\nabla\Psi(x^*)]_i$, we obtain in both cases that

$$\begin{aligned}&[(D_a)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i + \\ &\qquad [(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i[(\nabla F(x^*)/\nabla F(x^*)_{ff})(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i \leq 0.\end{aligned} \tag{47}$$

Note that this inequality holds for all indices $i \in I_{\bar{f}}$ such that $[\nabla\Psi(x^*)]_i \neq 0$. In addition, we can show in a similar way that the equality

$$\begin{aligned}&[(D_a)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i + \\ &\qquad [(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i[(\nabla F(x^*)/\nabla F(x^*)_{ff})(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_i = 0\end{aligned} \tag{48}$$

holds for all indices $i \in I_{\bar{f}}$ with $[\nabla\Psi(x^*)]_i = 0$.

Since $\nabla\Psi(x^*) \neq 0$ by assumption and since we already know that $[\nabla\Psi(x^*)]_f = 0$, it follows immediately from (44) and Lemma 6.1.9 (b) that $(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}$ is a nonzero vector. Therefore, since the Schur complement $\nabla F(x^*)/\nabla F(x^*)_{ff}$ is a P_0-matrix by assumption, there exists an index $i_0 \in I_{\bar{f}}$ such that

$$[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0} \neq 0 \quad \text{and}$$
$$[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0}[(\nabla F(x^*)/\nabla F(x^*)_{ff})(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0} \geq 0. \quad (49)$$

Now Lemma 6.1.9 (a), (47), (48) and (49) imply that

$$0 = [(D_a)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0}[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0}$$

and therefore $[(D_b)_{\bar{f}\bar{f}}(x^*)\Phi(x^*)_{\bar{f}}]_{i_0} = 0$ by Lemma 6.1.9 (b). This, however, contradicts the choice of the index i_0 in (49).

Hence we must have $\nabla\Psi(x^*) = 0$, so that Theorem 6.1.10 gives the desired result that x^* is a solution of the mixed complementarity problem. □

We note that, if we apply the main results of this section to the standard nonlinear complementarity problem, then we obtain some known properties [23, 34, 51] of the merit function Ψ.

6.1.3 Algorithmic Framework

We now present the algorithm used to solve the mixed complementarity problem and corresponding global and local convergence theory. We assume that we have a basic algorithm, which we will call **Algorithm LM** (for local method), with the following two properties:

(a) Given any point $x^k \in [\ell, u]$, if Algorithm LM is able to compute a search direction $d^k \in \mathbf{R}^n$, then this direction satisfies $x^k + d^k \in [\ell, u]$;

(b) Given any sequence $\{x^k\}$ converging to a strongly regular solution x^* of the mixed complementarity problem, Algorithm LM is able to compute a search direction d^k for all x^k sufficiently close to x^*, and this direction has the property that $\|x^k + d^k - x^*\| = o(\|x^k - x^*\|)$.

Property (a) is a very weak assumption; it does not even assume that Algorithm LM is able to do anything at an arbitrary given point x^k. For example, the subproblem might be inconsistent. However, if Algorithm LM is able to compute a search direction, we assume that it computes a search direction such that, if we take the full step, the new point $x^k + d^k$ stays in the feasible set $[\ell, u]$. Note that, due to the convexity of the feasible set $[\ell, u]$, this guarantees that all the points $x^k + t_k d^k, t_k \in [0, 1]$ are feasible, too. On the other hand, property (b) states that, under the strong regularity condition, Algorithm LM is locally well-defined and generates a locally superlinearly convergent search direction. Hence we view Algorithm LM as a feasible and locally superlinearly convergent method for the solution of the mixed complementarity problem. Note, however, that the above two conditions say nothing about the way in which we generate the sequence $\{x^k\}$.

Several methods satisfy the above two conditions. For example, one may take the Josephy-Newton method [73, 88] or the alternative method suggested in [108, 103]. The NE/SQP method [99] is another possible candidate as is the inexact QP-based solver in [74]. These latter two methods have been used to solve the standard complementarity problem only, but it is not difficult to extend both methods to mixed complementarity problems [5, 7].

The algorithm presented globalizes Algorithm LM using the merit function Ψ to measure any progress. If the point generated by Algorithm LM has a merit function value of Ψ sufficiently smaller than the previous one, it is accepted as the new iterate. Otherwise a projected gradient step for the smooth merit function Ψ is taken. In this way, we guarantee that all iterates stay in the feasible set $[\ell, u]$. In effect, the class of methods is an algorithmic framework for the solution of the box constrained optimization problem

$$\min \Psi(x) \quad \text{s.t.} \quad x \in [\ell, u]. \tag{50}$$

We now provide a detailed statement of the algorithm, where the projection of an arbitrary point $x \in \mathbf{R}^n$ on the feasible set $[\ell, u]$ is denoted by $\pi_{[\ell,u]}(x)$.

Algorithm GDM (General Descent Method)

(S.0) (Initialization)

Choose $x^0 \in [\ell, u], s > 0, \beta \in (0,1), \gamma \in (0,1)$ *and set* $k := 0$.

(S.1) (Termination Criterion)

If x^k *is a stationary point of (50): STOP.*

(S.2) (Compute Fast Search Direction)

Use Algorithm LM to compute a search direction d^k. *If this is not possible or if the condition*

$$\Psi(x^k + d^k) \leq \gamma \Psi(x^k) \tag{51}$$

is not satisfied, go to Step (S.4), else go to Step (S.3).

(S.3) (Accept Fast Search Direction)

Set $x^{k+1} := x^k + d^k, k \leftarrow k + 1$, *and go to Step (S.1).*

(S.4) (Take Projected Gradient Step)

Compute $t_k = \max\{s\beta^i \,|\, i = 0, 1, 2, \dots\}$ *such that*

$$\Psi(x^k(t_k)) \leq \Psi(x^k) - \sigma \nabla \Psi(x^k)^T (x^k - x^k(t_k)), \tag{52}$$

where $x^k(t) := \pi_{[\ell,u]}(x^k - t\nabla\Psi(x^k))$. *Set* $x^{k+1} := x^k(t_k), k \leftarrow k+1$, *and go to Step (S.1).*

Other methods have attempted to use projected gradient steps in conjunction with steps that give fast local convergence [48]. Unfortunately, these hybrid algorithms are difficult to implement and numerical testing has therefore only been carried out on small test examples. A key difference in the approach outlined here is that the implementation can be achieved as a modification to an existing code.

We now investigate the convergence properties of Algorithm GDM. To this end, we always assume implicitly that Algorithm GDM does not terminate in a finite number of steps. That is, none of the iterates x^k is a stationary point of (50).

The global convergence analysis consists of two parts: we first show that the algorithm is well-defined, and we then prove that any accumulation point of a sequence $\{x^k\}$ generated by Algorithm GDM is a stationary point of the bound constrained optimization problem

$$\min \Psi(x) \quad \text{s.t.} \quad x \in [\ell, u]. \tag{53}$$

Recall that Theorem 6.1.11 gives a relatively mild condition for a stationary point of (53) to be a solution of the mixed complementarity problem.

Theorem 6.1.12 *Algorithm GDM is well-defined for an arbitrary mixed complementarity problem with a continuously differentiable function F defined on an open set containing* *the rectangle $[\ell, u]$. Furthermore, every accumulation point of a sequence $\{x^k\}$ generated by Algorithm GDM is a stationary point of (53).*

Proof. To prove the algorithm is well-defined, we only have to show that the projected gradient step can be carried out at each iteration. That is, we can always find a finite step length $t_k > 0$ satisfying condition (52). However, since we assume that none of the iterates x^k is a stationary point of (50), this follows, e.g., from Proposition 2.3.3 (a) in Bertsekas [4].

For the second part of the theorem, let x^* be an accumulation point of the sequence $\{x^k\}$, and assume that $\{x^k\}_K$ is a subsequence converging to x^*. Suppose there are infinitely many $k \in K$ such that x^{k+1} is generated by using a projected gradient step for all these k. Since the iterates x^k belong to the feasible set $[\ell, u]$ for all $k \in \mathbb{N}$ and since the sequence $\{\Psi(x^k)\}$ is monotonically decreasing, it is not difficult to see that the proof of Proposition 2.3.3 (b) in Bertsekas [4] can be adapted in a straightforward manner to establish that x^* is a stationary point of the constrained reformulation (53).

Hence we can assume without loss of generality that all iterates $k \in K$ satisfy the descent condition (51). Due to the monotonic decrease of the sequence $\{\Psi(x^k)\}$, this implies that the entire sequence $\{\Psi(x^k)\}$ converges to 0. In particular, in view of the definition of the merit function, we see that the accumulation point x^* is a solution of the mixed complementarity problem and hence also a stationary point of problem (53). $\square$

We now show that Algorithm GDM is locally Q-superlinearly convergent under the strong regularity condition [106]. The proof is in two parts: we first show that the entire sequence generated by Algorithm GDM converges to a solution x^* if this solution satisfies the strong regularity assumption, and then we determine the rate of convergence. The critical tool to establish that the sequence converges is the following proposition.

Proposition 6.1.13 ([91]) *Assume that x^* is an isolated accumulation point of a sequence $\{x^k\}$ (not necessarily generated by Algorithm GDM) such that $\{\|x^{k+1}-x^k\|\}_K \to 0$ for any subsequence $\{x^k\}_K$ converging to x^*. Then the whole sequence $\{x^k\}$ converges to x^*.*

The basic device used to show Q-superlinear convergence of Algorithm GDM is to prove that eventually there are no projected gradient steps, so the method inherits the local convergence properties of the locally superlinearly convergent Algorithm LM used in Step (S.2). To simplify the proof, we invoke the following proposition.

Proposition 6.1.14 ([34, 76]) *Let $G : \mathbf{R}^n \to \mathbf{R}^n$ be locally Lipschitzian, $x^* \in \mathbf{R}^n$ with $G(x^*) = 0$ be such that all elements in $\partial G(x^*)$ are nonsingular, and assume that there are two sequences $\{x^k\} \subseteq \mathbf{R}^n$ and $\{d^k\} \subseteq \mathbf{R}^n$ (not necessarily generated by Algorithm GDM) with $\{x^k\} \to x^*$ and $\|x^k + d^k - x^*\| = o(\|x^k - x^*\|)$. Then $\|G(x^k + d^k)\| = o(\|G(x^k)\|)$.*

We can now state and prove the main local convergence result for Algorithm GDM. The convergence rate established depends critically on the main result of Section 6.1.1, namely Theorem 6.1.6.

Theorem 6.1.15 *Let $\{x^k\} \subseteq \mathbf{R}^n$ be a sequence generated by Algorithm GDM. Assume that this sequence has an accumulation point x^* which is a strongly regular solution of the mixed complementarity problem. Then the entire sequence $\{x^k\}$ converges to this point, and the rate of convergence is Q-superlinear.*

Proof. To establish convergence, we first note that a strongly regular solution is an isolated solution of the mixed complementarity problem, see [106]. Since Algorithm GDM generates a decreasing sequence $\{\Psi(x^k)\}$ and x^* is a solution of the mixed complementarity problem, the entire sequence $\{\Psi(x^k)\}$ converges to zero. Hence every accumulation point of the sequence $\{x^k\}$ must be a solution of the mixed complementarity problem. Therefore, the assumed strong regularity of x^* implies that x^* is an isolated accumulation point of the sequence $\{x^k\}$.

Now let $\{x^k\}_K$ denote any subsequence converging to x^*. Assume first that we take a projected gradient step for all sufficiently large $k \in K$. Then we obtain, using the nonexpansive property of the projection operator:

$$\begin{aligned} \|x^{k+1} - x^k\| &= \|x^k(t_k) - x^k\| \\ &= \|[x^k - t_k \nabla\Psi(x^k)]_+ - x^k\| \\ &= \|[x^k - t_k \nabla\Psi(x^k)]_+ - [x^k]_+\| \\ &\leq \|t_k \nabla\Psi(x^k)\| \\ &\leq s\|\nabla\Psi(x^k)\| \\ &\to 0 \end{aligned} \tag{54}$$

since x^* solves the mixed complementarity problem so that x^* is a global minimizer and hence a stationary point of the merit function Ψ.

On the other hand, if we calculate and accept the search direction d^k generated by Algorithm LM for infinitely many $k \in K$, then the assumptions on Algorithm LM and the assumed strong regularity imply that $\{d^k\} \to 0$ on this infinite subsequence, so that the updating rules from Algorithm GDM show that

$$\|x^{k+1} - x^k\| = \|d^k\| \to 0 \tag{55}$$

on this subsequence. Combining (55) and (54), we immediately obtain that

$$\{\|x^{k+1} - x^k\|\}_K \to 0.$$

Hence the assumptions of Proposition 6.1.13 are satisfied, and convergence follows from that result.

In order to establish the Q-superlinear rate, we note that the strong regularity of the solution x^* and Theorem 6.1.6 show that all elements in $\partial\Phi(x^*)$ are nonsingular. Since, in view of the assumptions about the search directions d^k generated by Algorithm LM, we have $\|x^k + d^k - x^*\| = o(\|x^k - x^*\|)$ for these search directions, Proposition 6.1.14 implies that

$$\|\Phi(x^k + d^k)\| = o(\|\Phi(x^k)\|)$$

and therefore

$$\Psi(x^k + d^k) = o(\Psi(x^k)).$$

This shows that the descent condition

$$\Psi(x^k + d^k) = \gamma\Psi(x^k)$$

is eventually satisfied in Step (S.2) of Algorithm GDM, i.e., for all $k \in \mathbb{N}$ sufficiently large, Algorithm GDM does not take any projected gradient steps. Hence Algorithm GDM has the same local convergence properties as Algorithm LM. Since $x^{k+1} = x^k + d^k$, this means that $\{x^k\}$ converges Q-superlinearly to x^*. $\square$

Obviously, if the basic Algorithm LM is locally Q-quadratically convergent, then the class of algorithms given in Algorithm GDM is also locally Q-quadratically convergent. Typically, this holds if we assume in addition that the Jacobian of F is locally Lipschitzian.

6.2 Implementation Features

PATH 4.x uses Algorithm GDM to globalize the Newton method from [103, 108]. This local method constructs a point-based approximation of the normal map [107]:

$$F(\pi_{[\ell,u]}(x)) + x - \pi_{[\ell,u]}(x)$$

where $\pi_{[\ell,u]}(x)$ represents the projection of x onto $[\ell, u]$. The resulting linear complementarity problem is solved using a pivotal code related to Lemke's method [81] to calculate a Newton point.

Two implementational points are of interest. We first need to reconcile the fact that the nonlinear code generates iterates in $[\ell, u]$, while the linear solver expects and returns a point in $\mathbf{R}^n$. The point based approximation is formed at $x^k \in [\ell, u]$. The iterate passed into the linear solver, y^k, is chosen to have the best normal map residual among all y such that $\pi_{[\ell,u]}(y) = x^k$. The point returned from the linear solver is termed x^N and is used in a backtracking search instead of the simple test given as (51). This search inspects points on the following arc parametrized by $t \in (0, 1]$:

$$\pi_{[\ell,u]}(x^k + t(x^N - x^k)).$$

Note that, in general, this is not the line segment joining x^k to x^N. The projection is necessary so that x^{k+1} remains in $[\ell, u]$. A projected gradient step was only taken if suitable descent did not occur for some minimum step length allowed. Secondly, the gradient of the merit function required for (52) was calculated using the formulas detailed in Propositions 6.1.7 and 6.1.2 and the method described in [5].

Many heuristics have been incorporated into the implementation of PATH 4.x. These are documented in the sequel. Note that previous version of PATH [25, 27, 28] are based entirely on the algorithms in [108, 103] and are globalized by using a pathsearch on the two-norm squared of the normal map. Hence, all of the searches were updating to use the Fischer-Burmeister merit function.

6.2.1 Crashing Method

The crashing technique [29] is used to quickly identify an active set from the user-supplied starting point. At this time, a proximal perturbation scheme [5, 6] is used to overcome problems with a singular basis matrix. The proximal perturbation is introduced in the crash method when the matrix factored is determined to be singular. The value of the perturbation is based on the current merit function value.

Even if the crash method is turned off, a perturbation is added to the model if the matrix that would have initially been factored in the crash method is numerically singular. This behavior is extremely useful for introducing a perturbation for singular models.

6.2.2 Nonmonotone Searches

The first line of defense against convergence to stationary points is the use of a nonmonotone linesearch [63, 64, 38]. In this case we define a reference value, R^k and we use this value in the test for sufficient decrease

$$\Psi(x^k(t_k)) \leq \mathbf{R}^k - \sigma \nabla \Psi(x^k)^T (x^k - x^k(t_k)).$$

Depending upon the choice of the reference value, this allows the merit function to increase from one iteration to the next. This strategy can not only improve convergence, but can also avoid local minimizers by allowing such increases.

We now need to detail the choice of the reference value. Let $\{M_1, \ldots, M_m\}$ be a finite set of values initialized to $\kappa\Psi(x^0)$, where κ is used to determine the initial set of acceptable merit function values and defaults to 20 in the code. Note $\kappa = 1$ indicates that we are not going to allow the merit function to increase beyond the initial value.

Having defined the values of $\{M_1, \ldots, M_m\}$ (where we use $m = 10$ by default), we can now calculate a reference value. Assuming that d^k is the Newton direction, we define $i_0 = \operatorname{argmax} M_i$ and $R^k = M_{i_0}$. After the nonmonotone linesearch finds t_k, we update the memory so that $M_{i_0} = \Psi(x^k + t_k d^k)$. That is, we remove the element from the memory having the largest merit function value.

When a gradient step is used, it is beneficial to let $x^k = x^{\text{best}}$ where x^{best} is the point with the absolute best merit function value encountered so far. We then recalculate $d^k = -\nabla\Psi(x^k)$ using the best point and let $R^k = \Psi(x^k)$. That is, we force decrease from the best iterate found whenever a gradient step is performed. After a successful step we set $M_i = \Psi(x^k + t_k d^k)$ for all $i \in [1, \ldots, m]$. This prevents future iterates from returning to the same problem area.

A watchdog strategy [12] is also available for use in the code. The method employed allows steps to be accepted when they are "close" to the current iterate. Nonmonotonic decrease is enforced every 10 iterations by default.

When the nonmonotone linesearch and projected gradient steps fail, the problem is restarted from the initial starting point with a modified set of options. These restarts change the algorithm in the hopes that the modified algorithm leads to a solution. The ordering and nature of the restarts were determined by empirical evidence based upon tests performed on real-world problems.

6.2.3 Linear Complementarity Subproblems

All versions of PATH solve a linear complementarity problem each major iteration. Let $M \in \mathbb{R}^{n \times n}$, $q \in \mathbb{R}^n$, and $[\ell, u]$ be given. $(\bar{z}, \bar{w}, \bar{v})$ solves the linear mixed complementarity problem defined by M, q, and $[\ell, u]$ if and only if it satisfies the following constrained system of equations:

$$Mz - w + v + q = 0 \tag{56}$$

$$w^T(z - \ell) = 0 \tag{57}$$

$$v^T(u - z) = 0 \tag{58}$$

$$z \in B, w \in \mathbb{R}^n_+, v \in \mathbb{R}^n_+, \tag{59}$$

where $x + \infty = \infty$ for all $x \in \mathbb{R}$ and $0 \cdot \infty = 0$ by convention. A triple, $(\hat{z}, \hat{w}, \hat{v})$, satisfying equations (56) - (58) is called a complementary triple.

The objective of the linear model solver is to construct a path from a given complementary triple $(\hat{z}, \hat{w}, \hat{v})$ to a solution $(\bar{z}, \bar{w}, \bar{v})$. The algorithm used to solve the linear problem is identical to that given in [25]; however, artificial variables are incorporated into the model and the residual is scaled by s. The augmented system is then:

$$Mz - w + v + Da + \frac{(1-t)}{s}(sr) + q = 0 \tag{60}$$

$$w^T(z - \ell) = 0 \tag{61}$$

$$v^T(u - z) = 0 \tag{62}$$

$$z \in B, w \in \mathbb{R}^n_+, v \in \mathbb{R}^n_+, a \equiv 0, t \in [0, 1] \tag{63}$$

where r is the residual, t is the path parameter, a is a vector of artificial variables, and D is a permuted diagonal matrix.

Artificial variables are used to construct an initial invertible basis consistent with the given starting point even under rank deficiency. The procedure consists of two parts:

constructing an initial guess as to the basis and then recovering from rank deficiency to obtain an invertible basis. The crash technique gives a good approximation to the active set which is used in the first phase of the algorithm to construct a basis by partitioning the variables into three sets:

1. $W = \{i \in \{1, \dots, n\} \mid \hat{z}_i = l_i \text{ and } \hat{w}_i > 0\}$
2. $V = \{i \in \{1, \dots, n\} \mid \hat{z}_i = u_i \text{ and } \hat{v}_i > 0\}$
3. $Z = \{1, \dots, n\} \setminus W \cup V$

Since $(\hat{z}, \hat{w}, \hat{v})$ is a complementary triple, $Z \cap W \cap V = \emptyset$ and $Z \cup W \cup V = \{1, \dots, n\}$. Using the above guess, we can recover an invertible basis consistent with the starting point by defining D appropriately. The technique relies upon the factorization to tell the linearly dependent rows and columns of the basis matrix. Some of the variables may be nonbasic, but not at their bounds. For such variables, the corresponding artificial will be basic.

We use a modified version of EXPAND [57] to perform the ratio test. Variables are prioritized as follows:

1. t leaving at its upper bound.
2. Any artificial variable.
3. Any z, w, or v variable.

If a choice as to the leaving variable can be made while maintaining numerical stability and sparsity, we choose the variable with the highest priority (lowest number above).

When an artificial variable leaves the basis and a z-type variable enters, we have the choice of either increasing or decreasing that entering variable because it is nonbasic but not at a bound. The determination is made such that t increases and stability is preserved.

If the code is forced to use a ray start at each iteration, then the code carries out Lemke's method, which is known [20] not to cycle. However, by default, we use a regular start to guarantee that the generated path emanates from the current iterate. Under appropriate conditions, this guarantees a decrease in the normal map residual. However, it is then possible for the pivot sequence in the linear model to cycle. To prevent this undesirable outcome, we attempt to detect the formation of a cycle with the heuristic that if a variable enters the basis more that a given number of times, we are cycling. The number of times the variable has entered is reset whenever t increases beyond its previous maximum or an artificial variable leaves the basis. If cycling is detected, we terminate the linear solver at the largest value of t and return this point.

Another heuristic is added when the linear code terminates on a ray. The returned point in this case is not the base of the ray. We move a slight distance up the ray and return this new point. The next linear subproblem should then have a different point-based approximation, which we attempt to solve.

Since the EXPAND pivot rules are used, some of the variables may be nonbasic, but slightly infeasible, at the solution. Whenever the linear code finishes, the nonbasic variables are put at their bounds and the basic variables are recomputed using the current factorization. The final step projects the recomputed basic variables onto their bounds. This correction procedure helps to find the best possible solution to the linear system.

6.2.4 Other Features

Other heuristics are incorporated into the code. During the first iteration, if the linear solver fails to find a Newton point, a Lemke ray start is used. Furthermore, if we repeatedly fail to solve the linear subproblems, an advanced ray start will be attempted. The advanced ray start chooses the "closest" extreme point of the $[\ell, u]$ polytope and selects a ray in the interior of the normal cone at this point. This technique helps to reduce the number of pivots required to solve the linear subproblem when compared to a Lemke ray start. However, when the basis corresponding to the cell of the normal manifold selected is not invertible the procedure fails and we resort to using a Lemke ray start. Computational experience has shown this to be an effective heuristic and generally results in solving the linear model. Using Lemke ray starts are not the default mode, since typically many more pivots are required.

The proximal point perturbation is shrunk each major iteration. However, when numerical difficulties are encountered, the value is increased to a fraction of the current merit function value.

We finally note, that when the merit function fails to show sufficient decrease over the last 100 iterates, a restart will be performed, as this tends to indicate that we are close to a stationary point.

6.3 Computational Results

In the following tables, we report the number of successes and failures (the latter in parenthesis) of the new code, PATH 4.x, compared to the PATH 2.9 code [25] from all starting points in the GAMSLIB, MCPLIB [26], and NETLIB [55] collections of test problems. The NETLIB problems are originally linear programs. We formed the optimality conditions for these linear programs and solved the resulting complementarity problems for the NETLIB tests.

In the reported results, PATH 4.x code is able to carry out at most 5 projected gradient steps when the direction provided by the linear subproblem is poor. Since the merit function in PATH 2.9 is nonsmooth, there is no possibility of carrying out a similar scheme in this code. The projected gradient steps enable progress to be made in PATH 4.x, which cannot occur in PATH 2.9. We note that a total of 361 projected gradient steps were taken during the course of 1400 runs, indicating the preference to take such steps only as a last resort (i.e. only after the nonmonotone search and the watchdog strategy fail). Several heuristic procedures were described and tested in [29]; all of these appear not to be beneficial to PATH 4.x.

PATH 2.9 did not encounter all of the subproblems for `endog`, `romer`, and `spill1` because of failures. Thus, there is a discrepancy between the totals reported in the MCPLIB test set. However, we can see from the results that PATH 4.x is considerably more reliable than PATH 2.9 even when restarts are not allowed. Note that there is no way to turn off the ad-hoc heuristics contained in PATH 2.9, so the appropriate comparison is with the version of PATH 4.x allowing restarts.

To conclude this section, Table 40 and Table 41 provide more complete information for PATH 4.x on a subset of the test problems. These problems were selected by taking every 10th run in an alphabetical ordering of all the test problems considered. However, we only report the first three instances of any given problem. We believe this is an unbiased sample of the test results.

The columns of these tables indicate the starting point number (SP), number of

Table 33: Comparison of PATH 4.x and PATH 2.9 on GAMSLIB

Problem	PATH 2.9	Without Restarts PATH 4.x	With Restarts PATH 4.x
cafemge	2 (0)	1 (1)	2 (0)
cammcp	1 (0)	1 (0)	1 (0)
cammge	1 (0)	1 (0)	1 (0)
cirimge	6 (0)	6 (0)	6 (0)
co2mge	6 (0)	6 (0)	6 (0)
dmcmge	2 (0)	2 (0)	2 (0)
ers82mcp	1 (0)	1 (0)	1 (0)
etamge	1 (0)	1 (0)	1 (0)
finmge	4 (0)	4 (0)	4 (0)
gemmcp	4 (0)	4 (0)	4 (0)
gemmge	4 (0)	4 (0)	4 (0)
hansmcp	1 (0)	1 (0)	1 (0)
hansmge	1 (0)	1 (0)	1 (0)
harkmcp	4 (0)	4 (0)	4 (0)
harmge	6 (0)	6 (0)	6 (0)
kehomge	3 (0)	3 (0)	3 (0)
kormcp	1 (0)	1 (0)	1 (0)
mr5mcp	1 (0)	1 (0)	1 (0)
nsmge	1 (0)	1 (0)	1 (0)
oligomcp	1 (0)	1 (0)	1 (0)
sammge	16 (0)	16 (0)	16 (0)
scarfmcp	1 (0)	1 (0)	1 (0)
scarfmge	4 (0)	4 (0)	4 (0)
shovmge	4 (0)	4 (0)	4 (0)
threemge	9 (0)	9 (0)	9 (0)
transmcp	3 (0)	3 (0)	3 (0)
two3mcp	2 (0)	2 (0)	2 (0)
unstmge	1 (0)	1 (0)	1 (0)
vonthmcp	1 (0)	0 (1)	1 (0)
vonthmge	1 (0)	1 (0)	1 (0)
wallmcp	1 (0)	1 (0)	1 (0)
Total	94 (0)	92 (2)	94 (0)

Table 34: Comparison of PATH 4.x and PATH 2.9 on MCPLIB

Problem	PATH 2.9	Without Restarts PATH 4.x	With Restarts PATH 4.x
asean9a	1 (0)	1 (0)	1 (0)
badfree	0 (1)	1 (0)	1 (0)
bai_haung	1 (0)	1 (0)	1 (0)
bert_oc	4 (0)	4 (0)	4 (0)
bertsekas	6 (0)	6 (0)	6 (0)
billups	0 (3)	0 (3)	0 (3)
bishop	1 (0)	1 (0)	1 (0)
box	352 (9)	361 (0)	361 (0)
bratu	1 (0)	1 (0)	1 (0)
cammcf	0 (1)	0 (1)	0 (1)
carbon	6 (1)	6 (1)	6 (1)
cgereg	19 (0)	19 (0)	19 (0)
choi	1 (0)	1 (0)	1 (0)
colvdual	4 (0)	2 (2)	4 (0)
colvnlp	6 (0)	6 (0)	6 (0)
congest	1 (0)	1 (0)	1 (0)
cycle	1 (0)	1 (0)	1 (0)
degen	1 (0)	1 (0)	1 (0)
denmark	34 (4)	38 (0)	38 (0)
dirkse1	0 (1)	1 (1)	1 (1)
duopoly	0 (1)	0 (1)	0 (1)
eckstein	0 (1)	1 (0)	1 (0)
ehl_k40	2 (1)	2 (1)	3 (0)
ehl_k60	3 (0)	2 (1)	3 (0)
ehl_k80	3 (0)	3 (0)	3 (0)
ehl_kost	3 (0)	3 (0)	3 (0)
electric	1 (0)	0 (1)	1 (0)
endog	0 (3)	23 (0)	23 (0)
eppa	8 (0)	8 (0)	8 (0)
eta2100	1 (0)	1 (0)	1 (0)
exemptions	181 (5)	186 (0)	186 (0)
explcp	1 (0)	1 (0)	1 (0)
exros	5 (0)	2 (3)	5 (0)

Table 35: Comparison of PATH 4.x and PATH 2.9 on MCPLIB (cont.)

Problem	PATH 2.9	Without Restarts PATH 4.x	With Restarts PATH 4.x
finance1	21 (0)	21 (0)	21 (0)
finance2	10 (0)	10 (0)	10 (0)
finance3	20 (0)	20 (0)	20 (0)
fixedpt	0 (1)	0 (1)	1 (0)
forcebsm	1 (0)	1 (0)	1 (0)
forcedsa	1 (0)	1 (0)	1 (0)
freebert	7 (0)	7 (0)	7 (0)
fried7	5 (0)	5 (0)	5 (0)
fried8	4 (1)	2 (3)	3 (2)
gafni	3 (0)	3 (0)	3 (0)
games	22 (3)	23 (2)	25 (0)
gei	2 (0)	2 (0)	2 (0)
golanmcp	1 (0)	1 (0)	1 (0)
hanskoop	10 (0)	10 (0)	10 (0)
hansmcf	1 (0)	1 (0)	1 (0)
hanson	1 (0)	1 (0)	1 (0)
ho	5 (0)	5 (0)	5 (0)
hydroc06	1 (0)	1 (0)	1 (0)
hydroc20	1 (0)	1 (0)	1 (0)
jel	2 (0)	2 (0)	2 (0)
jmu	1 (0)	0 (1)	1 (0)
josephy	8 (0)	8 (0)	8 (0)
keyzer	4 (2)	5 (1)	6 (0)
kojshin	8 (0)	8 (0)	8 (0)
kyh-scale	1 (1)	0 (2)	1 (1)
kyh	0 (2)	2 (0)	2 (0)
leyffer	1 (0)	1 (0)	1 (0)
lincont	1 (0)	1 (0)	1 (0)
lstest	0 (1)	0 (1)	0 (1)

Table 36: Comparison of PATH 4.x and PATH 2.9 on MCPLIB (cont.)

Problem	PATH 2.9	Without Restarts PATH 4.x	With Restarts PATH 4.x
markusen	18 (0)	18 (0)	18 (0)
mathinum	6 (0)	6 (0)	6 (0)
mathisum	6 (0)	6 (0)	6 (0)
methan08	1 (0)	1 (0)	1 (0)
mr5mcf	1 (0)	1 (0)	1 (0)
mrt	1 (0)	1 (0)	1 (0)
multi-v1	1 (0)	1 (0)	1 (0)
multi-v2	1 (0)	1 (0)	1 (0)
multi-v3	1 (0)	1 (0)	1 (0)
munson3	1 (0)	0 (1)	1 (0)
munson4	3 (0)	3 (0)	3 (0)
nash	4 (0)	4 (0)	4 (0)
ne-hard	0 (1)	1 (0)	1 (0)
nepal1	2 (1)	0 (3)	2 (1)
nepal2	0 (1)	1 (0)	1 (0)
obstacle	8 (0)	8 (0)	8 (0)
olg	1 (0)	1 (0)	1 (0)
opening	0 (2)	2 (0)	2 (0)
opt_cont	1 (0)	1 (0)	1 (0)
opt_cont127	1 (0)	1 (0)	1 (0)
opt_cont255	1 (0)	1 (0)	1 (0)
opt_cont31	1 (0)	1 (0)	1 (0)
opt_cont511	1 (0)	1 (0)	1 (0)
pgvon105	3 (3)	5 (1)	5 (1)
pgvon106	2 (4)	5 (1)	5 (1)
pies	1 (0)	1 (0)	1 (0)
pizer	3 (0)	3 (0)	3 (0)
powell	5 (1)	6 (0)	6 (0)
powell_mcp	6 (0)	6 (0)	6 (0)
qp	1 (0)	1 (0)	1 (0)

Table 37: Comparison of PATH 4.x and PATH 2.9 on MCPLIB (cont.)

Problem	PATH 2.9	Without Restarts PATH 4.x	With Restarts PATH 4.x
ralph	6 (1)	7 (0)	7 (0)
renger	0 (1)	0 (1)	0 (1)
romer	0 (1)	1 (1)	2 (0)
romer2	1 (6)	0 (7)	1 (6)
romer3	0 (1)	0 (1)	0 (1)
runge	7 (0)	7 (0)	7 (0)
scarfanum	4 (0)	4 (0)	4 (0)
scarfasum	4 (0)	4 (0)	4 (0)
scarfbnum	2 (0)	2 (0)	2 (0)
scarfbsum	1 (1)	2 (0)	2 (0)
shubik	43 (4)	43 (4)	47 (0)
simple-ex	0 (1)	0 (1)	0 (1)
simple-red	1 (0)	1 (0)	1 (0)
spill1	0 (0)	1 (0)	1 (0)
spill2	1 (0)	0 (1)	1 (0)
spill3	0 (1)	0 (1)	1 (0)
spill4	0 (1)	1 (0)	1 (0)
spill5	0 (1)	1 (0)	1 (0)
sppe	3 (0)	3 (0)	3 (0)
tinloi	58 (6)	56 (8)	64 (0)
tinloilp	4 (0)	4 (0)	4 (0)
tinsmall	63 (1)	63 (1)	64 (0)
titan	1 (1)	1 (1)	1 (1)
tobin	4 (0)	4 (0)	4 (0)
tqbilat	1 (0)	1 (0)	1 (0)
trade12	2 (0)	2 (0)	2 (0)
trafelas	2 (0)	2 (0)	2 (0)
trig	3 (0)	3 (0)	3 (0)
uruguay	4 (0)	4 (0)	4 (0)
venables	1 (1)	2 (0)	2 (0)
vonthmcf	1 (0)	1 (0)	1 (0)
water	2 (1)	2 (1)	3 (0)
xu1	7 (0)	7 (0)	7 (0)
xu2	7 (0)	7 (0)	7 (0)
xu3	7 (0)	7 (0)	7 (0)
xu4	7 (0)	7 (0)	7 (0)
xu5	7 (0)	7 (0)	7 (0)
Total	1124 (85)	1171 (60)	1207 (24)

Table 38: Comparison of PATH 4.x and PATH 2.9 on NETLIB

Problem		Without Restarts	With Restarts
	PATH 2.9	PATH 4.x	PATH 4.x
adlittle	1 (0)	1 (0)	1 (0)
afiro	1 (0)	1 (0)	1 (0)
agg	1 (0)	1 (0)	1 (0)
agg2	1 (0)	1 (0)	1 (0)
agg3	1 (0)	1 (0)	1 (0)
bandm	1 (0)	1 (0)	1 (0)
beaconfd	1 (0)	1 (0)	1 (0)
blend	1 (0)	1 (0)	1 (0)
bnl1	1 (0)	1 (0)	1 (0)
bnl2	0 (1)	1 (0)	1 (0)
boeing1	1 (0)	1 (0)	1 (0)
boeing2	1 (0)	1 (0)	1 (0)
bore3d	1 (0)	1 (0)	1 (0)
brandy	1 (0)	1 (0)	1 (0)
capri	1 (0)	1 (0)	1 (0)
cycle	0 (1)	0 (1)	0 (1)
czprob	1 (0)	1 (0)	1 (0)
d2q06c	0 (1)	0 (1)	0 (1)
d6cube	1 (0)	1 (0)	1 (0)
degen2	1 (0)	1 (0)	1 (0)
degen3	1 (0)	1 (0)	1 (0)
dfl001	0 (1)	0 (1)	0 (1)
e226	1 (0)	1 (0)	1 (0)
etamacro	1 (0)	1 (0)	1 (0)
fffff800	0 (1)	1 (0)	1 (0)
finnis	1 (0)	1 (0)	1 (0)
fit1d	1 (0)	1 (0)	1 (0)
fit1p	1 (0)	1 (0)	1 (0)
fit2d	0 (1)	0 (1)	0 (1)
fit2p	0 (1)	0 (1)	0 (1)
forplan	1 (0)	1 (0)	1 (0)
ganges	1 (0)	1 (0)	1 (0)
gfrd-pnc	0 (1)	1 (0)	1 (0)
greenbea	0 (1)	0 (1)	0 (1)
greenbeb	0 (1)	0 (1)	0 (1)
grow15	1 (0)	1 (0)	1 (0)
grow22	1 (0)	1 (0)	1 (0)
grow7	1 (0)	1 (0)	1 (0)

Table 39: Comparison of PATH 4.x and PATH 2.9 on NETLIB (cont.)

Problem		Without Restarts	With Restarts
	PATH 2.9	PATH 4.x	PATH 4.x
israel	1 (0)	1 (0)	1 (0)
kb2	1 (0)	1 (0)	1 (0)
lotfi	1 (0)	1 (0)	1 (0)
nesm	0 (1)	0 (1)	0 (1)
perold	0 (1)	1 (0)	1 (0)
pilot4	0 (1)	0 (1)	0 (1)
sc105	1 (0)	1 (0)	1 (0)
sc205	1 (0)	1 (0)	1 (0)
sc50a	1 (0)	1 (0)	1 (0)
sc50b	1 (0)	1 (0)	1 (0)
scagr25	1 (0)	1 (0)	1 (0)
scagr7	1 (0)	1 (0)	1 (0)
scfxm1	1 (0)	1 (0)	1 (0)
scfxm2	1 (0)	1 (0)	1 (0)
scfxm3	1 (0)	1 (0)	1 (0)
scorpion	1 (0)	1 (0)	1 (0)
scrs8	1 (0)	1 (0)	1 (0)
scsd1	1 (0)	1 (0)	1 (0)
scsd6	1 (0)	1 (0)	1 (0)
scsd8	1 (0)	1 (0)	1 (0)
sctap1	1 (0)	1 (0)	1 (0)
sctap2	1 (0)	1 (0)	1 (0)
sctap3	1 (0)	1 (0)	1 (0)
share1b	1 (0)	1 (0)	1 (0)
share2b	1 (0)	1 (0)	1 (0)
shell	1 (0)	1 (0)	1 (0)
ship04l	1 (0)	1 (0)	1 (0)
ship04s	1 (0)	1 (0)	1 (0)
ship08l	1 (0)	1 (0)	1 (0)
ship08s	1 (0)	1 (0)	1 (0)
ship12l	0 (1)	1 (0)	1 (0)
ship12s	1 (0)	1 (0)	1 (0)
stair	1 (0)	1 (0)	1 (0)
stocfor1	1 (0)	1 (0)	1 (0)
stocfor2	1 (0)	1 (0)	1 (0)
tuff	0 (1)	1 (0)	1 (0)
wood1p	0 (1)	1 (0)	1 (0)
Total	59 (16)	66 (9)	66 (9)

Table 40: Selected full results for PATH 4.x

Library	Problem	SP	MI	CI	R	PG	T
gamslib	cirimge	6	4	0	0	0	0.0 (0.0)
gamslib	etamge	1	7	0	0	0	0.1 (0.1)
gamslib	gemmge	3	5	0	0	0	0.3 (0.3)
gamslib	harmge	2	4	0	0	0	0.0 (0.0)
gamslib	nsmge	1	9	0	0	0	0.2 (0.1)
gamslib	sammge	11	3	0	0	0	0.0 (0.0)
gamslib	scarfmge	2	6	0	0	0	0.0 (0.0)
gamslib	threemge	6	4	0	0	0	0.0 (0.0)
gamslib	two3mcp	2	3	1	0	0	0.0 (0.0)
mcplib	bert_oc	3	0	3	0	0	0.6 (0.3)
mcplib	billups	3	16	2	3	0	* (*)
mcplib	box	9	3	0	0	0	0.0 (0.0)
mcplib	box	19	4	0	0	0	0.0 (0.0)
mcplib	box	29	3	0	0	0	0.0 (0.0)
mcplib	carbon	7	3	0	0	0	0.4 (0.4)
mcplib	cgereg	12	1	1	0	0	0.0 (0.0)
mcplib	cgereg	22	12	1	0	0	0.2 (0.2)
mcplib	colvnlp	5	6	1	0	0	0.0 (0.0)
mcplib	denmark	9	10	0	0	0	9.0 (8.5)
mcplib	denmark	26	12	0	0	0	4.2 (2.9)
mcplib	denmark	42	8	0	0	0	7.8 (10.6)
mcplib	ehl_k60	1	4	1	0	0	0.3 (0.2)
mcplib	endog	2	14	0	0	0	26.80 (*)
mcplib	endog	12	2	0	0	0	7.18 (*)
mcplib	endog	22	1	0	0	0	5.37 (*)
mcplib	eppa	8	4	0	0	0	1.7 (1.6)
mcplib	exemptions	10	7	0	0	0	3.6 (1.3)
mcplib	exemptions	20	4	0	0	0	2.3 (1.6)
mcplib	exemptions	30	8	0	0	0	3.8 (2.1)
mcplib	exros	2	115	1	1	0	14.2 (0.2)
mcplib	finance1	7	0	2	0	0	0.0 (0.0)
mcplib	finance1	17	0	1	0	0	0.0 (0.0)
mcplib	finance2	6	1	2	0	0	0.1 (0.0)
mcplib	finance3	6	0	1	0	0	0.0 (0.0)
mcplib	finance3	16	1	1	0	0	0.1 (0.0)
mcplib	freebert	3	3	1	0	0	0.0 (0.0)
mcplib	fried8	1	191	1	2	7	3.86 (*)
mcplib	games	3	42	1	0	0	0.2 (0.1)
mcplib	games	13	7	0	0	0	0.1 (0.0)
mcplib	games	23	5	1	0	0	0.0 (0.0)
mcplib	hanskoop	5	5	1	0	0	0.0 (0.0)
mcplib	ho	3	2	0	0	0	0.0 (0.0)

Table 41: Selected full results for PATH 4.x (cont.)

Library	Problem	SP	MI	CI	R	PG	T
mcplib	josephy	3	14	1	0	0	0.0 (0.0)
mcplib	keyzer	5	6	0	0	0	0.0 (0.0)
mcplib	kyh-scale	1	202	7	3	13	* (0.2)
mcplib	markusen	6	3	1	0	0	0.0 (0.0)
mcplib	markusen	24	5	0	0	0	0.0 (0.0)
mcplib	mathinum	6	6	1	0	0	0.0 (0.0)
mcplib	multi-v1	1	15	0	0	0	0.1 (0.0)
mcplib	nash	4	2	1	0	0	0.0 (0.0)
mcplib	obstacle	5	1	1	0	0	9.4 (1.0)
mcplib	opt_cont31	1	1	4	0	0	0.4 (0.2)
mcplib	pgvon106	3	18	1	0	0	0.3 (0.2)
mcplib	powell	3	6	1	0	0	0.0 (0.0)
mcplib	qp	1	1	1	0	0	0.0 (0.0)
mcplib	romer	2	25	0	0	1	1.33 (*)
mcplib	runge	2	23	0	0	0	0.0 (0.0)
mcplib	scarfasum	1	4	1	0	0	0.0 (0.0)
mcplib	shubik	3	9	1	0	0	0.0 (0.0)
mcplib	shubik	14	13	2	0	0	0.0 (*)
mcplib	shubik	24	2	1	0	0	0.0 (0.0)
mcplib	spill4	1	29	0	0	2	25.96 (*)
mcplib	tinloi	6	1	2	0	0	0.1 (0.0)
mcplib	tinloi	16	137	1	1	0	2.6 (*)
mcplib	tinloi	26	1	3	0	0	0.1 (0.0)
mcplib	tinloilp	2	3	2	0	0	114.1 (141.2)
mcplib	tinsmall	8	1	1	0	0	0.0 (0.0)
mcplib	tinsmall	18	2	1	0	0	0.0 (0.1)
mcplib	tinsmall	28	2	1	0	0	0.0 (0.1)
mcplib	tobin	2	7	2	0	0	0.0 (0.0)
mcplib	trig	3	68	1	0	1	0.1 (0.2)
mcplib	water	5	16	0	0	0	3.3 (30.0)
mcplib	xu2	3	4	1	0	0	0.0 (0.0)
mcplib	xu3	6	4	1	0	0	0.0 (0.0)
mcplib	xu5	2	5	1	0	0	0.0 (0.0)
netlib	agg3	1	22	2	0	0	1.1 (1.1)
netlib	capri	1	16	17	0	0	1.6 (2.6)
netlib	fffff800	1	27	0	0	0	15.8 (*)
netlib	greenbeb	1	9	29	0	0	* (*)
netlib	sc105	1	9	1	0	0	0.1 (0.1)
netlib	scrs8	1	15	1	0	0	3.8 (4.0)
netlib	ship04l	1	13	0	0	0	2.9 (2.6)
netlib	wood1p	1	4	0	0	0	465.18 (*)

major iterations (MI), crash iterations (CI), restarts (R), and projected gradient steps (PG) taken. The final column of this table reports the time for PATH 4.x in seconds, with the time for PATH 2.9 added in parentheses. All runs were carried out on a Sun UltraSparc 330 MHz processor with 768 MB RAM. A "*" indicates failure of the method. Both codes were compiled with the same compiler and options.

It is hard to draw firm conclusions from Table 40 and Table 41. The solution times of both algorithms are comparable, with some smaller times for PATH 4.x and some for PATH 2.9. There are only 6 problems in this subset which use projected gradient steps and restarts, the vast majority of them being solved without invoking these strategies. Overall, the theoretical extensions outlined and implemented result in the improved robustness of PATH 4.x over PATH 2.9 without any substantial changes in accuracy or speed.

6.4 User Documentation

The previous sections discussed the theoretical foundations of PATH 4.x, presented the implementation features, and demonstrated the reliability of the code. We now detail the PATH 4.x code from the perspective of a user. The log file informs a user about any progress made by the code. If the algorithm fails to solve a particular model, the log file can be used to track the reason for the failure. We document the output typically produced by PATH 4.x in Section 6.4.1, which assumes that the modeler is attempting to solve the `transport` model developed in Section 2.1. Section 6.4.2 discusses the available options used to modify the behavior of the algorithm. These options can be set by a modeler to improve the performance of the code for their particular complementarity problem.

```
Path 4.4a (Sat Feb 26 09:38:08 2000)

INITIAL POINT STATISTICS
Maximum of X. . . . . . . . . . -0.0000e+00 var: (x.seattle.new-york)
Maximum of F. . . . . . . . . .  6.0000e+02 eqn: (supply.san-diego)
Maximum of Grad F . . . . . . .  1.0000e+00 eqn: (demand.new-york)
                                            var: (x.seattle.new-york)

INITIAL JACOBIAN NORM STATISTICS
Maximum Row Norm. . . . . . . .  3.0000e+00 eqn: (supply.seattle)
Minimum Row Norm. . . . . . . .  2.0000e+00 eqn: (rational.seattle.new-york)
Maximum Column Norm . . . . . .  3.0000e+00 var: (p_supply.seattle)
Minimum Column Norm . . . . . .  2.0000e+00 var: (x.seattle.new-york)

Crash Log
major  func  diff  size  residual    step       prox   (label)
    0     0             1.0416e+03             0.0e+00 (demand.new-york)
    1     1     3     3 1.0029e+03  1.0e+00    1.0e+01 (demand.new-york)
pn_search terminated: no basis change.
```

Figure 35: Log File from PATH for solving `transmcp`

6.4.1 Anatomy of the Log File

We now describe the behavior of the PATH 4.x algorithm in terms of the output produced for the `transport` model. An example of the log for a particular run is given in Figure 35 and Figure 36.

After some basic memory allocation and problem checking, PATH 4.x checks if the modeler required an option file to be read. In this particular example, we did not request that an option file be used. However, if PATH 4.x is directed to read an option file, then the following output is generated after the PATH 4.x banner.

```
Reading options file PATH.OPT
 > output_linear_model yes;
Options: Read: Line 2 invalid: hi_there;
Read of options file complete.
```

If the option reader encounters an invalid option (as above), it reports this but carries on executing the algorithm. The user specifies option name, value pairs in the file. Note that

```
Major Iteration Log
major minor  func  grad  residual    step  type prox    inorm  (label)
    0     0     2     2 1.0029e+03           I 9.0e+00 6.2e+02 (demand.new-york)
    1     1     3     3 8.3096e+02  1.0e+00 SO 3.6e+00 4.5e+02 (demand.new-york)

...

   15     2    17    17 1.3972e-09  1.0e+00 SO 4.8e-06 1.3e-09 (demand.chicago)

FINAL STATISTICS
Inf-Norm of Complementarity . .  1.4607e-08 eqn: (rational.seattle.chicago)
Inf-Norm of Normal Map. . . . .  1.3247e-09 eqn: (demand.chicago)
Inf-Norm of Minimum Map . . . .  1.3247e-09 eqn: (demand.chicago)
Inf-Norm of Fischer Function. .  1.3247e-09 eqn: (demand.chicago)
Inf-Norm of Grad Fischer Fcn. .  1.3247e-09 eqn: (rational.seattle.chicago)

FINAL POINT STATISTICS
Maximum of X. . . . . . . . . .  3.0000e+02 var: (x.seattle.chicago)
Maximum of F. . . . . . . . . .  5.0000e+01 eqn: (supply.san-diego)
Maximum of Grad F . . . . . . .  1.0000e+00 eqn: (demand.new-york)
                                            var: (x.seattle.new-york)

 ** EXIT - solution found.

Major Iterations. . . . 15
Minor Iterations. . . . 31
Restarts. . . . . . . . 0
Crash Iterations. . . . 1
Gradient Steps. . . . . 0
Function Evaluations. . 17
Gradient Evaluations. . 17
Total Time. . . . . . . 0.020000
Residual. . . . . . . . 1.397183e-09
```

Figure 36: Log File from PATH for solving `transmcp` (cont.)

only the first three letters of each component of the option name is relevant. Therefore, the above option could be set with the string `out_lin_mod yes`. A complete listing of the options available are given in Section 6.4.2. Following the setting of the options, the algorithm starts working on the model.

6.4.1.1 Preprocessing

The first phase of PATH 4.x is to preprocess the model to reduce the size and complexity. However, the transportation model given to PATH 4.x is not reducible. For models that are reducible, the preprocessor reports its finding with output to the log file. An example of the preprocessing for the `forcebsm` model is presented below.

```
Zero:      0 Single:    112 Double:      0 Forced:      0
Preprocessed size: 72
```

The keywords indicate the type of elimination performed, while the corresponding integer tells the number of constraints where the rule was used. An indication of the size of the preprocessed model is also reported. See Chapter 4 for a complete description of the preprocessor.

On exit from the algorithm, we generate a solution for the original problem during a postsolve step. Following the postsolve, the residual of the solution generated for the original model is reported.

```
Postsolved residual: 1.0518e-10
```

This number should be approximately the same as the final residual reported for the presolved model.

6.4.1.2 Diagnostic Information

Following the preprocessing step, diagnostic information is generated at the starting point. This information is contained under the heading "INITIAL POINT STATISTICS". Included is information about the initial point and function evaluation. The log file tells the value of the largest element of the starting point and the variable where it occurs. Similarly, the maximum function value is displayed along with the name of the equation producing it. The maximum element in the gradient is also presented with the equation and variable where it is located.

The second block, "INITIAL JACOBIAN NORM STATISTICS", provides more information about the Jacobian at the starting point. This information can be used to help scale the model as detailed in Chapter 5.

6.4.1.3 Crash Log

The main computation in PATH 4.x begins with the crash procedure which attempts to quickly determine which of the inequalities should be active. This procedure is documented fully in [29], and an example of the Crash Log can be seen in Figure 35. The first column of the crash log is just a label indicating the current iteration number, the second gives an indication of how many function evaluations have been performed so far. Note that precisely one Jacobian (gradient) evaluation is performed per crash iteration. The number of changes to the active set between iterations of the crash procedure is shown under the "diff" column. The crash procedure terminates if this becomes small. Each iteration of this procedure involves a factorization of a matrix whose size is shown in the next column. The residual is a measure of how far the current iterate is from satisfying the complementarity conditions; it is zero at a solution. The column "step" corresponds to the step length taken in this iteration - ideally this should be 1. If the factorization fails, then the matrix is perturbed by an identity matrix scaled by the value indicated in the "prox" column. The "label" column indicates which row in the model is furthest away from satisfying the complementarity conditions. Typically, relatively few crash iterations are performed. Section 6.4.2 gives the options used to change the behavior of these steps.

6.4.1.4 Major Iteration Log

After the crash is completed, the main algorithm starts as indicated by the "Major Iteration Log" flag (see Figure 36). The columns that have the same labels as in the crash log have precisely the same meaning described above. However, there are some new columns that we now explain. Each major iteration attempts to solve a linear mixed complementarity problem using a pivotal method that is a generalization of Lemke's method [82]. The number of pivots performed `per` major iteration is given in the "minor" column.

The "grad" column gives the cumulative number of Jacobian evaluations used; typically one evaluation is performed per iteration. The "inorm" column gives the value of the error in satisfying the equation indicated in the "label" column.

At each iteration of the algorithm, several different step types can be taken, due to the use of nonmonotone searches [28, 38], which are used to improve robustness. In order to help the PATH 4.x user, we have added two code letters indicating the return code from the linear solver and the step type to the log file. Table 42 explains the return codes for the linear solver and Table 43 explains the meaning of each step type. The ideal output in this column is "SO", with "SD" and "SB" also being reasonable. Codes

Table 42: Linear Solver Codes

Code	Meaning
C	A cycle was detected.
E	An error occurred in the linear solve.
I	The minor iteration limit was reached.
N	The basis became singular.
R	An unbounded ray was encountered.
S	The linear subproblem was solved.
T	Failed to remain within tolerance after factorization was performed.

Table 43: Step Type Codes

Code	Meaning
B	A Backtracking search was performed from the current iterate to the Newton point in order to obtain sufficient decrease in the merit function.
D	The step was accepted because the Distance between the current iterate and the Newton point was small.
G	A Gradient step was performed.
I	Initial information concerning the problem is displayed.
M	The step was accepted because the Merit function value is smaller than the nonmonotone reference value.
O	A step that satisfies both the distance and merit function tests.
R	A Restart was carried out.
W	A Watchdog step was performed in which we returned to the last point encountered with a better merit function value than the nonmonotone reference value (M, O, or B step), regenerated the Newton point, and performed a backtracking search.

different from these are not catastrophic, but typically indicate that the solver is having difficulties due to numerical issues or nonmonotonicity in the model.

6.4.1.5 Minor Iteration Log

If more than 500 pivots are performed in the linear solver, a minor log is output that gives more details of the status of these pivots. A listing from `transmcp` model follows, where we have set the `output_minor_iteration_frequency` option to 1.

```
Minor Iteration Log
```

```
minor       t           z      w      v    art  ckpts  enter     leave
    1  4.2538e-01      8      2      0      0      0 t[     0] z[    11]
    2  9.0823e-01      8      2      0      0      0 w[    11] w[    10]
    3  1.0000e+00      9      2      0      0      0 z[    10] t[     0]
```

Note that `t` is a parameter that goes from zero to 1 for normal starts in the pivotal code. When the parameter reaches 1, we are at a solution to the subproblem. The `t` column gives the current value for this parameter. The next columns report the current number of problem variables z and slacks corresponding to variables at lower bound w and at upper bound v. Artificial variables are also noted in the minor log. Checkpoints are times where the basis matrix is refactored. The number of checkpoints is indicated in the `ckpts` column. Finally, the minor iteration log displays the entering and leaving variables during the pivot sequence.

6.4.1.6 Restart Log

The PATH 4.x code attempts to fully utilize the resources provided by the modeler to solve the problem. The algorithm tries to determine when a stationary point of the residual function has been encountered. When such a point is encountered and no progress is being made, the code is restarted from the initial point supplied with a different set of options. These restarts give the flexibility to change the algorithm behavior in the hopes that the modified algorithm leads to a solution. The ordering and nature of the restarts were determined by empirical evidence based upon tests performed on real-world problems.

The exact options set during the restart are given in the restart log, part of which is reproduced below.

```
Restart Log
proximal_perturbation 0
```

```
crash_method none
crash_perturb yes
nms_initial_reference_factor 2
proximal_perturbation 1.0000e-01
```

If a particular problem solves under a restart, a modeler can circumvent the wasted computation by setting the appropriate options as shown in the log. Note that sometimes an option is repeated in this log. In this case, the last value set is used.

6.4.1.7 Solution Log

A solution report is now given by the algorithm for the point returned. The first component, under the heading "FINAL STATISTICS", evaluates several different merit functions. Next, a display of some statistics concerning the final point is given. This report can be used detect problems with the model and solution as detailed in Chapter 5.

At the end of the log file, summary information regarding the algorithm's performance is given. The string "** EXIT - solution found" is an indication that PATH solved the problem. Any other EXIT string indicates termination at a point that may not be a solution.

6.4.2 Options Summary

The behavior of PATH 4.x can be modified by setting particular options. We give a list of the available options along with their default and meaning in Table 44 and Table 45. Note that only the first three characters of every word are significant.

The `proximal_perturbation` option can be used to overcome problems with a singular basis matrix. The results in a proximal point perturbation [5, 7] being applied. Linearly dependent columns can be output with the `output_factorization_singularities`

Table 44: PATH 4.x Options

Option	Default	Explanation
convergence_tolerance	1e-6	Stopping criterion
crash_iteration_limit	50	Maximum iterations allowed in crash
crash_merit_function	fischer	Merit function used in crash method
crash_method	pnewton	pnewton or none
crash_minimum_dimension	1	Minimum problem dimension needed to use crash technique
crash_nbchange_limit	1	Number of changes to the basis allowed
crash_perturb	yes	Perturb the problem using pnewton crash
crash_searchtype	line	Searchtype to use in the crash method
cumulative_iteration_limit	10000	Maximum minor iterations allowed
gradient_searchtype	arc	Searchtype to use when a gradient step is taken
gradient_step_limit	5	Maximum number of gradient step allowed before restarting
interrupt_limit	5	CTRL-C's required before killing job
lemke_start	automatic	Frequency of lemke starts (automatic, first, always)
major_iteration_limit	500	Maximum major iterations allowed
merit_function	fischer	Merit function to use (normal or fischer)
minor_iteration_limit	1000	Maximum minor iterations allowed in each major iteration
nms	yes	Allow line searching, watchdoging, and non-monotone descent
nms_initial_reference_factor	20	Controls size of initial reference value
nms_maximum_watchdogs	5	Maximum number of watchdog steps allowed
nms_memory_size	10	Number of reference values kept
nms_mstep_frequency	10	Frequency at which m steps are performed
nms_searchtype	line	Search type to use (line, or arc)
output	yes	Turn output on or off. If output is turned off, selected parts can be turned back on.
output_crash_iterations	yes	Output information on crash iterations
output_crash_iterations_frequency	1	Frequency at which crash iteration log is printed
output_errors	yes	Output error messages
output_factorization_singularities	yes	Output linearly dependent columns determined by factorization
output_final_degeneracy_statistics	no	Print information regarding degeneracy at the solution
output_final_point	no	Output final point returned from PATH
output_final_point_statistics	yes	Output information about the point, function, and Jacobian at the final point

Table 45: PATH 4.x Options (cont.)

Option	Default	Explanation
output_final_scaling_statistics	no	Display matrix norms on the Jacobian at the final point
output_final_statistics	yes	Output evaluation of available merit functions at the final point
output_final_summary	yes	Output summary information
output_initial_point	no	Output initial point given to PATH
output_initial_point_statistics	yes	Output information about the point, function, and Jacobian at the initial point
output_initial_scaling_statistics	yes	Display matrix norms on the Jacobian at the initial point
output_initial_statistics	no	Output evaluation of available merit functions at the initial point
output_linear_model	no	Output linear model each major iteration
output_major_iterations	yes	Output information on major iterations
output_major_iterations_frequency	1	Frequency at which major iteration log is printed
output_minor_iterations	yes	Output information on minor iterations
output_minor_iterations_frequency	500	Frequency at which minor iteration log is printed
output_model_statistics	yes	Turns on or off printing of all the statistics generated about the model
output_options	no	Output all options and their values
output_preprocess	yes	Output preprocessing information
output_restart_log	yes	Output options during restarts
output_warnings	no	Output warning messages
preprocess	yes	Attempt to preprocess the model
proximal_perturbation	0	Initial perturbation
restart_limit	3	Maximum number of restarts (0 - 3)
return_best_point	yes	Return the best point encountered or the absolute last iterate
time_limit	3600	Maximum number of seconds algorithm is allowed to run

option. For more information on the problems caused by singularity, we refer the reader to Chapter 5.

As a special case, PATH can emulate Lemke's method [20, 82] for LCP with the following options:

```
crash_method none;
crash_perturb no;
major_iteration_limit 1;
lemke_start first;
nms no;
```

If instead, PATH is to imitate the Successive Linear Complementarity method (SLCP, often called the Josephy-Newton method) [72, 88, 86] for MCP with an Armijo style linesearch on the normal map residual, then the options to use are:

```
crash_method none;
crash_perturb no;
lemke_start always;
nms_initial_reference_factor 1;
nms_memory size 1;
nms_mstep_frequency 1;
nms_searchtype line;
merit_function normal;
```

Note that the combination of `nms_memory_size 1` and `nms_initial_reference_factor 1` turns the nonmonotone linesearch off. The `nms_mstep_frequency 1` option disables watchdoging [12], while `nms_searchtype line` forces PATH to search the line segment between the initial point and the solution to the linear model. Finally, `merit_function normal` tells PATH to use the normal map for calculating the residual.

6.5 Summary

This chapter developed theory for the globalization of successive linearization algorithms using the Fischer-Burmeister merit function. Algorithm GDM was shown to be well-defined for arbitrary complementarity problems, as well as globally and locally fast convergent under a strong regularity assumption. This algorithm forms the basis for PATH 4.x which was shown to be more robust than previous versions of the code. In particular, PATH 4.x has been able to solve previously unsolvable models, such as the `romer` and `endog` models in MCPLIB. Finally, we documented the output generated by the PATH 4.x code and the options available to users.

We remark that PATH 2.9 was used for comparative purposes in this chapter as it was the last release of PATH written by Steven Dirkse and Michael Ferris. This code is based on the nonsmooth Newton method in [103, 108] and is documented in [25, 27, 28, 29].

The PATH 3.x series used PATH 2.9 as a template, but was completely rewritten. The main modification mades were the addition of artificial variables, the cycle detection rules, the EXPAND pivot rules, and an interpolating pathsearch. Diagnostic information and restarts were also introduced in the PATH 3.x series.

The PATH 4.x series is a major revision of PATH 3.x that implements Algorithm GDM and conforms to the solver interface in Chapter 3. We modified all of the searches (arc, line, and path) to use the Fischer-Burmeister function. Additional enhancements made in PATH 4.x are the correction step used at a the solution to the linear model, updates to the nonmonotone search for gradient steps, and preprocessing. We further note that the linear solver from PATH 4.x was reused in a predictor-corrector method [45].

We finally remark that an earlier version of Section 6.1 appeared in [37] and parts of

Section 6.2 and Section 6.4 appeared in [41, 42, 44].

Chapter 7

SEMI

PATH is currently the most widely used code for solving complementarity problems. While it may be argued that piecewise linear maps are more effective at approximating piecewise smooth maps, generating the “Newton” step typically involves the arduous task of solving a linear complementarity problem. A seemingly more attractive approach is to use an algorithm based on solving a system of linear equations to generate each “Newton” step. Recent theoretical work has outlined a host of methods with this property [15, 16, 23, 66, 98, 100]. Amongst these, the semismooth algorithm [23] appears to have some of the strongest associated theory.

SEMI is the result of an effort to develop a code based upon the semismooth algorithm. This chapter begins by briefly discussing the theoretical foundations of the semismooth algorithm in Section 7.1. Many of the results contained in this section are given without proof; instead, we provide references to the relevant literature. We then present the implementation details of the code. The main focus is on the numerical aspects of the code used to overcome problems with singularity and ill-conditioning, and strategies to recover from finding non-optimal stationary points of the merit function. Issues related to the use of iterative solvers for large scale problems are then outlined. We demonstrate the code’s robustness on the problems in the GAMSLIB and MCPLIB [26] collections by performing two tests, one using direct solution methods and another using only iterative techniques. Both indicate that the SEMI code is reliable and scalable.

7.1 Mathematical Foundation

The semismooth solver is based on a reformulation of the mixed complementarity problem as a nonlinear system of equations. We first recall that a mapping $\phi : \mathbb{R}^2 \to \mathbb{R}$ is called an *NCP-function* if it satisfies

$$\phi(a, b) = 0 \Longleftrightarrow 0 \le a \;\perp\; b \ge 0.$$

Two examples of NCP-functions are the Fischer-Burmeister [50] function

$$\phi_{FB}(a, b) := \sqrt{a^2 + b^2} - a - b \tag{64}$$

and the penalized Fischer-Burmeister [13] function

$$\phi_{CCK}(a, b) := \lambda \left(\sqrt{a^2 + b^2} - a - b\right) - (1 - \lambda) \max\{0, a\} \max\{0, b\}, \tag{65}$$

where $\lambda \in (0, 1)$ is a given parameter. To reformulate the mixed complementarity problem using these functions we introduced a partitioning of the index set $I = \{1, \dots, n\}$:

$$\begin{aligned}
I_l &:= \{i \in I \,|\, -\infty < l_i < u_i = +\infty\}, \\
I_u &:= \{i \in I \,|\, -\infty = l_i < u_i < +\infty\}, \\
I_{lu} &:= \{i \in I \,|\, -\infty < l_i < u_i < +\infty\}, \\
I_f &:= \{i \in I \,|\, -\infty = l_i < u_i = +\infty\},
\end{aligned}$$

where I_l, I_u, I_{lu} and I_f denote the set of indices $i \in I$ where there are finite lower bounds only, finite upper bounds only, finite lower and upper bounds and no finite bounds on the variable x_i, respectively. Hence, the subscripts in the above index sets indicate which bounds are finite, with the only exception of I_f which contains the free variables.

Let ϕ_1 and ϕ_2 belong to $\{\phi_{FB}, \phi_{CCK}\}$. Then we can extend the idea in [5] and define an operator $\Phi : \mathbf{R}^n \to \mathbf{R}^n$ componentwise as follows:

$$\Phi_i(x) := \begin{cases} \phi_1(x_i - l_i, F_i(x)) & \text{if } i \in I_l, \\ -\phi_1(u_i - x_i, -F_i(x)) & \text{if } i \in I_u, \\ \phi_2(x_i - l_i, \phi_1(u_i - x_i, -F_i(x))) & \text{if } i \in I_{lu}, \\ -F_i(x) & \text{if } i \in I_f. \end{cases}$$

We then have the following characterization of solutions to the mixed complementarity problem:

Proposition 7.1.1 *Let $x^* \in \mathbf{R}^n$ be given. Then the following are equivalent:*

(a) x^* is a solution of the mixed complementarity problem.

(b) $\Phi(x^*) = 0$.

We remark that Chapter 6 used the same reformulation of the complementarity problem for globalization purposes, but with $\phi_1 = \phi_2 = \phi_{FB}$. The above definition of Φ is more general in that we can choose any combination of ϕ_{FB} and ϕ_{CCK} for ϕ_1 and ϕ_2.

Note that Φ is not differentiable in general. However, if F is continuously differentiable then Φ is semismooth. A standard technique to solve the mixed complementarity problem is to apply a nonsmooth Newton method (see [102, 101]) to the system $\Phi(x) = 0$ and globalize it using the corresponding merit function

$$\Psi(x) := \frac{1}{2}\Phi(x)^T\Phi(x) = \frac{1}{2}\|\Phi(x)\|^2.$$

Assuming that Ψ is continuously differentiable and recalling that the *B-subdifferential* of Φ at a point $x \in \mathbf{R}^n$ is defined by [101]

$$\partial_B\Phi(x) := \{H \in \mathbf{R}^{n\times n} \,|\, \exists\{x^k\} \subseteq D_\Phi : x^k \to x \text{ and } \Phi'(x^k) \to H\},$$

where D_Φ denotes the set of differentiable points of Φ, we can follow the pattern from [23] and write down the basic semismooth solver for MCP.

Algorithm BSM (Basic Semismooth Method)

(S.0) (Initialization)

Choose $x^0 \in \mathbf{R}^n, \rho > 0, \beta \in (0,1), \sigma \in (0, 1/2), p > 2,$ *and set* $k := 0.$

(S.1) (Stopping Criterion)

If x^k *satisfies a suitable termination criterion: STOP.*

(S.2) (Search Direction Calculation)

Select an element $H_k \in \partial_B \Phi(x^k)$*. Find a solution* $d^k \in \mathbf{R}^n$ *of the linear system*

$$H_k d = -\Phi(x^k). \tag{66}$$

If this system is not solvable or if the descent condition

$$\nabla \Psi(x^k)^T d^k \le -\rho \|d^k\|^p \tag{67}$$

is not satisfied, set $d^k := -\nabla \Psi(x^k)$*.*

(S.3) (Line Search)

Compute $t_k := \max\{\beta^i \mid i = 0, 1, 2, \ldots\}$ *such that*

$$\Psi(x^k + t_k d^k) \le \Psi(x^k) + t_k \nabla \Psi(x^k)^T d^k.$$

(S.4) (Update)

Set $x^{k+1} := x^k + t_k d^k, k \leftarrow k+1,$ *and go to (S.1).*

Note that Algorithm BSM actually represents a whole class of methods since it depends on the definition of Φ which, in turn, is completely determined by the choice of ϕ_1 and

ϕ_2. Note that, usually, ϕ_1 plays the central role in the definition of Φ; for example, if there is no variable with both finite lower and upper bounds, then ϕ_2 is not used. In particular, this is the case for the standard nonlinear complementarity problem.

For the purpose of this chapter, we are particularly interested in the following two choices of Φ. We define

$$\Phi_{FB} := \Phi \quad \text{if} \quad \phi_1 = \phi_2 = \phi_{FB},$$

and

$$\Phi_{CCK} := \Phi \quad \text{if} \quad \phi_1 = \phi_{CCK},\, \phi_2 = \phi_{FB}.$$

The reader may wonder why we do not take $\phi_2 = \phi_{CCK}$ as well; the simple reason is that we were unable to prove some of the subsequent results for this case. In fact, basically all of these results are based on a suitable overestimation for the generalized Jacobian $\partial\Phi(x)$. Typically, such an overestimation can be obtained by exploiting Theorem 2.3.9 in [17] that contains a convex hull operation. It is often possible to remove this convex hull and to get a simpler overestimate for $\partial\Phi(x)$. However, when using $\phi_1 = \phi_2 = \phi_{CCK}$, we were not able to remove the convex hull, so we decided not to take $\phi_2 = \phi_{CCK}$ in the definition of the operator Φ_{CCK}.

Both from a theoretical and a numerical point of view [13], the operator Φ_{CCK} has stronger properties than Φ_{FB}, at least for the standard nonlinear complementarity problem. Hence, Φ_{CCK} will be used by default in the implementation of Algorithm BSM. However, in some situations, it is also helpful to have alternative operators like Φ_{FB}. For example, the implementation uses Φ_{FB} in one of the restarts.

We now summarize some of the properties of Φ_{FB} and Φ_{CCK} as well as of their corresponding merit functions

$$\Psi_{FB}(x) := \frac{1}{2}\Phi_{FB}(x)^T\Phi_{FB}(x) \quad \text{and} \quad \Psi_{CCK}(x) := \frac{1}{2}\Phi_{CCK}(x)^T\Phi_{CCK}(x).$$

The proofs of the results can be found in [5] and Section 6.1 for the case of $\Phi = \Phi_{FB}$. Since the proofs for $\Phi = \Phi_{CCK}$ are very similar (although quite technical and lengthy), we skip the proofs of all these results here.

Proposition 7.1.2 *Let Φ belong to $\{\Phi_{FB}, \Phi_{CCK}\}$ and Ψ be the corresponding merit function. Then the following hold:*

(a) Φ is semismooth.

(b) If F' is locally Lipschitzian, then Φ is strongly semismooth.

(c) Ψ is continuously differentiable on $\mathbf{R}^n$.

(d) If x^* is a strongly regular solution of MCP, then x^* is a BD-regular solution of $\Phi(x) = 0$.

We only note that (strong) semismoothness is one of the two ingredients which are needed to prove local superlinear (quadratic) convergence of a nonsmooth Newton method. The second ingredient is the BD-regularity assumption. See Chapter 1 for precise definitions of these terms.

We can now state the following convergence properties of Algorithm BSM. The proof is analogous to those given in [23] for Φ_{FB} and the standard nonlinear complementarity problem.

Theorem 7.1.3 *Let $\Phi \in \{\Phi_{FB}, \Phi_{CCK}\}$ and $\{x^k\}$ be a sequence generated by Algorithm BSM. Then any accumulation point of this sequence is a stationary point of Ψ. Moreover, if one of these accumulation points, say x^*, is a BD-regular solution of $\Phi(x) = 0$, then the following statements hold:*

(a) The entire sequence $\{x^k\}$ converges to x^*.

(b) *The search direction* d^k *is eventually given by the Newton equation (66).*

(c) *The full stepsize* $t_k = 1$ *is eventually accepted in Step (S.3).*

(d) *The rate of convergence is Q-superlinear.*

(e) *If* F' *is locally Lipschitzian, then the rate of convergence is Q-quadratic.*

7.2 The Linear System

The heart of the semismooth algorithm lies in the linear algebra. Most of the time is spent solving the Newton system, $H_k d = -\Phi(x^k)$, where in general H_k is neither symmetric nor positive definite. Effective mechanisms for solving this system using either iterative techniques or a direct method are indispensable and have great impact upon the success of the algorithm. The key advantage of the semismooth algorithm over PATH 4.x is that the former only solves a single linear system per iteration while the latter uses a pivotal based code to solve a linear complementarity problem. The pivotal based code relies upon the availability of a direct factorization and rank-1 updates which limits its applicability to medium sized or large, structured problems. Semismooth has no such restriction.

We begin the analysis by investigating iterative techniques for finding the Newton direction as these will enable the algorithm to solve very large problems. We present three of the methods considered and discuss time and space requirements and scaling issues. The methods were evaluated by applying them to two reasonably large models representative of the test suite. From the results, we conclude that LSQR [97] is the most reliable. Practical termination criteria are also mentioned.

We then discuss the issues involved and options available when using direct methods. The main difficulty encountered is singularity in the Newton system. Information on detecting singularity and using that knowledge to construct a useful direction even in this case is presented. The effects of the techniques considered on the singular models in the test set are given and the final choice of strategies for solving the linear system is provided.

Wherever possible, we want to use the best available technique to determine the Newton direction. While LSQR is very reliable, typically the best technique in terms of time is to use a direct method to factor H_k. However, when the size of the factors grows too large, we want to resort to the iterative technique. Therefore, the rules implemented are designed so that large, structured problems with sparse decompositions will use the factorization software, while problems that are either very dense or have large factors will not. Currently, a restriction on the size of the decomposition of 12 million nonzeros is imposed. We no longer consider using a direct method if this condition is violated.

7.2.1 Iterative Techniques

Three iterative techniques for finding the Newton direction were investigated: LSQR, GMRES, and QMR. We recall that the systems of equations solved will generally be neither symmetric nor positive definite. Therefore, we cannot directly use the popular techniques from optimization algorithms such as conjugate gradients. The algorithms tested are a representative set of those meeting the requirements.

LSQR [97] is based upon the bidiagonalization method developed in [58] which implicitly solves the least squares problem, $\min \left\| H_k d + \Phi(x^k) \right\|^2$. The method is essentially a reliable variant of conjugate gradients applied to the normal equation, $H_k^T H_k d =$

$-H_k^T\Phi(x^k)$. The code only requires a workspace of 3 n-vectors in addition to the storage of H_k, d, and $\Phi(x^k)$. The cost per iteration consists of $4nnz + 12n$ floating point operations, where nnz is the number of nonzero elements in H_k.

The GMRES [114, 119] method uses the Arnoldi procedure to construct an orthonormal basis for the Krylov subspace $\mathcal{K}_m\left(H_k, -\frac{\Phi(x^k)}{\|\Phi(x^k)\|}\right)$, where

$$\mathcal{K}_m(A, r) := \text{span}\left\{r, Ar, \ldots, A^{m-1}r\right\}$$

for some matrix $A \in \mathbf{R}^{n\times n}$ and a vector $r \in \mathbf{R}^n$. We then use this basis to find a vector in the generated subspace minimizing the residual, $\left\|H_k d + \Phi(x^k)\right\|^2$. The implementation uses Householder reflections for the orthogonalization process to preserve stability, maintains the current optimal value of the residual at each iteration using plane rotations, and restarts after m iterations. We remark that because the matrices are not guaranteed to be positive definite, restarted GMRES can stagnate and make no progress. The method has a workspace requirement of $(m+2)$ n- and 4 m-vectors and uses $2nnz+4n(1+2i)-4i^2$ operations per iteration where $i \in [1, m]$ is the iteration number. Furthermore, we use around $4mn$ operations at the end of each m iterations to generate the minimizing vector. The main difficulty with GMRES lies in choosing the restart frequency. If it is too small, we can fail to converge entirely, and if it is too large, the per iteration cost and storage requirements become significant.

The QMR [114] algorithm uses the Lanczos biorthogonalization algorithm to construct bases for the Krylov subspaces $\mathcal{K}_m\left(H_k, -\frac{\Phi(x^k)}{\|\Phi(x^k)\|}\right)$ and $\mathcal{K}_m\left(H_k^T, -\frac{\Phi(x^k)}{\|\Phi(x^k)\|}\right)$ satisfying a biorthogonality condition. The bases generated are then used to find a vector with approximate minimum residual in $\mathcal{K}_m\left(H_k, -\frac{\Phi(x^k)}{\|\Phi(x^k)\|}\right)$. The QMRPACK [54] code tested uses the coupled two-term recurrence variant of the look-ahead Lanczos algorithm. We allowed m look-ahead steps to be tried, which results in a workspace of $10m + 1$ n- and

$8m+18$ m-vectors. Typically, m is chosen to be small. The code uses at least $4nnz+14n$ floating point operations per iteration and requires a backsolve at the end to determine the vector with approximate minimal residual.

Scaling the linear system is crucial to the success of the iterative algorithms. We define a matrix scaling using the following diagonal matrices $R, C \in \mathbb{R}^{n\times n}$ with diagonal entries:

$$\begin{aligned} R_{i,i} &= \frac{1}{\max\left\{\sqrt{\Phi_i(x^k)^2+\sum_j (H_k)_{i,j}^2}, 10^{-10}\right\}}, \\ C_{j,j} &= \frac{1}{\max\left\{\sqrt{\sum_i (RH_k)_{i,j}^2}, 10^{-10}\right\}}. \end{aligned} \tag{68}$$

We solve the linear system $RH_kCC^{-1}d = -R\Phi(x^k)$ by defining $\tilde{d} := C^{-1}d$, solving the system $RH_kC\tilde{d} = -R\Phi(x^k)$, and recovering the Newton direction as $d = C\tilde{d}$. This procedure scales the rows and then the columns so that each has a two norm of 1. The constants in the max operator are used to avoid division by zero errors. When a row or column has a two norm close to zero, the scaling has no effect. Scaling significantly reduces the number of iterations required in all cases and thus the total time spent in the iterative solver.

We remark that the use of preconditioned versions of LSQR, GMRES, and QMR could further reduce the number of iterations taken and time spent to solve the system of equations. This is a topic of future research.

7.2.1.1 Evaluation

We selected two reasonably large models with known solutions from the test suite on which to evaluate the iterative techniques. The `uruguay` model has 2,281 rows and columns and 90,206 nonzeros and is interesting because a direct factorization incurs a large amount of fill-in. `opt_cont255` is a structured problem with 8,192 rows and columns and 147,200 nonzeros. For each complementarity model, we only solved the first linear system arising from (66). The termination criteria for these tests was based upon the relative residual, $\frac{\left\|H_0 d_i + \Phi(x^0)\right\|}{\|\Phi(x^0)\|}$. The iterative methods terminated when the relative residual is less than 10^{-8}. In all cases, we chose an initial guess of $d_0 := 0$. We did not investigate other choices.

When using the GMRES method, we need to choose the restart frequency. We varied the value of m by choosing values between 10 and 200. The results for each value of m tested are given in the accompanying tables. For QMR, we need to determine the number of look-ahead steps allowed. We placed an upper limit of 20 on these steps, although smaller values would suffice.

All of the trials were run on the same machine using the same executable so that we can make a valid comparison. Tables 46 and 47 report the results on the two test problems. The total iterations and time, exit status, and relative residual at the final point are given. Based upon the available information, we conclude that LSQR is the best choice for this purpose. This method is quite effective for the models we have generated. However, it may require a large number of iterations in order to converge. In these situations we are willing to sacrifice speed in exchange for reliability. Note that results shown later demonstrate the robustness of LSQR on the entire test suite. The robustness and effectiveness of LSQR is also in accordance with the results of [24].

7.2.1.2 Termination Rules

While the termination rule given above is reasonable for evaluation, we now return to the subject of practical termination rules. The relative residual calculated above is not applicable as a termination criterion unless we know a priori that the linear model has

Table 46: Iterative Method Results: `uruguay`

Method	Iterations	Time	Status	Relative Error
LSQR	908	43	Solved	1.0e-8
GMRES 10	1,766	47	Solved	1.0e-8
GMRES 20	1,346	46	Solved	1.0e-8
GMRES 50	799	45	Solved	9.9e-9
GMRES 100	481	45	Solved	9.7e-9
GMRES 200	419	54	Solved	9.9e-9
QMR	466	41	Solved	6.5e-9

Table 47: Iterative Method Results: `opt_cont255`

Method	Iterations	Time	Status	Relative Error
LSQR	2,848	90	Solved	9.9e-9
GMRES 10	100,000	2,114	Iteration Limit	8.7e-4
GMRES 20	100,000	3,194	Iteration Limit	2.6e-4
GMRES 50	100,000	6,217	Iteration Limit	6.0e-5
GMRES 100	100,000	11,498	Iteration Limit	1.1e-5
GMRES 200	100,000	22,136	Iteration Limit	5.4e-8
QMR	100,000	5,288	Iteration Limit	1.6e-7

a solution. This is an unreasonable assumption to make. Therefore, the implementation of LSQR uses the termination rules developed in [97]. They are to terminate if any of the following holds:

1. $\mathrm{cond}(H_k) \geq \mathrm{CONLIM}$
2. $\|r_i\| \leq \mathrm{BTOL}\left\|\Phi(x^k)\right\| + \mathrm{ATOL}\|H_k\|\,\|d_i\|$
3. $\frac{\left\|H_k^T r_i\right\|}{\|H_k\|\|r_i\|} \leq \mathrm{ATOL}$

where $r_i = -(H_k d_i + \Phi(x^k))$. Justification of these rules and a demonstration of their effectiveness is given in [97]. We note that LSQR builds up estimates of $\|H_k\|$ and $\mathrm{cond}(H_k)$ by performing a small amount of additional computation per iteration of the code. The exact tolerances used are $ATOL = \epsilon^{\frac{2}{3}}$, $BTOL = \epsilon^{\frac{2}{3}}$, and $CONLIM = \frac{1}{10\sqrt{\epsilon}}$ where ϵ is the machine precision. Furthermore, an iteration limit of $\min\{100000, 20n\}$ was used. These tolerances force the iterative method to find a point close to the exact solution of the linear system if it exists. We did not investigate using less stringent termination criteria.

7.2.2 Direct Methods

We now look at the issues involved in using direct methods to solve the Newton system. The factorization software needs to have routines to factor and solve, and should be able to uncover singularity problems and make a good approximation of the linearly dependent rows and columns in such cases. For reasonably sized problems we use the LUSOL [56] sparse factorization routines contained in the MINOS [93] nonlinear programming solver to factor H_k and solve for the Newton direction. The authors of this package have investigated the effects of modifying tolerances in the factorization on general linear systems and have suggested defaults which we have adopted for all the results.

The major difficulty with the direction finding problem is dealing with those instances where the Newton system does not have a solution. These singularity problems frequently occur in real world applications. However, the theoretical algorithm only provides a crude mechanism in this case, i.e. the use of a gradient step, while other approaches may be more effective. Clearly, any practical implementation of the semismooth algorithm must include appropriate procedures to deal with singularity.

Following the success of scaling for the iterative techniques, we first investigate the applicability of scaling in conjunction with direct methods to avoid ill-conditioned systems. We then look at techniques to determine a useful direction when the model is singular, including using gradient steps, diagonal perturbations of H_k, and finding least squares solutions to the linear system. Empirical evidence is provided upon which we evaluate the methods.

7.2.2.1 Scaling

LUSOL contains routines to detect when a matrix is singular or nearly singular. We study in this subsection the effects of scaling the linear problems in an effort to improve the conditioning of the matrices that we request to factor. We want to determine if we can significantly reduce the number of occurrences where the factorization package determines that the matrix is singular. By using scaling we hope to improve the overall reliability of the code on ill-conditioned problems.

Two different scaling schemes were tested on the problems in the test set along with the default of no scaling. The first technique is the diagonal scaling used in the PATH

Table 48: Scaling Effects on Direct Methods

Scaling	Detected Singular Matrices	Failures	Time
None	2138	28	13,922
Diagonal	2248	30	12,780
Matrix	1331	30	12,901

4.x solver. In this case, we define a row scaling by looking at elements of the diagonal of H_k which are large and scale the entire row of the problem. Formally, we define a diagonal matrix R such that if $|(H_k)_{i,i}| > 100$ then $R_{i,i} = \frac{10}{|(H_k)_{i,i}|}$ and $R_{i,i} = 1$ otherwise. We then try to factorize the scaled matrix RH_k.

The other scaling method is the matrix scaling as used in the iterative techniques. We use the diagonal matrices defined in (68) and attempt to factorize RH_kC.

There are costs associated with scaling. Of the two methods, matrix scaling is more expensive per iteration because it requires looking at the data twice. We tested all of these scalings on the models in the entire test set and report in Table 48 the number of detected singular solves, failures of the algorithm to find a solution, and total time in seconds over the entire test set. When a singular model was detected we use the least squares recovery method detailed in Section 7.2.2.2. Since diagonal scaling does not improve significantly over no scaling, we disregard this method. The reason for this poor behavior is probably due to the fact that the diagonal elements do not necessarily reflect the actual scaling of the problem. Matrix scaling significantly reduces the number of singular systems detected. However, it does result in additional failures of the algorithm. Since we were unable to definitively choose between no scaling and matrix scaling, in the next section we look at recovery techniques and report results for both.

Before continuing, we note that the scaling investigated here is not very exhaustive and more complex schemes might be tested. Furthermore, the scaling is being performed on the linear model, when it might be more appropriate to look at the nonlinear model to determine the scaling. Finally, we did not investigate modifying other parameters, such as those encountered in the nonlinear model in Section 7.3, in conjunction with scaling, which might lead to improved reliability and performance.

7.2.2.2 Singularity

Having looked at scaling we need to establish procedures to recover from the singularity problem and generate a reasonable direction. We have investigated three techniques. The first is the theoretical standby of using only gradient steps when the Newton system is unsolvable. A second technique is to use a diagonal perturbation of H_k to regularize the problem. The final method is to use LSQR to calculate a least squares solution of the system. All of the results in this section are only given for those models where singularity was detected by the linear solver. Only a few options were changed for each run, with the rest being held constant.

Gradient Steps The naive recovery technique, and the simplest of those considered, is to simply resort to a gradient step whenever a singular model is detected. This approach is theoretically justified, but in practice frequently leads to a stationary point that does not solve the complementarity problem. However, we use this approach as the baseline against which we evaluate the rest of the methods. The results for the 194 singular models are given in Table 49 where we report the number of times the algorithm failed and total time for both scaled and unscaled models. In this case, we note that matrix scaling performs better than no scaling.

Table 49: Gradient Results on Singular Models

Scaling	Failures	Time
none	28	4,836
matrix	24	3,630

Perturbation Perturbation involves replacing the linear model with one which does not have a singularity problem. We investigated using a diagonal perturbation where we replace H_k with $H_k + \lambda I$ for some $\lambda > 0$. The value for λ was chosen in the interval $[\alpha, \beta]$, with $\lambda = \gamma\Psi$ whenever possible. We used values of $\alpha = 10^{-8}$, $\beta = 1$ and $\gamma = \frac{1}{10}$. When the perturbation is insufficient to overcome the singularity, we increase λ to $\delta\lambda$ for some $\delta > 1$. We currently use $\delta = 10$, and allow the perturbation to increase only one time per iteration.

The other choice to make is when to add the perturbation. There are two options investigated:

- If the first model encountered is singular, calculate a λ and monotonically decrease it from one iteration to the next. The new value is $\min\{\kappa_1\lambda, \kappa_2\Psi\}$ where $\kappa_1 = 0.4$ and $\kappa_2 = 0.1$ by default. This strategy is used in the PATH 4.x code.

- Every time a singular model is encountered, calculate a value for λ.

When scaling was used, we first perturbed the problem and then scaled it.

To test these strategies, we ran the set of singular models using each of the options. We report the number of failures on the 194 singular models and total time in Table 50. When the perturbation fails to find a nonsingular matrix, the least squares method (to be described in the next section) was used to calculate a direction. The use of perturbation leads to fewer total failures than only using the gradient method. The effects of scaling

Table 50: Perturbation Effects on Singular Models

Scaling	Strategy	Failures	Time
none	first	21	8,910
	demand	21	3,285
matrix	first	20	8,210
	demand	23	2,703

Table 51: LSQR Results on Singular Models

Scaling	Failures	Time
none	9	8,513
matrix	11	7,534

the problem are mixed, with scaling leading to a decrease in total time, but sometimes resulting in additional total failures.

Least Squares Method Finally, we investigate the use of the LSQR iterative scheme to find a solution to the least squares problem min $\left\|H_k d + \Phi(x^k)\right\|^2$ and use the resulting d as the Newton direction. The practical termination rules mentioned in 7.2.1.2 were used.

We investigated using scaling and no scaling in the linear model that we try to factor. We present the results on the 194 singular models in Table 51 where we report the total number of failures in the algorithm and time. The major downside to using the iterative technique to solve the least squares problem is that it is fairly slow because we allow many iterations and have low tolerances. We did not study the effect of changing the termination criteria. However, the results indicate that this method is better than the others tested in terms of reliability.

7.2.3 Summary

The empirical results given above provide clear choices. For both large scale work and calculating a direction when the Newton system is singular we will use the LSQR iterative technique. We remark that while this is the most reliable choice, it is perhaps not the most efficient method. While we always scale the linear system when an iterative solver is used, the effects of scaling the matrix we try to factor are indeterminate and we made the decision to use no scaling to achieve simplicity in the code. The effect of choices made in the nonlinear model which are discussed in the next section have a great impact upon the success of the algorithm. However, we did not investigate modifying those strategies in conjunction with the strategies in the linear solver.

7.3 The Nonlinear Model

At the nonlinear level of the algorithm, we are concerned with properties of the algorithm affecting convergence. These include numerical issues related to the merit function and calculation of H_k as well as crashing and the recourse taken when a stationary point of the merit function is encountered. These issues are discussed in the following subsections. We then summarize the results and present the final strategies chosen.

A difficulty with the semismooth code occurs when F is ill-defined because no guarantee is made that the iterates will remain feasible with respect to the box $[\ell, u]$. Such problems arise when using log functions or real powers which frequently occur in applications. Backtracking away from places where the function is undefined and restarting is typically sufficient for these models.

7.3.1 $\Phi(x^k)$ and H_k

As mentioned in Section 7.1, the implementation of the semismooth algorithm for mixed complementarity problems will use the penalized Fischer-Burmeister merit function. The value of λ chosen in (65) can have a significant impact upon the performance of the semismooth algorithm. We note that small values of λ, say less than 0.5, should not be used. In the case of nonlinear complementarity problems, small values for λ emphasize the $\max\{0, F_i(x^k)\} \max\{0, x_i^k\}$ term of the penalized Fischer-Burmeister function. This term is related to the complementarity error, but does not enforce $F_i(x^k) \geq 0$ and $x_i^k \geq 0$. Since we use inexact arithmetic and terminate when the merit function is small, i.e. less than 10^{-12}, a small value of λ could result in finding a point satisfying the termination tolerance which is not necessarily close to a solution of the MCP. The default choice in the implementation is to have $\lambda = 0.8$ which can be changed using the `chen_lambda` option.

Furthermore, despite the fact that the penalized Fischer-Burmeister function is typically superior, there are some situations where the original function might be more appropriate. Therefore, when using restarts (see Section 7.3.3.2) we also might change the merit function. This is done by modifying the `merit_function` option to `fischer` or `chen` for the standard and penalized Fischer-Burmeister functions.

We also note that when $\phi_1 \neq \phi_2$, there are two different representations for Φ. This situation only occurs when both variables are bounded (which is the only case when ϕ_2 is used). In this case, we use

$$\phi_2(x_i - l_i, \phi_1(u_i - x_i, -F_i(x))).$$

An alternative is to use

$$-\phi_2(u_i - x_i, \phi_1(x_i - l_i, F_i(x))).$$

Both of these functions are equally valid and can lead to different sequences of iterates being generated by the algorithm. We did not investigate this option further.

The calculation of $\phi(a, b)$ needs to be performed in such a way as to minimize the effect of roundoff error. If we have $a = 10^{-4}$ and $b = 10^4$ and are using a machine with 6 decimal places of accuracy, a naive calculation of $\phi(a, b) = \sqrt{a^2 + b^2} - a - b$ would produce zero, meaning that we are at a solution to the problem when in fact we are not, as the following calculation indicates:

$$
\begin{aligned}
\sqrt{10^{-8} + 10^8} - 10^{-4} - 10^4 \\
= \sqrt{10^8} - 10^{-4} - 10^4 \\
= 10^4 - 10^{-4} - 10^4 \\
= 10^4 - 10^4 \\
= 0.
\end{aligned}
$$

The actual value of $\phi(a, b)$ should be on the order of -10^{-4}. A better way to calculate $\phi(a, b)$ is as follows:

1. If $|a| > |b|$ then $\phi(a, b) = (\sqrt{a^2 + b^2} - a) - b$.

2. Otherwise $\phi(a, b) = (\sqrt{a^2 + b^2} - b) - a$.

This method gives a more accurate value of ϕ. The square root operation needed is computed by defining $s = |a| + |b|$. If $s = 0$ then the value is zero, otherwise $s\sqrt{(\frac{a}{s})^2 + (\frac{b}{s})^2}$ is the value computed. This eliminates overflow problems.

In order to calculate H_k, we use the procedure developed in [5, 23] for the Fischer-Burmeister function, and a modification of the method in [13] extended for MCP models for the penalized Fischer-Burmeister function.

7.3.2 Crashing

Projected gradient crashing before starting the main algorithm can improve the performance of the algorithm by generating a more reasonable starting point. To do this, we use a technique already tested in [24] and add a new step, S.0a, to the algorithm between S.0 and S.1. Let $\pi_{[\ell,u]}$ be the projection of onto the box $[\ell, u]$. We then perform the following in this new step.

Algorithm PGC (Projected Gradient Crash)

(S.1) Let $j = 0$.

(S.2) Calculate $d^j = -\nabla\Psi(x^j)$.

(S.3) Let $t_j := \max\{\beta^i \mid i = 0, 1, 2, \ldots\}$ *such that*

$$\Psi(\pi_{[\ell,u]}(x^j + t_j d^j)) \leq \Psi(x^j) - \tau\nabla\Psi(x^j)^T(x^j - \pi_{[\ell,u]}(x^j + t_j d^j)).$$

(S.4) If $t_j < \tau$ *stop and set* $x^0 = x^j$.

(S.5) Otherwise let $x^{j+1} = \pi_{[\ell,u]}(x^j + t_j d^j)$ *and* $j = j + 1$. *Go to (S.2).*

In the code $\tau = 10^{-5}$ and $\beta = 0.5$; furthermore, we only allow 10 iterations of the projected gradient crash method.

The crashing technique presented has iterates that remain feasible and improve upon the initial point with respect to the merit function. We believe that this is the key benefit from crashing – all iterates remain in $[\ell, u]$. Otherwise poor values of x^0 can frequently lead to failures in the semismooth algorithm. In particular, the crashing technique can significantly affect the iterates generated during a restart. Crashing can be turned off with the option `crash_method none`.

7.3.3 Stationary Points

While stationary point termination is typically adequate for nonlinear optimization, determining a stationary point of the residual function that is not a zero is considered a "failure" by complementarity modelers. Much theoretical work has been carried out determining the weakest possible assumptions that can be made on the problem (and/or the algorithm) in order to guarantee that a stationary point of the merit function is in fact a solution of the complementarity problem. Some of these results restrict the problem class considered by employing assumptions that cannot be easily verified for arbitrary models. Other techniques (such as nonmonotone linesearching) rely on a combination of heuristics and theory, while others are entirely heuristic in nature. The basic strategies we used to improve the reliability of the semismooth solver include nonmonotone linesearching and restarting. The positive effects of these strategies have been demonstrated in the literature and we just present the basic idea and any modifications made.

7.3.3.1 Nonmonotone Linesearch

The first line of defense against convergence to stationary points is the use of a nonmonotone linesearch [63, 64, 38]. In this case we define a reference value, R^k and we use this value to replace the test in step S.3 of Algorithm BSM with the nonmonotone test:

$$\Psi(x^k + t_k d^k) \leq R^k + t_k \nabla\Psi(x^k)^T d^k.$$

Depending upon the choice of the reference value, this allows the merit function to increase from one iteration to the next. This strategy can not only improve convergence, but can also avoid local minimizers by allowing such increases.

We now need to detail the reference value choice. We begin by letting $\{M_1, \ldots, M_m\}$ be a finite set of values initialized to $\kappa\Psi(x^0)$, where κ is used to determine the initial set of acceptable merit function values. The value of κ defaults to 1 in the code and can be modified with the `nms_initial_reference_factor` option; $\kappa = 1$ indicates that we are not going to allow the merit function to increase beyond its initial value.

Having defined the values of $\{M_1, \ldots, M_m\}$ (where the code by default uses $m = 4$), we can now calculate a reference value. We must be careful when we allow gradient steps in the code. Assuming that d^k is the Newton direction (or a least squares solution to the Newton system in the presence of singularity, see 7.2.2.2), we define $i_0 = \text{argmax } M_i$ and $R^k = M_{i_0}$. After the nonmonotone linesearch rule above finds t_k, we update the memory so that $M_{i_0} = \Psi(x^k + t_k d^k)$. That is, we remove an element from the memory having the largest merit function value.

When we decide to use a gradient step, it is beneficial to let $x^k = x^{\text{best}}$ where x^{best} is the point with the absolute best merit function value encountered so far. We then recalculate $d^k = -\nabla\Psi(x^k)$ using the best point and let $R^k = \Psi(x^k)$. That is to say that we force decrease from the best iterate found whenever a gradient step is performed. After a successful step we set $M_i = \Psi(x^k + t_k d^k)$ for all $i \in [1, \ldots, m]$. This prevents future iterates from returning to the same problem area.

A watchdog strategy [12] is also available for use in the code. The method employed allows steps to be accepted when they are "close" to the current iterate. Nonmonotonic decrease is enforced every m iterations, where m is set by the `nms_mstep_frequency` option. Currently, we use this strategy only used in a restart.

7.3.3.2 Restarting

The rules for nonmonotone linesearching and crashing are extremely useful in practice, but do not preclude convergence to a nonoptimal stationary point. One observation

Table 52: Restart Definitions

Restart Number	Parameter Values
1	crash_method none
2	chen_lambda 0.95 crash_method none nms_mstep_frequency 4 nms_initial_reference_factor 5
3	crash_method none merit_function fischer nms_mstep_frequency 1 nms_initial_reference_factor 1

relevant for complementarity solvers is that we know a priori the optimal value of the merit function at a solution if one exists. If the code detects that the current iterate is a stationary point that is not a solution, a recovery strategy can be invoked. One successful technique is the restart strategy introduced in PATH where the recovery mechanism involves starting over from the user supplied starting point with a different set of options, thus leading to a different sequence of iterates being investigated. For the semismooth algorithm we use the restarts defined in Table 52. In addition, the first restart of the semismooth code will also include the option `chen_lambda 0.95` if no crash iterations were performed in its first run; otherwise, we would waste computational resources by generating the exact same sequences of iterates as the first attempt.

Restarts should be applicable to general models and not written to overcome difficulties with a specific model. That is, if we were to use the restart definition as the default, we should still solve most problems in the test set. We present in Table 53 the numbers of failures on a subset of GAMSLIB and MCPLIB models when the particular restart options were used. Note that the restarts (0, 1 and 2) using the penalized Fischer-Burmeister function given in equation (65) outperform the one using the standard

Table 53: Restart Performance on GAMSLIB and MCPLIB Problems

	Failures	
Restart Number	GAMSLIB	MCPLIB
0 (first run)	3	63
1	3	63
2	3	64
3	12	76

Fischer-Burmeister function defined in equation (64).

7.4 Computational Results

The LUSOL [56] sparse factorization routines contained in the MINOS [93] nonlinear programming solver were used for factorization purposes. All of the linear algebra and other basic mechanisms are exactly the same for both SEMI and PATH 4.x. Therefore, the comparison made is as close to a true comparison of the algorithms as we can make. We note that PATH 4.x is much more mature than the semismooth implementation. Both codes are continually being improved when deficiencies are uncovered.

Finally, we remark that the factorization routine is a key component of the implementation. For the semismooth algorithm, rank-1 updates to the matrix are not required. Therefore, factorization routines other than those provided by LUSOL might be more effective for SEMI. However, we sacrifice some potential speed gains in preference for having a consistent choice for the direct method among the codes tested.

To test the standard algorithm, we ran the code on all of the problems in the GAMSLIB and MCPLIB [26] suites of test problems. The following tables report the number of successes, followed by the number of failures in parenthesis. In order to test the reliability of the iterative method, we ran all of the models using only the iterative LSQR technique to calculate the Newton direction. For comparison purposes, we also show the performance of PATH 4.x from Chapter 6 on the same problems. These latter two results are given in the columns labeled "Iterative" and "PATH 4.x" respectively. We split the results into those that allow restarts and those that do not.

These results indicate that while SEMI is not as reliable as PATH 4.x, it does well and is robust on the GAMSLIB and MCPLIB problem sets. We remark that the `box` model accounts for over 80 failures in the MCPLIB test set. This particular test problem modifies a parameter within a loop and solves each successive problem using the solution from the previous iteration as a starting point. Therefore, a failure to solve one particular problem leads to extreme difficulty solving the next. Hence, we have a large number of accumulated failures for this problem. We could argue for the removal of the `box` model from the test set. However, for completeness we report results for the `box` model.

The results also indicate that the iterative method is robust. Since we are only using iterative techniques in this code, the memory requirements are much less. 42 of the additional failures in the "Iterative" column occur in the `asean9a`, `denmark`, and `opt_cont511` models in which the time limit was encountered. We also note that the restart heuristic significantly improves the robustness of both SEMI and PATH 4.x on MCPLIB.

We currently do not have any results on very large problems, but believe that based on the evidence, the code will scale well to the larger problems. In particular, the iterative version of the SEMI requires significantly less memory than PATH 4.x, allowing the possibility of solving huge models. The current drawback of the semismooth code is the time taken by LSQR to solve the linear systems. We designed the code for robustness, and therefore chose parameters in the code to enhance reliability. This choice results in

Table 54: Comparison of SEMI and PATH 4.x on GAMSLIB

Problem	With Restarts			Without Restarts		
	SEMI	Iterative	PATH 4.x	SEMI	Iterative	PATH 4.x
cafemge	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	1 (1)
cammcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
cammge	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
cirimge	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
co2mge	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
dmcmge	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
ers82mcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
etamge	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
finmge	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
gemmcp	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
gemmge	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
hansmcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
hansmge	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
harkmcp	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
harmge	6 (0)	3 (3)	6 (0)	6 (0)	3 (3)	6 (0)
kehomge	3 (0)	3 (0)	3 (0)	2 (1)	2 (1)	3 (0)
kormcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
mr5mcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
nsmge	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
oligomcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
sammge	16 (0)	16 (0)	16 (0)	16 (0)	16 (0)	16 (0)
scarfmcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
scarfmge	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
shovmge	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
threemge	9 (0)	9 (0)	9 (0)	9 (0)	9 (0)	9 (0)
transmcp	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
two3mcp	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
unstmge	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
vonthmcp	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	0 (1)
vonthmge	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
wallmcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
Total	93 (1)	90 (4)	94 (0)	91 (3)	88 (6)	92 (2)

Table 55: Comparison of SEMI and PATH 4.x on MCPLIB

Problem	With Restarts			Without Restarts		
	SEMI	Iterative	PATH 4.x	SEMI	Iterative	PATH 4.x
asean9a	1 (0)	0 (1)	1 (0)	1 (0)	0 (1)	1 (0)
badfree	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
bai_haung	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
bert_oc	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
bertsekas	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
billups	0 (3)	0 (3)	0 (3)	0 (3)	0 (3)	0 (3)
bishop	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
box	277 (84)	280 (81)	361 (0)	270 (91)	271 (90)	361 (0)
bratu	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
cammcf	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
carbon	6 (1)	6 (1)	6 (1)	6 (1)	6 (1)	6 (1)
cgereg	17 (2)	18 (1)	19 (0)	16 (3)	18 (1)	19 (0)
choi	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
colvdual	4 (0)	4 (0)	4 (0)	3 (1)	3 (1)	2 (2)
colvnlp	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
congest	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
cycle	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
degen	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
denmark	38 (0)	0 (40)	38 (0)	28 (10)	0 (40)	38 (0)
dirkse1	0 (2)	0 (2)	1 (1)	0 (2)	0 (2)	1 (1)
duopoly	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
eckstein	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
ehl_k40	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	2 (1)
ehl_k60	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	2 (1)
ehl_k80	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
ehl_kost	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
electric	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	0 (1)
endog	21 (2)	0 (2)	23 (0)	20 (3)	0 (2)	23 (0)
eppa	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)
eta2100	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
exemptions	182 (4)	182 (4)	186 (0)	182 (4)	182 (4)	186 (0)
explcp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
exros	5 (0)	5 (0)	5 (0)	5 (0)	5 (0)	2 (3)

Table 56: Comparison of SEMI and PATH 4.x on MCPLIB (cont.)

Problem	With Restarts			Without Restarts		
	SEMI	Iterative	PATH 4.x	SEMI	Iterative	PATH 4.x
finance1	21 (0)	21 (0)	21 (0)	21 (0)	21 (0)	21 (0)
finance2	10 (0)	10 (0)	10 (0)	10 (0)	10 (0)	10 (0)
finance3	20 (0)	20 (0)	20 (0)	20 (0)	20 (0)	20 (0)
fixedpt	0 (2)	0 (2)	1 (0)	0 (2)	0 (2)	0 (1)
forcebsm	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
forcedsa	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
freebert	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
fried7	5 (0)	5 (0)	5 (0)	5 (0)	5 (0)	5 (0)
fried8	2 (3)	2 (3)	3 (2)	1 (4)	1 (4)	2 (3)
gafni	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
games	25 (0)	25 (0)	25 (0)	25 (0)	25 (0)	23 (2)
gei	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
golanmcp	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
hanskoop	10 (0)	10 (0)	10 (0)	8 (2)	8 (2)	10 (0)
hansmcf	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
hanson	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
ho	5 (0)	5 (0)	5 (0)	5 (0)	5 (0)	5 (0)
hydroc06	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
hydroc20	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
jel	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
jmu	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	0 (1)
josephy	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)
keyzer	5 (1)	5 (1)	6 (0)	3 (3)	3 (3)	5 (1)
kojshin	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)
kyh-scale	0 (2)	0 (2)	1 (1)	0 (2)	0 (2)	0 (2)
kyh	0 (2)	0 (2)	2 (0)	0 (2)	0 (2)	2 (0)
leyffer	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
lincont	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
lstest	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)

Table 57: Comparison of SEMI and PATH 4.x on MCPLIB (cont.)

Problem	With Restarts			Without Restarts		
	SEMI	Iterative	PATH 4.x	SEMI	Iterative	PATH 4.x
markusen	18 (0)	18 (0)	18 (0)	17 (1)	17 (1)	18 (0)
mathinum	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
mathisum	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
methan08	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
mr5mcf	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
mrt	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
multi-v1	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
multi-v2	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
multi-v3	1 (0)	1 (0)	1 (0)	0 (1)	1 (0)	1 (0)
munson3	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	0 (1)
munson4	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
nash	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
ne-hard	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
nepal1	1 (2)	1 (2)	2 (1)	1 (2)	1 (2)	0 (3)
nepal2	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
obstacle	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)	8 (0)
olg	1 (0)	1 (0)	1 (0)	0 (1)	0 (1)	1 (0)
opening	2 (0)	2 (0)	2 (0)	1 (1)	1 (1)	2 (0)
opt_cont	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
opt_cont127	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
opt_cont255	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
opt_cont31	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
opt_cont511	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
pgvon105	2 (4)	2 (4)	5 (1)	2 (4)	2 (4)	5 (1)
pgvon106	1 (5)	1 (5)	5 (1)	1 (5)	0 (6)	5 (1)
pies	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
pizer	1 (2)	1 (2)	3 (0)	1 (2)	1 (2)	3 (0)
powell	6 (0)	6 (0)	6 (0)	4 (2)	4 (2)	6 (0)
powell_mcp	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)	6 (0)
qp	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)

Table 58: Comparison of SEMI and PATH 4.x on MCPLIB (cont.)

Problem	With Restarts			Without Restarts		
	SEMI	Iterative	PATH 4.x	SEMI	Iterative	PATH 4.x
ralph	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
renger	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)	0 (1)
romer	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	1 (1)
romer2	0 (7)	0 (7)	1 (6)	0 (7)	0 (7)	0 (7)
romer3	1 (0)	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)
runge	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
scarfanum	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
scarfasum	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
scarfbnum	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
scarfbsum	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
shubik	46 (1)	46 (1)	47 (0)	38 (9)	38 (9)	43 (4)
simple-ex	1 (0)	1 (0)	0 (1)	0 (1)	0 (1)	0 (1)
simple-red	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
spill1	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
spill2	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	0 (1)
spill3	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	0 (1)
spill4	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
spill5	0 (1)	0 (1)	1 (0)	0 (1)	0 (1)	1 (0)
sppe	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)
tinloi	64 (0)	64 (0)	64 (0)	64 (0)	64 (0)	56 (8)
tinloilp	0 (4)	0 (4)	4 (0)	0 (4)	0 (4)	4 (0)
tinsmall	64 (0)	64 (0)	64 (0)	64 (0)	64 (0)	63 (1)
titan	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)	1 (1)
tobin	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
tqbilat	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)	1 (0)
trade12	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
trafelas	1 (1)	0 (2)	2 (0)	1 (1)	0 (2)	2 (0)
trig	2 (1)	2 (1)	3 (0)	0 (3)	0 (3)	3 (0)
uruguay	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)	4 (0)
venables	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)	2 (0)
vonthmcf	0 (1)	1 (0)	1 (0)	0 (1)	1 (0)	1 (0)
water	3 (0)	3 (0)	3 (0)	3 (0)	3 (0)	2 (1)
xu1	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
xu2	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
xu3	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
xu4	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
xu5	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)	7 (0)
Total	1081 (151)	1024 (189)	1207 (24)	1034 (198)	987 (226)	1171 (60)

PATH 4.x being much faster than the semismooth code. However, we believe that many of the standard techniques for improving iterative linear equation solvers are applicable in the context of the semismooth algorithm.

7.5 Summary

We have shown that a code based on semismooth algorithm can be implemented as a robust MCP solver. Particular care needs to be taken in implementing the evaluation of Φ and Ψ, and in treating numerical issues related to constructing the Newton step. The resulting SEMI code is robust and has great potential in the very large scale setting as indicated by the numerical results.

We remark that SEMI is the first implementation of a semismooth algorithm for solving mixed complementarity problems that is available in the AMPL and GAMS modeling languages and the MATLAB and NEOS tools. We also note that an earlier version of this chapter appeared in [92].

Chapter 8

Conclusion

Interest in formulating and solving large-scale complementarity problems has become significant and continues to grow. One of the reasons for this growth is the availability of high-quality software built on strong theoretical foundations that can be accessed within convenient modeling environments.

Environments, such as the AMPL and GAMS modeling languages, and, to a lesser extent, the MATLAB and NEOS packages, have given practitioners the ability to quickly model complementarity problems and apply standard techniques to solve them. We surveyed the features of each of these environments in Chapter 2 and presented the low-level solver interface used by PATH 4.x and SEMI to support them in Chapter 3. The main contributions found in these two chapters were the introduction of the MATLAB and NEOS links and the specification of the solver interface, which can be used to make PATH 4.x or SEMI available in other environments.

Theory is used to ensure that an algorithm is well-defined and efficiently processes models. Chapter 6 developed a globalization strategy for successive linearization methods and proved that the method is well-defined for arbitrary complementarity problems, and globally and locally-fast convergent under a strong regularity assumption. This method was later used in the PATH 4.x algorithm.

Sufficient conditions guaranteeing that an algorithm converges have been studied in the literature. However, models can be easily generated which will cause most algorithms to fail. There are many reasons for this failure. For example, a code will fail when the model has no solution. In some cases, the preprocessor in Chapter 4 can detect this infeasibility and report the variables or constraints causing the infeasibility. Other examples include cases where the function is ill-defined, or the Jacobian matrix is ill-conditioned, poorly-scaled, or singular. The diagnostic information documented in Chapter 5 and provided by PATH 4.x and SEMI can be used to detect some of these problems.

Finally, sophisticated algorithm implementations, such as PATH 4.x and SEMI, are used to calculate solutions to complementarity problems. Chapter 6 presented the PATH 4.x code, which is a nonsmooth Newton method globalized with the Fischer-Burmeister merit function. The SEMI algorithm was developed in Chapter 7 and based on a semismooth algorithm applied to the penalized Fischer-Burmeister function. We studied methods for overcoming numerical problems in the code and investigated the use of iterative methods to solve the linear systems of equations. The resulting PATH 4.x and SEMI implementations were shown to be very robust on available test sets.

Bibliography

[1] E. Andersen and K. Andersen. Presolving in linear programming. *Mathematical Programming*, 71:221–245, 1995.

[2] L. Armijo. Minimization of functions having Lipschitz-continuous first partial derivatives. *Pacific Journal of Mathematics*, 16:1–3, 1966.

[3] K. Arrow and G. Debreu. Existence of equilibrium for a competitive economy. *Econometrica*, 22:265–290, 1954.

[4] D. P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 1995.

[5] S. C. Billups. *Algorithms for Complementarity Problems and Generalized Equations*. PhD thesis, University of Wisconsin–Madison, Madison, Wisconsin, August 1995.

[6] S. C. Billups. Improving the robustness of descent-based methods for semi-smooth equations using proximal perturbations. *Mathematical Programming*, 87:153–176, 2000.

[7] S. C. Billups and M. C. Ferris. QPCOMP: A quadratic program based solver for mixed complementarity problems. *Mathematical Programming*, 76:533–562, 1997.

[8] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR 2.0 user's guide. Mathematics and Computer Science Division Report ANL/MCS–TM–192, Argonne National Laboratory, Argonne, Illinois, 1995.

[9] J. Bisschop and A. Meeraus. On the development of a general algebraic modeling system in a strategic planning environment. *Mathematical Programming Study*, 20:1–29, 1982.

[10] A. Brearley, G. Mitra, and H. Williams. Analysis of mathematical programming problems prior to applying the simplex algorithm. *Mathematical Programming*, 8:54–83, 1975.

[11] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A User's Guide*. The Scientific Press, South San Francisco, CA, 1988.

[12] R. M. Chamberlain, M. J. D. Powell, and C. Lemaréchal. The watchdog technique for forcing convergence in algorithms for constrained optimization. *Mathematical Programming Study*, 16:1–17, 1982.

[13] B. Chen, X. Chen, and C. Kanzow. A penalized Fischer-Burmeister NCP-function: Theoretical investigation and numerical results. Preprint 126, Institute of Applied Mathematics, University of Hamburg, Hamburg, Germany, 1997.

[14] B. Chen and P. T. Harker. A non-interior-point continuation method for linear complementarity problems. *SIAM Journal on Matrix Analysis and Applications*, 14:1168–1190, 1993.

[15] B. Chen and P. T. Harker. Smooth approximations to nonlinear complementarity problems. *SIAM Journal on Optimization*, 7:403–420, 1997.

[16] Chunhui Chen and O. L. Mangasarian. Smoothing methods for convex inequalities and linear complementarity problems. *Mathematical Programming*, 78:51–70, 1995.

[17] F. H. Clarke. *Optimization and Nonsmooth Analysis*. John Wiley & Sons, New York, 1983.

[18] T. F. Coleman and A. Verma. *ADMAT User Guide*, 2000.

[19] R. W. Cottle. *Nonlinear programs with positively bounded Jacobians*. PhD thesis, Department of Mathematics, University of California, Berkeley, California, 1964.

[20] R. W. Cottle and G. B. Dantzig. Complementary pivot theory of mathematical programming. *Linear Algebra and Its Applications*, 1:103–125, 1968.

[21] R. W. Cottle, J. S. Pang, and R. E. Stone. *The Linear Complementarity Problem*. Academic Press, Boston, 1992.

[22] J. Czyzyk, M. P. Mesnier, and J. J. Moré. The Network-Enabled Optimization Server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[23] T. De Luca, F. Facchinei, and C. Kanzow. A semismooth equation approach to the solution of nonlinear complementarity problems. *Mathematical Programming*, 75:407–439, 1996.

[24] T. De Luca, F. Facchinei, and C. Kanzow. A theoretical and numerical comparison of some semismooth algorithms for complementarity problems. *Computational Optimization and Applications, forthcoming*, 1999.

[25] S. P. Dirkse. *Robust Solution of Mixed Complementarity Problems*. PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1994. Available from ftp://ftp.cs.wisc.edu/math-prog/tech-reports/.

[26] S. P. Dirkse and M. C. Ferris. MCPLIB: A collection of nonlinear mixed complementarity problems. *Optimization Methods and Software*, 5:319–345, 1995.

[27] S. P. Dirkse and M. C. Ferris. The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems. *Optimization Methods and Software*, 5:123–156, 1995.

[28] S. P. Dirkse and M. C. Ferris. A pathsearch damped Newton method for computing general equilibria. *Annals of Operations Research*, pages 211–232, 1996.

[29] S. P. Dirkse and M. C. Ferris. Crash techniques for large-scale complementarity problems. In Ferris and Pang [46], pages 40–61.

[30] S. P. Dirkse and M. C. Ferris. Traffic modeling and variational inequalities using GAMS. In Ph. L. Toint, M. Labbe, K. Tanczos, and G. Laporte, editors, *Operations Research and Decision Aid Methodologies in Traffic and Transportation Management*, volume 166 of *NATO ASI Series F*, pages 136–163. Springer-Verlag, 1998.

[31] S. P. Dirkse, M. C. Ferris, P. V. Preckel, and T. Rutherford. The GAMS callable program library for variational and complementarity solvers. Mathematical Programming Technical Report 94-07, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1994.

[32] B. C. Eaves. On the basic theorem of complementarity. *Mathematical Programming*, 1:68–87, 1971.

[33] F. Facchinei, A. Fischer, and C. Kanzow. A semismooth Newton method for variational inequalities: The case of box constraints. In Ferris and Pang [46], pages 76–90.

[34] F. Facchinei and J. Soares. A new merit function for nonlinear complementarity problems and a related algorithm. *SIAM Journal on Optimization*, 7:225–247, 1997.

[35] M. C. Ferris, R. Fourer, and D. M. Gay. Expressing complementarity problems and communicating them to solvers. *SIAM Journal on Optimization*, 9:991–1009, 1999.

[36] M. C. Ferris and C. Kanzow. Complementarity and related problems: A survey. In P. M. Pardalos and M. G. C. Resende, editors, *Handbook of Applied Optimization, forthcoming*. Oxford University Press, 2000.

[37] M. C. Ferris, C. Kanzow, and T. S. Munson. Feasible descent algorithms for mixed complementarity problems. *Mathematical Programming*, 86:475–497, 1999.

[38] M. C. Ferris and S. Lucidi. Nonmonotone stabilization methods for nonlinear equations. *Journal of Optimization Theory and Applications*, 81:53–71, 1994.

[39] M. C. Ferris, A. Meeraus, and T. F. Rutherford. Computing Wardropian equilibrium in a complementarity framework. *Optimization Methods and Software*, 10:669–685, 1999.

[40] M. C. Ferris, M. P. Mesnier, and J. Moré. NEOS and Condor: Solving nonlinear optimization problems over the Internet. *ACM Transactions on Mathematical Software, forthcoming*, 1999.

[41] M. C. Ferris and T. S. Munson. Case studies in complementarity: Improving model formulation. In M. Théra and R. Tichatschke, editors, *Ill–Posed Variational Problems and Regularization Techniques*, number 477 in Lecture Notes in Economics and Mathematical Systems, pages 79–98. Springer Verlag, Berlin, 1999.

[42] M. C. Ferris and T. S. Munson. Interfaces to PATH 3.0: Design, implementation and usage. *Computational Optimization and Applications*, 12:207–227, 1999.

[43] M. C. Ferris and T. S. Munson. Preprocessing complementarity problems. Mathematical Programming Technical Report 99-07, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1999.

[44] M. C. Ferris and T. S. Munson. Complementarity problems in GAMS and the PATH solver. *Journal of Economic Dynamics and Control*, 24:165–188, 2000.

[45] M. C. Ferris, T. S. Munson, and D. Ralph. A homotopy method for mixed complementarity problems based on the PATH solver. In D. F. Griffiths and G. A. Watson, editors, *Numerical Analysis 1999*, Research Notes in Mathematics, pages 143–167, London, 2000. Chapman and Hall.

[46] M. C. Ferris and J. S. Pang, editors. *Complementarity and Variational Problems: State of the Art*, Philadelphia, Pennsylvania, 1997. SIAM Publications.

[47] M. C. Ferris and J. S. Pang. Engineering and economic applications of complementarity problems. *SIAM Review*, 39:669–713, 1997.

[48] M. C. Ferris and D. Ralph. Projected gradient methods for nonlinear complementarity problems via normal maps. In D. Du, L. Qi, and R. Womersley, editors, *Recent Advances in Nonsmooth Optimization*, pages 57–87. World Scientific Publishers, 1995.

[49] M. Fiedler and V. Pták. On matrices with nonpositive off-diagonal elements and positive principal minors. *Czechoslovak Mathematics Journal*, 12:382–400, 1962.

[50] A. Fischer. A special Newton–type optimization method. *Optimization*, 24:269–284, 1992.

[51] A. Fischer. A new constrained optimization reformulation for complementarity problems. Technical Report MATH-NM-10-1995, Institute of Numerical Mathematics, Technical University of Dresden, Dresden, 1995.

[52] R. Fourer, D. M. Gay, and B. W. Kernighan. A modeling language for mathematical programming. *Management Science*, 36:519–554, 1990.

[53] R. Fourer, D. M. Gay, and B. W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 1993.

[54] R. M. Freund and N. M. Nachtigal. QMRPACK: A package of QMR algorithms. *ACM Transactions on Mathematical Software*, 22:46–77, 1996.

[55] D. M. Gay. Electronic mail distribution of linear programming test problems. *COAL Newsletter*, 13:10–12, 1985.

[56] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and Its Applications*, 88/89:239–270, 1987.

[57] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.

[58] G. H. Golub and W. Kahan. Calculating the singular values and pseudoinverse of a matrix. *SIAM Journal on Numerical Analysis*, 2:205–224, 1965.

[59] M. S. Gowda. Applications of degree theory to linear complementarity problems. *Mathematics of Operations Research*, 18:868–879, 1993.

[60] M. S. Gowda. An analysis of zero set and global error bound properties of a piecewise affine function via its recession function. *SIAM Journal on Matrix Analysis and Applications*, 17:594–609, 1996.

[61] M. S. Gowda and J. S. Pang. Stability analysis of variational inequalities and nonlinear complementarity problems, via the mixed linear complementarity problem and degree theory. *Mathematics of Operations Research*, 19:831–879, 1994.

[62] M. S. Gowda and R. Sznajder. Generalizations of P_0– and P– properties; extended vertical and horizontal LCPs. *Linear Algebra and Its Applications*, 223/224:695–715, 1995.

[63] L. Grippo, F. Lampariello, and S. Lucidi. A nonmonotone line search technique for Newton's method. *SIAM Journal on Numerical Analysis*, 23:707–716, 1986.

[64] L. Grippo, F. Lampariello, and S. Lucidi. A class of nonmonotone stabilization methods in unconstrained optimization. *Numerische Mathematik*, 59:779–805, 1991.

[65] P. T. Harker and J. S. Pang. Finite–dimensional variational inequality and nonlinear complementarity problems: A survey of theory, algorithms and applications. *Mathematical Programming*, 48:161–220, 1990.

[66] P. T. Harker and B. Xiao. Newton's method for the nonlinear complementarity problem: A B–differentiable equation approach. *Mathematical Programming*, 48:339–358, 1990.

[67] G. W. Harrison, T. F. Rutherford, and D. Tarr. Quantifying the Uruguay round. *The Economic Journal*, 107:1405–1430, 1997.

[68] P. Hartman and G. Stampacchia. On some nonlinear differential-functional equations. *Acta Mathematica*, 115:271–310, 1966.

[69] J. Huang and J. S. Pang. Option pricing and linear complementarity. *Journal of Computational Finance*, 2:31–60, 1998.

[70] ILOG CPLEX Division, 889 Alder Avenue, Incline Village, Nevada. *CPLEX Optimizer*. http://www.cplex.com/.

[71] N. H. Josephy. A Newton method for the PIES energy model. Technical Summary Report 1971, Mathematics Research Center, University of Wisconsin–Madison, Madison, Wisconsin, 1979.

[72] N. H. Josephy. Newton's method for generalized equations. Technical Summary Report 1965, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, 1979.

[73] N. H. Josephy. *Newton's Method for Generalized Equations and the PIES Energy Model*. PhD thesis, Department of Industrial Engineering, University of Wisconsin–Madison, 1979.

[74] C. Kanzow. An inexact QP–based method for nonlinear complementarity problems. *Numerische Mathematik, forthcoming*, 1999.

[75] C. Kanzow and H. Kleinmichel. A new class of semismooth Newton-type methods for nonlinear complementarity problems. *Computational Optimization and Applications*, 11:227–251, 1998.

[76] C. Kanzow and H. D. Qi. A QP-free constrained Newton-type method for variational inequality problems. *Mathematical Programming*, 85:81–106, 1999.

[77] S. Karamardian. The complementarity problem. *Mathematical Programming*, 2:107–129, 1972.

[78] W. Karush. Minima of functions of several variables with inequalities as side conditions. Master's thesis, Department of Mathematics, University of Chicago, 1939.

[79] H. W. Kuhn and A. W. Tucker. Nonlinear programming. In J. Neyman, editor, *Proceedings of the Second Berkeley Symposium on Mathematical Statistics and Probability*, pages 481–492. University of California Press, Berkeley and Los Angeles, 1951.

[80] C. E. Lemke. Bimatrix equilibrium points and mathematical programming. *Management Science*, 11:681–689, 1965.

[81] C. E. Lemke. On complementary pivot theory. In G. B. Dantzig and A. F. Veinott, editors, *Mathematics of the Decision Sciences: Part 1*, volume 11 of *Lectures in Applied Mathematics*, pages 95–114. American Mathematical Society, Providence, 1968.

[82] C. E. Lemke and J. T. Howson. Equilibrium points of bimatrix games. *SIAM Journal on Applied Mathematics*, 12:413–423, 1964.

[83] J. L. Lions and G. Stampacchia. Variational inequalities. *Communications on Pure and Applied Math*, 20:493–19, 1967.

[84] Z.-Q. Luo and P. Tseng. Error bound and convergence analysis of matrix splitting algorithms for the affine variational inequality problem. *SIAM Journal on Optimization*, 2:43–54, 1992.

[85] O. L. Mangasarian. *Nonlinear Programming*. McGraw–Hill, New York, 1969. SIAM Classics in Applied Mathematics 10, SIAM, Philadelphia, 1994.

[86] L. Mathiesen. Computation of economic equilibria by a sequence of linear complementarity problems. *Mathematical Programming Study*, 23:144–162, 1985.

[87] L. Mathiesen. Computational experience in solving equilibrium models by a sequence of linear complementarity problems. *Operations Research*, 33:1225–1250, 1985.

[88] L. Mathiesen. An algorithm based on a sequence of linear complementarity problems applied to a Walrasian equilibrium model: An example. *Mathematical Programming*, 37:1–18, 1987.

[89] MATLAB. *User's Guide*. The MathWorks, Inc., 1992.

[90] R. Mifflin. Semismooth and semiconvex functions in constrained optimization. *SIAM Journal on Control and Optimization*, 15:957–972, 1977.

[91] J. J. Moré and D. C. Sorensen. Computing a trust region step. *SIAM Journal on Scientific and Statistical Computing*, 4:553–572, 1983.

[92] T. S. Munson, F. Facchinei, M. C. Ferris, A. Fischer, and C. Kanzow. The semismooth algorithm for large scale complementarity problems. Mathematical Programming Technical Report 99-06, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1999.

[93] B. A. Murtagh and M. A. Saunders. MINOS 5.0 user's guide. Technical Report SOL 83.20, Stanford University, Stanford, California, 1983.

[94] J. F. Nash. Equilibrium points in N–person games. *Proceedings of the National Academy of Sciences*, 36:48–49, 1950.

[95] J. F. Nash. Non-cooperative games. *Annals of Mathematics*, 54:286–295, 1951.

[96] J. M. Ortega and W. C. Rheinboldt. *Iterative Solution of Nonlinear Equations in Several Variables*. Academic Press, San Diego, California, 1970.

[97] C. C. Paige and M. A. Saunders. LSQR: An algorithm for sparse linear equations and sparse least squares. *ACM Transactions on Mathematical Software*, 8:43–71, 1982.

[98] J. S. Pang. A B–differentiable equation based, globally and locally quadratically convergent algorithm for nonlinear programs, complementarity and variational inequality problems. *Mathematical Programming*, 51:101–132, 1991.

[99] J. S. Pang and S. A. Gabriel. NE/SQP: A robust algorithm for the nonlinear complementarity problem. *Mathematical Programming*, 60:295–338, 1993.

[100] J. S. Pang and L. Qi. Nonsmooth equations: Motivation and algorithms. *SIAM Journal on Optimization*, 3:443–465, 1993.

[101] L. Qi. Convergence analysis of some algorithms for solving nonsmooth equations. *Mathematics of Operations Research*, 18:227–244, 1993.

[102] L. Qi and J. Sun. A nonsmooth version of Newton's method. *Mathematical Programming*, 58:353–368, 1993.

[103] D. Ralph. Global convergence of damped Newton's method for nonsmooth equations, via the path search. *Mathematics of Operations Research*, 19:352–389, 1994.

[104] D. Ralph. On branching numbers of normal manifolds. *Nonlinear Analysis, Theory, Methods and Applications*, 22:1041–1050, 1994.

[105] S. M. Robinson. Generalized equations and their solution: Part I: Basic theory. *Mathematical Programming Study*, 10:128–141, 1979.

[106] S. M. Robinson. Strongly regular generalized equations. *Mathematics of Operations Research*, 5:43–62, 1980.

[107] S. M. Robinson. Normal maps induced by linear transformations. *Mathematics of Operations Research*, 17:691–714, 1992.

[108] S. M. Robinson. Newton's method for a class of nonsmooth functions. *Set Valued Analysis*, 2:291–305, 1994.

[109] S. M. Robinson. A reduction method for variational inequalities. *Mathematical Programming*, 80:161–169, 1998.

[110] R. T. Rockafellar. *Convex Analysis*. Princeton University Press, Princeton, New Jersey, 1970.

[111] T. F. Rutherford. MILES: A mixed inequality and nonlinear equation solver. Working Paper, Department of Economics, University of Colorado, Boulder, 1993.

[112] T. F. Rutherford. Extensions of GAMS for complementarity problems arising in applied economic analysis. *Journal of Economic Dynamics and Control*, 19:1299–1324, 1995.

[113] T. F. Rutherford. Applied general equilibrium modeling with MPSGE as a GAMS subsystem: An overview of the modeling framework and syntax. *Computational Economics*, 14:1–46, 1999.

[114] Y. Saad. *Iterative Methods for Sparse Linear Systems*. PWS Publishing Company, Boston, Massachusetts, 1996.

[115] M. W. P. Savelsbergh. Preprocessing and probing techniques for mixed integer programming problems. *ORSA Journal on Computing*, 6:445–454, 1994.

[116] H. Sellami and S. M. Robinson. Implementation of a continuation method for normal maps. *Mathematical Programming*, 76:563–578, 1997.

[117] J. Tomlin and J. Welch. Finding duplicate rows in a linear programming model. *Operations Research Letters*, 5(1):7–11, 1986.

[118] P. Tseng. Growth behavior of a class of merit functions for the nonlinear complementarity problem. *Journal of Optimization Theory and Applications*, 89:17–37, 1996.

[119] H. F. Walker. Implementation of the GMRES method using Householder transformations. *SIAM Journal on Scientific Computing*, 9:152–163, 1988.

[120] L. Walras. *Elements of Pure Economics*. Allen and Unwin, London, 1954.

[121] J. G. Wardrop. Some theoretical aspects of road traffic research. *Proceeding of the Institute of Civil Engineers, Part II*, pages 325–378, 1952.

[122] P. Wilmott, J. Dewynne, and S. Howison. *Option Pricing*. Oxford Financial Press, Oxford, England, 1993.

[123] S. J. Wright. *Primal–Dual Interior–Point Methods*. SIAM, Philadelphia, Pennsylvania, 1997.