ROBUST SOLUTION OF MIXED COMPLEMENTARITY PROBLEMS

By

Steven P. Dirkse

A THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN – MADISON

1994

Abstract

Robust Solution of Mixed Complementarity Problems
Steven P. Dirkse
Under the Supervision of Associate Professor Michael C. Ferris
at the University of Wisconsin–Madison

This thesis is concerned with algorithms and software for the solution of the Mixed Complementarity Problem, or MCP. The MCP formulation is useful for expressing systems of nonlinear inequalities and equations; the complementarity allows boundary conditions be to specified in a succinct manner. Problems of this type occur in many branches of the sciences, including mathematics, engineering, economics, operations research, and computer science.

The algorithm we propose for the solution of MCP is a Newton based method containing a novel application of a nonmonotone stabilization technique previously applied to methods for solving smooth systems of equalities and for unconstrained minimization. In order to apply this technique, we have adapted and extended the path construction technique of Ralph (1994), resulting in the PATH algorithm. We present a global convergence result for the PATH algorithm that generalizes similar results obtained in the smooth case. The PATH solver is a sophisticated implementation of this algorithm that makes use of the sparse basis updating package of MINOS 5.4.

Due to the widespread use of algebraic modeling languages in the practice of operations research, economics, and other fields from which complementarity problems are drawn, we have developed a complementarity facility for both the GAMS and AMPL modeling languages, as well as software interface libraries to be used in hooking up a complementarity solver as a solution subsystem. These interface libraries provide the algorithm developer with

a convenient and efficient means of developing and testing an algorithm, while also benefiting the modeling community by providing ready access to the latest advances in algorithmic development.

The library interface routines are used to read a number of complementarity models formulated in the GAMS and AMPL modeling languages. We define the syntax required for these models and describe their derivation. These models have been collected to form a library and have been made publicly available so that others may benefit from this work.

We present extensive computational results for the PATH solver and other solution techniques, many of which are obtained by using the interface library and the library of complementarity models developed for this purpose.

Acknowledgements

Of the many debts of gratitude I owe, none is greater than the one owed to my wife, Pat. She has been the best part of my life, and words cannot express my appreciation for the support and encouragement she has given me and the sacrifices she has made on my behalf.

I thank my parents for providing a secure home, invaluable discipline, and constant encouragement. They have given up much to pay for my education, and have fostered my love of learning for as long as I can remember.

Any success I have met with in graduate school has been due in large part to my advisor, Michael Ferris. His advice, ideas, and guidance have been invaluable to me. Working with him has been a joyful and enlightening experience. I thank James Morris and Robert Meyer for serving on my thesis committee and Olvi Mangasarian and Stephen Robinson for serving as readers. Their interest in my work has been a source of encouragement, while their commments and suggestions have greatly improved my work. In addition, Renato DeLeone, Ferris, Mangasarian, Meyer and Robinson have served as classroom instructors and have helped to stimulate my interest in mathematical programming. Gerard Venema of Calvin College was instrumental in my decision to pursue an education in mathematics and did much to make my graduate career possible.

Many colleagues have lent a hand. Danny Ralph introduced me to the path concept and provided comments during the early stages of this work. Tom Rutherford's pioneering work in using the GAMS language provided a foundation for much of my research. GAMS Development Corporation donated a GAMS system, while Alex Meeraus, Ramesh Raman, and Erwin Kalvelagen answered many questions. David Gay's suggestions helped lead to the design of the AMPL interface library. S. Chan Choi provided data for an interesting equilibrium model. Pete TerMaat introduced me to EMACS, while Jon Cargille helped me to use it wisely. I have enjoyed frequent discussions and infrequent golf outings with my office mate, Stephen Billups.

Finally, I would like to thank Rick and Jill Poel for providing a reminder that my life is not my own, but that I belong to one who provides all that I need.

> Call to me, and I will answer you, and I will tell you great and mighty things, which you do not know.
>
> *Jeremiah 33:3*

This research was partially funded by Fellowships from the National Science Foundation and the Wisconsin Alumni Research Foundation, as well as by the National Science Foundation under Grant CCR–9157632 and the Air Force Office of Scientific Research under Grants AFOSR–89–0410 and F49620-94-1-0036.

Contents

Abstract **ii**

Acknowledgements **iv**

1 Introduction **1**

1.1 Notation 3

1.2 The Mixed Complementarity Problem 4

1.3 Newton-based Equation Solvers 10

1.4 Modeling Languages 13

1.5 Chapter Outline 15

2 Modeling Language Interfaces **17**

2.1 Interface Tasks 18

2.2 GAMS/MCP 21

2.2.1 Developer-Written Subroutines 25

2.2.2 CPLIB Subroutines 26

2.2.3 Communication and Control 32

2.2.4 The C Interface 33

2.3 AMPL/MCP 36

2.4 Interface comparisons 42

3 MCPLIB: A Model Library **49**

3.1 MCP Syntax for GAMS and AMPL 50

3.2 The Model Library 53

3.2.1 Computing a Nash Equilibrium 53

3.2.2 A Spatial Price Equilibrium Model 56
3.2.3 A Walrasian Equilibrium Model 60
3.2.4 A Traffic Assignment Model 63
3.2.5 Computing an Invariant Capital Stock 65
3.2.6 Extended Linear-Quadratic Programming 67
3.2.7 An Obstacle Problem 72
3.2.8 The Elastohydrodynamic Lubrication Problem 73

4 The PATH Solver **76**
4.1 Approximation 77
4.2 Path Generation 78
4.3 Pathsearch Damping 85
4.4 Nonmonotone Stabilization 88
4.5 A Global Convergence Result 94

5 Computational Results **101**
5.1 Comparison of PATH to Josephy-Newton and MILES 102
5.2 Comparison of PATH to B-DIFF and NE/SQP 118
5.3 Comparison of PATH to Other Techniques 120
5.4 Conclusions 123

6 Preprocessing and Other Extensions **124**
6.1 Preprocessing 124
6.2 Other Extensions 137
6.3 Conclusions 138

Chapter 1

Introduction

In this thesis, we are concerned with the robust solution of nonlinear mixed complementarity problems (MCP's) arising in practical situations. In particular, we describe a novel method for the solution of these problems, prove a convergence result for this method, and give extensive computational results demonstrating the effectiveness of the proposed method as compared to other techniques considered in the literature. Since computational tests on a wide variety of problems are essential in determining the relative merit of the many algorithms proposed for the complementarity problem, we have developed a library of test problems, formulated in the GAMS and AMPL modeling languages and drawn from a number of different fields, with which to test our solver and others. The development of this library, along with the software necessary to make use of it, is also described in this work.

It would be difficult to overestimate the importance of the complementarity problem. Since its definition nearly 30 years ago, it has been the subject of intense study regarding the existence, uniqueness, and computability of its solutions. Originally noted in its linear form as a unifying framework for quadratic and linear programming and as a useful tool in game theory, the complementarity problem and its close relative, the variational inequality (VI), have become fundamental problems in the field, due in part to the fact that the optimality conditions for most problems in constrained and unconstrained optimization can be expressed as a VI.

Even more importantly, due to asymmetry of the Jacobian matrix, many complementarity problems are difficult to express as smooth optimization problems, that is, as the minimization of a smooth function subject to a number of smooth constraints. Thus, techniques for solving these types of complementarity problems efficiently and robustly are especially important. Prime examples of these types of problems include the general equilibrium models which arise in economics. The Jacobian matrix for these models is often asymmetric, so that the usefulness of a smooth minimization approach is limited. These models are used in tax policy analysis for the U.S. and elsewhere, in setting corporate average fuel economy standards, in analyzing potential growth patterns in an economy, and in analyzing the present and future effects of policy changes on the environment and global carbon emissions. Other complementarity problems occur in the areas of mechanical engineering, in Nash and spatial equilibrium models, and in transportation and regional science.

In order to effectively solve the mixed complementarity problem, we will rely primarily on a damped variant of Newton's method applied to a reformulation of the MCP as a nonsmooth system of equations (described in Section 1.2). This approach is motivated by both theoretical and practical considerations. Newton's method has been shown to perform well on a wide range of problems encountered in practice, while it possesses excellent local convergence properties that can be generalized via linesearch or trust region techniques. We will show that our proposed algorithm, a pathsearch damped, nonmonotonically stabilized version of Newton's method for the MCP, is globally convergent under conditions similar to those used to show the convergence of other Newton-type algorithms for equations outlined in Section 1.3.

Many of the applications mentioned above are taken from economics and related fields. In these disciplines, the GAMS modeling language (Brooke, Kendrick & Meeraus 1988) is widely accepted and extensively used to formulate linear, nonlinear, and mixed integer programs. It was only natural, then, that complementarity problems be formulated in GAMS as well. In order to do so, it was necessary to extend the GAMS language and write an interface library of software routines used in linking a MCP solver to GAMS. This was done by Dirkse, Ferris, Preckel & Rutherford (1994); the resulting complementarity format is known as GAMS/MCP. A demonstration of the use of the GAMS/MCP system for equilibrium analysis and game theory is provided by Rutherford (Rutherford 1994*b*). In addition, Rutherford (Rutherford 1994*a*) has embedded MPSGE, a modeling language designed specifically for solving Arrow-Debreu economic equilibrium models, in GAMS/MCP. In fact, the widely publicized estimate of $250 billion for annual economic benefits for the GATT world trade agreement was produced by GAMS/MCP and the MPSGE subsystem.

While the GAMS modeling language is the standard in economics, it is not so dominant in other fields, such as mathematical programming and operations research, where a more recent modeling language called AMPL has been gaining increased acceptance. AMPL has a number of features not found in GAMS, such as a facility for "defining" variables in terms of a function and a syntax more suited to those accustomed to the notation of mathematical programming. Motivated by these factors, we have developed a technique whereby complementarity problems can be expressed in the AMPL language, and have written an interface library which can be used to quickly and easily hook a solver to AMPL.

As an aid in testing our solver, and in order to compare it to other available software, we have formulated MCPLIB, a library of GAMS/MCP models drawn from a wide variety of disciplines. Many of the models in the library have been coded in AMPL as well. This model library, when coupled with the appropriate interface software, provides a ready source of test problems for anyone wishing to develop or evaluate a complementarity solver. In addition, Brooke et al. (1988) and Fourer, Gay & Kernighan (1993) show that the use of a modeling language in problem formulation has a number of advantages over the use of a programming language such as C or Fortran.

1.1 Notation

A word about notation is in order. The set of real numbers is denoted by $\mathbb{R}$ and the extended reals by $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$. The nonnegative orthant in $\mathbb{R}^n$ is denoted by $\mathbb{R}^n_+$. Matrices and vectors in $\mathbb{R}^{m \times n}$ and $\mathbb{R}^n$ are denoted by upper and lower case Arabic letters, respectively. The transpose of a matrix A is denoted by $A^\top$, and similarly for vectors. The vector $e \in \mathbb{R}^n$ is the vector whose components are all one, while the vector $e_i \in \mathbb{R}^n$ is the the vector whose components are all equal to zero except the i'th component, which is one. The Euclidean unit ball is denoted by $\mathbb{B} := \{x \mid \|x\|_2 \leq 1\}$. Two vectors ℓ and u in $\overline{\mathbb{R}}^n$ define a box or rectangle $B = [\ell, u]$, where $[\ell, u] := \{z \mid \ell \leq z \leq u\}$. The relative interior of a convex set C is denoted by $ri\, C$ and is defined to be the interior of C with respect to its affine hull (Rockafellar 1970, Section 6).

Assuming the set $C \subset \mathbb{R}^n$ is closed and convex, we denote the projection operator onto the set C as $\pi_C(\cdot)$; $\pi_C(x)$ is the unique point in C which minimizes the Euclidean norm $\|c - x\|_2$ for $c \in C$. The projection of a vector x onto $\mathbb{R}^n_+$ is denoted more simply by x_+.

The *negative* of the projection onto the negative orthant is denoted by x_-, so that while $x_- \geq 0$, $x = x_+ - x_-$ decomposes x into its positive and negative parts.

We write $s \downarrow 0$ to mean $s \to 0, s > 0$. Given a scalar or vector function $h(s)$, we say $h(s) = o(s)$ (as $s \downarrow 0$) if $h(s)/\|s\| \to 0$ in norm as $s \downarrow 0$; similarly, $h(s) = O(s)$ if $h(s)/\|s\|$ is bounded as $s \downarrow 0$. Similar definitions hold for the cases where $s \to \infty$. A function F is Lipschitz of modulus $L \geq 0$ on a subset X_0 of $\mathbb{R}^n$ if $\|F(x) - F(y)\| \leq L \|x - y\| \ \forall\, x, y \in X_0$. A function F is Lipschitz invertible of modulus $L \geq 0$ if F is bijective and its inverse mapping is Lipschitz of modulus L.

1.2 The Mixed Complementarity Problem

In this section, we define the Mixed Complementarity Problem (MCP), the chief problem of interest for this thesis, and a number of related problems. Both in this section and throughout, we assume that $F : C \mapsto \mathbb{R}^n$ is a continuously differentiable mapping from an open set containing C, where $C \subset \mathbb{R}^n$ is a closed convex set.

When the set $C \equiv \mathbb{R}^n_+$ (the nonnegative orthant), we have the well-known nonlinear complementarity problem (NCP) defined by F: find $z \in \mathbb{R}^n$ such that

$$0 \leq F(z) \quad \perp \quad z \geq 0, \tag{NCP}$$

where $\perp$ indicates a complementarity relationship between F and z (in this case, $\langle F(z), z \rangle = 0$). The NCP bears a close relationship to the variational inequality $\mathrm{VI}(F, C)$, that of finding $z \in C$ such that

$$\langle F(z), c - z \rangle \geq 0 \quad \forall\, c \in C. \tag{VI}$$

It is not difficult to show that z solves NCP iff z solves $\mathrm{VI}(F, \mathbb{R}^n_+)$, so that NCP is a special case of VI. This result is a special case of Theorem 2 below which relates the VI to the mixed complementarity problem.

Given a function F and a box $B := [\ell, u]$, we define below the mixed complementarity problem $\mathrm{MCP}(F, B)$. Suppressing the F and B, we will refer to the problem as MCP when it is convenient to do so.

Definition 1 (MCP) *Given a box* $B := [\ell, u]$ *and a function* $F : B \to \mathbb{R}^n$*,*

$$\text{find} \quad z \in \mathbb{R}^n, \quad w, v \in \mathbb{R}^n_+$$

$$\text{s.t.} \quad \begin{aligned} F(z) &= w - v &\quad (1.1a)\\ \ell \le z &\le u &\quad (1.1b)\\ \langle w, z - \ell \rangle &= 0 &\quad (1.1c)\\ \langle v, u - z \rangle &= 0 &\quad (1.1d) \end{aligned}$$

In the remainder of the thesis, we shall use the notation

$$F(z) \quad \perp \quad z \in [\ell, u]$$

to express the complementarity conditions (1.1). When convenient, we will include the implied bounds on F and use an inequality to indicate the finite variable bounds ℓ and u, as in (NCP).

The MCP can be viewed in at least two ways. On one hand, it can be seen as a generalization of the NCP to the case of general (and perhaps infinite) lower and upper bounds on the variables z, rather than the nonnegativity condition imposed in the NCP. Just as any practical implementation of an interior point or simplex method for linear programming must explicitly consider lower and upper variable bounds and free variables, so too must an algorithm for solving complementarity problems. Thus, the w and v in the above definition can be viewed as simply the positive and negative parts of $F(z)$, which must be complementary to the difference between z and its lower and upper bounds ℓ and u, respectively. Note that the choice of z completely determines w and v, so that we can speak of either (z, w, v) or z solving MCP, as convenience dictates.

Many problems commonly considered in the literature are equivalent or can be reduced to MCP, including nonlinear equations ($B := \mathbb{R}^n$) and nonlinear complementarity problems. MCP reduces to NCP when the box B defined by ℓ and u is the positive orthant (i.e. $\ell := 0$ and $u := \infty$). These bounds imply that $z \ge 0$, while (1.1d) implies that $v \equiv 0$, so that $F(z) = w \ge 0$, while $\langle F(z), z \rangle = 0$ follows from (1.1c).

The MCP can also be viewed as a special case of the VI where the set C is replaced by a box $B = [\ell, u]$, as we show in the following theorem.

Theorem 2 *Given a rectangular set $B := [\ell, u]$ and function $F : B \to \mathbb{R}^n$, the vector z solves MCP(F, B) $\Leftrightarrow$ z solves VI(F, B).*

Proof ($\Rightarrow$) Assume z solves MCP. Then $z \in B$, and for all $c \in B$,

$$\begin{aligned} \langle F(z), c - z \rangle &= -\langle F_+(z), z - c \rangle - \langle F_-(z), c - z \rangle \\ &\geq -\langle F_+(z), z - \ell \rangle - \langle F_-(z), u - z \rangle \\ &= 0. \end{aligned}$$

($\Leftarrow$) If z solves VI, then $\ell \leq z \leq u$. Define $w := F_+(z)$, $v := F_-(z)$, so that $F(z) = w - v$. For any index $i \in 1, \ldots, n$, assume $w_i > 0$ and $z_i > \ell_i$, so that

$$\langle F(z), (z + (\ell_i - z_i)e_i) - z \rangle = w_i(\ell_i - z_i) < 0.$$

This cannot be the case (since z solves VI), so that either $w_i = 0$ or $z_i - \ell_i = 0$. Since i was arbitrary, $\langle w, z - \ell \rangle = 0$ as well. Similarly, $\langle v, u - z \rangle = 0$, so that z is a solution to MCP. $\square$

The normal cone is closely related to the VI and is an important and useful tool. Given a closed convex set $C \subset \mathbb{R}^n$ and a point $z \in \mathbb{R}^n$, the normal cone to C at z is defined to be the set of all directions making an obtuse angle with any direction in C emanating from z, i.e.

$$N_C(z) := \begin{cases} \{y \mid \langle y, c - z \rangle \leq 0 \quad \forall\, c \in C\} & \text{if } z \in C \\ \emptyset & \text{if } z \notin C \end{cases}$$

Clearly, $\bar{z}$ solves VI$(F, C) \iff -F(\bar{z}) \in N_C(\bar{z})$. Thus, solutions to the VI can be described in terms of the normal cone.

As Theorem 2 shows, VI(F, C) is equivalent to MCP(F, C) when C is rectangular. When C is polyhedral rather than rectangular, VI(F, C) can be reduced to an MCP by explicitly including the dual variables to the constraints defining C.

Theorem 3 *Given a box $B := [\ell, u]$ and a set $X := \{z \mid Az \leq b\}$, where $A \in \mathbb{R}^{m \times n}$, VI$(F, B \cap X)$ is equivalent to MCP$(H, B \times \mathbb{R}^m_+)$, where*

$$H(z, u) := \begin{bmatrix} F(z) + A^\top u \\ -Az + b \end{bmatrix}.$$

Proof If $\bar{z}$ solves $\mathrm{VI}(F, B \cap X)$, then $-F(\bar{z}) \in N_B(\bar{z}) + N_X(\bar{z})$, since B and X are both polyhedral. If we partition the rows of A into those corresponding to constraints active ($\mathcal{A}$) and inactive ($\mathcal{I}$) at $\bar{z}$, we can express the normal cone to X at $\bar{z}$ as $N_X(\bar{z}) = \{A_{\mathcal{A}}^{\top} u \mid u \geq 0\}$, so that there exists a $\bar{u} \geq 0$ such that $-(F(\bar{z}) + A^{\top}\bar{u}) \in N_B(\bar{z})$, where $\bar{u}_i = 0$ for all $i \in \mathcal{I}$. Since $A\bar{z} - b \leq 0$ and by choice of $\bar{u}$, we have $A\bar{z} - b \in N_{\mathbb{R}^m_+}(\bar{u})$ as well, so that $-H(\bar{z}, \bar{u}) \in N_{B \times \mathbb{R}^m_+}(\bar{z}, \bar{u})$. Thus, $(\bar{z}, \bar{u})$ solves $\mathrm{VI}(H, B \times \mathbb{R}^m_+)$.

If $(\bar{z}, \bar{u})$ solves $\mathrm{MCP}(H, B \times \mathbb{R}^m_+)$, then $-H(\bar{z}, \bar{u}) \in N_{B \times \mathbb{R}^m_+}(\bar{z}, \bar{u})$. Thus, $-(F(\bar{z}) + A^{\top}\bar{u}) \in N_B(\bar{z})$ and $A\bar{z} - b \in N_{\mathbb{R}^m_+}(\bar{u})$. Consequently, we see that $\bar{z} \in X$, $\bar{u} \geq 0$, and $\bar{u}_i = 0$ for all $i \in \mathcal{I}$. Thus, $A^{\top}\bar{u} \in N_X(\bar{z})$, so that $-F(\bar{z}) \in N_B(\bar{z}) + N_X(\bar{z})$, and $\bar{z}$ solves $\mathrm{VI}(F, B \cap X)$.

□

As formulated above, the MCP is not amenable to solution via the powerful Newton-based techniques used in finding zeros to systems of equations. To express the MCP as a zero-finding problem, the normal map of Eaves (1971) and Robinson (1990, 1992) is used. In what follows, we show how the normal map can be derived as a natural result of the MCP under consideration, thus providing some intuition into the relationship between these two problems.

Since MCP is equivalent to the box-constrained VI, $z \in B$ solves MCP if and only if

$$\langle -F(z), c - z \rangle \leq 0 \qquad \forall\, c \in B. \tag{1.2}$$

If we define $x := z - F(z)$, then the inequality

$$\langle x - z, c - z \rangle \leq 0 \qquad \forall\, c \in B \tag{1.3}$$

follows from inequality (1.2). But (1.3) is the projection inequality (Hiriart-Urruty & Lemaréchal 1993); assuming $z \in B$, (1.3) holds if and only if $z := \pi_B(x)$, the Euclidean projection of x onto B. Hence, the equation

$$-F(\pi_B(x)) = x - \pi_B(x) \tag{1.4}$$

is satisfied. Conversely, if (1.4) holds, then since the projection inequality (1.3) holds for $z = \pi_B(x)$, it follows that (1.2) holds as well. Thus, $\pi_B(x)$ solves MCP, so that solving equation (1.4) is equivalent to solving MCP. This is made precise in the following definition and theorem.

Definition 4 (Normal Map) *Given a closed convex set $B \subset \mathbb{R}^n$ and a function $F : B \to \mathbb{R}^n$, the normal map $F_B(\cdot)$ induced on F by B is defined as*

$$F_B(x) := F(\pi_B(x)) + (x - \pi_B(x)).$$

The corresponding normal map equation is then defined as

$$0 = F_B(x) = F(\pi_B(x)) + (x - \pi_B(x)). \tag{NME}$$

Theorem 5 follows directly from equations (1.2), (1.3), and (1.4) above and the discussion surrounding them.

Theorem 5 *Given a rectangular set $B := [\ell, u]$ and function $F : B \to \mathbb{R}^n$, the vector $x \in \mathbb{R}^n$ solves NME $\Rightarrow z := \pi_B(x)$ solves MCP, while z solves MCP $\Rightarrow x := z - F(z)$ solves NME.*

Since the projection mapping π_B is continuous (Hiriart-Urruty & Lemaréchal 1993), a necessary and sufficient condition for the continuity of F_B is the continuity of F on B. However, since π_B is in general nondifferentiable, F_B also fails to be differentiable. In order to better understand the nondifferentiability of F_B, we must take a closer look at the projection π_B.

We first define the faces of $B = [\ell, u]$. In this case, the faces are essentially determined by forcing some of the defining inequalities of B, namely $\ell \leq z \leq u$, to be satisfied as equalities. Thus, if I and J are disjoint subsets of $\{1, \ldots, n\}$, then a corresponding face of B is $\{z \in B \mid z_I = \ell_I,\ \ z_J = u_J\}$. For example, if $B = \mathbb{R}^n$, then B has only one nonempty face, namely B itself. On the other hand, if $B = \mathbb{R}^2_+$, the nonnegative orthant of $\mathbb{R}^2$, then the four nonempty faces of B are $(0, 0)$, $0 \times \mathbb{R}_+$, $\mathbb{R}_+ \times 0$, and $\mathbb{R}^2_+$. These faces are critically related to π_B. Given a face F of the set B, let σ represent all the points in $\mathbb{R}^n$ that are projected onto F by π_B. The collection of all such σ is called the *normal manifold*. The sets σ are called cells of the normal manifold. Robinson (1992) has shown that each cell is polyhedral, has dimension n, and is of the form $F + N_F$, where N_F is defined to be the normal cone on $ri\, F$. In addition, we note that the sets $\sigma' := ri\, F + N_F$ form a partition of $\mathbb{R}^n$. Returning to our two examples above, when $B = \mathbb{R}^n$, the only cell is $\sigma = \mathbb{R}^n$, which has dimension n and partitions $\mathbb{R}^n$. For $B = \mathbb{R}^2_+$, the four cells σ are the orthants of $\mathbb{R}^2$, each of which has dimension 2, while the four sets σ' partition $\mathbb{R}^2$.

The normal manifold in $\mathbb{R}^2$ corresponding to the box $B := [0, \infty) \times [0, 1]$ is given in Figure 1. The six cells of this manifold, in clockwise order, are B, $[0, \infty) \times (-\infty, 0]$, $(-\infty, 0] \times (-\infty, 0]$, $(-\infty, 0] \times [0, 1]$, $(-\infty, 0] \times [1, \infty)$, and $[0, \infty) \times [1, \infty)$.

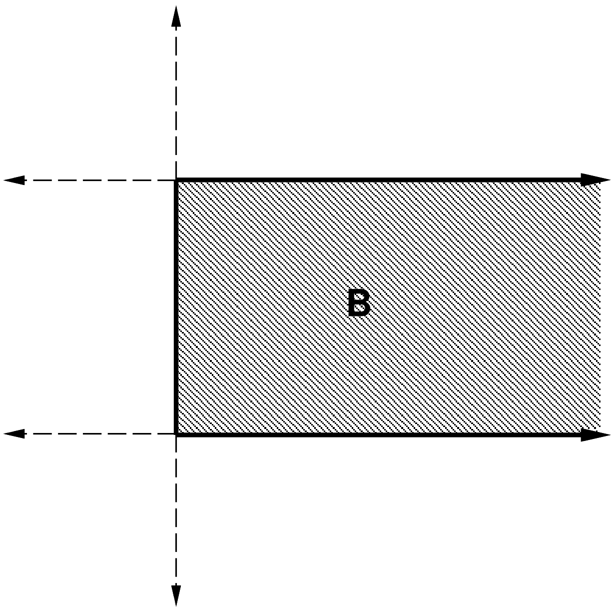


Figure 1: Normal Manifold for $B = [0, \infty) \times [0, 1]$

The projection $\pi_B(x)$ onto the box $B := [\ell, u]$ can be computed component-wise as follows

$$(\pi_B(x))_i = \begin{cases} \ell_i & \text{if } x_i < \ell_i, \\ x_i & \text{if } \ell_i \leq x_i \leq u_i, \\ u_i & \text{if } u_i < x_i. \end{cases} \tag{1.5}$$

In this case, $\mathbb{R}^n$ is partitioned into at most 3^n rectangular cells where in each cell the function used to compute $\pi_B(x)_i$ is affine. Thus, the restriction of the projection operator π_B to each of these cells is affine.

The normal manifold provides a useful tool for working with the normal map, since it partitions $\mathbb{R}^n$ into a number of cells on each of which π_B, and hence F_B, is smooth. This allows us to view F_B as a smooth nonlinear function on the interior of each each of these cells, or as a piecewise-smooth function over the whole space. The smoothness properties of

F_B will be essential in developing Newton methods for the solution of (NME) and in proving sufficient conditions for their convergence.

1.3 Newton-based Equation Solvers

In this section, we describe some of the algorithms previously proposed for the solution of systems of nonlinear equations, for the mixed complementarity problem, and for a number of related problems. Since the solvers considered in this thesis are primarily of Newton type, we first review Newton's method for solving systems of equations, as well as some of the extensions applied to the basic method. This method and its extensions will serve as a model for our proposed solution methods for MCP.

Newton's method for solving the equation

$$F(x) = 0 \tag{NLE}$$

consists of two steps, approximation and zero-finding, applied repeatedly to produce a sequence of iterates $\{x^k\}$. In an approximation step, the function F is approximated, or linearized, at the point x^k by the affine function $A_k(\cdot)$ defined by

$$A_k(x) := F(x^k) + F'(x^k)(x - x^k). \tag{1.6}$$

The Newton point x_N^k is a zero of the approximation A_k, i.e. $A_k(x_N^k) = 0$. If the Jacobian matrix $F'(x^k)$ is nonsingular, this zero is unique, and is conceptually easy to find. Upon solving the matrix equation $F'(x^k)d^k = -F(x^k)$, the Newton point is given by $x_N^k = x^k + d^k$, where d^k is the Newton direction. The next iterate in the Newton process is the Newton point x_N^k, so that

$$x^{k+1} := x^k + d^k.$$

Under certain assumptions, the sequence $\{x^k\}$ can be shown to converge to a solution x^* of NLE. Typical of this type of result is the domain of attraction result found in (Ortega & Rheinboldt 1970, Theorem 10.2.2), which shows quadratic convergence of $\{x^k\}$ to x^* in a neighborhood of x^*, assuming Lipschitz continuity of $F'(x)$ near x^* and nonsingularity of $F'(x^*)$. Another result is the Newton-Kantorovich theorem (Ortega & Rheinboldt 1970, Theorem 12.6.2), which shows the existence of and convergence to a point x^*, a zero of F, given the Lipschitz continuity of F' and the existence of a point x^0 for which both $\|F'(x^0)^{-1}\|$

and $\|F'(x^0)^{-1}F(x^0)\|$ are sufficiently small. This is a strong result, since it does not assume the existence of the solution *a priori*.

The generalized equation (GE) of Robinson (1979) is a zero-finding problem for a set-valued mapping defined in terms of F and the normal cone to the closed convex set C, i.e. find z such that

$$0 \in F(z) + N_C(z). \tag{GE}$$

Note that when $C \equiv \mathbb{R}^n$, $N_C(z) = \{0\}$ for all $z \in \mathbb{R}^n$, so that GE reduces to NLE in this case. In general, however, $N_C(z)$ will not be a singleton. Note also that GE is an equivalent formulation of VI, expressed in terms of the normal cone. Josephy (1979*b*) describes a Newton method for GE in which the linearizations obtained by replacing F with A_k are solved to obtain the successive iterates. Under an assumption of strong regularity (Robinson 1980) at a solution z and an assumption regarding the Lipschitz continuity of F', Josephy (1979*b*) shows that a domain of attraction result holds for Newton's method for GE, and that quadratic convergence is achieved. A Newton-Kantorovich result is shown to hold under assumptions on the initial point x^0 similar to those mentioned earlier.

While the convergence results for the basic Newton method indicate that fast convergence may be expected in the neighborhood of a solution, this neighborhood may be very small. Thus, convergence to a solution depends on the choice of initial iterate x^0. In order to reduce or eliminate this dependency on x^0, globalization methods of either the trust region or linesearch damping type are used (Fletcher 1987). Linesearch damping (Armijo 1966, Goldstein 1967) was originally proposed in the context of the unconstrained minimization of a function $f : \mathbb{R}^n \mapsto \mathbb{R}$. Given d, a direction of descent for the function f at a point x^k, a linesearch is applied in order to find a steplength λ such that $f(x^k + \lambda d) < f(x^k)$. Under appropriate conditions on the descent direction d and the choice of steplength λ, convergence of the iterates to a minimizer of f can be shown. In a linesearch damped Newton method for NLE, the function to be minimized is often chosen to be $\|F(x)\|_2^2$. The Newton direction d^k, a descent direction for $\|F(x)\|_2^2$, is searched for a point that reduces $\|F(x)\|_2^2$.

Motivated by the success of the damped Newton method for NLE, Ralph (1994) has proposed a similar method for the solution to the normal map equation (NME) formulation of the complementarity problem defined in Section 1.2. Like the damped method in the smooth case, Ralph's algorithm constructs a sequence of iterates, each one resulting from an approximation obtained from the previous iterate. Instead of searching a line from the current point to the Newton point, a piecewise-linear path connecting these points is searched, resulting in a reduction in $\|F_B\|$. Convergence results for this algorithm similar to those mentioned above are given by Ralph (1994).

Other Newton-type methods for complementarity problems include the B(ouligand)–differentiable equations approach proposed by Pang (1990), in which the B–derivative is substituted for the F(réchet)–derivative in approximating a function H. When

$$H(x) := \min(x, F(x)),$$

$0 = H(x)$ if and only if x solves NCP, so that a B–Newton method for finding a zero of F solves NCP. The B–Newton direction can be linesearched, so that the method can be shown to be globally convergent. However, a required assumption for both local and global convergence is the F–differentiability of H at the solution point. Robinson (1993) has shown that for a class of nonsmooth functions for which a *point-based approximation* exists (a class which includes the normal map F_B), a Newton method can be applied. Convergence is shown without assuming the F–differentiability of the function at the solution point. A computational study of B–Newton's method was performed by Harker & Xiao (1990), comparing B-DIFF, an implementation of B–Newton's method for NCP, to Josephy-Newton's method on a number of nonlinear complementarity problems. A method proposed for minimizing $\|H\|$ is the NE/SQP method of Pang & Gabriel (1993), in which a quadratic programming problem is used to construct a descent direction for $\|H\|$. The sequence of QP's solved leads to a sequence of iterates which can be shown to converge to a zero of the nonsmooth equation H.

The solution methods for NCP mentioned thus far all involve a reformulation of the problem as a system of nonsmooth equations. Other solution methods based on a reformulation as a smooth minimization problem have also been explored. These methods all involve the minimization of a function $\Theta : \mathbb{R}^n \mapsto \mathbb{R}_+$ such that $\Theta(x) = 0$ if and only if x solves NCP. Various functions Θ have been proposed by Mangasarian (1976), by Mangasarian & Solodov (1993), and by Geiger & Kanzow (1994). Computational tests using these formulations have been done by Ferris & Lucidi (1991) and by Geiger & Kanzow (1994).

Recently, Fukushima (1992) has shown how the asymmetric variational inequality can be formulated as a differentiable optimization problem, without any compactness or strong convexity assumptions being made on the feasible set C. His formulation makes use of a penalty term added to the gap function of Hearn (1982) so that the resulting function is bounded. Fukushima gives a formula for the gradient of his modified gap function, and shows that when F' is positive definite for all x, a descent direction for this mapping can be easily computed, without making use of $F'(x)$.

1.4 Modeling Languages

An algebraic modeling language is a tool for expressing a mathematical programming problem in an algebraic notation that is easily understood by both human and computer. Notable examples include the GAMS modeling language (Brooke et al. 1988), first introduced in the late 1970's, and AMPL (Fourer et al. 1993), a more recent entry into the field; many other systems exist. Both GAMS and AMPL come with a book describing the language and a number of models illustrating how the system is used and what is possible. Each is available in student and professional versions on a wide range of platforms and with a growing number of available solvers.

Prior to the development of modeling languages, an optimization problem might be expressed via a number of Fortran routines providing function and gradient evaluation (both objective and constraint), bound information, and the initial point. Once the problem had been correctly specified and debugged, perhaps by a programmer not familiar with the problem being modeled, the code was difficult for others to read and even more difficult to modify. A change in the data of the model could be a time consuming task, requiring the help of the model's programmer, while a change in the model structure might be unthinkable. In addition, the specification syntax varied with the computing environment, so that the cost of moving a model to a different machine or switching to a different solution method could be prohibitive. In short, the low-level problem description was difficult to write, read, modify, and move.

The difficulties described above motivate a number of fundamental concepts underlying the design of a modeling language. Data independence refers to a model being specified independently of the data it uses. This allows a user to look at and change the form of the model independently of the data, and vice versa. The concise algebraic notation used to enter a model makes the job of writing the model simpler and less time-consuming. The derivatives are computed symbolically, resulting in fewer errors, since less code needs to be written. The algebraic notation used is more self-documenting than comparable code in a language such as Fortran or C, while comments can also be included freely in the statement of the model. Finally, the model is specified independently of any solution algorithm or computing platform used to solve it. This solver independence, perhaps the most important feature of a modeling language, allows many solvers to be easily applied to a common model specification, thus both allowing the most efficient solver to be applied and providing a benchmark for comparison between different solution procedures. In addition, a new solution method, when implemented as a solver for a modeling language, can immediately be tested on the many models already formulated, regardless of the platform on which the solver runs or on which the models were originally specified. This solver independence is also acheived in the CUTE system of Bongartz, Conn, Gould & Toint (1993) and the well-known MPS format for linear programs, although these systems do not provide the ease of use and simplicity of model formulation found in a modeling language. In addition, the MPS format suffers from a loss of precision on some systems.

Modeling languages were originally developed to formulate and solve linear programs, but due to their success and popularity were soon modified to permit the formulation of nonlinear and mixed integer programs as well. This development has continued with the extension of the GAMS language to allow the formulation of mixed complementarity problems (Dirkse et al. 1994). GAMS is especially popular in the field of economics, and since many applications of the MCP are found in this field, a complementarity facility in GAMS serves a large number of potential users and provides a convenient access to a large number of real-world applications which might not otherwise be available. In addition, a similar extension to the AMPL modeling language has been developed. Thus, the benefits of using a modeling language now accrue to practitioners formulating their problems in a mixed complementarity format and to those developing algorithms for complementarity problems. Both the extension to AMPL and the extension to GAMS are described and documented in Chapter 2.

A modeling language functions as follows: The model to be solved must first be read in by a compiler. Communication between a modeling language and a solver is done almost entirely by files. A modeling language does not dispense with code for evaluating functions and gradients, etc., but merely automates its formation. Each time a model is solved, a compiler writes files which contain all the problem data and machine-readable code to evaluate the functions determining the model. Typically, a solver uses a software library to read these files, determine the form of problem to be solved, and calculate the functions and gradients at the required points. Thus, each solve requires a number of temporary files to be written to and read from disk. The time required to read the model and read and write the temporary files can represent a significant fraction of total solution time, even though the temporary files are often written in a compact binary format. In addition, it may be difficult or impossible to take advantage of any special structure the problem may possess.

The paragraph above shows that the benefits of using a modeling language are not obtained without cost. The file I/O required for each solve and the generality of the problem statement required by the interface are balanced by the ease and speed of model formulation and modification. In an environment where the time required for model formulation dominates the time required for model solution, a modeling language is a valuable time-saver.

1.5 Chapter Outline

In this introductory chapter, we have defined the MCP and a number of related problems, chiefly the Normal Map Equation. The NME will be used in the development of a Newton method for MCP. After a survey of related work in this area, we have introduced algebraic modeling languages, and briefly discussed their origin, their function, and the benefits involved in their use.

In Chapter 2, we discuss the fundamentals involved in the design of a complementarity interface to an algebraic modeling language. These fundamentals apply to the interfaces for both GAMS and AMPL, so that the sections describing these two interfaces have much in common. This commonality and a number of important differences are discussed in the closing section of this chapter, along with numerical results comparing the performance of the two interface libraries.

A library of complementarity problems written in the GAMS and AMPL modeling languages is presented in Chapter 3. In this chapter, we describe the derivation of some of the more complex models and the parameters these models contain. We also provide a brief tutorial on the syntax used to express these models in GAMS and in AMPL.

Chapter 4 describes the PATH algorithm, a path-following Newton method for the solution of MCP. This algorithm can be viewed as a generalization of a linesearch damped Newton method for smooth equations; our treatment of the PATH algorithm is along these lines. We also give the details of a nonmonotone stabilization technique applied to the underlying Newton algorithm, concluding with a convergence proof.

Chapter 5 contains extensive computational results obtained by solving a large number of problems with a number of different algorithms. Most of the results presented were obtained using the interface and model libraries of Chapters 2 and 3, respectively. A number of general equilibrium models obtained from the GAMS model library and expressed in GAMS/MPSGE format were solved as well. We also present results comparing the PATH solver to algorithms for which computational results have been published.

Chapter 6 concludes this thesis; in it we describe a projected Newton preprocessor for MCP, give some very promising computational results obtained using this technique, and indicate a number of possible extensions to the interface libraries.

Chapter 2

Modeling Language Interfaces

The primary purpose of a modeling language is to aid the modeler in preparing a model *for solution* and to report the results of the solution process to the modeler. The modeling language does this by providing a convenient, portable, algebraic means of expressing the problem at hand. While this formulation is human-readable, it is not so useful (in the algebraic form) to a solver, which requires specific instructions as to how to evaluate the required functions and gradients defining the problem, as well as other problem data. A modeling language usually writes all this information to a file or files in a compact, binary format. An interface exists to interpret these files for a solver and provide the solver with the functions and data these files contain.

In this chapter, we will be concerned with interfaces for hooking a complementarity solver to a modeling language. In Section 2.1, we will indicate what type of information and functionality will be required from such an interface, and the data structures and computation necessary to provide this. Since many modeling languages, including GAMS and AMPL, were not designed to formulate complementarity problems, attaining this functionality is a challenging and nontrivial task. Sections 2.2 and 2.3 describe in detail two interface libraries used to hook up complementarity solvers (including PATH) to GAMS and AMPL, respectively. In Section 2.4, we compare the GAMS and AMPL interface libraries, noting their similarities and differences and discussing the consequences of each.

2.1 Interface Tasks

Since the MCP is defined by a function F and a box $B = [\ell, u]$, minimum requirements for a solver interface are routines to evaluate F and provide B. We also include routines to evaluate J, the Jacobian of F. Since F is nonlinear, techniques for the solution of MCP may depend heavily on the choice of initial iterate z^0; the interface must provide this as well. Finally, the interface must provide the means to report a solution z^* to the modeling language and hence to the modeler. In addition to these minimal requirements, an interface may provide a number of convenience routines, such as a way to pass algorithm parameter values from a model to a solver. In order to report the solver's progress to the modeler, additional routines may be necessary, such as those to write a status or log file, report solution statistics, and provide the names of functions or variables used in the model (so that a solver can report on the variable "price('corn')" rather than "z[156]"). Depending on the modeling language being used, other routines may also be necessary or desirable. It should not be necessary for a solver developer to communicate directly with the modeler; the interface must provide for all the input and output required.

Most algebraic modeling languages, including GAMS and AMPL, are designed to express constrained optimization problems and pass them to a solver. As a rule, it is impractical to express the many complex constraints of these models in terms of a *single* function or a *single* vector of variables. Typically, a modeling language allows a number of nonlinear constraints, expressed in terms of a number of named variables, to be specified, and forms one *collective* constraint function based on these component functions. Similarly, the many named variables are combined into one *collective* vector of variables. Thus, a modeling language converts a problem expressed in terms of many named variables and many constraints into a problem expressed in terms of a single constraint function and a single vector of variables. The function F and box B of the MCP must be extracted from this collective constraint function and variable. How this is done is best illustrated by example.

A simple Walrasian equilibrium problem is given by Mathiesen (1987) and has equilibrium conditions

$$0 \le b + Ay - d(p) =: S(p, y) \quad \perp \quad p \ge 0 \tag{2.1a}$$

$$0 \le -A^\top p =: L(p) \quad \perp \quad y \ge 0, \tag{2.1b}$$

where the demand function $d(\cdot)$ is defined by

$$d_i(p) \;:=\; \frac{a_i \;\sum_k b_k p_k}{p_i},$$

$p \in \mathbb{R}^m$, $y \in \mathbb{R}^n$, and the data a, b and A are given. The vectors p and y represent prices of goods and levels of production activity, while the functions S and L represent excess supply of goods and loss per unit activity level, respectively. Two different yet equivalent ways of expressing this problem as an MCP are to define

$$F(y,p) := \begin{bmatrix} L(p) \\ S(y,p) \end{bmatrix}, \qquad B := \mathbb{R}^n_+ \times \mathbb{R}^m_+, \tag{2.2}$$

or, equivalently, to define

$$F(p,y) := \begin{bmatrix} S(y,p) \\ L(p) \end{bmatrix}, \qquad B := \mathbb{R}^m_+ \times \mathbb{R}^n_+\,. \tag{2.3}$$

In either case, the functions $S(y,p)$ and $L(p)$ combine to form F, while the variables y and p, together with their bounds, combine to form z and B. The order in which these components are combined in the collective function and variable may depend upon the order in which they are declared in the model, in which case it would be possible to declare p, y, L, and S in such a way that the functions and variables are combined as in (2.2) or (2.3). An interface *might* depend on the modeler to do exactly that.

However, this approach lacks flexibility, is prone to error, and does not allow the interface to perform more than a simple check for model consistency. Models specified in this manner would be difficult to read and modify. In addition, this approach assumes that the modeling language provides the constraints and variables to the solver in the order in which they are specified by the modeler. This will not always be the case, as some languages (notably AMPL) may provide constraints and variables in a different order from that in which they are specified. Therefore, the above approach is not used. Rather, the modeler defines the component functions of the model (using the constraint syntax), the variables used together with their bounds, and a list of function-variable pairs. Given the collective constraint function, the collective variables are permuted so that each component function is complementary to the variable with which it is paired. This allows the modeler to explicitly define the complementarity relationship desired, independent of the order in which the functions and variables are defined in the model. The list of pairs allows the interface to check the consistency of the model described by the modeler, a very useful function for more complex models consisting of many pairs. Thus, the Walrasian model described above would be specified by the pairs $\langle S.p \rangle$ and $\langle L.y \rangle$, assuming S and L are defined as in (2.1).

The indexing required to permute the collective constraint function and collective variable (the rows and columns) serves a dual purpose. When variable components are fixed (i.e. $\ell_i = u_i$), they can be removed from the vector z, along with their associated equation F_i. The indexing required for this is already in place, so the interface can perform this task with little additional overhead, and in conjunction with the consistency check and the permutation of rows and columns. This process occurs only once per problem, while the work files are being read. In what follows, we will assume that the removal and addition of fixed variables takes place while the rows and columns are being permuted.

Once a problem has been read in, the interface is ready to accept requests from the solver for evaluations of F and J, the bounds ℓ and u, and the initial point z^0. To get z^0, the interface calls a routine to get the initial values of the collective variable. These values are then permuted and returned to the solver. The bounds ℓ and u are obtained in a similar fashion. To compute F, the level values supplied by the solver are permuted by the interface into the order in which they appear in the collective constraint function. These values are then passed as input to a routine which uses the instructions in the work file to evaluate the constraint function. This routine must understand the binary format of the work file, and should be supplied with the modeling language being used. The function value returned by this routine is then permuted by the interface and returned to the solver. A similar process is used to compute the Jacobian J. However, it may be necessary to permute both the rows and columns of the constraint Jacobian, since the variable z in the MCP is a permuted version of the collective variable determining the constraint Jacobian.

Once the solver has computed a solution, it calls an interface routine to report the solution z^* and the function value $F(z^*)$ to the modeler. The solution data is written to a work file in much the same way as the initial point is read in. In addition, the solver can send a message or set some status variables to indicate why the solver has terminated (solution found, iteration or resource limit exceeded, error, etc). How or if this is done depends on the modeling language being used. Routines for writing solution status files or log files, performing other kinds of file I/O, and the convenience routines for reading and writing parameter values and setting algorithm tolerances and parameters also vary greatly between interfaces. The particular form these routines take will be described in the sections dealing with the interface libraries for the GAMS and AMPL modeling languages.

2.2 GAMS/MCP

The GAMS modeling language has recently been extended to enable the formulation of MCP in a format known as GAMS/MCP (Rutherford 1994*b*). The GAMS Callable Program Library (CPLIB) is an interface designed to simplify and speed the process of hooking up complementarity solvers for use as GAMS/MCP solution subsystems. CPLIB is a set of Fortran routines that use the GAMS I/O library (Kalvelagen 1992) to read and write the binary instruction files used to communicate between GAMS and a solver. The I/O library also contains routines to evaluate the constraint function and its gradient, using the instructions found in the instruction files. Designed for use in a Fortran environment, CPLIB provides a dynamic memory allocation feature that is useful when hooking up a Fortran solver.

The relationship between CPLIB, the GAMS I/O library, and the routines written by an algorithm developer is presented in Figure 2. The developer-written routines are indicated by dashed boxes. When a CPLIB solver begins execution, control lies with the developer-written Fortran `MAIN` routine, whose only purpose is to call the `cpmain` subroutine, a part of CPLIB. The `cpmain` routine, which has no arguments, is the top-level CPLIB routine, from which calls to the GAMS I/O library, CPLIB, and the developer-written routines (`corerq` and `solver`) are made. `cpmain` first calls routines from the I/O library in order to estimate the size of the problem to be solved. It then calls the developer-written `corerq`, which should contain code to specify the solver type and the amount of memory required by the solver. The amount of memory required is calculated in the `corerq` routine, and is based on the estimates of problem size previously obtained. These will always be *over*estimates. Although a more accurate determination of problem size is made later, this cannot be done until the problem data is read in by CPLIB. Unfortunately, this cannot take place until memory is allocated for CPLIB. Since the GAMS I/O library allows the dynamic allocation of *only one* memory block, the memory for CPLIB and the solver must be allocated together. Hence, the actual problem size in not available at the time that a request for solver memory has to be made in `corerq`.

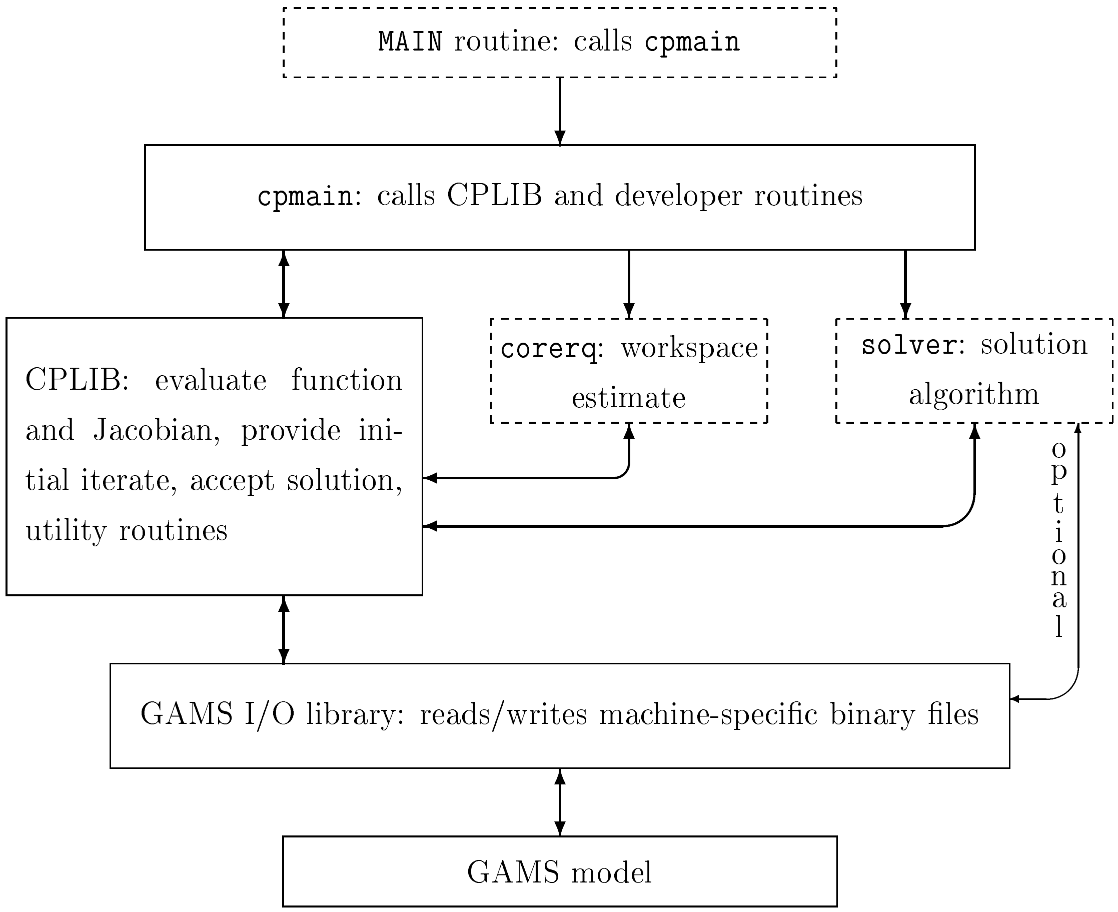


Figure 2: Interrelationship of Developer Code and CPLIB

Once the request for solver memory has been made, the total memory requirement (for both solver and CPLIB) can be computed and a large block of memory allocated. At this point, CPLIB reads in the instruction files (using I/O library routines) and formulates the function F and box B defining the MCP. The data structures used to convert between F and z and the collective function and variable computed by the I/O library are set up at this time as well.

After CPLIB has formulated the problem and is ready to evaluate the function F, the developer-written `solver` routine is called. This routine performs the work of problem solution. To do so, it can call CPLIB routines to:

i) obtain values of machine-dependent parameters,

ii) obtain the variable bounds ℓ and u, along with an initial iterate z^0,

iii) evaluate the function F and its Jacobian, and

iv) return the computed solution, or an indication of why a solution has not been found.

Once the solution has been found or the reason for failure has been reported, the `solver` routine returns control to `cpmain`, which must close the instruction files before the solution process terminates.

In order to simplify development and maintenance of CPLIB, parameter values are generally not passed as subroutine arguments. Instead, communication between developer code and the library takes place through calls to the "scalar interrogation" and "scalar return" routines `cpget*` and `cpput*` described in Section 2.2.2. By keeping the use of argument lists to a minimum, it is easier to provide backward compatibility in future revisions of the library.

A sample solver coded in Fortran and illustrating the use of the CPLIB routines described in Section 2.2.2 is available for anonymous ftp at `ftp.cs.wisc.edu` in directory `/math-prog/solvers/pg_sample_f/`. A subset of the code from this solver is included in Figure 3. In the next two sections we provide more information regarding the developer-written subroutines and CPLIB routines, respectively.

```
program pgrad
call cpmain
end

subroutine corerq
call cpgeti ('N',n)              ! size of problem
call cpgeti ('NADIM',nnz)        ! nonzeros in Jacobian
nwucor = 4 * n + 0 * nnz         ! we won't use Jacobian
call cpputi ('ISTYPE',2)         ! solves general MCP's
call cpputi ('NWUCOR', nwucor)  ! solvers memory requirement
return

subroutine solver (work, nwucor)
call cpgeti ('N',n)
call projgrad (n, work(1), work(n+1), work(2*n+1), work(3*n+1))
return

subroutine projgrad (n, bl, bu, z, F)
...
call cpbnds (z, bl, bu, n)
call cpfunf (z, F, n)
...
call cpputi ('MODSTA', 11)     ! model not solved
call cpputi ('SOLSTA', 2)      ! iteration interrupt
return
```

Figure 3: Sample Fortran Solver Code - GAMS Link

2.2.1 Developer-Written Subroutines

Of the three developer-written subroutines required by CPLIB, the main program is trivial, and must only call `cpmain`. In writing the other two, it is good policy to avoid introducing external symbols beginning with the letters `cp` or `gf`, since names of the CPLIB and GAMS I/O library routines begin with these pairs of letters.

Requesting Memory: `corerq`

The `corerq` subroutine has no arguments. It is used to communicate to CPLIB the capabilities of the solver and its workspace needs. It does this by calls to the routine `cpputi`, which sets integer parameters associated with string constants. If a solver can only process problems with unbounded variables (i.e. systems of nonlinear equations), `corerq` must set the `"istype"` parameter to 1. Otherwise, any other value may be used. To request memory, the `"nwucor"` parameter must be set to the number of "words" (double-precision real equivalents) required by the `solver` routine. This amount can be computed by calls to `cpgeti` using the strings `"n"`, `"nadim"`, and `"intw"` (see Table 2).

Problem Solution: `solver`

```
subroutine solver(work, nwucor)
integer nwucor
double precision work(nwucor)
```

`nwucor`	input	number of words (double-precision reals) of memory requested by the solver in the previous call to `corerq`
`work`	input	workspace array of `nwucor` words

The `solver` routine is responsible for solving the MCP at hand. Typically, it acts as an interface to another routine used to solve the problem. If properly coded, the solution routine called by `solver` will differ only slightly from a standalone version of the same routine. For example, much of the same code is used in the GAMS, AMPL, and standalone versions of the PATH solver. Conditional compilation and different calling routines (`solver` in the GAMS version) account for the only differences in the versions of the PATH solver. For a Fortran solver using dynamically allocated memory, the `solver` routine is typically used to partition a large block of memory and pass this memory on to a subroutine as a number of smaller, separate arrays.

In order to solve a problem, the `solver` routine and its subsidiaries will call CPLIB routines for problem data. These routines are described in the next section.

2.2.2 CPLIB Subroutines

Variable Bounds and Level Values: `cpbnds`

```
subroutine cpbnds(z, bl, bu, n)
integer n
double precision z(n), bl(n), bu(n)
```

`z`	output	initial values of the problem variables
`bl`	output	lower bounds
`bu`	output	upper bounds
`n`	input	problem dimension.

CPLIB passes three values for each variable to the solver: the initial level value, the lower bound, and the upper bound. The bit patterns used to represent plus and minus infinity in `bl` and `bu` should be obtained via calls to `cpgetd`, using the strings `"plinfy"` and `"mninfy"`.

Abnormal Interrupt: `cppunt`

Normally, the `solver` routine will process a problem and return control to `cpmain`. However, a good solver will include checks for errors in the data and in programming, especially when under development. When these errors occur, the solver may wish to terminate the program immediately. Instead of using a Fortran `stop` statement or calling the `exit()` routine, the solver should call the `cppunt` routine. This routine, which has no arguments, sets status indicators used to report the result of the solution process to the GAMS modeler, as well as making other arrangements for a graceful exit.

Function and Jacobian: `cpfunf`, `cpsprj`

```
subroutine cpfunf (z,F,n)
integer n
double precision z(n), F(n)

subroutine cpsprj(z, F, J, Jrow, Jcol, Jlen, n, nadim)
```

```
integer n, nadim, Jrow(nadim), Jcol(n), Jlen(n)
double precision z(n), F(n), J(nadim)
```

`z`	input	point at which to evaluate F and J
`F`	output	value of F evaluated at `z`
`J`	output	nonzero coefficients of the Jacobian evaluated at `z`
`Jrow`	output	row indices of the coefficients stored in `J`
`Jcol`	output	pointers to columns starts in `J`
`Jlen`	output	lengths of the columns in `J`
`n`	input	problem dimension.
`nadim`	input	number of nonzero components in `J`.

The subroutine `cpfunf` evaluates the nonlinear function F at a given point without evaluating the Jacobian J. The `cpsprj` routine evaluates the function F and its Jacobian J, the matrix of first partial derivatives of F with respect to its arguments. The Jacobian is returned in the well-known row index, column pointer, column length format. The coefficients for the nonzero entries of the k'th column of J are stored in the vector `J`, in positions `Jcol(k)`, `Jcol(k)+1`, ..., `Jcol(k)+Jlen(k)-1`. The row indices for these coefficients are stored in the corresponding positions of `Jrow`.

Reporting Solution: `cpsoln`

```
subroutine cpsoln (z, n)
integer n
double precision z(n)
```

`z`	input	solution estimate at solver termination
`n`	input	problem dimension.

Before the `solver` routine returns control to `cpmain`, it may explicitly return the computed solution z^* to CPLIB. Use of the `cpsoln` routine is optional. CPLIB keeps track of the best values encountered during the course of solution, and writes these to disk if `cpsoln` is not used.

Scalar Interrogation: `cpgetd`, `cpgeti`, `cpgetl`

```
subroutine cpgetd (name, dparam)
character*(*) name
double precision dparam

subroutine cpgeti (name, iparam)
character*(*) name
integer iparam

subroutine cpgetl (name, lparam)
character*(*) name
integer lparam
```

`name`	input	the name of the parameter to be returned
`dparam`	output	real parameter returned
`iparam`	output	integer parameter returned
`lparam`	output	logical parameter returned.

These routines, which "get" parameter values from CPLIB, return double precision (real), integer and logical parameters, respectively. Character string identifiers for which these subroutines produce useful values are listed in Tables 1, 2, and 3, along with definitions of the results.

Scalar Return: `cpputd`, `cpputi`

```
subroutine cpputd (name, dparam)
character*(*) name
double precision dparam

subroutine cpputi (name, iparam)
character*(*) name
double precision iparam
```

`name`	input	the name of the parameter to be passed
`dparam`	input	real parameter passed
`iparam`	input	integer parameter passed.

These routines pass double precision (real) and integer values, respectively, to the CPLIB library. Character string identifiers used as input to these routines are listed in Tables 4 and 5, along with definitions of the results.

Table 1: `CPGETD` Arguments

String	Result returned in double precision argument
“clock”	Current elapsed time (for checking resource limit “reslim”).
“eps”	The smallest positive number that can be added to 1.0 to obtain a result different from 1.0.
“huge”	The largest positive number representable on the machine.
“maxexp”	The largest positive decimal exponent representable on the machine.
“minexp”	The largest negative decimal exponent representable on the machine.
“mninfy”	Value currently used for $-\infty$.
“obj’	Merit function associated with the most recent function evaluation.
“plinfy”	Value currently used for $+\infty$.
“precis”	The number of significant decimal digits.
“real1” – “real5”	Five real values can be set in a user’s GAMS program using option statements of the form: `option real3 = 0.1;` these should be used only during solver development.
“reslim”	The resource limit in CPU seconds.
“tiny”	The smallest positive number representable on the machine.

Table 2: `CPGETI` Arguments

String	Result returned in integer argument
"domerr"	Number of domain errors encountered
"domlim"	Maximum number of domain errors allowed before the I/O library terminates execution
"integer1" – 'integer5"	Five integer values can be set in a user's GAMS program using option statements of the form: `option integer3 = 525;` these should be used only during solver development.
"intw"	The number of integers per "word" (1 word = 1 double precision real)
"iolog"	The unit number of the log file
"ioopt"	The unit number of the options file (cf. `cpgetl("useopt")`)
"iosta"	The unit number of the status file
"iterlim"	An iteration limit set via the GAMS iterlim option; default = 1000
"maxcol"	The maximum number of nonzeros in any column of the matrix
"n"	The number of equations / variables in the MCP
"nadim"	estimated number of nonzeros in the Jacobian of F.
"screen"	The unit number of the screen.

Table 3: `cpgetl` Arguments

String	Result returned in logical argument
"useopt"	If true, the solver should attempt to read the user's options file, whose format and syntax are solver-defined.
"sysout"	If true, GAMS will copy the complete status file to the listing file.

Table 4: `cpputd` arguments

String	Description of associated value
"contol"	Convergence tolerance – used to identify infeasible equations in the solution listing.

Table 5: String arguments to subroutine `cpputi`

String	Description of associated value
"istype"	Indicator of solution algorithm capability (passed from `corerq`): 1 nonlinear equations ($l = -\infty,\ u = +\infty$) 2 general MCP ($-\infty \leq l \leq u \leq +\infty$)
"itsusd"	The number of iterations used by the solver. If not set, this records the number of function/derivative evaluations.
"modsta"	Model status indicator. Values relevant to MCP models are: 1 model solved 7 intermediate nonoptimal 13 error - no solution (GAMS triggers a SYSOUT)
"nwucor"	Words of memory requested for solver (passed from `corerq`)
"solsta"	Solver status indicator. Values relevant to MCP algorithms are: 1 normal completion 2 iteration interrupt 3 resource interrupt 4 terminated by solver (GAMS triggers a SYSOUT) 5 evaluation error limit 11 internal solver error
"startc"	Start copying status file output to the listing file
"stopc"	Stop copying status file output to the listing file

2.2.3 Communication and Control

There are a number of conventions that a proper GAMS solution subsystem is expected to follow. These conventions exist so that the GAMS user may better control the behavior of the solver and be informed of its progress.

To inform the user of its progress, the solver writes to the status file, the log file, and the screen. The unit numbers for these files are obtained through calls to `cpgeti`. The status file contains two classes of information – that which is always copied to the GAMS listing (`.lst`) file and that which is copied to the listing file only when the GAMS user specifies the option `sysout = on`. The first type of output is identified by first calling `cpputi` with the string `"startc"` (the integer argument is ignored). Subsequent solver output to the status file will then appear in the GAMS listing. To stop copying to the listing file, call `cpputi` with the string `"stopc"` (again, the integer argument is ignored).

The log file and the screen are typically the same unit. On interactive platforms, the solver may send messages to the log file to indicate progress towards a solution. This can be particularly reassuring when the solution process progresses slowly. It is possible, however, for the user to redirect the log file. (A user might do this when operating over a slow phone line, or in order to save the log file output for examination later.) Only when information (such as a copyright notice) is always to be displayed on the screen should the screen unit be used.

A GAMS user can control the behavior of a solution system in two ways, through GAMS options and an options file. The GAMS `iterlim` and `reslim` options can be set in a GAMS model and passed to a solver via `cpgeti` and `cpgetd` calls, respectively. It is a GAMS convention that for algorithms with major and minor iterations, `iterlim` refers to the cumulative minor iterations performed. Other types of solvers will interpret this limit differently. The default value is 1000. The real value corresponding to `reslim` is a resource (time) limit, in seconds, requested by the user. This defaults to 1000 as well. It is up to the solver developer to see that these limits are adhered to. Note that the `"solsta"` indicator should be set (via `cpputi`) to indicate when these limits have been exceeded (see Table 5).

All algorithms have a number of controlling parameters which can be adjusted to affect performance and tune for particular problems. Nondefault settings for these are best specified in an options file whose form and content depend on the algorithm developer. Examples of options file formats are the SPEC file of MINOS 5.0 (Murtagh & Saunders 1983) and the keyword-value syntax of MINOS 5.4, Lancelot (Conn, Gould & Toint 1992), and PATH (Dirkse & Ferris 1994). The GAMS user can specify whether an options file is to be used by setting the `optfile` option. This logical value is passed to the solver via a call to `cpgetl`, using the `"optfil"` string. Depending on what language the solver is coded in, CPLIB can be requested to open the options file and return its unit number to the solver or to return only the name of the options file. The former is done using the `cpgeti` routine with `"ioopt"`, while the latter is done using the C routine `c_gfopti`.

Once the solver has been compiled into an executable format, GAMS must be made aware of it, and arrangements must be made to call the executable properly. Details of how this is done on a PC running DOS are discussed in (Dirkse et al. 1994). The details for a UNIX installation are quite similar.

2.2.4 The C Interface

On many platforms, the Fortran CPLIB can be used in conjunction with solvers written in the C programming language. While the tasks of linking a Fortran solver and a C solver are quite similar, there are some important differences. Because of these differences, we have written a set of C routines which act as an interface to the routines in CPLIB. These C routines allow the writer of a C solver to ignore many of the (perhaps platform-specific) cross-language issues he or she would otherwise have to consider in making direct calls to Fortran CPLIB subroutines; instead, a C routine is called, which performs the dirty work. In this section, we discuss the issues involved in the design of such routines, and indicate how these routines can best be used.

There are a number of standard conventions used in calling Fortran routines from C and vice versa. Perhaps most importantly, Fortran arguments are "called-by-reference" (pointers to data are passed, not the actual data values), while C passes by value. Of course, arrays are stored column-major in Fortran, but row-major in C. Also, on some systems, a Fortran subroutine named `foo` gets an underscore appended to its name before being passed to the loader, so a C call to `subroutine foo` must actually call `foo_`. Case is significant in the C code, while Fortran names are all generally converted to lower case. Calls from C to CPLIB which use only integer and real arguments can be made easily, and in a portable manner, by keeping these conventions in mind. An extra interface layer in these cases is not necessary.

While passing numeric arguments is simple, the interrogation routines (`cpgeti`, `cpputi`, etc.) in CPLIB require that a character string be passed to a Fortran subroutine. Passing this string from a C routine is a bit more complicated than passing a numeric value; the code necessary to do this may vary from machine to machine. Because of this, we have chosen to write C routines which act as logical replacements for the CPLIB interrogation routines. The details of passing a string from C to Fortran are taken care of in the body of these C routines; the algorithm developer need not be aware of how this is done. In addition to making programming easier, these interface routines serve to isolate much of the code used to make CPLIB calls. This eases the task of porting the C solver to a different architecture, since changes need be made only to the interface routines; the calls to them in the solver remain unchanged.

From a solver writer's perspective, the essential details of the C interface are contained in the header file `c_cplib.h`. The first lines of this file define `BOOLEAN`, `CHAR`, `DREAL`, and `INT` to be the C type declarators for logical, character, floating-point and integer types, respectively. When writing a C solver, it is recommended that these type declarators be used for all variables which will be passed to CPLIB functions or to the C interface. The declarators have been defined to assure correspondence in size and type to the Fortran variables used in CPLIB; their use increases solver portability.

Declarations for routines called from the solver are also included in the header file `c_cplib.h` . The functions `c_cpget*` have a single string pointer argument, and return a value of the appropriate type. The functions `c_cpput*` have two arguments, a string pointer and the value to be put. The `c_print_msg` routine is used to print messages to the various Fortran I/O units opened by CPLIB. Its first (integer) argument is the unit number to print to; its second argument is a pointer to the string to be printed. This string must be null-terminated. Thus, one technique for writing to the CPLIB status and log files from a C solver is to use `sprintf` to write to a message buffer, and to pass a pointer to this buffer to the `c_print_msg` routine. This is the technique used in a sample solver written in C and available via anonymous ftp at `ftp.cs.wisc.edu` in directory `/math-prog/solvers/pg_sample_c/`. The remaining calls to CPLIB routines (`cpbnds`, `cpfunf`, etc.) are made without an interface. In making these calls, care must be taken to observe the conventions described above. C-type declarations for the CPLIB routines are included in `c_cplib.h` to aid in error detection. A subset of the code from the sample C solver is included in Figure 4.

```
#include "c_cplib.h"
DREAL *z, *lower, *upper, *F;

void corerq_ (void)
{
  c_cpputi ("nwucor", 1);       /* one "word", it won't be used */
  c_cpputi ("istype", 2);       /* solve general MCP's */
  return;
}

void solver_ (DREAL *work, INT *nwucor)
{
  INT n;

  n = c_cpgeti ("n");
  projected_gradient (n);
}

void projected_gradient (INT n)
{
  lower = MEMALLOC (DREAL,n); /* similarly for upper, z, F */
  cpbnds_ (z, lower, upper, &n);
  cpfunf_ (z, F, &n);
  ..
  c_cpputi ("modsta", MODEL_NOT_SOLVED);
  c_cpputi ("solsta", SOLU_ITERATION);
  return;
}
```

Figure 4: Sample C Solver Code - GAMS Link

2.3 AMPL/MCP

Like the GAMS modeling language, AMPL was not designed to formulate and solve complementarity problems. However, it was designed in a sufficiently general way so that an MCP can be extracted from a properly specified set of constraints expressed in the AMPL language. The approach we use to specify a complementarity problem in AMPL/MCP differs significantly from that used in GAMS/MCP and CPLIB, and is made possible by the AMPL solver interface library written and made publicly available by Gay (1993).

In a GAMS/MCP model, the component functions are defined as constraints, while the complementarity pattern is defined in the `model` statement, where a list of function-variable pairs is given, as described in Section 2.2. Thus, CPLIB merely permutes the rows and columns of the constraint function, its Jacobian, and the associated variables to arrive at F, J, and z. The AMPL language lacks a model statement, so the above approach cannot be used. Instead, a function and variable are associated via a *pseudo-constraint*. Consider for example the Walrasian equilibrium problem from Section 2.1 (page 18). The corresponding pseudo-constraints can be written down directly from the complementarity conditions (2.1), as follows,

$$p \cdot [S(p,y)] = 0 \tag{2.4a}$$

$$y \cdot [L(p)] = 0, \tag{2.4b}$$

where the variable bounds $\ell_p \le p$, $\ell_y \le y$ are assumed to be specified elsewhere. We refer to (2.4) as a system of pseudo-constraints because they may not hold at a solution to MCP (e.g. $S(p,y) > 0$, $p = \ell_p > 0$). If the pseudo-constraints (2.4) are decomposed into $\begin{bmatrix} p \\ y \end{bmatrix}$ and $\begin{bmatrix} S \\ L \end{bmatrix}$, the complementarity relationship between $z := \begin{bmatrix} p \\ y \end{bmatrix}$ and $F := \begin{bmatrix} S \\ L \end{bmatrix}$, as well as the function F itself, can be recovered. This decomposition is performed by the AMPL MCP interface library.

We mention here a crucial point regarding the presolve step that the AMPL compiler performs when writing a problem to disk in response to a `solve` or `write` command. Since the pseudo-constraints specifying the complementarity pattern and function are not *true* constraints, they are not properly understood or processed by AMPL's pre-solver. Thus, it is imperative that the AMPL compiler skip the presolve stage when generating a model. This is accomplished by the options setting `option presolve 0;`, which can be set at the AMPL command line, or more conveniently, in the default AMPL initialization file. If there are fixed variables in the model, they will be eliminated from the problem by the AMPL MCP library routines, not the AMPL presolve step. In a similar vein, we note that unless an AMPL variable will always be treated as a constant and is not associated with any function via a pseudo-constraint, the AMPL `fix` statement used to fix variables at their current values *should not* be used.

Since both the AMPL MCP and AMPL solver routines are written in C, the need to dynamically allocate memory places no constraints on the organization of the interface library or solver. The only requirements are that the solver calls an initialization routine prior to calling any other AMPL MCP routines, and that a routine to write solution values and/or send a termination message is called to report the solver's progress prior to the solver's termination. The MCP interface library is simple to both use and describe. The organization of an AMPL/MCP solver is illustrated in Figure 5.

The AMPL solver interface library communicates with AMPL by writing and reading files whose names have the form *stub*.*suffix*. AMPL calls its solvers with two command line arguments, the filename *stub* and the string `"-AMPL"`. Unless the solver can be called in a non-AMPL mode, the `"-AMPL"` string can be ignored. The filename *stub* and an indication of what type of Jacobian, sparse or dense, is required is input to the `mcp_init` routine, the first routine called by a typical AMPL/MCP solver.

Interface Initialization: `mcp_init`

```
int mcp_init (char *stub, int do_sparse, int *n, int *nnz);
```

	return	OK or error indication (see `mcp.h`)
`stub`	input	filename *stub* for nonlinear instruction, solution files
`do_sparse`	input	if true, set up to compute a sparse Jacobian; otherwise, set up for a dense Jacobian (you can't do both).
`n`	output	problem dimension
`nnz`	output	number of Jacobian nonzeros.

The return values for the `mcp_init` routine are described in the header file `mcp.h`; this file is part of the AMPL MCP library and should be `#include`d in the solver source code. The

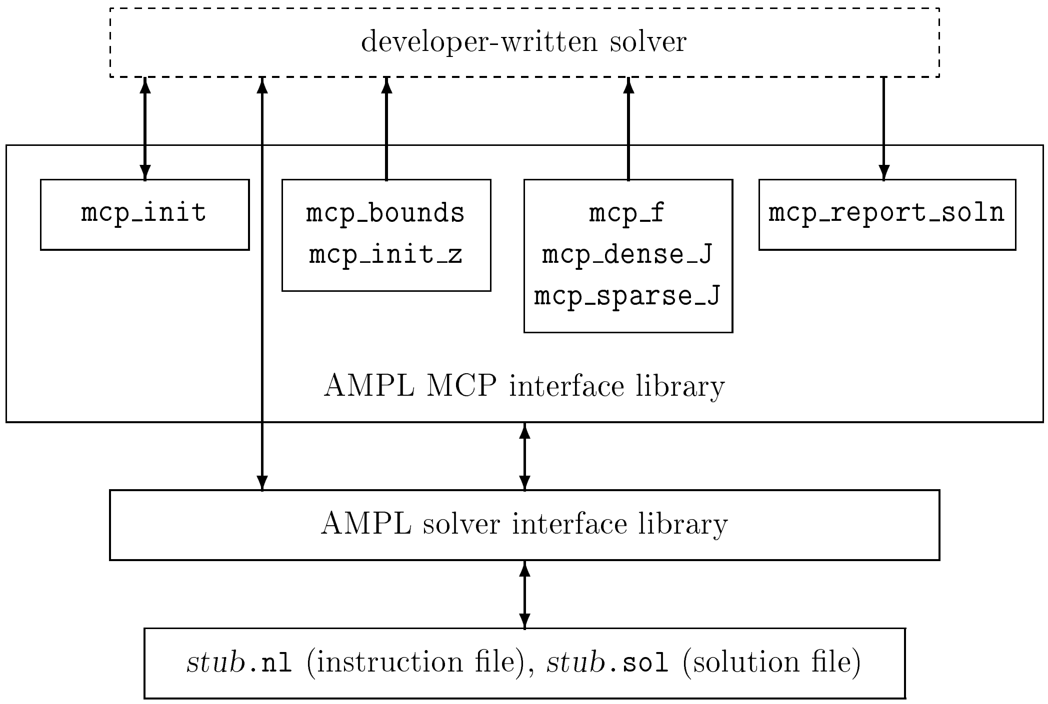


Figure 5: Interrelationship of Developer Code and AMPL MCP library

`mcp_init` routine reads in the problem described in *stub*.`nl` and decomposes the pseudo-constraints it contains, checking that the result will be a valid MCP. In the course of doing so, the `mcp_init` routine sets up the data structures necessary to calculate F, B, and z and returns the size of the problem. In addition, `mcp_init` initializes the global variables `col_len`, `col_start`, and `row_idx`. When a dense Jacobian is requested, these variables are unnecessary and are set to `NULL`. When `do_sparse` is true, they are set to point to integer arrays describing the nonzero structure of J. Before calling the `mcp_init` routine with `do_sparse = true`, the global variable `Fortran` should be set to either 0 or 1. See the discussion following the `mcp_sparse_J` routine for details on `Fortran` and the integer pointers mentioned above.

Once the AMPL MCP library has been initialized, the other library routines can be called in any order. Typically, the routines to read in the bounds and the initial point will be called first.

Variable Bounds and Level Values: `mcp_bounds`, `mcp_init_z`

```
void mcp_bounds (int n, DOUBLE *l, DOUBLE *u);
void mcp_init_z (int n, DOUBLE *z);
```

`n`	input	problem dimension (size of `l`, `u`, and `z`)
`l`	output	lower bound
`u`	output	upper bound
`z`	output	initial iterate.

Function and Jacobian: `mcp_F`, `mcp_dense_J`, `mcp_sparse_J`

```
int mcp_F (int n, DOUBLE *z, DOUBLE *F);
int mcp_dense_J (int n, DOUBLE *z, DOUBLE *J, DOUBLE *F);
int mcp_sparse_J (int n, int nnz, DOUBLE *z, DOUBLE *J, DOUBLE *F);
```

	return	OK or domain error (see `mcp.h`)
`n`	input	problem dimension
`nnz`	input	number of nonzeros in J (size of `J`)
`z`	input	point at which to evaluate the function F
`F`	output	value of F evaluated at `z`.

J	output	value of J evaluated at z; if dense, stored column major in J[0] ... J[n*n-1]; if sparse, only the nonzero coefficients are stored.

In order to remain compatible with solvers coded in C, Fortran, and other languages, only one-dimensional vectors are used in the AMPL MCP library. The mcp_dense_J routine, used to evaluate F and its Jacobian J, returns J in column-major order in a vector of size n^2. To obtain a sparse representation of the Jacobian, the mcp_sparse_J routine is used. The nonzero structure of the Jacobian matrix is determined in function mcp_init and indicated by the three global integer vectors col_len, col_start, and row_idx having length n, n+1, and nnz, respectively. The nonzero elements of the k'th column of J are stored in the vector J in positions col_start[k], col_start[k]+1, ..., col_start[k] + col_len[k] - 1. The row indices for these coefficents are stored in the corresponding positions of row_idx. The extra element in the vector col_start is provided so that all column lengths can be computed using only the col_start vector, via the formula col_len[k] = col_start[k+1] - col_start[k]. Since these global vectors are computed only once, care should be taken not to overwrite their values.

To facilitate the use of both Fortran and C language solvers, the Fortran global variable can be set. When Fortran == 0 (the default), the indices stored in col_start and row_idx assume that array indices begin with 0, so *col_start = 0, etc., and the values stored in row_idx range from 0 to $n-1$. This is appropriate for most C solvers. When Fortran == 1, the array indexing used is suitable for a Fortran solver, i.e., *col_start = 1 and the values stored in row_idx range from 1 to n.

Solver Termination: mcp_report_soln

```
void mcp_report_soln (char *msg, int n, DOUBLE *z, DOUBLE *F);
```

msg	input	termination message for the AMPL user
n	input	problem dimension
z	input	if non-NULL, computed solution
F	input	if non-NULL, value of F evaluated at z.

Before terminating, the solver should send a message to the AMPL user indicating why the solver has terminated, as well as sending the computed solution, if available. If no solution has been computed, NULL pointers should be passed to the mcp_report_soln routine instead of the vectors z and F.

A sample solver coded in C and illustrating the use of some of the library routines described in this section is available for anonymous ftp at `ftp.cs.wisc.edu` in directory `/math-prog/solvers/pg_sample_c/`. A subset of the code from this solver is included in Figure 6.

```
#include "jacdim.h"              /* AMPL interface header */
#include "mcp.h"                 /* AMPL/MCP interface header */

void main (int argc, char **argv)
{
  INT n,                         /* dimension of system to solve */
  nnz;                           /* number of nonzeroes in Jacobian */

  Fortran = 1;                   /* indices start with 1 */
  if ((mcp_init (argv[1], TRUE, &n, &nnz)) != OK) {
    fprintf (stdout, "Routine mcp_init returns error.\n");
    exit (-1);
  }
  projected_gradient (n);        /* this is actually the call to solve */
}

DREAL *z, *bl, *bu, *F;

void projected_gradient (INT n)
{
  bl = MEMALLOC (DREAL,n);       /* similarly for bu, z, F */

  mcp_bounds (n, bl, bu);
  mcp_init_z (n, z);
  mcp_F (n, z, F);
  ...
  mcp_report_soln ("Solution not found: iteration limit", n, NULL, NULL);
  return;
}
```

Figure 6: Sample C Solver Code - AMPL Link

While it is possible to write an AMPL MCP solver using only the MCP library routines, using some of the routines from the AMPL solver library as well may result in a more user-friendly implementation. For example, an AMPL user can specify option values during an

AMPL session that can be passed on to a solver, thus overriding both the default parameter values and those found in the solver-specific options file. These options are passed to the solver in the form of an environment variable named *solver*`_options`. Thus, the AMPL version of the PATH solver first gets parameter values from the options file `path.opt`, and then checks the environment variable `path_options`, using the C `getenv` function. The PATH solver uses the same code to parse both the options file and the options string. The AMPL solver interface library routine `b_search` exists for those not wanting to write their own code for parsing an options string. A description of the `b_search` routine, as well as other useful routines and global variables available to all solvers hooked up as AMPL subsystems, is provided by Gay (1993).

As mentioned earlier, AMPL communicates with its solvers through files of the form *stub.suffix*. The only file required by the AMPL MCP library is the nonlinear instruction file *stub*`.nl`; this file contains the problem description read by the AMPL solver interface library routines. When developing or debugging a solver, the *stub*`.nl` file can be created and saved in an AMPL session by using the `write` command, as described in the AMPL manual (Fourer et al. 1993). The solver can then be called directly from the command line with arguments *stub* and `-AMPL`.

2.4 Interface comparisons

In this section, we compare the MCP interface libraries for the GAMS and AMPL modeling languages. The similarities between these libraries have several implications on the design of software for solving complementarity problems. The differences between these libraries arise largely out of the languages used to code them, and also out of the differences between the two modeling languages used. We conclude this chapter with a performance comparison of the two libraries.

Both GAMS and AMPL exist for the purpose of formulating constrained optimization problems and passing these problems on to a solver. Neither language included the formulation of complementarity problems as a major goal, so that each interface library must first interpret a set of constraints, verify that they specify a valid MCP, and construct this problem before any requests from a solver for problem data can be fulfilled. Each language communicates with its solvers through files, each provides one derivative (the Jacobian) obtained symbolically, and each expects to provide the Jacobian in the same sparse row index, column pointer format.

Both of the interface libraries are designed to be used with a solver written to be called as a subroutine. This solver should make use of dynamic memory allocation, and should also handle matrices stored in the sparse format used by both libraries. Naturally, the evaluation of functions and gradients should be done using subroutines. Finally, input from and output to the user should be confined to as few places as possible, since each modeling language has its own conventions for I/O that must be adhered to.

There are a number of differences between the two libraries. Perhaps the most important difference lies in the programming languages used. The AMPL MCP library is written in C, while CPLIB is written in Fortran. Each library is written so that solvers written in other languages can be used. However, CPLIB allows a solver to request memory, a feature useful only for a Fortran solver, while the AMPL MCP library requires a solver to allocate its own memory. The latter approach has the advantage of separating the memory required for the interface and the memory required for the solver. It also allows a solver to request exactly the amount of memory it requires; recall that the request for memory using CPLIB is made using estimates only. Finally, the AMPL approach prevents the solver from depending too closely on the interface; a solver that allocates its own memory can be decoupled more easily and used in other applications.

The CPLIB interface requires that the GAMS dictionary file be read in when setting up the complementarity problem. This file contains the complementarity relationship specified in the `model` statement and provides indices to the components of the functions and variables named in the model statement. The dictionary file is an ASCII file; for large, sparse models, reading and writing it can consume a significant percentage of solution time. Space must also be reserved for storing the contents of this file in memory. The AMPL approach does not require that similar AMPL files (*stub*`.row`, *stub*`.col`) be written or read, since the complementarity relationship is implied in the pseudo-constraint, which is written and read in a binary rather than an ASCII format.

The different ways of specifying the complementarity relationship have other implications on the interfaces as well. The GAMS model is "overdetermined" in the sense that the bounds on the function are given implicitly by the variable bounds and explicitly by the bounds used to specify the constraints. This can be used as a consistency check, but it can also lead to some confusion as to where the bounds are specified. In the GAMS/MCP system, as in the MCP definition, it is the variable bounds that determine the bounds on the function; the relational operator used to specify the constraint is extraneous. In addition, it is not possible to consistently define a GAMS equation associated with a bounded variable. Although seemingly well suited for this case, the GAMS "no constraint" relational operator (`"=n="`) cannot be used, since it drops the constant terms from the constraint. On the other hand, the pseudo-constraint syntax used in the AMPL model sets no extraneous constraints on the function F defining the MCP; the only bounds the user provides are those explicitly placed on the AMPL variables. While this is more consistent with the problem definition, it does not allow the user to specify the intended sign of F explicitly. Therefore, a consistency check cannot be performed by the interface.

When CPLIB is requested to evaluate a function or Jacobian, it merely permutes the input and output to the GAMS I/O library routines and accommodates the removal of fixed variables. While simple to program, this approach does unnecessary work when fixed variables are present in the model. The AMPL MCP routines typically evaluate only those function and Jacobian components not corresponding to fixed variables, thus avoiding unnecessary work on the part of the interface.

The differences in form and function between the two modeling languages cannot be overlooked. One salient feature of AMPL is the "defined variable" (Chapter 13 of Fourer et al. (1993)), where a variable is defined in terms of other variables and used in turn to define still other variables or to define constraints. In addition to making model writing much simpler and reducing the errors in model formulation, the use of defined variables can lead to significant gains in performance (see Table 6, page 45). When a variable is defined as it is declared, it will be substituted out of the model automatically by the AMPL compiler. When a variable is defined using a constraint declaration, it will only be substituted out of the model if `option substout` is set to 1.

The GAMS `model` statement provides a convenient way to specify the complementarity problem; in addition, variables and functions can be temporarily left out of the model easily by eliminating them from the `model` statement. These functions and variables must be commented out of an AMPL model. The GAMS loop statment is also a useful feature not yet included in the AMPL language.

In order to compare the performance of the two interface libraries, we have tested them on a number of models. In these tests, AMPL and GAMS models for the same problems have been read in and the functions and Jacobians evaluated a number of times. In Table 6, we compare both the times required to initialize the libraries for function and Jacobian evaluation and the average time required to evaluate the function and Jacobian. These two measurements include practically all of the computation done in the MCP interface libraries; the interface time spent doing other work is negligible. What is not measured is the time required for the respective compilers to process the model description and write the work files used to define the model for the solver.

These tests were performed on a DECstation 5000/125, using Version 2 of CPLIB and an experimental version of the AMPL MCP interface library. The AMPL models marked with an asterisk ("*") indicate where defined variables are used. The data in Table 6 represent the average of the results obtained over 5 trials, where in each trial the problem was set up, the function and Jacobian were evaluated 100 times, and the setup and average evaluation times were reported. The results are illustrated in the bar graphs of Figures 7 and 8.

Problem Size	Model		Setup Time	Evaluation Time
14	choi	GAMS	6.2	1.9
		AMPL	4.0	1.0
		AMPL*	1.1	.23
100	ehl_kost	GAMS	8.6	2.2
		AMPL	7.7	1.7
		AMPL*	1.2	.23
300	bratu	GAMS	4.6	.08
		AMPL	.4	.09
300	obstacle	GAMS	4.4	.02
		AMPL	.3	.07

Table 6: Interface Library Execution Times (in seconds)

As illustrated in Figure 7, the AMPL library is able to read in the instruction file and set up the complementarity problem in less time than is taken by the GAMS CPLIB. This

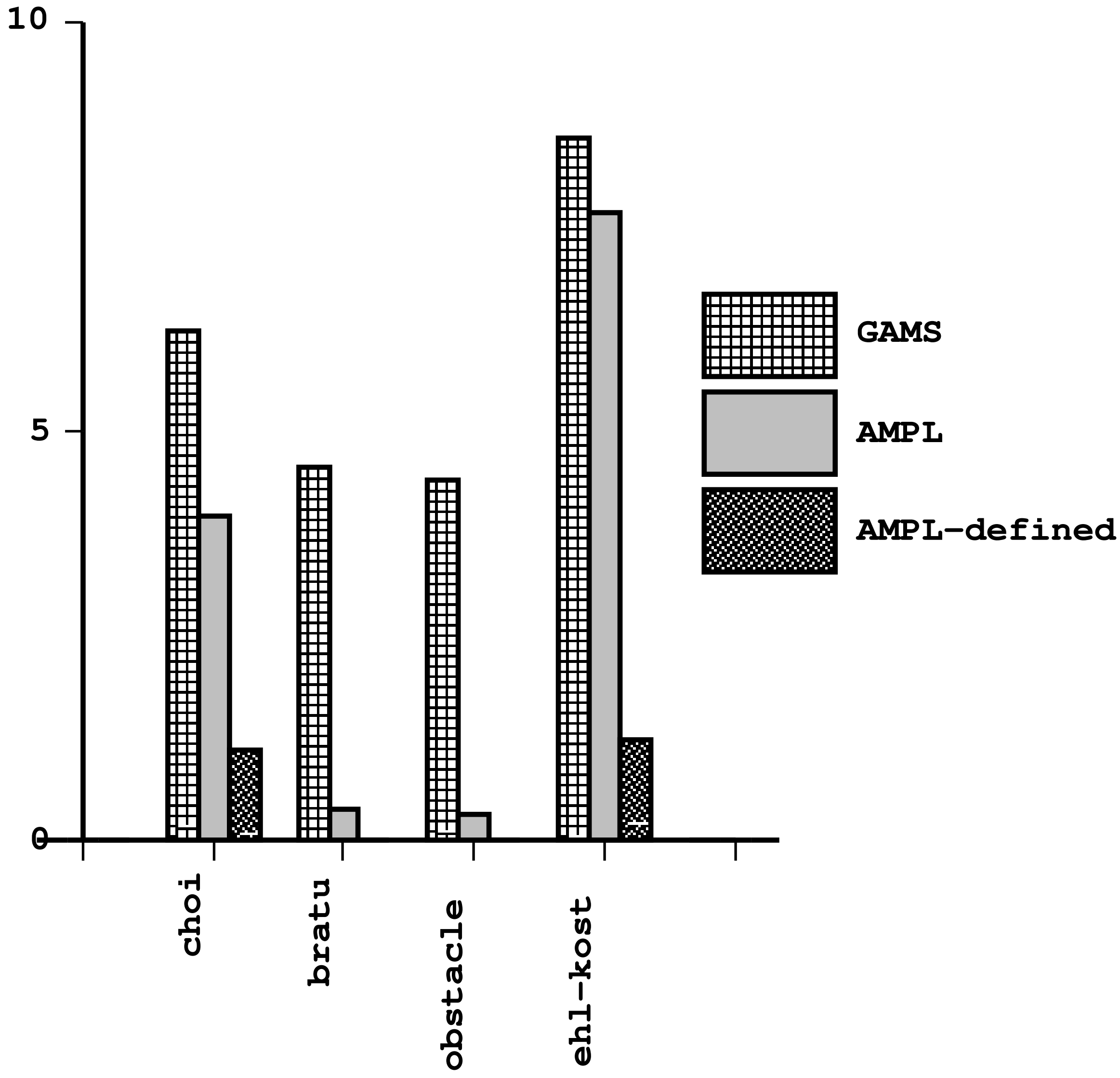


Figure 7: Interface Library Setup Times (in seconds)

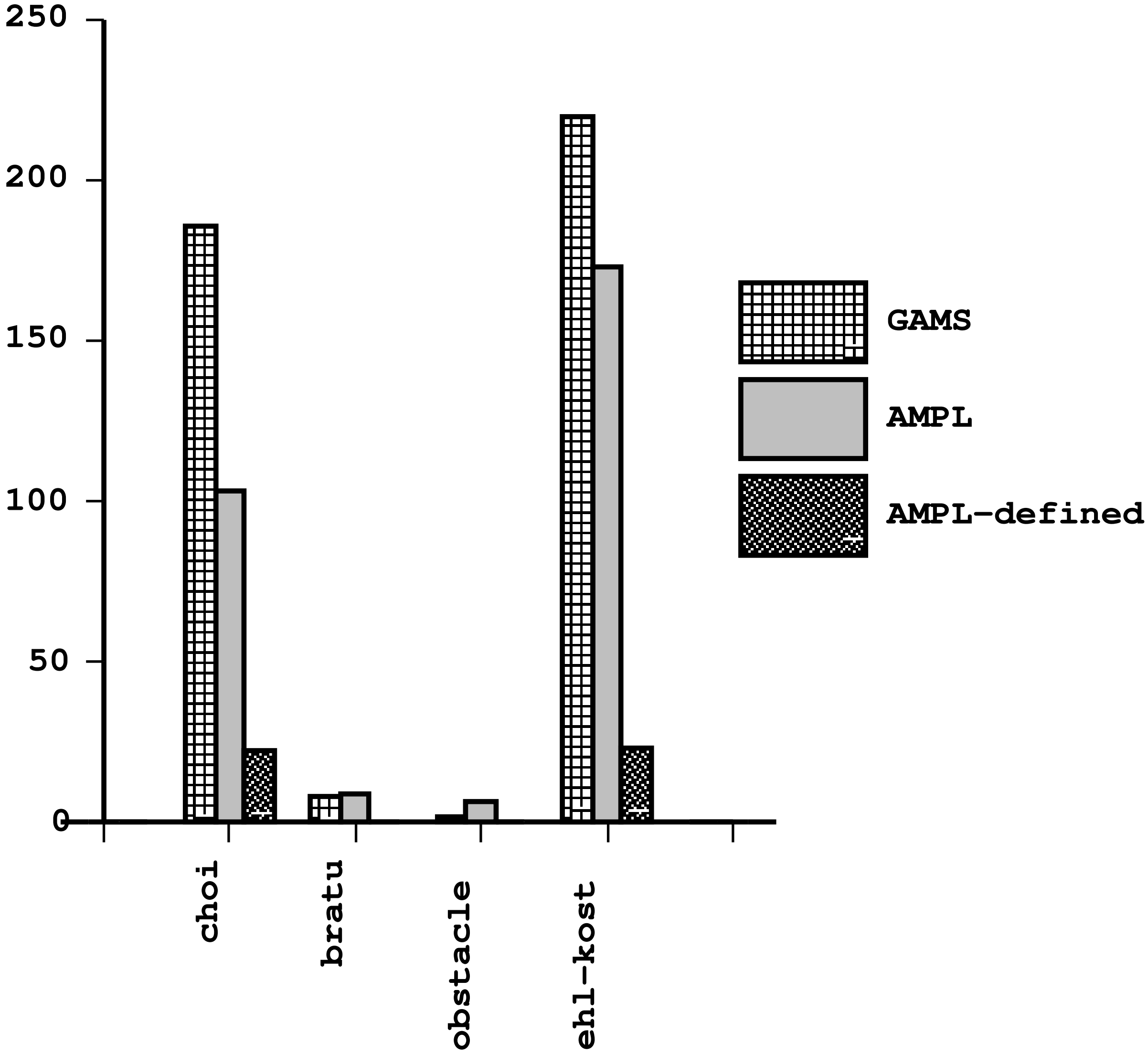


Figure 8: Interface Library Evaluation Times (in seconds)

advantage is increased when defined variables are used in the AMPL model. Also, we can expect the difference in setup times to increase as the models increase in size. However, neither interface library exhibits a clear superiority in function and Jacobian evaluation times. For the more complex models, and for those in which defined variables can be used, the AMPL interface is faster, and is to be preferred to the GAMS interface, while for very simple models (especially the linear one) the GAMS model is able to evaluate the function and Jacobian more quickly. For the large, sparse, simple models, the question of which library requires the least amount of total computing time depends on the number of function and gradient evaluations required. Since a linear problem such as the obstacle problem should require a very small number of function evaluations, the AMPL interface is to be preferred for this problem, and unless a very large number of function evaluations are necessary, the AMPL interface would be preferred for the bratu problem as well.

Chapter 3

MCPLIB: A Model Library

In this chapter, we describe MCPLIB, a library of nonlinear mixed complementarity problems formulated in the GAMS and AMPL modeling languages. The problems defined in this library can be used in conjunction with the interfaces discussed in Chapter 2. Together with the MCP interfaces, the library of test problems provides a uniform basis for testing and comparing currently available MCP algorithms, as well as those under development. The library greatly simplifies the task of thoroughly testing an algorithm on a large number of problems drawn from a number of different fields. The problems in this library will thus serve both as test problems for new algorithms and as a standard of comparison between existing algorithms for solving the MCP.

In addition, the problems in this library also serve as examples of how many different types of problems can be formulated as MCP's, and how these MCP's can be expressed in the GAMS and AMPL languages. The usefulness of the complementarity format is aptly demonstrated by the number and breadth of the problems included. Using these problems as examples, researchers in many areas will be able to more easily formulate their problems, and in a way which gives access to a number of different solution algorithms. It is hoped that this library will act as a catalyst for further use of the complementarity facilities recently added to the GAMS and AMPL languages, thereby providing even more models with which to test and compare solution algorithms.

In Section 3.1, we describe the syntax used to express MCP's in the GAMS and AMPL modeling languages, using the KKT conditions for a quadratic program as an example. Section 3.2 contains descriptions of some of the models in the library, as well as a discussion of their derivation. Included in the library are all the problems attempted by Harker & Xiao (1990), Pang & Gabriel (1993), and Dirkse & Ferris (1994). New and larger problems from extended linear-quadratic programming (Rockafellar 1987) and other areas are included as well. The GAMS and AMPL models for these problems are available via anonymous ftp from `ftp.cs.wisc.edu:~/math-prog/mcplib/`.

3.1 MCP Syntax for GAMS and AMPL

In order to describe the syntax used to formulate MCP's in GAMS and AMPL, we will consider the KKT conditions of a QP in the following form:

$$\begin{array}{ll} \underset{x}{\text{minimize}} & \frac{1}{2}x^\top Qx + c^\top x \\ \text{subject to} & Ax \le b. \end{array} \tag{QP}$$

We will assume that Q is symmetric. (If not, Q can be replaced with $\frac{Q+Q^\top}{2}$.) The Karush–Kuhn–Tucker conditions (Mangasarian 1969) associated with (QP) are as follows:

$$\begin{aligned} 0 = Qx + A^\top u + c \;&\perp\; x \\ 0 \le -Ax + b \;&\perp\; u \ge 0. \end{aligned} \tag{KKT}$$

Although it would be quite simple to express (KKT) as an MCP, we will not do so directly. Instead, we write down the complementary function-variable pairs of (KKT) in the GAMS or AMPL model. These pairs are then used by the appropriate MCP library to construct an MCP, as described in Chapter 2.

The model fragments given in Figures 9 and 10 exclude parameter and set definitions, etc. Those unfamiliar with how these are defined should consult the GAMS (Brooke et al. 1988) or AMPL (Fourer et al. 1993) user's manuals.

In a GAMS/MCP model, the component functions are defined using the GAMS constraint syntax, while the complementarity pattern is defined in the `model` statement by a list of constraint–variable pairs. The bounds on the variables are given using the normal GAMS techniques, as shown in Figure 9.

When formulating an MCP via GAMS, it is important to keep in mind the simple rules GAMS/MCP follows in obtaining functions from constraints. Regardless of the relational operator used (`=e=`, `=l=`, `=g=`, or `=n=`), the function defining the constraint is "normalized"

```
variables     x(J),
              u(I);
u.lo(I) = 0;

equations     dx(J),
              du(I);

dx(J) ..      sum(K, Q(J,K)*x(K)) + sum(I, u(I)*A(I,J)) + c(J)
              =e= 0;
du(I) ..      b(I) =g= sum(J, A(I,J)*x(J));

model qp      / dx.x, du.u /;

solve qp using mcp;
```

Figure 9: Fragment of GAMS/MCP model of KKT conditions for QP

by moving all the terms to the left-hand side of the equation; this normalized function is then used in defining the MCP. Thus, the `du(I)` constraint in the model of Figure 9 could have been written as `-sum(J, A(I,J)*x(J)) + b(I) =g= 0` or as `0 =g= sum(J, A(I,J)*x(J)) - b(I)`, but *not* as `sum(J, A(I,J)*x(J)) =l= b(I)`.

As mentioned in Chapter 2, the permissible sign of a component function is determined solely by the bounds on the associated variable; the relational operator used to define the constraint is extraneous. The extra information the relational operator provides is used as a consistency check. However, the relational operator `=n=` cannot be used, so there is no consistent way to express models containing bounded variables. In this case, another operator must be used, and the consistency check is not performed. Note that in writing down the KKT conditions for QP, we associated the nonnegative dual variables u with the nonnegatively constrained function $b - Ax$; this is consistent with the MCP format. Associating u with the nonpositively constrained function $Ax - b$ is *not* consistent. Care should be taken in writing down KKT or complementarity conditions that are consistent with the MCP. For example, in the GAMS model of Figure 9, the `model` statement associates the unbounded variable x with the equation $dx = 0$ and the nonnegative variable u with the nonnegatively constrained function $du := b - Ax \geq 0$. Similarly, a variable bounded above should be associated with a function whose sign must be nonpositive. The function name must precede the variable name in each pair specified in the `model` statement.

In an AMPL/MCP model, there are no extraneous constraints placed on the functions.

Rather, each variable to appear in the MCP is used in a particular way in a pseudo-constraint that both defines the associated function and links it to the variable used. An AMPL/MCP model for the KKT conditions for QP is given in Figure 10.

```
var {j in 1..N} x;
var {i in 1..M} u >= 0;

dx {j in 1..N}:
            x[j] *
            ( sum {k in 1..N} Q[j,k]*x[k]
            + sum {i in 1..M} u[i]*A[i,j] + c[j] )
            = 0;
du {i in 1..M}:
            u[i] *
            ( b[i] - sum {j in 1..N} A[i,j]*x[j] )
            = 0;

solve;
```

Figure 10: AMPL/MCP model of KKT conditions for QP

The variables included in the AMPL/MCP model are those appearing in the AMPL constraints. Each pseudo-constraint must be of the form

var `* (` *expr*`) = 0;`.

This pseudo-constraint is decomposed by the interface, and results in the variable *var* being associated with the function defined by *expr* in the resulting model.

As when formulating a GAMS/MCP model, it is crucial that the complementarity conditions be consistent with the MCP format, i.e. variables bounded below associated with functions whose sign must be nonnegative, etc. Given such a consistent set of complementarity conditions, the (redundant) bounds on the functions can be deduced from the variable bounds. Thus, the MCP contains no explicit bounds on the function; neither does the AMPL/MCP model. The only information contained in the pseudo-constraints are the variable bounds, the function definitions, and the complementarity pattern; no extraneous bounds can be placed on the functions.

Since the pseudo-constraints do not represent *true* constraints, they may not be correctly interpreted by AMPL's presolve stage. Thus, the presolve option should not be used when formulating AMPL/MCP models; it can be turned off by specifying `option presolve 0;`.

3.2 The Model Library

The AMPL/MCP models included in the model library are given in Table 7, while the GAMS/MCP models are given in Table 8. Due to the more recent development of the AMPL/MCP format and to difficulties encountered with some of the earlier versions of the AMPL compilers, the AMPL library currently contains only a partial list of the models contained in the GAMS library.

Table 7: AMPL/MCP Models

Model origin	AMPL file	Size
Nonlinear programming		
Quadratic programming	qp.mod	4
NLP test problem #2 from Colville	colvncp.mod	15
Dual of Colville problem #2	colvdual.mod	20
Obstacle problems	obstacle.mod	N
Obstacle Bratu problems	bratu.mod	N
Nonlinear complementarity		
	josephy.mod	4
	kojshin.mod	4
Elastohydrodynamic lubrication	ehl_kost.mod	N
Variational inequalities		
Nash equilibrium	nash.mod	10
" "	choi.mod	14
Walrasian equilibrium	mathi*.mod	4
" "	scarfa*.mod	14
" "	scarfb*.mod	40
Traffic assignment	gafni.mod	5
Invariant capital stock	hanskoop.mod	14
Project Independence energy system (PIES)	pies.mod	42
Von Thünen land use	vonthun.mod	186
Extended linear-quadratic programming		
Optimal control	opt_cont.mod	N

3.2.1 Computing a Nash Equilibrium

The problem of computing a Nash equilibrium appears often in the literature. As studied by Murphy, Sherali & Soyster (1982), Harker (1988), and Harker & Xiao (1990), the problem

Table 8: GAMS/MCP Models

Model origin	GAMS file	Size
Nonlinear equations		
Distillation column modeling	hydroc20.gms	99
" "	hydroc06.gms	39
" "	methan08.gms	39
Nonlinear programming		
Quadratic programming	qp.gms	4
NLP test problem #2 from Colville	colvncp.gms	15
Dual of Colville problem #2	colvdual.gms	20
Obstacle problems	obstacle.gms	N
Obstacle Bratu problems	bratu.gms	N
Nonlinear complementarity		
	josephy.gms	4
	kojshin.gms	4
Elastohydrodynamic lubrication	ehl_kost.gms	N
Variational inequalities		
Nash equilibrium	nash.gms	10
" "	choi.gms	14
Spatial price equilibrium	sppe.gms	27
" "	tobin.gms	42
Walrasian equilibrium	mathi*.gms	4
" "	scarfa*.gms	14
" "	scarfb*.gms	40
Traffic assignment	gafni.gms	5
Invariant capital stock	hanskoop.gms	14
Project Independence energy system (PIES)	pies.gms	42
Von Thünen land use	vonthun.gms	186
Extended linear-quadratic programming		
Optimal control	opt_cont.gms	N

concerns a number of firms, each competitively producing a *common* good. We define the following:

N	number of firms, indexed $i = 1, \dots, N$
$x = (x_i)$	production vector; firm i produces a quantity x_i of the good
ξ	$e^\top x$, the sum total of the quantity being produced
$p(\xi)$	inverse demand function; $p(\xi)$ is the unit price at which consumers will demand (and actually purchase) a quantity ξ
$C_i(x_i)$	the production cost for firm i; note that this is the total cost, not a per–unit cost.

The firms comprise a market that we assume evolves over a number of time periods. At the beginning of each period, each firm sets its production level x_i so as to maximize its own profit, under the assumption that the production for all other firms remains constant at some level $x_j^*, j \neq i$. (These firms are said to operate in a *Nash manner*, i.e., they assume the other firms' decisions remain constant.) Intuitively, a Nash equilibrium point x^* is a production pattern in which *no* firm can increase its profit by unilaterally changing its level of production. Since no firm chooses to change its production in the current period, there is no change in the market, hence the equilibrium. Mathematically, a *Nash equilibrium* is a vector x^* such that

$$\forall\, i, \qquad x_i^* \;\in\; \operatorname*{arg\,max}_{x_i \geq 0} \qquad x_i\, p(x_i + \sum_{j \neq i} x_j^*) - C_i(x_i) \tag{3.1}$$

The KKT conditions for (3.1) take the following simple form:

$$\forall\, i, \qquad 0 \leq \nabla C_i(x_i) - p(\xi) - x_i \nabla p(\xi) \quad \perp \quad x_i \geq 0 \tag{NE}$$

which we call the Nash equilibrium conditions. In conformity with generally accepted economic behavior, the inverse demand function p is assumed to be strictly decreasing, the cost function C to be convex, and the "industry revenue curve" $\xi p(\xi)$ to be concave for $\xi \geq 0$. Under these assumptions, the objective function in (3.1) is concave (Murphy et al. 1982). Mangasarian (1969) shows that under these conditions, the Nash equilibrium conditions (NE) are both necessary and sufficient for x^* to maximize (3.1). By combining the Nash equilibrium conditions for each i, we get an NCP in N variables.

The functions p and C used in the problem defined by Murphy et al. (1982) are defined below; c_i, L_i, β_i, and γ are parameters, with $\gamma > 1$.

$$p(\xi) = 5000^{\frac{1}{\gamma}}\, \xi^{\frac{-1}{\gamma}}$$

$$C_i(x_i) = c_i x_i \; + \; \frac{\beta_i}{1+\beta_i} \, L_i^{\frac{1}{\beta_i}} \, x_i^{\frac{\beta_i+1}{\beta_i}}$$

The parameter values are given in the models `nash.gms` and `nash.mod`.

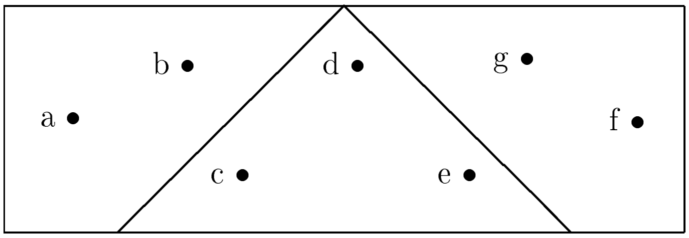
Another Nash equilibrium problem is given by Choi, DeSarbo & Harker (1990). In this problem, the firms are differentiated by the characteristics of the analgesic pain relievers they produce, in addition to their production costs, while demand is determined by the prices and ingredient lists of the pain relievers. Each firm acts by setting its price, rather than its production level. Since the goods produced by the competing firms are not identical, the demand function for each good (and hence, the revenue function for each firm) depends on the prices of each of the other goods; firms continue to act so as to maximize revenue. Data for this problem, and a description of the demand and revenue functions, are given in the files `choi.gms` and `choi.mod`.

3.2.2 A Spatial Price Equilibrium Model

Harker (1986) gives a number of models which describe the spatial and competitive structure of markets embedded in a network (i.e. a set of nodes and the arcs connecting them). Each node represents a unit or site separated spatially from the others. In each model, a *spatial price equilibrium* is sought. One competitive structure modeled is an oligopoly, a market situation in which a few producers control the deliveries to and demands from a large number of buyers. In our example, each producer tries to maximize the profit associated with his production of a single commodity common to all producers. We define the following:

L	set of distinct production units or *sites*
$W \subset L \times L$	set of transportation arcs between the sites in L
Q	set of producers, or *firms*, operating in the market
$I_q \in L$	set of sites controlled by firm $q \in Q$. The set of sites L is partitioned among the sets $I_q, q \in Q$.

Example 6 *Eight sites partitioned among 3 producers.*



$L = \{\mathrm{a}, \ldots, \mathrm{g}\}$
$Q = \{1, 2, 3\}$
$I_1 = \{\mathrm{a}, \mathrm{b}\}$
$I_2 = \{\mathrm{c}, \mathrm{d}, \mathrm{e}\}$
$I_3 = \{\mathrm{f}, \mathrm{g}\}$

$s_l, l \in L$	amount of commodity supplied (produced) by site l
$C_l(s_l)$	total cost of producing s_l units of output at site l (integral of inverse supply function)
$d_l, l \in L$	amount of commodity delivered (demanded) at site l
$\theta_l(d_l)$	purchase price dictated by the delivery to site l (inverse demand function)
$t_{ij}, ij \in W$	flow from site i to site j
$c_{ij}(t_{ij}), ij \in W$	unit transportation cost at level t_{ij}
d_{lq}	amount of commodity *produced by firm* q delivered to site l.

We will assume that each firm q acts in a Nash manner (see Section 3.2.1) when making decisions regarding the following quantities:

$s_i, i \in I_q$	the amounts produced at the sites q controls
$d_{lq}, l \in L$	amount of firm q's production delivered to each site in L
$t_{ij}, i \in I_q, j \in L$	flow from sites under firm q's control to each site in L.

The aggregation of these variables is firm q's strategy vector x_q. The constraints on x_q are those which ensure a conservation of flow at each site. Constraints for sites which firm q controls are more complicated than those for sites outside of firm q's control. The supply, delivery, and transportation variables are subject to lower and upper bounds, which we have taken to be 0 and $+\infty$, respectively. Thus, the set X_q of feasible strategies for the firm q is

$$X_q = \left\{ x_q := \begin{bmatrix} s_i \\ d_{lq} \\ t_{ij} \end{bmatrix} \geq 0 \;\middle|\; \begin{array}{lr} d_{lq} + \sum\limits_{j \in L} t_{lj} = s_l + \sum\limits_{i \in I_q} t_{il} \qquad (\forall\, l \in I_q) & \text{(3.2a)} \\ d_{lq} = \sum\limits_{i \in I_q} t_{il} \qquad (\forall\, l \in L \setminus I_q) & \text{(3.2b)} \end{array} \right\}.$$

Let $X := \prod_{q \in Q} X_q$, so that $x \in X$ is a feasible strategy for all firms. Firm q's profit is then given by the function f_q:

$$f_q(x) \;:= \textstyle\sum_{l \in L} \theta_l(\sum\limits_{j \in L} t_{jl}) d_{lq} - \sum\limits_{i \in I_q} C_i(s_i) - \sum\limits_{i \in I_q} \sum\limits_{j \in L} c_{ij}(t_{ij}) t_{ij}, \tag{3.3}$$

so that firm q wishes to find a strategy x_q which solves the following problem:

$$\begin{array}{ll} \underset{x_q \in X_q}{\text{maximize}} & f_q(x) \\ \text{subject to} & x_p = \bar{x}_p \quad \forall\, p \neq q, \end{array} \tag{3.4}$$

where $\bar{x}_p$ is the current strategy employed by firm p. If we assume that, for all $l, i, j \in L$, $\theta_l(d_l)$ is a decreasing function, $C_l(s_l)$ is a convex function, and $c_{ij}(t_{ij})$ is an increasing function, then f_q is convex. If f_q is defined on the feasible set X and X contains a positive point, then, by applying Theorem 27.4 from Rockafellar (1970), we see that problem (3.4) is equivalent to VI($\nabla f_q, X_q$), where f_q is differentiated with respect to x_q. A spatial price equilibrium (Harker 1986) is therefore a point x which solves the following VI:

$$\begin{array}{ll} \text{find} & \bar{x} \in X \\ \text{s.t.} & \sum_{q \in Q} \nabla f_q(\bar{x})^\top (x_q - \bar{x}_q) \geq 0 \quad \forall\, x \in X \end{array} \tag{3.5}$$

A GAMS or AMPL model for this problem can be obtained from (3.5) or, more directly, from the KKT conditions for (3.4). The particular model formulated contains 3 sites and 3 firms, so that each firm controls only one site; the relevant functions are defined as follows:

$$C_l(s_l) := \alpha_l s_l + \beta_l s_l^2, \qquad \theta_l(d_l) := \rho_l - \eta_l d_l, \qquad c_{ij}(t_{ij}) := \gamma_{ij} + \mu_{ij} t_{ij}^2.$$

While this particular example is somewhat limited, the GAMS model `sppe.gms` is coded for the general situation, where each firm controls multiple sites.

Tobin (1988) describes a spatial price equilibrium in a multi-commodity market modeled as a network. In this example, the variables are the prices at the various nodes in the network. These prices determine supply and demand, and not conversely, as in Harker's SPE model. The competitive structure assumed in this example is one of perfect competition; it's "every node for itself". We define the following:

$l = 1, \ldots, n$	the *nodes* (markets) in the network
$k = 1, \ldots, p$	the commodities being traded in the network
$\pi = (\pi_{lk})$	price vector; for each node-commodity pair (l, k), π_{lk} is the unit price of commodity k at node l
$D_{lk}(\pi)$	demand for commodity k at node l
$S_{lk}(\pi)$	supply of commodity k at node l

$a = (ij)$	an arc in the network, from node i to node j
$A = [A_{la}]$	the standard node-arc incidence matrix. A is mainly zeros, with these exceptions: if $a = (ij), A_{ia} = 1$ & $A_{ja} = -1$.
$t = (t_{ak})$	flow vector; for each arc-commodity pair (a, k), t_{ak} is the flow of commodity k on arc a
$c_{ak}(t_{ak})$	unit cost of transportation service for commodity k on arc a.

Section 2 of (Tobin 1988) gives the following conditions for a spatial price equilibrium (SPE):

Nonnegative flows, prices, demands, & supplies:

$$t_{ak} \geq 0, \;\; \pi_{lk} \geq 0, \;\; D_{lk} \geq 0, \;\; S_{lk} \geq 0 \qquad \forall\, a, l, k \tag{3.6a}$$

Conservation of flow at each node:

$$S_{lk} + \sum_i t_{(il)k} = D_{lk} + \sum_j t_{(lj)k} \qquad \forall\, l, k \tag{3.6b}$$

Delivered price exceeds local price:

$$\pi_{ik} + c_{(ij)k}(t) \geq \pi_{jk} \qquad \forall\, a \;:= (ij), k \tag{3.6c}$$

Delivered/local price difference or path flow $= 0$

$$\left\langle \pi_{ik} + c_{(ij)k} - \pi_{jk}, t_{ak} \right\rangle = 0 \qquad \forall\, a \;:= (ij), k. \tag{3.6d}$$

A set of flows and prices are feasible if they satisfy conditions (3.6a) and (3.6b). Condition (3.6c) and the complementarity condition (3.6d) imply that if the delivered price strictly exceeds the local price, no commodity is being delivered, and that if there is a commodity being delivered, its delivered price equals the local price.

If we relax the conservation of flow constraint (3.6b) to allow excessive supply, we get the following NCP:

$$0 \leq c(t) + A^{\top}\pi \quad \perp \quad t \geq 0, \tag{3.7a}$$

$$0 \leq S(\pi) - D(\pi) - At \quad \perp \quad \pi \geq 0, \tag{3.7b}$$

The following lemma gives conditions under which the conditions for a SPE are equivalent to the NCP defined in (3.7).

Lemma 7 *Suppose the arc cost functions $c(t) > 0$ and the demand and supply functions are such that*

$$\pi_{lk} = 0 \;\Rightarrow\; D_{lk}(\pi) - S_{lk}(\pi) > 0 \tag{3.8}$$

Then a set of flows and prices $(\bar{t}, \bar{\pi})$ is a spatial price equilibrium iff it solves the NCP defined by (3.7a) - (3.7b); furthermore, $\bar{\pi} > 0$.

Proof If $(\bar{t}, \bar{\pi})$ is a SPE, then clearly it solves the NCP as well. To show the converse, we need only show that (3.6b) is satisfied at a solution to (3.7). Assume then that $(\bar{t}, \bar{\pi})$ is a solution to the NCP. For the sake of contradiction, assume that $\pi_{lk} = 0$ for some node l and commodity k. Then we have

$$[At]_{lk} \leq S_{lk} - D_{lk} < 0, \tag{3.9}$$

so that node l is a net *importer* of commodity k. Thus, for some node $i, t_{(il)k} > 0$. This and condition (3.7a) of the NCP imply that $\pi_{ik} + c(t_{(il)k}) = \pi_{lk}$. However, this last equation and the positivity of $c(f)$ implies that $\pi_{lk} > 0$, which is the desired contradiction to our original assumption. Since $\pi_{lk} > 0$, (3.7b) implies the conservation of flow constraints must be satisfied exactly; hence, we have a SPE. □

A similar result is proved by Friesz, Tobin, Smith & Harker (1983); there, the inequality in (3.8) is not strict, and as a result, the optimal prices need not be positive; the equivalence between the NCP solution and the SPE still holds. In (Tobin 1988), the positivity of the optimal prices is said to follow from the weaker version of (3.8); this is false, as is easily shown by a small counterexample.

Condition (3.8) is a reasonable one; we can expect demand to exceed supply when something is free. If this is true, we can model the problem as an MCP in two ways: as an NCP (using only non-negativity constraints), or by letting the price vector be free and enforcing the conservation of flow constraints (3.6b) directly.

In the GAMS model `tobin.gms`, the relevant functions are defined as follows:

$$\begin{aligned} c_{ak}(t) &:= \Gamma_{ak} + \Omega_{ak} t_{ak}^4 + \sum_{m \neq k} \Delta_{akm} t_{am} \\ S_{lk}(\pi) &:= \ B_{lk} + J_{lk}\pi_{lk}^2 + \sum_{i \neq l} u_{lik}\pi_{ik} \\ D_{lk}(\pi) &:= \ E_{lk} - G_{lk}\pi_{lk}^2 + \sum_{i \neq l} w_{lik}\pi_{ik} \end{aligned}$$

3.2.3 A Walrasian Equilibrium Model

An equilibrium can be characterized as *Walrasian* if there are no goods for which demand strictly exceeds supply (Varian 1978). Mathiesen (1987) describes an economy containing a number of goods, a number of utility-maximizing consumers, and a number of profit-maximizing producers. Both consumers and producers act as price-takers, that is, they assume that the market price for each good does not change as a result of their actions. The role of the consumers here is to demand goods; this demand is determined by the prices. The producers determine their optimal levels of production based on these demands. Our objective is to find an equilibrium, or a steady state, for the economy. More specifically, we define the following:

$i = 1, \ldots, m$	indices corresponding to the m types of goods or commodities in the economy
$j = 1, \ldots, n$	index corresponding to the n *sectors* or types of production processes in the economy
$p = (p_i)$	vector of prices for the goods
$b = (b_i)$	vector of initial endowments for the goods (i.e. the amount of each good initially available)
$d(p) = (d_i(p))$	consumer demand functions; given a price vector, the demand for good i is $d_i(p)$
$y = (y_j)$	vector of activites; y_j is the activity or production level in sector j
$A = (a_{ij})$	technology matrix; a unit production level in sector j results in an output of a_{ij} units of good i. Negative values of a_{ij} indicate an input of good i is required for activity j. Column $A_{\cdot j}$ describes the process of sector j, while row A_i indicates where good i is used and produced.

The equilibrium conditions given in Definition 5.1.3 of Scarf (1973) are as follows:

No activity earns a positive profit:	$A^\top p \le 0$	(3.10a)
No good is in excess demand:	$b + Ay - d(p) \ge 0$	(3.10b)
No prices or activity levels are negative:	$p \ge 0 \quad y \ge 0$	(3.10c)
An activity earning a deficit is not run; an operated activity runs at zero profit:	$\langle y, -A^\top p \rangle = 0$	(3.10d)
A good in excess supply has a zero price; a positive price implies market clearance:	$\langle p, b + Ay - d(p) \rangle = 0$	(3.10e)

At equilibrium, no activity earns a positive profit; if this were the case, others would step in to duplicate the activity, driving the profit to zero. Condition (3.10b) characterizes the equilibrium as Walrasian; there is no excess demand for any good. Condition (3.10e) implies that goods in excess supply have a zero price; if we assume that the goods are "desirable", (i.e. any good with a zero price must be in demand), then (3.10e) implies that all markets clear, or that supply equals demand.

A noteworthy property of Walrasian models is the assumption that the demand function $d(p)$ is homogeneous of degree 0 (i.e. $d(p) = d(tp) \quad \forall\, t > 0$). As a consequence, the equilibrium price vector is not unique; if p^* is an equilibrium price vector, so is tp^* for $t > 0$. An additional consequence of the homogeneity of d, shown by Mathiesen (1987), is the singularity of the matrix $\nabla d(p)$. This singularity can make finding a solution difficult. Two customary ways of avoiding this singularity are normalizing the price vector or fixing one of the prices, called the numéraire price.

In the example given by Mathiesen (1987), the consumer demand function $d(p)$ is determined by a single consumer; there is one production activity, and 3 goods. The problem is a difficult one because of the singularity of the Jacobian of the NCP formulation when no "fix" is applied, and because of the form of d:

$$d_i(\pi) \;:=\; \frac{a_i \;\sum_k b_k \pi_k}{\pi_i}$$

If we require that $\sum_i a_i = 1$, then a_i determines the fraction of the budget $\sum_k b_k \pi_k$ spent on good i.

Scarf (1973) describes two similar Walrasian models, the smaller of which contains six commodities, eight activity sectors, and 6 consumers. Each consumer n has an initial asset e_{in} of each good i; the initial endowment b_i of good i is given by summing over all the consumers n. The individual initial assets are used in computing the demand function d, which is the sum of the individual consumers' demands. The equilibrium conditions (3.10) are the optimality conditions for this problem as well.

If α_{in} is the demand share parameter for good i and consumer n, and β_n is the elasticity of substitution for consumer n, then the demand function for this problem is

$$d_i(\pi) \;:=\; \sum_n \alpha_{in} \pi_i^{\beta_n} \frac{\sum_k e_{kn} \pi_k}{\sum_k \alpha_{kn} \pi_k^{1-\beta_n}}$$

3.2.4 A Traffic Assignment Model

Bertsekas & Gafni (1982) give a traffic assignment problem where there are 5 cities connected by a network of one–way links (see Figure 11). In each city i, there is a shipper who must ship d_i units of a commodity to city $(i+3)$. Thus, there are 5 origin-destination (OD) pairs in the network. There are only two paths or routes linking each OD pair, the inside and the outside paths. On each of these paths, a delay is incurred, which is equal to the sum of the delays on the links in that path. The delay on a link k is determined by the flow on and near link k, and is given in terms of a convex function g and a parameter $\gamma \geq 0$; we have taken $g(x) := 1 + x + x^2$. Figure 11 gives the configuration of the network, and the link delay functions. It is assumed that all flow not intended for a city will bypass that city.

Let x_i denote the amount shipped from city i via the outside path, and y_i the amount shipped via the inside path. Then the vectors $x = (x_i)$ and $y = (y_i)$ determine the flow on the paths, and also on each of the links. A flow is said to be feasible if

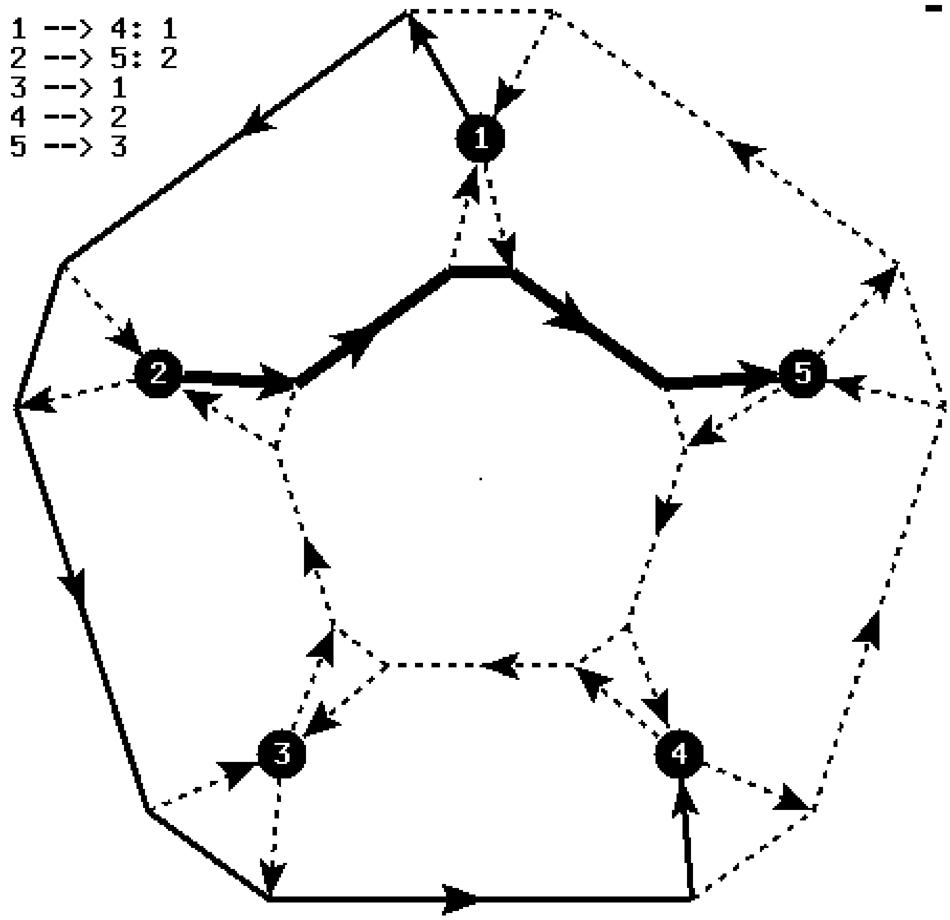
$$\begin{pmatrix} x \\ y \end{pmatrix} \in X := \left\{ \begin{pmatrix} x \\ y \end{pmatrix} \middle| \; x_i + y_i = d_i, \quad x, y \geq 0 \right\}.$$

Given a flow $\begin{pmatrix} x \\ y \end{pmatrix}$, we define the *effective delay* between two cities in an OD pair to be the maximum delay among paths with *nonzero* flow between the two cities. The problem is to find a feasible flow in which each user has minimized her effective delay, subject to all other users' flows remaining constant. This occurs when the delay on every path with *nonzero* flow is the minimum among all paths between the corresponding OD pair. This flow is optimal in the sense that no user can reduce her effective delay by adjusting the flows she controls, while remaining feasible.

The conditions described in the above paragraph can be encapsulated by the optimality conditions VI(T, X), where

$$T \begin{pmatrix} x \\ y \end{pmatrix} := \begin{pmatrix} \text{outside-delay}(x) \\ \text{inside-delay}(y) \end{pmatrix}. \tag{3.11}$$

This VI in 10 variables and 5 demand constraints can be written simply as an NCP in 15 variables, if the demand constraints are relaxed to permit excess flow (there is no excess flow at the solution; clearly, sending excess flow increases any user's effective delay.) The



highway links	An arrow near midpoint indicates direction of flow. Delay on highway link k: $10g[flow_k] + 2\gamma g[flow_{\text{exit from }k}]$.
exit ramps	An arrowhead indicates flow from a highway to a city. Delay on exit ramp k: $g[flow_k]$.
entrance ramps	An arrowhead indicates flow from a city to a highway. Delay on exit ramp k: $g[flow_k] + \gamma g[flow_{\text{bypass of }k}]$.
bypass links	No arrows; flow direction clear from figure. Delay on bypass link k: $g[flow_k]$.

Solid lines indicate positive flow.

Figure 11: Traffic Network

simple demand constraints lead to NCP(G), where

$$G\begin{pmatrix} x \\ y \\ u \end{pmatrix} := \begin{pmatrix} \text{outside-delay}(x) - u \\ \text{inside-delay}(y) - u \\ x + y - d \end{pmatrix}.$$

The problem can be expressed even more compactly by taking advantage of the constraint $x + y = d$ and the generality of the MCP model. Let $B := \{z \mid 0 \leq z \leq d\}$; then

$$X = \{a + Az \mid z \in B\}, \quad a = \begin{bmatrix} 0 \\ d \end{bmatrix}, \quad A = \begin{bmatrix} I \\ -I \end{bmatrix}.$$

Expressing VI(T, X) in term of z, we have the condition

$$\langle T(a + A\bar{z}), (a + Az) - (a + A\bar{z})\rangle = \langle A^{\top} T(a + A\bar{z}), z - \bar{z}\rangle \geq 0 \quad \forall\, z \in B,$$

so that for $F(z) := A^{\top} T(a + Az)$, VI$(T, X)$ is equivalent to VI(F, B).

The intuition behind this latest VI is the clearest of any yet offered: $F_i(\bar{z})$ represents the difference in delay between the outside and inside paths from node i at optimality. When the difference is positive, the outside path is more expensive; all flow from node i should go to the inside. When the difference is negative, the inside path is more expensive; all flow from node i should go to the outside. When the difference is 0, any flow pattern from node i which satisfies the demand constraints is acceptable. Since the feasible set B is rectangular, the VI(F, B) is an MCP. Thus, we need only solve an MCP in 5 variables, rather than the forty-plus variables in the problem on the links, or the 15 variables in NCP(G).

3.2.5 Computing an Invariant Capital Stock

Hansen & Koopmans (1972) consider the problem of determining an invariant optimal capital stock. In this problem, an economy is assumed to grow over an infinite number of time periods. The technology (i.e., the production processes that can be run) and the available resources are assumed constant over all time periods. At the beginning of each time period, the economy invests its *capital goods* into the production processes, which produce both capital goods and *consumption goods*. The capital produced will be invested in the next period, while the consumption goods produced determine the utility of the investment. The total utility is a discounted sum; that is, the utility earned by an investment of capital at time t is discounted by a factor of α^t, where the discount factor $\alpha \in (0, 1)$. We wish to find an initial endowment of capital for which the investment strategy necessary to maximize the discounted sum of the utilities is constant. More formally, we have the following:

r	index for the set of resources types
i	index for the set of capital good types to be invested in production.
j	index for the set of production processes to run; each process consumes capital and resources, and produces capital and consumption goods.
$w = (w_r)$	The resources available at the beginning of each time period; this is assumed constant over time.
$z_t = (z_i)_t$	A *capital stock*; the amount of capital goods available for investment at the beginning of time period t.
$x_t = (x_j)_t$	The level at which to run the production processes during time period t. This effectively determines the investment of the capital stock z_t.
$v(x)$	Utility derived from the production/investment specified by x.
$A = (a_{ij})$	capital input matrix; running production process j at unit level requires a_{ij} units of capital good i ($A \geq 0$)
$B = (b_{ij})$	capital output matrix; running production process j at unit level produces b_{ij} units of capital good i ($B \geq 0$)
$C = (c_{rj})$	resource input matrix; running production process j at unit level requires c_{rj} units of resource good r ($C \geq 0$)
$0 < \alpha < 1$	discount factor for future utility

Assuming an integer time variable t, and given an initial capital stock z_0, we might wish to optimize our growth by solving the following:

$$
\begin{array}{ll}
\underset{x_t, z_t}{\text{maximize}} & \sum_{t=0}^{\infty} \alpha^t v(x_t) \\
\text{subject to} & \begin{array}{rcl} Ax_t & \leq & z_t \\ Bx_t & \geq & z_{t+1} \\ Cx_t & \leq & w \\ x_t & \geq & 0 \end{array}
\end{array}
\tag{3.12}
$$

A solution of (3.12) maximizes the discounted sum of the utilities v; the feasibility conditions ensure that the *growth path* $\{(z_t, x_t)\}$ determining these utilities is consistent with the given technology and resource constraints. Notice that in (3.12), the initial capital stock z_0 is given; this stock determines the optimal growth path. Note also that the sequence of capital stocks $\{z_t\}$ is not fixed explicitly by the constraints in (3.12). However, it is possible that, over time, some optimal pattern of investment and return may evolve; that is, the growth path approaches a constant value.

This motivates the following problem: an initial capital stock z_0 is desired for which the optimal growth path does not vary. It should be noted that one cannot merely require that the path be constant, and optimize the choice of z_0. The invariance of the path must be a result of the optimality conditions in (3.12) and the choice of z_0, not of any explicit constraint. We will not derive here the conditions for a z_0 with a constant optimal growth path, since the motivation for the result is rather lengthy, and the proof longer still. The interested reader is referred to (Hansen & Koopmans 1972), or to (Cottle, Pang & Stone 1992) for an example where v is linear.

We will assume that the utility function to be maximized in (3.12) is concave and continuously differentiable. Under some reasonable constraints on the technology, and a regularity condition on z_0, an initial capital stock z_0 whose optimal growth path (z_t, x_t) is constant satisfies the following NCP:

$$0 \le -\nabla v(x) + (A - \alpha B)^\top y + C^\top u \quad \perp \quad x \ge 0, \tag{3.13a}$$

$$0 \le (B - A)x \quad \perp \quad y \ge 0, \tag{3.13b}$$

$$0 \le -Cx + w \quad \perp \quad u \ge 0. \tag{3.13c}$$

A solution to NCP (3.13) suffices to determine an initial capital stock whose optimal growth path is constant; no regularity condition on z_0 is necessary in this direction. If $(\bar{x}, \bar{u}, \bar{y})$ satisfy (3.13), the capital stock $z_0 = A\bar{x}$.

3.2.6 Extended Linear-Quadratic Programming

A number of recent papers have proposed an extended linear-quadratic programming (ELQP) model (Rockafellar 1988, Rockafellar 1990) as a means of taking advantage of the special structure found in large-scale problems in multi-stage optimization (Rockafellar 1991), stochastic programming (Rockafellar & Wets 1986a), and optimal control (Rockafellar 1988). While problems formulated in this way are generally more difficult to solve than the conventional quadratic program, there exists an elegant duality theory for ELQP, which can be exploited in solution procedures. In this section, the ELQP is defined, and a significant special case is shown to be an instance of the MCP.

A problem in extended linear-quadratic programming is defined using the primal variables $u \in \mathbb{R}^n$, the dual variables $v \in \mathbb{R}^m$, and the nonempty, polyhedral sets $U \subset \mathbb{R}^n$ and $V \subset \mathbb{R}^m$. Let $p \in \mathbb{R}^n$ and $P \in \mathbb{R}^{n \times n}$, and let $q \in \mathbb{R}^m$ and $Q \in \mathbb{R}^{m \times m}$, where Q and P are both symmetric positive semi-definite. In the ELQP model, some constraints are incorporated into a penalty or *monitoring function* added to the objective, rather than being considered explicitly. Given the set V and the matrix Q, this monitoring function is defined as

$$\rho_{VQ}(w) := \sup_{v \in V} \; w^\top v - \tfrac{1}{2} v^\top Q v \qquad \text{for } w \in \mathbb{R}^m \tag{3.14}$$

An extended linear-quadratic program may be defined using either a primal or dual form, both of which follow:

$$\operatorname*{minimize}_{u \in U} \qquad f(u) := p^\top u + \tfrac{1}{2} u^\top P u + \rho_{VQ}(q - Ru) \tag{P}$$

$$\operatorname*{maximize}_{v \in V} \qquad g(v) := q^\top v - \tfrac{1}{2} v^\top Q v - \rho_{UP}(R^\top v - p) \tag{D}$$

The difficulties in solving problems (P) and (D) arise from the monitoring functions ρ.

Theorem 8 (Proposition 2.3, Rockafellar (1987)) *The function ρ_{VQ} is lower semicontinuous, convex, and piecewise linear-quadratic: its effective domain*

$$\operatorname{dom} \rho_{VQ} := \{ w \in \mathbb{R}^m \;\mid \rho_{VQ}(w) < \infty \}$$

is a nonempty convex polyhedron that can be decomposed into finitely many polyhedral convex sets, on each of which ρ_{VQ} is quadratic (or linear); a similar result holds for ρ_{UP} and its effective domain.

Thus, the objective function f is convex and *piecewise* linear-quadratic, as is $-g$. This makes it difficult to apply techniques from smooth optimization in a straightforward manner. However, duality theory can be used to show that problems (P) and (D) above are related through the following Lagrangian function:

$$L(u, v) := p^\top u + \tfrac{1}{2} u^\top P u + q^\top v - \tfrac{1}{2} v^\top Q v - v^\top R u, \tag{3.15}$$

with $f(u) = \sup_{v \in V} L(u, v)$ and $g(v) = \inf_{u \in U} L(u, v)$. The following theorem from Rockafellar (1987) characterizes a pair of solutions to (P) and (D) as a saddle point of L.

Theorem 9 *It is always true that* $\inf(P) \geq \sup(D)$*. Furthermore, a pair* $(\bar{u}, \bar{v})$ *is a saddle point of the Lagrangian* $L(u, v)$ *on* $U \times V$ *if and only* $\bar{u}$ *solves (P),* $\bar{v}$ *solves (D), and the optimum values are equal.*

The characterization of an optimal solution pair $(\bar{u}, \bar{v})$ as a saddle point leads to a characterization in terms of a VI. We define

$$T\begin{pmatrix} u \\ v \end{pmatrix} := \begin{pmatrix} \nabla_u L(u, v) \\ -\nabla_v L(u, v) \end{pmatrix} = \begin{pmatrix} P & -R^\top \\ R & Q \end{pmatrix} \begin{pmatrix} u \\ v \end{pmatrix} + \begin{pmatrix} p \\ -q \end{pmatrix} \tag{3.16}$$

and note from Theorem 9 that the pair $(\bar{u}, \bar{v})$ is optimal for (P) and (D) if and only if $(\bar{u}, \bar{v})$ solves $\mathrm{VI}(T, U \times V)$.

Any ELQP can be reformulated as a conventional QP, and hence as a complementarity problem (Rockafellar & Wets 1986*a*). Unfortunately, this may greatly increase the problem size and disguise any special problem structure. Although specialized techniques can solve ELQP's quickly, we show that a frequently occurring special case of ELQP can be reformulated as an equivalent MCP, without any increase in size or loss of special structure. In a common practical situation (Rockafellar & Wets 1986*b*, Rockafellar & Wets 1986*a*, Rockafellar 1990), the feasible sets U and V are rectangular. In this case, the $\mathrm{VI}(T, U \times V)$ defined by (3.16) is one involving only rectangular constraints, so that no reformulation is necessary to solve the problem as an MCP. In the remainder of this section, we discuss a continuous-time optimal control problem whose discretization results in a problem of this type.

Given a fixed time interval $[t_0, t_1]$, we define the primal problem in terms of the instantaneous control variables $u(t) \in U \subset \mathbb{R}^k$ and the left endpoint control variables $u^L \in U_L \subset \mathbb{R}^{k_L}$; the free state variables $x(t) \in \mathbb{R}^n$ depend on these control variables. The data for the problem (i.e. the matrices $\tilde{A}, \tilde{B}, \tilde{C}, \tilde{D}, \tilde{P}$, and $\tilde{Q}$, the vectors $\tilde{b}, \tilde{c}, \tilde{p}$, and $\tilde{q}$, and the feasible sets U and V) are generally assumed to vary continuously in t; we will assume that these matrices are constant as well. We seek to minimize the functional

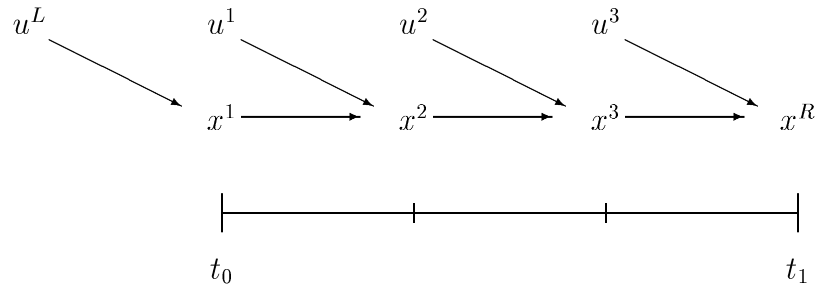
$$\begin{aligned} \mathcal{F}(u^L, u) := & \int_{t_0}^{t_1} [\tilde{p}u(t) + \frac{1}{2}u(t)\tilde{P}u(t) - \tilde{c}x(t)]\, dt \quad + \quad p^L u^L + \frac{1}{2}u^L P_L u^L - c^R x(t_1) \\ & + \int_{t_0}^{t_1} \rho_{V\tilde{Q}}(\tilde{q} - \tilde{C}x(t) - \tilde{D}u(t))\, dt \quad + \quad \rho_{V_R Q_R}(q^R - C_R x(t_1)) \end{aligned}$$

over the state trajectory

$$\frac{dx}{dt}(t) = \tilde{A}x(t) + \tilde{B}u(t) + \tilde{b}, \qquad x(t_0) = B_L u^L + b^L, \tag{3.17}$$

where the subscripts L and R denote data and variables used to define boundary conditions at the left and right endpoints, respectively. In this model, the feasible sets U, U_L, V, and V_R are bounded rectangular sets.

The ELQP model arises as a discretization of the continuous problem above. The interval $[t_0, t_1]$ is divided into N segments, so that the variables $u(t)$ and $x(t)$ are discretized as follows,



$N = 3$

where the arrows indicate the dependence of the state variables on previous states and controls, as determined by (3.17). If we assume that $t_1 - t_0 = 1$, the resulting discrete-time ELQP is that of minimizing

$$\begin{aligned}&\frac{1}{N}\sum_1^N[\tilde{p}u^i + \frac{1}{2}u^i\tilde{P}u^i - \tilde{c}x^i] \quad + \quad p^L u^L + \frac{1}{2}u^L P_L u^L - c^R x^R \\ &+\frac{1}{N}\sum_1^N \rho_{V\tilde{Q}}(\tilde{q} - \tilde{C}x^i - \tilde{D}u^i) \quad + \quad \rho_{V_R Q_R}(q^R - C_R x^R)\end{aligned}$$

subject to the state constraints

$$x^1 = B_L u^L + b^L \tag{3.18}$$

$$x^{i+1} = x^i + \frac{1}{N}(\tilde{B}u^i + \tilde{A}x^i + \tilde{b}) \quad i = 1, \ldots, N-1 \tag{3.19}$$

$$x^R = x^N + \frac{1}{N}(\tilde{B}u^N + \tilde{A}x^N + \tilde{b}). \tag{3.20}$$

If we define $A := I + \frac{1}{N}\tilde{A}$, $B := \frac{1}{N}\tilde{B}$, $b := \frac{1}{N}\tilde{b}$, $C := \frac{1}{N}\tilde{C}$, $c := \frac{1}{N}\tilde{c}$, $D := \frac{1}{N}\tilde{D}$, $P := \frac{1}{N}\tilde{P}$,

$p := \frac{1}{N}\tilde{p}$, $Q := \frac{1}{N}\tilde{Q}$, and $q := \frac{1}{N}\tilde{q}$, we obtain the following ELQP:

$$\begin{aligned}
\underset{u^L,u^i,x^i,x^R}{\text{minimize}} \quad & \mathcal{F}_D(u^L,u^i,x^i,x^R) := \\
& \sum_1^N [pu^i + \frac{1}{2}u^i P u^i - cx^i] \;+\; p^L u^L + \frac{1}{2}u^L P_L u^L - c^R x^R \\
& + \sum_1^N \rho_{VQ}(q - Cx^i - Du^i) \;+\; \rho_{V_R Q_R}(q^R - C_R x^R)
\end{aligned}$$

subject to the constraints

$$\begin{aligned}
x^1 &= B_L u^L + b^L \\
x^{i+1} &= Bu^i + Ax^i + b \quad i = 1, \ldots, N-1 \\
x^R &= Bu^N + Ax^N + b.
\end{aligned}$$

Using (3.16), we can express the optimality conditions for the discrete-time minimization problem as the VI($F, U_L \times U^N \times \mathbb{R}^{n(N+1)} \times V^N \times V_R \times \mathbb{R}^{n(N+1)}$), with

$$F\begin{pmatrix} u \\ x \\ v \\ y \end{pmatrix} = \begin{bmatrix} \bar{P} & 0 & -\bar{D}^\top & -\bar{B}^\top \\ 0 & 0 & -\bar{C}^\top & I - \bar{A}^\top \\ \bar{D} & \bar{C} & \bar{Q} & 0 \\ \bar{B} & \bar{A} - I & 0 & 0 \end{bmatrix} \begin{bmatrix} u \\ x \\ v \\ y \end{bmatrix} + \begin{bmatrix} \bar{p} \\ -\bar{c} \\ -\bar{q} \\ \bar{b} \end{bmatrix},$$

where

$$\bar{P} := \begin{bmatrix} P_L & & & \\ & P & & \\ & & \ddots & \\ & & & P \end{bmatrix}, \quad \bar{D} := \begin{bmatrix} 0 & D & & \\ & 0 & \ddots & \\ & & \ddots & D \\ & & & 0 \end{bmatrix}, \quad \bar{B} := \begin{bmatrix} B_L & & & \\ & B & & \\ & & \ddots & \\ & & & B \end{bmatrix},$$

$$\bar{C} := \begin{bmatrix} C & & & \\ & \ddots & & \\ & & C & \\ & & & C_R \end{bmatrix}, \quad \bar{A} := \begin{bmatrix} 0 & & & \\ A & 0 & & \\ & \ddots & \ddots & \\ & & A & 0 \end{bmatrix}, \quad \bar{Q} := \begin{bmatrix} Q & & & \\ & \ddots & & \\ & & Q & \\ & & & Q_R \end{bmatrix},$$

$$\bar{p} := \begin{bmatrix} p^L \\ p \\ \vdots \\ p \end{bmatrix}, \quad \bar{c} := \begin{bmatrix} c \\ \vdots \\ c \\ c^R \end{bmatrix}, \quad \bar{q} := \begin{bmatrix} q \\ \vdots \\ q \\ q^R \end{bmatrix}, \quad \bar{b} := \begin{bmatrix} b^L \\ b \\ \vdots \\ b \end{bmatrix},$$

and the dots represent replication N times.

In the GAMS implementation, the data elements for the continous-time problem are generated randomly, where the matrices $\tilde{P}$ and $\tilde{Q}$ are generated to be positive (semi)definite. The division by N takes place during the formation of the discretized problem. Note that the discrete-time problem makes use of the function $\mathcal{F}_D(u^L, u^i, x^i, x^R)$ in the variables u and x, while the continuous problem is expressed as a minimization over u only. While it is possible to express the discrete time problem without using the x variables, this results in a dense problem. For this reason, the state variables x and y are retained in the MCP formulation.

3.2.7 An Obstacle Problem

The obstacle problem (Ciarlet 1978) consists of finding the equilibrium position of an elastic membrane subject to a vertical force f pushing upwards. In our example, we consider a membrane with height v on a domain $\mathcal{D} := (0,1) \times (0,1)$. We restrict our attention to those functions v in the space $H_0^1(\mathcal{D})$ of functions with compact support in $\mathcal{D}$ such that v and $\|\nabla v\|^2$ belong to the square integrable class $L^2(\mathcal{D})$. Note that this implies that $v = 0$ on the boundary of $\mathcal{D}$. In addition, we have lower and upper bounds v_ℓ and v_u on v which represent the position of solid objects below and above the membrane, respectively. The membrane's equilibrium position is its position of minimum energy, where the energy of the membrane is given by the quadratic functional $q(v)$ in the following quadratic program:

$$\begin{array}{ll} \underset{v}{\text{minimize}} & q(v) = \frac{1}{2} \int_{\mathcal{D}} \|\nabla v\|^2 \, d\mathcal{D} - \int_{\mathcal{D}} f v d\mathcal{D} \\ \text{subject to} & v \in H_0^1(\mathcal{D}) : v_\ell \leq v \leq v_u \end{array}. \tag{3.21}$$

In (Moré & Toraldo 1991), the force f is taken to be the constant $c = 1$.

In order to solve this problem numerically, the domain $\mathcal{D}$ is discretized by a triangulation of a rectangular grid with grid spacing $h := \frac{1}{N+1}$ in both the X and Y axes. The function v is then approximated by a piecewise linear function which can be represented by its values $v_{i,j}$, for $i, j = 1, \ldots, N$, at the N^2 interior vertices of the triangulation. Using this approximation, the objective function q in (3.21) can be reduced (see for example (Moré & Toraldo 1991)) to a quadratic function

$$q(v) := \tfrac{1}{2} v^\top M v - q^\top v, \tag{3.22}$$

where the components of $v \in \mathbb{R}^{N^2}$ are the values $v_{i,j}$ at the vertices of the triangularization,

$q_{i,j} = ch^2$, and M is the usual pentadiagonal matrix obtained via a difference approximation of the Laplacian operator (diagonal entries of 4, off-diagonal entries of -1). Given the constraints $v_\ell \leq v \leq v_u$, the optimality conditions for minimizing the discretized $q(\cdot)$ can be written as the following MCP:

$$F(v) := Mv - q \quad \perp \quad v \in [v_\ell, v_u]. \tag{3.23}$$

If the force f acting on the membrane is taken taken to be the nonlinear function λe^v, the *obstacle Bratu* problem results. This problem, solved by Miersemann & Mittelmann (1989) and Hoppe & Mittelmann (1989), differs from the one just described in that the components of the vector q are no longer constant but are a function of v, i.e., $q_{i,j} = \lambda e^{v_{i,j}}$.

3.2.8 The Elastohydrodynamic Lubrication Problem

The problem of the elastohydrodynamic lubrication of cylinders in line contact is considered by Kostreva (1984). A particular example considers (cylindrical) roller bearings lubricated by oil. Earlier work by Cryer & Dempster (1980) considers the case where the bearing is rigid, rather than elastic, resulting in a linear complementarity problem. The standard mathematical model for the elastic problem is governed by 3 equations: a linear integral equation for the deformation of the cylinders, Reynolds' differential equation for the pressure in the lubricant, and a linear integral equation which represents a balance of load constraint. If the lubricant pressure at position x is represented by $p(x)$, then the thickness h of the lubricant film between the cylinders at position x is given by

$$h(x) = x^2 + k - \frac{2}{\pi} \int_a^b p(s) \ln |x - s| ds, \tag{3.24}$$

where k is a free variable of the model, x_a is an inlet point and x_b is an outlet point to be determined from the model solution, with $x_a < x_b$. The pressure will be positive between the inlet and outlet points, while the boundary conditions are $p(x_a) = p(x_b) = p'(x_b) = 0$. In the region of positive pressure, Reynolds' equation, which relates lubricant pressure to lubricant film thickness, holds:

$$R(p,k) := -\frac{d}{dx}\left(\frac{h(x)^3}{e^{\alpha p}}\frac{dp}{dx}\right) + \lambda \frac{dh}{dx} = 0. \tag{3.25}$$

Downstream of x_b, the pressure will be 0, so that Reynolds' equation need not be satisfied; in this area, $R(p,k)$ is allowed to become positive and reduces to $\lambda \frac{dh}{dx}$. Since $\lambda > 0$, this represents a divergence of the cylinders downstream of the outlet point. The final equation represents a constraint placed on the cumulative pressure required by the specified load on the cylinders:

$$T(p,k) := 1 - \frac{2}{\pi}\int_a^b p(s)ds = 0. \tag{3.26}$$

Given the inlet point x_a, the complementarity form of this problem makes use of finite difference approximations to R and T on the interval $[x_a, x_F]$, where x_F is chosen to be far downstream, so that $x_F > x_b$. Given a uniform grid of N intervals such that $x_F = x_a + N\Delta x$, let $p_i = p(x_a + i\Delta x)$ and let $h_j = h(x_a + j\Delta x)$ for $i = 1, \ldots, N,\ j = i \pm \frac{1}{2}$. The values of h_j at the intermediate points can be approximated by numerical integration of (3.24) or by the following, computationally recommended, integral obtained from (3.24) via integration by parts:

$$h(x) = x^2 + k + 1 + \frac{2}{\pi}\int_{x_a}^{x_b} (s-x)\ln|x-s|\left(\frac{dp}{ds}\right)ds.$$

In the GAMS model, both h_j and T are approximated using the trapezoidal rule. The formula for h_j is substituted into the finite difference approximation to Reynolds' equation at the points x_i for $i = 1, \ldots, N$ as follows:

$$\begin{aligned} R_i(k,p) := &-\frac{1}{(\Delta x)^2}\left[\frac{(h_{i+\frac{1}{2}})^3}{\exp(\alpha p_{i+\frac{1}{2}})}(p_{i+1} - p_i) - \frac{(h_{i-\frac{1}{2}})^3}{\exp(\alpha p_{i-\frac{1}{2}})}(p_i - p_{i-1})\right] \\ &+ \frac{\lambda}{\Delta x}(h_{i+\frac{1}{2}} - h_{i-\frac{1}{2}}). \end{aligned}$$

The final MCP is given by

$$\begin{aligned} 0 = T(k,p) \quad &\perp \quad k \\ 0 \leq R_i(k,p) \quad &\perp \quad p_i \geq 0, \qquad \text{for } i = 1, \ldots, N. \end{aligned}$$

As mentioned earlier, the location of the free boundary x_b is not known *a priori*; it is determined as part of the solution to the complementarity problem. This is in contrast to other methods proposed for this problem, which rely on heuristics to locate the free boundary. Kostreva (1984) considers examples where the free boundary has been mislocated by these heuristic techniques, as well as other examples where the computed film thickness h differs from previous results.

The elastohydrodynamic lubrication model is interesting both because of its highly nonlinear nature and because of its potentially large size. Unfortunately, it is fully dense, so that sparse techniques cannot be used to improve performance. In his computational work, Kostreva (1984) used a grid of size 0.05 on an interval of length 5, resulting in a highly nonlinear model with 100 equations. However, for higher pressure and load conditions, the solution to this problem develops a large pressure spike, which can be difficult to compute, and necessitates the use of finer grid approximations and larger problems.

Chapter 4

The PATH Solver

The PATH solver is an implementation of a stabilized Newton method for solving MCP. Much of the motivation and previous work behind Newton methods in this context has already been given in Chapter 1. In this chapter, we introduce and describe a stabilization scheme as it applies to Newton methods for nonsmooth equations and present a global convergence result for the damped Newton method that results. In order to do so, it will be convenient to express the MCP as the normal map equation

$$F_B(x) = 0, \tag{NME}$$

where F_B is the normal map of Robinson (1992) imposed on F by the rectangular set $B := \{z \mid \ell \leq z \leq u\}$:

$$F_B(x) := F(\pi_B(x)) + x - \pi_B(x).$$

The normal map is a generalization of the Minty map (Minty 1962) defined when $B := \mathbb{R}^n_+$. Theorem 5 of Chapter 1 shows that a solution to NME leads directly to a solution to MCP, and vice versa. Thus, we can view the MCP as the problem of finding a zero of an equation, albeit a potentially nonsmooth one. This framework enables us to apply Newton-type techniques from equation solving, including the method to be described, to find solutions for the MCP.

In the classical Newton's method, the smooth function F is approximated at a point x^k via the linearization A_k defined by

$$A_k(x) := F(x^k) + F'(x^k)(x - x^k). \tag{4.1}$$

This linearization A_k is said to be a *first order approximation* of F at x_k. The *Newton point* x_N^k is a zero of this approximation, that is, $A_k(x_N^k) = 0$. Assuming nonsingularity of the Jacobian matrix, this zero is unique, and it is a conceptually simple task to find it; one merely solves the linear system $F'(x^k)d^k = -F(x^k)$. The Newton point is defined by $x_N^k := x^k + d^k$, and the Newton direction by d^k. The next iterate in the Newton process is determined by a linesearch along this direction, that is,

$$x^{k+1} := x^k + \lambda^k d^k,$$

where λ^k satisfies appropriate conditions. Thus, a linesearch-damped Newton method can be divided into three parts: linearization, direction-finding, and linesearching. Our presentation will be organized similarly; the PATH solver analogues of these three parts are approximation, path generation and pathsearch damping. These are described in the first three sections of this chapter. In Section 4.4, we describe a nonmonotone stabilization scheme which we have incorporated into the algorithm. Finally, in Section 4.5 we present a convergence proof for the stabilized method.

4.1 Approximation

Due to the piecewise-linear nature of the projection operator $\pi_B(\cdot)$, it is in general impossible to approximate F_B well with a linear function. Instead, a first-order approximation (Robinson 1993) is used, which generalizes the familiar linearization used for smooth functions.

Definition 10 *Let $x^k \in \mathbb{R}^n$. A* first-order approximation *of F_B at x^k is a mapping $A_k : \mathbb{R}^n \mapsto \mathbb{R}^n$ such that*

$$\lim_{x \to x^k} \|F(x) - A_k(x)\| \,/\, \left\|x - x^k\right\| = 0.$$

This is expressed more compactly by saying $F(x) - A_k(x)$ is $o(x - x^k)$. A first-order approximation *of F_B on $X_0 \subset \mathbb{R}^n$ is a mapping $\mathcal{A}$ on X_0 such that for each $x \in X_0$, $\mathcal{A}(x)$ is a first-order approximation of F_B at x.*

Let $\mathcal{A}$ be a first-order approximation of F on X_0. $\mathcal{A}$ is a uniform *first-order approximation (with respect to X_0) if there exists $h : (0, \infty) \mapsto [0, \infty]$, with $h(s) = o(s)$, such that for any $x, y \in X_0$,*

$$\|\mathcal{A}(x)(y) - F(y)\| \leq h(\|x - y\|). \tag{4.2}$$

Note the fundamental difference between first-order approximations to a function at a point and on a set; the approximation on a set is an operator by which approximations at the points in that set can be obtained (e.g. for $x_k \in X_0$, $A_k := \mathcal{A}(x^k)$).

The nondifferentiability of the normal map F_B is due to the piecewise-linear nature of the projection operator $\pi_B(\cdot)$. The standard first-order approximation of F_B at x^k is the point-based approximation of Robinson (1993) obtained by linearizing F around $\pi_B(x^k)$ and leaving the projection operator alone. This yields

$$A_k(x) := M\pi_B(x) + q + x - \pi_B(x), \tag{4.3}$$

where

$$M := F'(\pi_B(x^k)) \qquad \text{and} \qquad q := F(\pi_B(x^k)) - M\pi_B(x^k).$$

A Newton point x_N^k is defined to be a zero of the approximation A_k. This point may not be unique. However, we will continue to use the notation x_N^k to refer to the unique Newton point found by the path generation technique described in the next section. Much of the difficulty in computing a zero of A_k is caused by the projection operator $\pi_B(\cdot)$, which is nonsmooth. Our method for finding a zero depends on the notion of a path, which we now introduce.

4.2 Path Generation

An essential part of the algorithm is the path constructed between the current point x^k and the Newton point x_N^k. The general form of the path construction technique is due to Ralph (1994). This path generalizes the Newton direction d^k in the smooth case, and serves two purposes: it provides us with the Newton point, and it is the backbone of a pathsearch scheme which serves to damp our Newton method and improve on its convergence properties. This piecewise linear path is constructed using pivotal techniques; each pivot step results in a new linear piece of the path. In this section, we describe a parametric method used to construct the desired path from x^k to x_N^k. However, we first describe the equivalence between the approximation A_k of (4.3) and another system more amenable to pivotal techniques, and we review Lemke's method as a type of path construction technique.

Instead of attempting to find a zero of the approximation A_k directly, this approximation is cast as a linear MCP, and solved using a pivotal technique. This technique yields a path to the Newton point x_N^k; furthermore, there is a simple relationship between the variables used in the pivotal technique and those of (NME) which allows an easy transition from points $x \in \mathbb{R}^n$ to points $z \in B \subset \mathbb{R}^n$, and vice versa. This will be crucial in the pathsearch stage of the algorithm. Set

$$\begin{aligned} z &= \pi_B(x), \\ v &= (x - z)_+, \\ w &= (z - x)_+. \end{aligned} \tag{4.4}$$

Since v and w are the positive and negative parts of $x - z$, it follows that $v - w = x - z$ and

$$x = z - w + v,$$

where

$$w \geq 0, \qquad v \geq 0, \qquad w^\top v = 0 \tag{4.5a}$$

$$z \in B, \tag{4.5b}$$

$$z = \pi_B(z - w + v). \tag{4.5c}$$

Definition 11 *Let $B \subset \mathbb{R}^n$ be rectangular, and $x \in \mathbb{R}^n$.*

1. *The vectors z, w, and v defined by (4.4) are said to be the* components *of x.*

2. *Vectors z, w, and v satisfying (4.5) are said to* comprise *x; (z, w, v) is called a* triple.

It is clear that there is a 1-1 correspondence between triples (z, w, v) and the points $x \in \mathbb{R}^n$; a triple (z, w, v) comprises x precisely when z, w, and v are the components of x. Moreover, the vector x solves $F_B(x) = 0$ exactly when its components solve the MCP:

Definition 12 (MCP) *Given a box $B := [\ell, u]$ and a function $F : B \to \mathbb{R}^n$,*

$$\text{find} \qquad z \in \mathbb{R}^n, \;\; w, v \in \mathbb{R}^n_+$$

s. t.

$$
\begin{aligned}
F(z) &= w - v \\
\ell \leq z &\leq u \\
\langle w, z - \ell \rangle &= 0 \\
\langle v, u - z \rangle &= 0
\end{aligned}
$$

The approximation A_k can be written using the components of x to obtain

$$
\begin{aligned}
0 = A_k(x) &= M\pi_B(x) + q + x - \pi_B(x) \\
&= Mz + q - w + v,
\end{aligned} \tag{4.6}
$$

where (z, w, v) comprise x. It is in the form (4.6) that the zero of the approximation A_k is computed. In the course of solving (4.6), valid triples (z, w, v) are maintained throughout; the vectors x comprised by these triples form a path.

We now define the formal notion of a path, using the definition from Ralph (1994).

Definition 13 *A* path *in* $\mathbb{R}^n$ *is a continuous function* $p : [0, T] \mapsto \mathbb{R}^n$*, where* $T \in [0, 1]$*. The Newton path satisfies the following additional conditions:*

$$
p^k(0) = x^k, \tag{4.7a}
$$

$$
A_k(p^k(t)) = (1 - t)F_B(x^k), \quad \forall\, t \in [0, T]. \tag{4.7b}
$$

The path p^k may be denoted simply by p when the context makes the meaning clear. Note that (4.7b) implies that the norm of the approximation at points on p decreases linearly as a function of $1 - t$, and that the point $p(1)$ is a Newton point. To avoid ambiguity, we will assume that the notation x_N^k refers to this Newton point, which is unique, if it exists. Note also that (4.7a) requires that the path begin at the current point x^k.

When the feasible set $B = \mathbb{R}^n_+$, the approximation (4.6) reduces to the linear complementarity problem (LCP), to which Lemke's method can be applied. We now consider Lemke's method as a path construction technique.

In Lemke's method (Lemke 1965, Cottle & Dantzig 1968), an extra column (called a covering vector) is added to the matrix M, along with an artificial variable λ. Typically, the covering vector is taken to be the unit vector e. This vector is introduced to achieve feasibility for an augmented system, while also maintaining complementarity in the original variables, that is, (4.6) is replaced by

$$\begin{bmatrix} M & -I & e \end{bmatrix} \begin{bmatrix} z \\ w \\ \lambda \end{bmatrix} = -q \tag{4.8}$$

$$z, w, \lambda \geq 0,$$

where $(z, w, v \equiv 0)$ comprise x. A ray start is performed, in which λ_0 is set to $\min\{\lambda \mid \lambda \geq 0,\ e\lambda + q \geq 0\}$. This ray start leads directly to an initial basic feasible solution (BFS) (Chvátal 1983) of the system (4.8). Note that the variables z and w are feasible for the LCP (i.e., $z \geq 0$, $w \geq 0$, $w = Mz + q$) only if $\lambda = 0$. In general, λ will be basic in the initial BFS, with value $\lambda_0 > 0$, as a result of the ray start. Thus, Lemke's method specifies pivoting rules which determine a sequence of entering and leaving variables and BFS which maintain the complementarity of z and w. The algorithm terminates successfully when a pivot results in λ leaving the basis at 0. At this point, the LCP has been solved; the original variables z and w are both complementary *and* feasible. The solution to this LCP is the Newton iterate, and a path, parameterized by $t = 1 - \frac{\lambda}{\lambda_0}$ and leading from the initial BFS to the Newton iterate, has been constructed by the sequence of Lemke pivots. At every point in this path, z and w comprise a vector x. Unfortunately, the Lemke path is not quite what is needed, since in general it does not include the current point (z^k, w^k), violating condition (4.7a).

In the general case, the approximation (4.6) is expressed using the triple (z, w, v); any path p from x^k to x_N^k can be expressed as a triple by letting $(z(t), w(t), w(t))$ be the components of $p(t)$, that is,

$$p(t) := z(t) - w(t) + v(t),$$

for all $t \in [0, T]$. The requirements for a feasible path (4.7) require that

$$(1 - t)F_B(x^k) = A_k(p(t)),$$

or applying (4.6) that

$$(1 - t)r = Mz(t) + q - w(t) + v(t), \tag{4.9}$$

where $r := F_B(x^k)$ is the "residual" vector at the start of the path. Clearly, setting $p(0) = x^k$ satisfies (4.9), while the triple $(z(1), w(1), v(1))$ comprises the Newton point x_N^k. The path from x^k to x_N^k is now determined by a sequence of pivots, which are analogous to the pivots used in Lemke's method above and which we now describe.

In the PATH solver we use r as the covering vector. Thus in the general case, (4.8) becomes:

$$\begin{gathered}
\begin{bmatrix} M & -I & I & r \end{bmatrix} \begin{bmatrix} z \\ w \\ v \\ t \end{bmatrix} = -q + r \\
\ell \leq z \leq u \\
w, v \geq 0 \\
0 \leq t \leq 1.
\end{gathered} \tag{4.10}$$

The initial BFS is determined by the triple (z^k, w^k, v^k), where $t = 0$. If the triple is non-degenerate (i.e., for all $j \in 1, \ldots, n$, exactly one of z_j^k, w_j^k, and v_j^k is not at a bound), then the choice of basis corresponding to the triple is unique; the basis consists of columns corresponding to variables not at bound. The *first* entering variable is always t, which enters the basis at its lower bound 0, and forces a variable to leave the basis. The leaving variable, chosen by a ratio test, must be one of four types, and determines the choice of entering variable according to the following **pivot rules**:

w_j: If w_j leaves the basis, the next entering variable will be z_j, which will enter at its lower bound ℓ_j.

v_j: If v_j leaves the basis, the next entering variable will be z_j, which will enter at its upper bound u_j.

z_j: If z_j leaves the basis at lower bound, w_j enters at 0. If z_j leaves at upper bound, v_j enters at 0.

t: If t leaves the basis at upper bound 1, the Newton point x_N^k has been computed and can be recovered from the basis.

The choice of entering variable drives a new pivot step. The path-construction algorithm continues taking pivot steps, using the pivot rules indicated above, until t leaves the basis at 1 (successful termination), t leaves at lower bound, or the ratio test results in no leaving variable (ray termination). Note that once t enters the basis, the lower bound of 0 for t may be relaxed or ignored. Relaxing this bound has proved useful in practice; some of the linearizations solved admit a Newton point only after a sequence of pivots in which t oscillates and takes values less than 0. Each pivot step described above results in a new (linear) piece of the path. Thus, it is possible for a path to have a very large number of pieces; consequently, it may be quite expensive to store. Since the techniques used for storing the path depend upon how the path is to be searched, we defer a discussion of path storage techniques to Section 4.3, where the pathsearch is considered.

It is clear from (4.9) that the residual vector $r(t) := A_k(p(t))$ goes to 0 linearly in t as t goes from 0 to 1. In Figure 12, we have plotted the contours of the Euclidean norm of this residual vector for an approximation A_k, along with two paths, each leading to a zero of this approximation. The piecewise nature of the path is clearly illustrated by this figure, as well as the smooth nature of the approximation A_k on each cell of the normal map defined by the box $B := [0, \infty) \times [0, 1]$ and the nonsmoothness of the approximation on the boundaries of each cell. We see as well that the direction taken at each pivot step (each crossing of the boundary) is one which minimizes the norm of A_k on the current cell.

It is possible that the basis corresponding to the triple (z^k, w^k, v^k) be rank deficient. In this case, it is impossible to construct the path from the current point to the Newton point by the path generation method described; instead, the PATH solver constructs a path from a new point to the Newton point. This new point is chosen so as to correspond to a basis; for all constrained variables, slack columns are made basic. If there are no free variables, this all-slack basis is guaranteed to be of full rank. Furthermore, is is possible, by a simple choice of the basic values for the slack variables w and v, to duplicate exactly the sequence of pivots performed by Lemke's complementary pivot algorithm. In the case where there are free variables, it may not be possible to choose a full-rank basis corresponding to any valid triple, since the columns corresponding to the free variables must always be in the basis; a sufficient condition is that the principal submatrix M_f corresponding to the free variables be of full rank. In this case, a basis can be obtained by choosing as many slack columns as possible. Cao & Ferris (1992) describe a scheme whereby the lineality of the feasible set B can be factored out, and a new problem solved over a reduced space. The full rank condition on M_f is a necessary condition in that context. However, in a more recent paper (Cao & Ferris 1994), the lineality is removed under a copositive-plus assumption only. Neither of

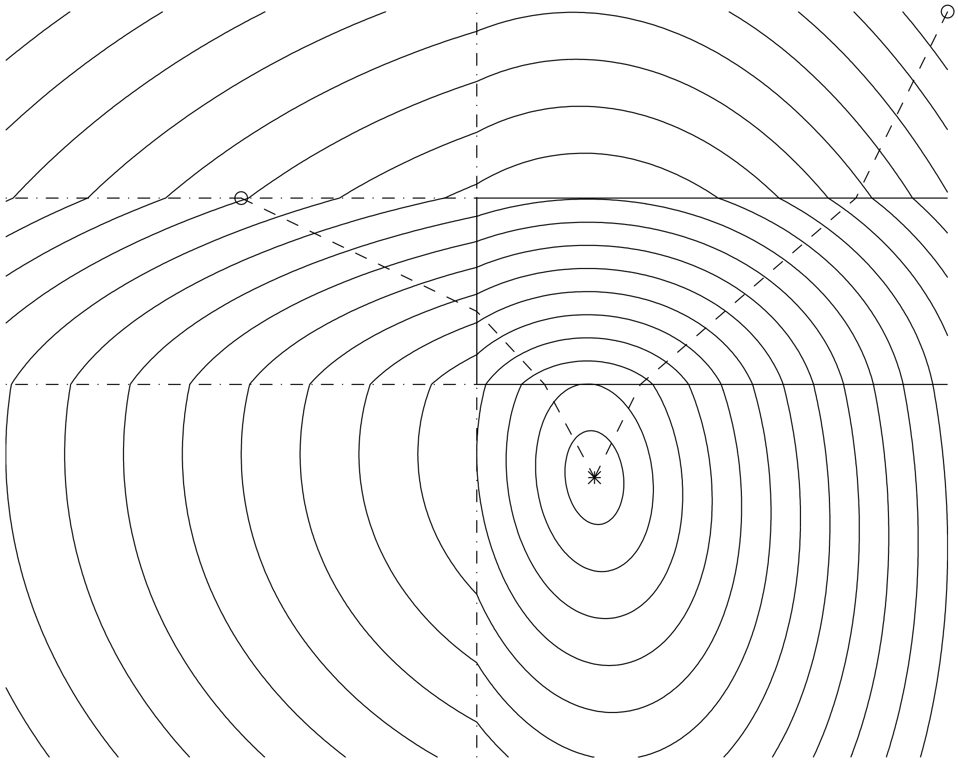


Figure 12: Contour Plots for $\|A_k\|$

these techniques is currently included in PATH, since this case has not occurred in practice.

Having generated the path, we now return to the nonlinear model and describe our globalization strategy, pathsearch damping.

4.3 Pathsearch Damping

In pathsearch-damped Newton's method for finding a zero of the function F_B, the path from x^k to x^k_N is searched for a point satisfying some descent condition. This condition is often a sufficient reduction in the norm of F_B or in some other *merit function*. These merit functions are nonnegative functions whose zeroes coincide exactly with those of F_B. Thus, while the path is computed in order to find a zero of A_k, the next iterate will be a point on this path yielding a suitable decrease in the merit function. In the smooth case, the Newton direction yields a zero of the approximation and also serves as a descent direction for the merit function (Dennis & Schnabel 1983). The paths we construct have similar properties.

Recall from Section 4.2, (4.7b) that the norm of the approximation A_k goes to zero linearly in $(1-t)$ on the path p. We use this and the approximation properties of A_k to show that the norm of F_B must decrease on the path near $t=0$. Let $\sigma \in (0,1)$; then

$$\begin{aligned} \|F_B(p(t))\| &= \|A_k(p(t)) + o(t)\| \\ &= (1-t)\,\|F_B(x_k)\| + o(t) \\ &\leq (1-\sigma t)\,\|F_B(x_k)\|, \end{aligned} \tag{4.11}$$

for some $\bar{t} \in (0,1)$ and all $t \in [0,\bar{t})$. Thus, the norm of F_B decreases on a section of the path near 0, so that p is a "descent path" for F_B. Note that the relaxation parameter $\sigma < 1$, so that the norm of F_B for acceptable points on p does not have to be as small as predicted by the approximation A_k; in practice, σ will be chosen to be close to 0, so that almost any decrease in $\|F_B\|$ will be sufficient for acceptance.

For solving $F(x) = 0$, one possible merit function $\Theta(\cdot)$ is

$$\Theta(x) := \frac{1}{2}F(x)^\top F(x),$$

the norm function of F. In this case, the Newton direction $d = -F'(x^k)^{-1}F(x^k)$ is a descent direction for Θ; note that

$$\begin{aligned}\Theta'(x^k)\, d &= F(x^k)^\top F'(x^k)\, d \\ &= F(x^k)^\top F'(x^k)(-F'(x^k)^{-1})F(x^k) \\ &= -F(x^k)^\top F(x^k) < 0.\end{aligned}$$

In order to find a value of t which satisfies (4.11), an Armijo search (Armijo 1966, Dennis & Schnabel 1983) can be performed on the path p. In a typical implementation of this technique, a parameter $\gamma \in (0,1)$ is chosen, and the points $p(1),\ p(\gamma^1),\ p(\gamma^2),\ \ldots$ are tried, until a value of t is found for which

$$\|F_B(p(t))\| \leq (1-\sigma t)\,\|F_B(x_k)\|\,.$$

In the smooth case, the path p consists of the line between x^k and x_N^k, so that both storing p and computing $p(t)$ are trivial tasks. This is not the case when p is piecewise linear; in this case, it is necessary to modify the standard linesearch techniques to accommodate the special form of the path. Ralph (1994) suggests two approaches to pathsearching. The first, called the *forward pathsearch*, checks that the descent condition (4.11) is satisfied as the path is being constructed. Assuming that each pivot step in the path generation algorithm results in an increase in the value of t from t_{old} to t_{new}, the forward pathsearch ensures that t_{new} satisfies (4.11). The path generation / pathsearch routine terminates when the Newton point is found, or when a pivot step results in an unacceptable value of t_{new}. In the latter case, the line segment between $p(t_{\text{old}})$ and $p(t_{\text{new}})$ is searched for an acceptable point, which becomes the next iterate x^{k+1}. The primary advantage of the forward pathsearch lies in its simplicity; the path is searched as it is constructed, so that it does not need to be stored.

As reported by Ralph (1994), the chief drawback of the forward pathsearch lies in the fact that the search begins on the wrong end of the path. When the Newton point $p(1)$ is acceptable, all the function evaluations performed in checking that the descent condition is satisfied during path construction are essentially wasted. Also, the forward pathsearch is too restrictive in the sense that an acceptable Newton point may exist at the end of a path which has been terminated due to a failure to satisfy (4.11) at an intermediate point on the path. Since we wish to accept the Newton step as often as possible, it seems reasonable to check the Newton point first. Recognizing this, Ralph (1994) suggests a *backward pathsearch*, in which the path is constructed without checking the descent condition. Instead, a list is made, with each element in the list containing the values of the variables at a breakpoint in the path. When the path has been fully constructed, the endpoint is checked. If this point satisfies (4.11), this point is accepted as the next iterate; if not, a recursive bisection search is carried out on the list. This bisection search checks (4.11) for a point near the center of the list; if this point is acceptable, the second half of the list is searched; if not, the first half is searched. In this way, the Newton point is checked first, when it is part of the path. Ralph demonstrates that this leads to fewer function evaluations. Unfortunately, the backward pathsearch also requires that the sequence of pivot steps be recorded. This may require a large amount of space, which cannot be estimated before path construction. The amount of storage required at each pivot step is $O(n)$, while the number of pivot steps may be exponential in n. This is a serious drawback for large-scale problems. Also, a bisection of the list may not lead to a bisection of the current section of the path to be searched; the change in t at each pivot varies widely and nonuniformly.

Motivated by the success of the backwards pathsearch in reducing the number of necessary function evaluations and in increasing the chance of accepting the Newton point, we have implemented a *backtracing* pathsearch. As in the backwards pathsearch, we construct the path without searching it; path generation is completed when the Newton point is found or when ray termination occurs, but not when t oscillates or when the descent criteria are violated. However, instead of saving all of the variable values at each pivot, only information about the entering variable is stored, on a stack which grows with the path. When path generation terminates at a point $p(T)$, the only information about the path that exists is the current basis and a record of the entering variables which led to this basis. At this point, the backtracing pathsearch traces the path in the reverse direction from that of its construction, using the information about the entering variables from the stack to "unpivot", i.e., to reconstruct the breakpoints of the path, along with their associated bases. Backtracing ends when an acceptable point is found. Backtracing is essentially as expensive as is constructing the path; however, this expense is only incurred when a backtrace is necessary. When the point $p(T)$ is acceptable, the information about the path (which can be saved quite cheaply) can be thrown away, without any real backtracing taking place. Also, in this case, at most one function evaluation will be performed.

The forward pathsearch requires little storage, at the cost of added computation (in the form of function evaluations) and reduced robustness. The backward pathsearch requires a minimal number of function evaluations and increases robustness, at the cost of a large storage requirement. The backtracing pathsearch possesses the advantages of both of the above methods, while its drawback (the computational cost of reconstructing parts of the path) is evidenced only when $p(T)$ is unacceptable and a nontrivial pathsearch must be performed. The nonmonotone stabilization techniques discussed in the next section serve to reduce the number of pathsearches performed and make the backtracing pathsearch an even better choice.

4.4 Nonmonotone Stabilization

In a linesearch-damped Newton method, the line from x^k to x^k_N is searched for a point satisfying some descent condition, usually expressed in terms of a decrease in some merit function. Implementations of these methods invariably require a monotonic decrease in this merit function, although there is evidence which indicates that this requirement may impede or block convergence to the solution of the equation (Grippo, Lampariello & Lucidi 1986, Grippo, Lampariello & Lucidi 1991, Ferris & Lucidi 1994). Various nonmonotone stabilization (NMS) schemes for Newton's method have been proposed, each seeking to improve efficiency by relaxing the requirement of monotone descent. The PATH solver implements a scheme of this type, modified to incorporate a pathsearch rather than a linesearch.

The NMS scheme implemented makes use of the *watchdog* technique proposed by Chamberlain, Powell & Lemaréchal (1982) to reduce the number of pathsearches performed, and allows a nonmonotonic decrease in the merit function associated with the points chosen as a result of these pathsearches. The number of pathsearches is reduced by taking a *d-step* in the majority of cases. A d-step is acceptable if the point returned by the path generation procedure is suitably close to the current point. The measure of closeness, Δ, decreases as the algorithm progresses. In order to monitor these steps, the nonmonotone descent criteria for the merit function are checked at least once every $\bar{n}$ number of iterations. The current merit function value is compared with a *reference value* $\mathcal{R}$, which is computed from previous function values. Steps in which these checks on the current merit function value occur are called *m-steps*. The points at which these criteria are checked *and satisfied* are called *check* points. An m-step is also taken when a d-step is unacceptable, that is, when it is too large. A watchdog step occurs when descent criteria are violated; when this occurs, the algorithm returns to the most recent check point, re-generates the path from the check point (if necessary), and backtraces the path until the nonmonotone descent criteria are satisfied.

For future reference we introduce a new index j which is set initially to $j = 0$ and incremented each time we define a new check point. If $\ell(j)$ is the index of the jth check point, then we indicate by $\{x^{\ell(j)}\}$ the sequence of check points (where the merit function has been evaluated) and by $\{\mathcal{R}_j\}$ the sequence of reference values associated with the check points. Each check point $x^{k+1} := p^k(t_k)$ is chosen so that the step length t_k satisfies equation (NmD) below, a generalization of a descent condition for the monotone linesearch: given a reference value $\mathcal{R} \geq \left\|F_B(x^k)\right\|$, the step length t_k satisfies

$$\left\|F_B(p^k(t))\right\| \leq (1 - \sigma t)\,\mathcal{R}. \tag{NmD}$$

If T_k satisfies (NmD), then the pathsearch will choose the step length $t_k := T_k$. If not, we require that the step length be chosen to be large enough, in some sense. This is accomplished by making the technical assumption that t_k be at least τ times as large as the largest interval $[0, T]$ on which (NmD) is satisfied, for some $\tau \in (0, 1)$. Thus, we require that the step length t_k satisfy the following:

$$\begin{gathered}\text{(NmD) holds for } T_k \text{ implies } t_k := T_k;\ \text{otherwise,} \\ \exists\, \tau \in (0,1) \text{ s.t. } t_k \geq \tau \sup\{T \mid \text{(NmD) holds } \forall\, t \in [0,T]\}.\end{gathered} \tag{NmPs}$$

Note that the backtracing pathsearch described in Section 4.3 yields a value t_k which satisfies (NmPs). To see this, note that, given a linear segment of the path from $p(t_{\text{old}})$ to $p(t_{\text{new}})$ for which (NmD) holds at t_{old} but not at t_{new}, the segment can be searched using an Armijo technique for a point at which (NmPs) is satisfied, where τ depends on the pathsearch parameters used.

In order to complete the description of the algorithm we must specify the rule employed for updating $\mathcal{R}_j$, the reference value for the merit function. This is initially set to $\|F_B(x^0)\|$. Whenever a point $x^{\ell(j)}$ is generated such that $\left\|F_B(x^{\ell(j)})\right\| < \mathcal{R}_j$, the reference value is updated by taking into account the memory (that is, a fixed number $m(j) \leq \bar{m}$ of previous values) of the merit function. To be precise, we require the updating rule for $\mathcal{R}_{j+1}$ to satisfy the following condition.

Reference Updating Rule: Given $\bar{m} \geq 0$, let $m(j+1)$ be such that

$$m(j+1) \leq \min\left[m(j) + 1, \bar{m}\right],$$

let

$$\mathcal{M}_{j+1} := \max_{0 \le i \le m(j+1)} \left\| F_B(x^{\ell(j+1-i)}) \right\|, \tag{4.12}$$

and choose the value $\mathcal{R}_{j+1}$ to satisfy

$$\left\| F_B(x^{\ell(j+1)}) \right\| \le \mathcal{R}_{j+1} \le \mathcal{M}_{j+1}. \tag{4.13}$$

These conditions on the reference values include several ways of determining the sequence $\{\mathcal{R}_j\}$ in an implementation of the algorithm. For example, any of the following updating rules can be used:

$$\mathcal{R}_{j+1} = \mathcal{M}_{j+1} = \max_{0 \le i \le m(j+1)} \left\| F_B(x^{\ell(j+1-i)}) \right\|, \tag{4.14}$$

$$\mathcal{R}_{j+1} = \max \left[\left\| F_B(x^{\ell(j+1)}) \right\|, \frac{1}{m(j+1)+1} \sum_{i=0}^{m(j+1)} \left\| F_B(x^{\ell(j+1-i)}) \right\| \right], \tag{4.15}$$

$$\mathcal{R}_{j+1} = \min \left[\mathcal{M}_{j+1}, \frac{1}{2} \left(\mathcal{R}_j + \left\| F_B(x^{\ell(j+1)}) \right\| \right) \right]. \tag{4.16}$$

We note that (4.14) is the easiest to satisfy and is used in the PATH solver, while (4.15) and (4.16) define conditions which guarantee "mean descent".

We should stress at this point that the stabilization technique differs from standard linesearch techniques in two ways. Firstly, the acceptance criteria for the pathsearch are relaxed significantly by replacing the current merit function value by a reference value, typically taken to be the maximum over a fixed number of previous merit function values. Secondly, the pathsearch is skipped entirely when the Newton point is close to the current point (within the d-step tolerance Δ) and an m-step is not required.

The algorithm can be outlined as follows:

Algorithm PATH

1) [Initialization] Let x^0, $\bar{n} >= 1$, $\Delta = \bar{\Delta} > 0$, $\beta \in (0, 1)$ be given:

   set $k = 0$, check_point $= 0$, j $= 0$, $\Delta_0 = \Delta$, $\mathcal{R}_0 = \|F_B(x^0)\|$.

2) If $F_B(x^k) = 0$, stop.

3) Using the approximation A_k, generate a path $p^k : [0, T_k] \mapsto \mathrm{I\!R}^n, T_k \in (0, 1]$, satisfying (4.7).

4) If ($k <$ check_point $+ \bar{n}$) then

d-step:

if ($\left\| p^k(T_k) - p^k(0) \right\| < \Delta$), the step is small enough; accept it:

set $x^{k+1} := p^k(T_k)$;

set $\Delta = \Delta * \beta$;

else the step is too large; go to m-step:

else

m-step:

if ($\left\| F_B(p^k(T_k)) \right\| \leq (1 - \sigma T_k)\mathcal{R}_j$), accept the step:

set $x^{k+1} := p^k(T_k)$;

else perform a watchdog step:

set $k =$ check_point, $\Delta = \Delta_j$;

if necessary, generate the path p^k from x^k to $p^k(T_k)$;

backtrace p^k to find $t_k \in (0, T_k]$ satisfying (NmPs); set $x^{k+1} := p^k(t_k)$;

increment j; update $\mathcal{R}_j$; set $\Delta_j = \Delta$; set check_point $= k + 1$.

5) Increment k, and go to Step 2.

In Step 3 above, it is assumed that T_k is as large as possible, i.e., that if $T_k < 1$, A_k is not continuously invertible near $p^k(T_k)$. For practical reasons and robustness, the PATH solver checks whether the function F_B is defined at a given point before accepting that point. This check, not described in the algorithm above, yields a function value at each point. When this function value is computed after a d-step and found to be lower than the reference value, the reference value and check point are updated. If the function at the prospective new iterate is undefined, a watchdog step is performed in the same manner as that performed for a failing m-step.

In order to illustrate the behavior of the PATH algorithm and the stabilization techniques incorporated into it, we have solved a modified version of the ETAMACRO model from the

GAMS MPSGE library and saved the iteration log, which is given in Table 9 and forms the basis for Figure 13. The modification to the model consisted only of lengthening the time horizon and decreasing the size of each period within this time horizon, resulting in a larger problem.

Iterate	pivots	F evals	t value	$\|F_B\|$
x^0	0	1		5.1863e+01
$\hat{x}^1$	16	1	0	3.1458e+06
$\hat{x}^2$	12	1	0	6.0384e+06
x^0	0			5.1863e+01
x^1	16	18	.874	4.5335e+01
$\bar{x}^2$	12	1	0	3.1459e+06
$\bar{x}^3$	12	1	0	6.0384e+06
x^1	0			4.5335e+01
x^2	12	13	.596	3.4070e+01
x^3	2	1	0	3.8274e+01
x^4	1	1	0	1.5150e+01
x^5	1	1	0	6.0979e+00
x^6	1	1	0	1.7764e+00
x^7	1	1	0	2.5578e-01
x^8	1	1	0	7.0422e-03
x^9	1	1	0	5.6718e-06
x^{10}	1	1	0	3.6886e-12

Table 9: PATH Solver Output for Modified ETAMACRO Problem

The data in Table 9 indicate that the first two subproblems terminate at the Newton point. Note that for numerical reasons, the PATH solver forces t from 1 down to 0 instead of vice versa, so that a terminal t value of 0 corresponds to the Newton point. Note also that $\|F_B(\hat{x}^1)\|$ and $\|F_B(\hat{x}^2)\|$ are very much larger than $\|F_B(x^0)\|$. These residual norms are not used by the algorithm, but are computed to demonstrate its progress. The algorithm parameters were set to check the descent condition after two iterations, so that instead of accepting $\hat{x}^2$, the algorithm returns to the check point x^0 (i.e. it performs a watchdog

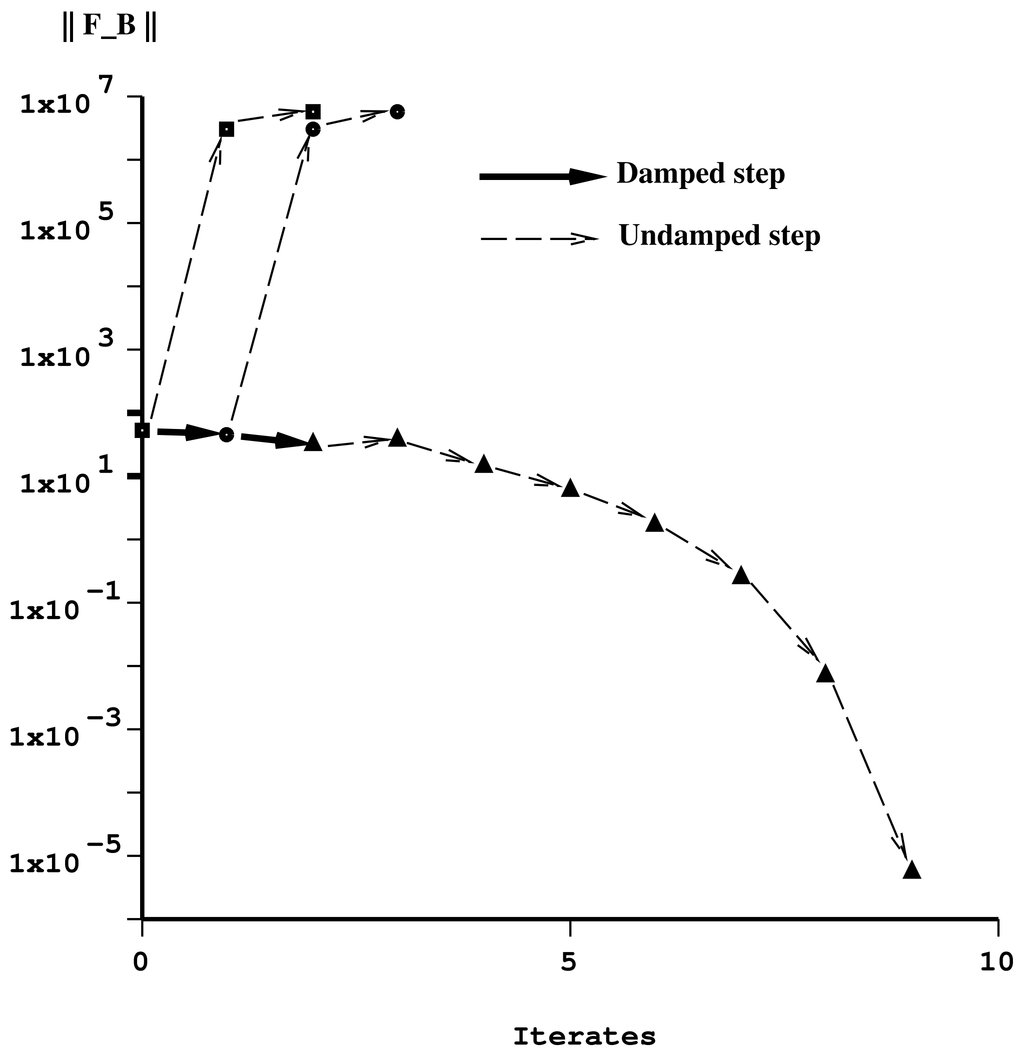


Figure 13: PATH Solver Output for Modified ETAMACRO Problem

step). In doing so, the points $\hat{x}^1$ and $\hat{x}^2$ are discarded. This is indicated in Table 9 by the first horizontal line . The path is reconstructed from x^0, and instead of accepting the resulting Newton point, a backtracing pathsearch is performed. This pathsearch terminates at $t = .874$ and a point x^1, which becomes the new check point. The pathsearch required 18 function evaluations and resulted in a decrease in $\|F_B\|$. The algorithm continues from x^1 by taking two Newton steps $\bar{x}^2$ and $\bar{x}^3$, but again, the descent conditions are not satisfied. Thus, a watchdog step is performed and the algorithm returns to the check point x^1, as indicated by the second horizontal line in Table 9. The path from x^1 is reconstructed, and the nonmonotone linesearch procedure gives $t = .596$ and the new check point x^2.

The Newton point x^3 does not satisfy any monotone descent criterion; it is, however, accepted by the watchdog method. This is fortunate, since from this point on, the PATH solver computes Newton points which satisfy any reasonable descent criteria. Note that the optimal basis has been reached at this point, so that each succeeding iteration requires only one pivot step. Each of these single pivot steps result in a linear path from the current iterate to the Newton point.

In Figure 13, the norm of the residual F_B at each of the iterates is plotted on a logarithmic scale. In addition, the type of step taken to reach each of these iterates, whether damped or undamped, is indicated.

4.5 A Global Convergence Result

In this section, we present a global convergence result for the PATH solver. This result generalizes the work of Ralph (1994) through the addition of the watchdog technique described earlier. Before doing so, we include, without proof, a path lifting result from Ralph (1994) which guarantees the existence of the paths used in our algorithm.

Lemma 14 *Let $\Phi : X \mapsto Y$, $x \in X$ and $\Phi(x) \neq 0$. Suppose the restricted mapping $\bar{\Phi} := \Phi \mid_U : U \mapsto V$ is continuously invertible, where U and V are neighborhoods of x and $\Phi(x)$, respectively. If U is open, and $\epsilon > 0$ is such that $\Phi(x) + \epsilon \mathbb{B} \subset V$, then, for $0 \leq T \leq \min\{\frac{\epsilon}{\|\Phi(x)\|}, 1\}$, the unique path p of domain $[0, T]$ such that*

$$p(0) = x$$

$$\Phi(p(t)) = (1 - t)\,\Phi(x) \quad \forall\, t \in [0, T]$$

is given by

$$p(t) = \bar{\Phi}^{-1}((1-t)\Phi(x)) \quad \forall\, t \in [0,T].$$

We now present our main result, which gives the convergence properties of the PATH algorithm. Note that (A3), the third assumption below, is a technical one which states that the domains of the paths used by the algorithm can be closed (see also (Ortega & Rheinboldt 1970, Definition 5.3.1)).

Theorem 15 *Let $F : \mathbb{R}^n \mapsto \mathbb{R}^n$ be continuous, and let $\alpha_0 > 0$ and $X_0 := \{x \in \mathbb{R}^n \mid \|F_B(x)\| \leq \alpha_0\}$. Let $\sigma, \beta \in (0,1)$, $\bar{\Delta} > 0$, and $\bar{m}, \bar{n} \in \mathbb{N}$ be the parameters governing the pathsearch, and let $X_{\bar{\Delta}\bar{n}} := X_0 + \bar{\Delta}\bar{n}\,\mathbb{B}$. Suppose*

(A1) $\mathcal{A}$ is a uniform first-order approximation of F on X_0, i.e., (4.2) holds.

(A2) $\mathcal{A}(x)$ is uniformly Lipschitz invertible near each $x \in X_{\bar{\Delta}\bar{n}}$; i.e., for some δ, ϵ, and $L > 0$ and for each $x \in X_{\bar{\Delta}\bar{n}}$, there exist sets U_x and V_x containing $x + \delta\,\mathbb{B}$ and $F_B(x) + \epsilon\,\mathbb{B}$ respectively, such that $\mathcal{A}(x) \mid_{U_x} : U_x \mapsto V_x$ is Lipschitz invertible of modulus L.

(A3) For each $x \in X_{\bar{\Delta}\bar{n}}$ and $T \in (0,1]$, if $p : [0,T) \mapsto \mathbb{R}^n$ is such that $p(0) = x$ and, for each $t \in [0,T)$, $\mathcal{A}(p(t)) = (1-t)F_B(x)$ and $\mathcal{A}(x)$ is continuously invertible near $p(t)$, then there exists $p(T) := \lim_{t \uparrow T} p(t)$ with $\mathcal{A}(x)(p(T)) = (1-T)F_B(x)$.

Then for any $x^0 \in X_0$, Algorithm PATH produces a sequence $\{x^k\}$ such that either $F_B(x^k) = 0$ for some $k \geq 0$ or the sequence $\{x^k\}$ converges to a zero x^* of F_B at a Q-superlinear rate.

Furthermore, the residuals $F_B(x^k)$ converge to zero, and the sequence of reference values $\{\mathcal{R}_j\}$ converges to zero at an R-linear rate. If for some $c > 0$ the approximation $\mathcal{A}$ satisfies $\|\mathcal{A}(x)(x^*) - F_B(x^*)\| \leq c\|x - x^*\|^2$ on some neighborhood of x^*, the sequence $\{x^k\}$ converges to x^* at a Q-quadratic rate.

Proof Assume that $F_B(x^k) \neq 0$ for each k. We note first that given $x^k \in X_{\bar{\Delta}\bar{n}}$, there exists a unique path $p^k : I \mapsto \mathbb{R}^n$ of largest domain I such that the following hold:

1. $p(0) = x^k$;
2. $I = [0,T]$ for some $T \in (0,1]$;
3. $\forall\, t \in I$, $A_k(p(t)) = (1-t)F_B(x^k)$; and

4. $\forall\, t < T, A_k$ is continuously invertible near $p^k(t)$.

To see this, note that Ralph (1994) has shown that the sets U_x and V_x in (A2) can be assumed to be open. Let $\hat{A}_k$ be the Lipschitz invertible mapping obtained by restricting A_k to the neighborhood U_{x^k} around x^k. (A2), Lemma 14, and (A3) imply the existence of the path described above, where the technical assumption (A3) is used to close the domain of the path.

Thus, the paths required by the algorithm are guaranteed to exist when $\{x^k\} \subset X_{\bar{\Delta}\bar{n}}$. Note that the PATH algorithm will construct these paths, and that the backtracing pathsearch described in Section 4.3 will yield a point which satisfies (NmPs). (See note following definition of (NmPs)).

To see that the algorithm is well defined, we need only show that the sequence of iterates remains in $X_{\bar{\Delta}\bar{n}}$, where the pathsearch is well defined. To do this, we show that the algorithm can take only a limited number of bounded steps before the iterates are forced to return to X_0. It will be convenient to define the index

$$\jmath(k) := \max\,[j \mid \ell(j) \leq k].$$

Thus $\ell(\jmath(k))$ is the largest iteration index not exceeding k at which the merit function has been evaluated. For example,

$$\begin{array}{rcllllllllllll}
k & = & 0 & 1 & 2 & \ldots & & 10 & 11 & \ldots & & 57 & \ldots & \ldots \\
j & = & 0 & & & & & 1 & & & & 2 & & \ldots \\
\ell(j) & = & 0 & & & & & 10 & & & & 57 & & \ldots \\
\jmath(k) & = & 0 & & & \ldots & 0 & 1 & & \ldots & 1 & 2 & 2 & \ldots
\end{array}$$

We use the notation

$$d^k := p^k(T_k) - p^k(0) \tag{4.17}$$

to denote the difference between the initial and terminal points of the path p^k. The d^k above should not be confused with the notation for the search direction d used in the smooth case; rather, $\left\|d^k\right\|$ is the size of a possible d-step.

If the point x^{k+1} is a check point, then it has been generated as the result of an m-step, so that

$$\left\|F_B(x^{k+1})\right\| < \mathcal{R}_{\jmath(k)} \leq \left\|F_B(x^0)\right\|$$

and $x^{k+1} \in X_0$.

If the point x^{k+1} is *not* a check point, then it has been generated as the result of a (bounded) d-step, so that x^{k+1} satisfies

$$x^{k+1} = x^{\ell(\jmath(k))} + \sum_{i=\ell(\jmath(k))}^{k} d^i,$$

where $\|d^i\| \leq \bar{\Delta}$ and $k - \ell(\jmath(k)) < \bar{n}$. Since $x^{\ell(\jmath(k))}$ is a check point, it must be in X_0, so that $x^{k+1} \in X_{\bar{\Delta}\bar{n}}$.

Thus, we have shown that the algorithm is well-defined and that every iterate $x^k \in X_{\bar{\Delta}\bar{n}}$. We now demonstrate the global convergence of our method. We first show that F_B converges to 0 (i.e $\lim_{k\to\infty} \left\|F_B(x^k)\right\| = 0$). This result is used to show convergence of the iterates, and to derive rates for their local convergence.

To show convergence of $\{F_B(x^k)\}$ to zero, the sequence $\{x^k\}$ can be split into two subsequences: $\{x^{\ell(k)}\}$, the points at which a reference value has been defined, and $\{x^{r(k)}\}$, the remainder of the points.

If the sequence $\{x^{r(k)}\}$ is finite, then the algorithm will eventually take only m-steps. Once this point is reached, a pathsearch is performed at each iteration, and there can be no further watchdog steps taken. Ralph (1994) shows that in this case, the residual norms $\left\|F_B(x^k)\right\|$ converge linearly to zero.

Assume then that $\{x^{r(k)}\}$ is an infinite sequence. We show first that for large enough k, $x^{r(k)} \in X_0$. Recall that $x^{r(k)}$ is the result of a d-step, so that $x^{r(k)} = x^{r(k)-1} + d^{r(k)-1}$, where $d^{r(k)-1}$ is bounded as follows:

$$\left\|d^{r(k)-1}\right\| \leq \bar{\Delta}\ \beta^k.$$

Thus, $\lim_{k\to\infty} \left\|d^{r(k)-1}\right\| = 0$. Choose K so that $\left\|d^{r(k)-1}\right\| \leq \hat{\xi}\ \forall\, k \geq K - \bar{n}$, where $\hat{\xi}$ is such that $h(\xi) \leq \dfrac{\xi}{L}\ \ \forall\, \xi \leq \hat{\xi}$. (Recall from Definition 10 that $h(s)$ is $o(s)$.) The Lipschitz invertibility of $\mathcal{A}$ yields

$$\|p(t) - p(0)\| = \left\|\hat{A}_k^{-1}((1-t)F_B(x^k)) - \hat{A}_k^{-1}(F_B(x^k))\right\| \leq Lt \left\|F_B(x^k)\right\|, \qquad (4.18)$$

so that for all $k \leq K - \bar{n}$, we have, by the uniformity of $\mathcal{A}$,

$$\begin{aligned}
\left\| F_B(p(T_{r(k)-1})) \right\| &\leq \left\| A_{r(k)-1}(p(T_{r(k)-1})) \right\| + h(\left\| d^{r(k)-1} \right\|) && \text{by (4.2)} \\
&\leq (1 - T_{r(k)-1}) \left\| F_B(x^{r(k)-1}) \right\| + \frac{1}{L} \left\| d^{r(k)-1} \right\| && \text{by (4.7b)} \\
&\leq (1 - T_{r(k)-1}) \left\| F_B(x^{r(k)-1}) \right\| + T_{r(k)-1} \left\| F_B(x^{r(k)-1}) \right\| && \text{by (4.17),(4.18)} \\
&\leq \left\| F_B(x^{r(k)-1}) \right\|.
\end{aligned}$$

Since the PATH algorithm takes at most $\bar{n}$ d-steps before taking an m-step, and we have shown previously that all the points resulting from m-steps are in X_0, the above result shows that $x^{r(k)} \in X_0$ for $k \geq K$.

We now show that the sequence $\{\mathcal{R}_j\}$ converges linearly to 0. Recall from the algorithm description that the number of consecutive d-steps is bounded above by $\bar{n}$, after which an m-step must occur. Let $\{x^{s(k)}\}$ be the sequence of iterates which have occurred as the result of an m-step, but whose predecessors have occurred as the result of a d-step. The algorithm requires that $x^{s(k)} := p^{s(k)-1}(T_{s(k)-1})$ satisfies

$$\left\| F_B(x^{s(k)}) \right\| \leq (1 - \sigma T_{s(k)-1}) \mathcal{R}_{\jmath(s(k)-1)}. \tag{4.19}$$

Since $x^{s(k)-1}$ is the result of a d-step, $\left\| F_B(x^{s(k)-1}) \right\| \leq \alpha_0$ for large enough k, where α_0 is an upper bound for $\|F_B\|$ on the level set X_0. Since by (A2), $A_{s(k)-1}$ is continuously invertible in an ϵ-neighborhood of $F_B(x^{s(k)-1})$, we can use Lemma 14 to show that

$$T_{s(k)-1} \geq \min\{\frac{\epsilon}{\alpha_0}, 1\},$$

thus bounding $(1 - \sigma T_{s(k)-1})$ away from 1. We now need only show that a result similar to (4.19) holds for $x^{\ell(k)}$ when $x^{\ell(k)}$ is the result of *consecutive* m-steps. This is precisely what Ralph (1994, Theorem 9) has shown in proving convergence for his algorithm; this and (4.19) imply that for large enough k,

$$\left\| F_B(x^{\ell(k)}) \right\| \leq (1 - \sigma \hat{T}) \mathcal{R}_{\jmath(\ell(k)-1)} \tag{4.20}$$

holds for some $\hat{T} \in (0, 1)$. Applying (4.20) and our rule (4.14) for updating the reference values, we have

$$\mathcal{R}_{\jmath(\ell(k+\bar{m}))} \leq (1 - \sigma \hat{T}) \mathcal{R}_{\jmath(\ell(k))},$$

thus demonstrating the R-linear convergence of the reference values at the rate of $(1-\sigma\hat{T})^{\frac{1}{\bar{m}}}$. The entire sequence $\{F_B(x^k)\}$ converges to 0 as well, since for large enough k,

$$\mathcal{R}_{\jmath(\ell(k))} \geq \left\|F_B(x^i)\right\| \quad \text{for } i \geq \ell(k),$$

since all d-steps following iteration $\ell(k)$ result in a decrease in $\|F_B\|$, while all m-steps following $\ell(k)$ result in iterates x^i at which $\mathcal{R}_{\jmath(\ell(k))} > \|F_B(x^i\|$.

Thus, we have established that $\{\left\|F_B(x^k)\right\|\}$ converges to 0. We show now that, after a certain point, the algorithm takes only Newton steps. Let $\gamma := \min\{\epsilon, \frac{\hat{\xi}}{L}\}$, where ϵ is defined in (A2) and $\hat{\xi}$ is such that $h(\xi) \leq \dfrac{\xi(1-\sigma)}{L} \ \ \forall \xi \leq \hat{\xi}$. Let K be chosen so that $\left\|F_B(x^k)\right\| \leq \gamma$ for $k \geq K$. Then for $k \geq K$, the following hold:

$$\left\|F_B(x^k)\right\| \leq \epsilon, \tag{4.21}$$

$$\left\|F_B(x_N^k)\right\| \leq (1-\sigma)\left\|F_B(x^k)\right\|, \tag{4.22}$$

where (4.21) follows directly from the choice of γ. To see (4.22), note that (4.21) and Lemma 14 imply that the PATH algorithm finds the Newton point $x_N^k := p^k(1)$, so that by the uniformity of $\mathcal{A}$,

$$\begin{aligned}
\left\|F_B(x_N^k)\right\| &\leq \left\|A_k(x_N^k)\right\| + h(\left\|d^k\right\|) = h(\left\|d^k\right\|) \\
&\leq h(L\left\|F_B(x^k)\right\|) && \text{by (4.18)} \\
&\leq \frac{L\left\|F_B(x^k)\right\|(1-\sigma)}{L} && \text{by choice of } \gamma \\
&\leq (1-\sigma)\left\|F_B(x^k)\right\|.
\end{aligned}$$

Hence by (4.22), $x^{k+1} = x_N^k$ for $k \geq K$.

We show now that the sequence $\{x^k\}$ is Cauchy. For $k \geq K$,

$$\begin{aligned}
\left\|x^{k+1} - x^k\right\| &= \left\|\hat{A}_k^{-1}(0) - \hat{A}_k^{-1}(F_B(x^k))\right\| \\
&\leq L\left\|F_B(x^k)\right\| \\
&\leq L(1-\sigma)^{k-K}\left\|F_B(x^K)\right\|,
\end{aligned}$$

the last inequality following from (4.22). Choosing $s \geq r \geq K$ implies that

$$
\begin{aligned}
\|x^s - x^r\| &\leq \sum_{k=r}^{\infty} \left\|x^{k+1} - x^k\right\| \\
&\leq \sum_{k=r}^{\infty} \frac{L \left\|F_B(x^K)\right\|}{(1-\sigma)^K} (1-\sigma)^k \\
&= \frac{L \left\|F_B(x^K)\right\|}{(1-\sigma)^K \sigma} (1-\sigma)^r \to 0 \quad \text{as } r \to \infty.
\end{aligned}
$$

This implies convergence of $\{x^k\}$. Let $x^* := \lim_{k\to\infty} x^k$. Since $\{\left\|F_B(x^k)\right\|\} \to 0$, the continuity of F_B implies $F_B(x^*) = 0$.

To see the Q-superlinear rate of convergence for the iterates, note that for some $\bar{K} \geq K$, $k \geq \bar{K}$ implies that $x^* \in x^k + \delta\,\mathbb{B}$ (i.e., x^* is in the range of the inverse of the linearization $\hat{A}_k^{-1}$), and the following applies:

$$
\begin{aligned}
\left\|x^{k+1} - x^*\right\| &= \left\|\hat{A}_k^{-1}(F_B(x^*)) - \hat{A}_k^{-1}(A_k(x^*))\right\| \\
&\leq L \ \|F_B(x^*) - A_k(x^*)\| \qquad (4.23) \\
&\leq L \ h(\left\|x^k - x^*\right\|), \qquad (4.24)
\end{aligned}
$$

where the last inequality depends on the uniformity of $\mathcal{A}$. Since $h(s)$ is $o(s)$, inequality (4.24) shows convergence at a Q-superlinear rate.

If the approximation $\mathcal{A}$ also satisfies the inequality

$$
\left\|\mathcal{A}(x^k)(x^*) - F_B(x^*)\right\| \leq c \left\|x^k - x^*\right\|^2 \qquad (4.25)
$$

for some $c > 0$ and on some neighborhood of x^*, then for large enough k, (4.23) and (4.25) together yield

$$
\left\|x^{k+1} - x^*\right\| \leq cL\left\|x^k - x^*\right\|^2,
$$

so that a quadratic rate of convergence is achieved. □

Note that although Theorem 15 deals with the normal map F_B, the result holds for more general nonsmooth mappings; the restriction to the normal map F_B is made only for the sake of consistency with the rest of the thesis.

Chapter 5

Computational Results

In this chapter, we present computational results obtained from several complementarity problems considered in the literature, using a number of different solution algorithms. In Sections 5.1 and 5.2, the following algorithms are compared:

PATH	The PATH solver described in Chapter 4.
J-N	The classic Josephy-Newton's method, as described by Josephy (1979*a*). The results shown were obtained by running the PATH solver with the options file set to emulate Josephy-Newton's method.
MILES	MILES is a Mixed Inequality and nonLinear Equations Solver developed by Rutherford (1993). This solver is an adaptation of Josephy-Newton's method in which warm start and basis-crashing techniques are used to reduce the number of pivot steps required.
B-DIFF	The B-differentiable equations approach of Harker & Xiao (1990), in which each major iteration involves a linesearch of a direction determined by solving a system of equations.
NE/SQP	Pang & Gabriel (1993) describe a method in which the search direction is determined by solving a quadratic program; this direction is linesearched as well.

In Section 5.3, we compare the PATH solver to solution algorithms proposed by Geiger & Kanzow (1994) and Sellami (1994).

Unless otherwise noted with an asterisk (*), the results for the PATH solver were obtained using default values for all parameters; in no case was the code modified to improve performance for any particular problem. For B-DIFF and NE/SQP we include only results available in the literature (Harker & Xiao 1990, Pang & Gabriel 1993).

5.1 Comparison of PATH to Josephy-Newton and MILES

In this section, we compare the Josephy-Newton method, MILES, and PATH, three solvers available as GAMS solution subsystems. The Josephy-Newton (J-N) solver is obtained as a special case of the PATH algorithm (i.e. no pathsearch is carried out, and a particular choice of initial basis and basic values is made for each subproblem), by using an options file to set options included in the PATH solver for just this purpose. This makes possible a meaningful comparison of solution times, as any differences are the result of the algorithm used and not of the implementation. The MILES solver of Rutherford (1993) which we used for these tests is nearly identical to the one distributed with the GAMS compiler, the difference being some code added to report solution time to the log file. Unless taken from published sources, the results in this chapter were obtained on a Sun SPARCstation 10. All solution times are reported in seconds. In each instance, the CPU time reported is the sum of the user time and the time spent in system mode on behalf of the user's process, each obtained via the `getrusage` system call.

Since each of the above algorithms is available as a GAMS solver, we can easily compare their performance by using them to solve a number of complementarity problems expressed in the GAMS language. We have run each of the three solvers on a total of 57 different input files. The number of variables in each model and the number of nonzeros in its Jacobian are indicated in the tables below by the columns headed n and nnz, respectively. For each model solved, we compare the number of major and minor iterations, the number of function and Jacobian evaluations, and the amount of CPU time required for problem solution. Some of the models are solved from multiple starting points or using different values for model parameters, resulting in additional rows in the tables below. Twelve of the models are general MCP's from the GAMS model library (distributed with GAMS); these models have filename stubs ending in `mcp` to distinguish them from the other library models. The results for these models are given in Tables 10 and 11. Fifteen of the models are MPSGE models (Rutherford 1994*a*) also taken from the GAMS model library; these models have filename stubs ending in `mge`. The results for these models are given in Tables 12 through 15. The remaining 30 models are taken from MCPLIB, and are described in Chapter 3. The results for these models are given in Tables 16 and 23.

Some of the models solved in this section contain solve statements for which no solution is intended, or for which the solution process is trivial. For example, some of the MPSGE models use a solve statement to obtain a function value in order to calibrate the model. The results for these solves have been omitted, so that the initial points for each model are not numbered consecutively.

For the `tobin` model, the default minor iterations limit of 1000 was reached several times, due to the cycling of bases observed to occur during some of the major iterations for this problem. This is why the solution time is so high for the second run of the PATH solver on this model. For the hydrocarbon refinery problem (`hydroc20`), the memory size and initial reference factor for the nonmonotone stabilization technique were both set to unity, so that the watchdog technique would return to the initial iterate to perform a damped step. The initial iterate for this problem is a very good estimate of the solution, as required by the highly nonlinear nature of this problem. The `dmcmge` model was run with the m-step frequency set to one to prevent function evaluation errors.

The initial points used to obtain the data in Tables 10 through 23 can be obtained from the GAMS models, as can the settings for most model parameters. A GAMS user may wish to adjust many of these parameters to effect a model's size or difficulty of solution. The models in Tables 22 and 23 make use of a discretization process whose mesh size can be easily changed. For the obstacle and bratu problems, we have chosen a mesh size of 75×75, resulting in a problem with 5625 variables. The bounds for the obstacle problem varied over the runs, and are given in the GAMS model; for the bratu runs, we chose $\lambda = 6$ and set the lower and upper bounds at 0 and 4, respectively. For the optimal control problem of Bertsekas (1982), we have chosen a mesh of size 1000, resulting in a problem in 5000 variables. For the ELQP models from optimal control (`opt_cont`N), we varied the mesh and kept the number of controls and states constant, so that model `opt_cont`N has a total of $32(N+1)$ variables. Thus, the largest problem solved is one having 16,384 variables, and 278,272 nonzeros in its Jacobian.

It is clear from these results that the PATH solver represents an increase in robustness over the undamped method of Josephy. In addition, the PATH solver requires considerably

Table 10: Comparison of Major and Minor Iteration Counts - GAMSLIB

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
242	1380	cammcp 1	fail	4	4	fail	37	4
232	1321	ers82mcp 1	fail	5	5	fail	5	5
262	2532	gemmcp 1	fail	1	1	fail	1	1
262	2532	gemmcp 2	fail	0	0	fail	0	0
262	2536	gemmcp 3	fail	5	5	fail	5	5
262	2532	gemmcp 4	fail	4	5	fail	4	5
262	2532	gemmcp 5	fail	4	4	fail	4	4
43	356	hansmcp 1	22	4	27	871	41	97
32	100	harkmcp 1	8	6	8	242	58	34
32	103	harkmcp 2	4	4	4	124	4	4
32	103	harkmcp 3	4	4	4	132	4	4
92	329	harkmcp 4	4	4	4	350	337	90
78	346	kormcp 1	fail	3	3	fail	3	3
350	1338	mr5mcp 1	fail	6	6	fail	6	6
6	16	oligomcp 1	5	7	5	35	7	5
11	24	transmcp 1	1	1	1	9	1	9
11	27	transmcp 2	1	0	0	9	0	0
11	24	transmcp 3	1	1	1	9	1	1
11	27	transmcp 4	5	5	5	55	15	6
6	24	two3mcp 1	5	5	5	35	5	5
		two3mcp 2	4	4	4	28	4	4
125	636	vonthmcp 1	fail	13	11	fail	110	184
6	20	wallmcp 1	fail	2	2	fail	2	2

Table 11: Comparison of Func/Jac. Evals & Solution Times - GAMSLIB

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
cammcp 1	fail	5	fail	5	fail	0.78	0.41
ers82mcp 1	fail	6	fail	6	fail	0.71	0.52
gemmcp 1	fail	2	fail	2	fail	0.30	0.27
gemmcp 2	fail	1	fail	1	fail	0.03	0.05
gemmcp 3	fail	6	fail	6	fail	1.17	1.13
gemmcp 4	fail	6	fail	6	fail	0.96	1.16
gemmcp 5	fail	5	fail	5	fail	1.00	0.93
hansmcp 1	23	28	23	28	1.06	0.15	0.40
harkmcp 1	9	9	9	9	0.15	0.17	0.07
harkmcp 2	5	5	5	5	0.09	0.07	0.03
harkmcp 3	5	5	5	5	0.09	0.07	0.03
harkmcp 4	5	5	5	5	0.69	0.62	0.20
kormcp 1	fail	4	fail	4	fail	0.14	0.07
mr5mcp 1	fail	7	fail	7	fail	1.39	0.97
oligomcp 1	6	6	6	6	0.02	0.10	0.02
transmcp 1	2	2	2	2	0.02	0.02	0.01
transmcp 2	2	1	2	1	0.01	0.01	0.01
transmcp 3	2	2	2	2	0.01	0.02	0.01
transmcp 4	6	6	6	6	0.03	0.08	0.02
two3mcp 1	6	6	6	6	0.02	0.06	0.03
two3mcp 2	5	5	5	5	0.03	0.05	0.03
vonthmcp 1	fail	12	fail	12	fail	0.96	0.68
wallmcp 1	fail	3	fail	3	fail	0.02	0.02

Table 12: Comparison of Major and Minor Iteration Counts - MPSGE

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
47	316	cafemge 2	6	11	6	289	15	8
129	1731	cammge 1	fail	4	4	fail	4	4
219	1601	co2mge 1	fail	9	17	fail	60	52
170	1728	dmcmge 1	fail	9	19*	fail	462	376*
194	1532	dmcmge 2	fail	7	6	fail	9	8
114	941	etamge 1	fail	11	21	fail	29	101
183	2871	finmge 2	4	6	4	838	11	212
153	2806	finmge 3	3	3	3	506	3	3
183	2853	finmge 4	4	4	4	821	13	11
153	2796	finmge 5	4	4	4	688	4	4
323	7036	gemmge 2	4	4	4	1364	1328	5
323	7012	gemmge 3	9	5	9	3105	668	355
323	7012	gemmge 4	5	5	5	1732	701	692
323	6962	gemmge 5	5	22	5	1728	3513	360
43	793	hansmge 1	3	3	3	114	42	40
9	81	harmge 2	3	3	3	30	3	3
		harmge 3	5	5	5	50	5	5
		harmge 4	5	5	5	50	5	5
9	81	kehomge 1	7	7	7	70	7	7
		kehomge 2	23	7	7	228	7	7
		kehomge 3	23	6	6	228	6	6

Table 13: Comparison of Func/Jac. Evals & Solution Times - MPSGE

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
cafemge 2	7	7	7	7	1.63	2.40	1.35
cammge 1	fail	5	fail	5	fail	0.56	0.62
co2mge 1	fail	18	fail	18	fail	1.57	1.49
dmcmge 1	fail	300*	fail	20*	fail	3.17	9.65*
dmcmge 2	fail	7	fail	7	fail	1.20	0.83
etamge 1	fail	52	fail	22	fail	1.11	1.77
finmge 2	5	5	5	5	6.31	4.46	1.73
finmge 3	4	4	4	4	3.77	0.96	0.76
finmge 4	5	5	5	5	6.09	1.31	0.92
finmge 5	5	5	5	5	4.64	1.28	0.95
gemmge 2	5	5	5	5	13.84	11.21	1.75
gemmge 3	10	10	10	10	31.30	6.87	5.79
gemmge 4	6	6	6	6	17.96	8.20	6.69
gemmge 5	6	6	6	6	18.65	130.00	4.60
hansmge 1	4	4	4	4	0.22	0.21	0.17
harmge 2	4	4	4	4	0.04	0.04	0.03
harmge 3	6	6	6	6	0.05	0.08	0.04
harmge 4	6	6	6	6	0.05	0.06	0.05
kehomge 1	8	8	8	8	0.07	0.12	0.08
kehomge 2	24	8	24	8	0.21	0.17	0.09
kehomge 3	24	7	24	7	0.22	0.11	0.07

Table 14: Comparison of Major and Minor Iteration Counts - MPSGE

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
14	148	sammge 2	4	4	4	76	4	4
14	150	sammge 3	5	5	5	89	5	5
14	170	sammge 4	5	5	5	107	5	5
18	324	scarfmge 1	6	6	6	110	24	24
20	348	scarfmge 2	6	6	6	126	6	6
20	348	scarfmge 3	8	8	8	167	11	11
20	348	scarfmge 4	9	9	9	188	15	13
		shovmge 2	4	3	4	44	3	4
10	100	shovmge 3	5	5	5	55	5	5
		shovmge 4	5	5	5	69	5	5
5	25	unstmge 1	21	7	13	124	12	16
80	842	vonthmge 1	13	14	13	979	81	74

Table 15: Comparison of Func/Jac. Evals & Solution Times - MPSGE

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
sammge 2	5	5	5	5	0.06	0.08	0.06
sammge 3	6	6	6	6	0.07	0.09	0.05
sammge 4	6	6	6	6	0.09	0.09	0.06
scarfmge 1	7	7	7	7	0.11	0.17	0.11
scarfmge 2	7	7	7	7	0.14	0.15	0.13
scarfmge 3	9	9	9	9	0.16	0.21	0.14
scarfmge 4	10	10	10	10	0.18	0.24	0.16
shovmge 2	5	5	5	5	0.05	0.05	0.04
shovmge 3	6	6	6	6	0.05	0.08	0.05
shovmge 4	6	6	6	6	0.06	0.08	0.05
unstmge 1	22	17	22	14	0.11	0.09	0.07
vonthmge 1	14	14	14	14	1.89	1.00	0.54

Table 16: Comparison of Major and Minor Iteration Counts - MCPLIB

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
15	60	bertsekas 1	4	36	4	55	49	17
		bertsekas 2	4	4	4	60	6	6
		bertsekas 3	12	11	12	150	25	28
13	169	choi 1	4	4	4	56	4	4
20	149	colvdual 1	4	3	4	69	21	28
		colvdual 2	4	3	4	67	21	21
15	99	colvnlp 1	4	3	4	41	18	15
		colvnlp 2	4	3	4	41	16	17
101	10200	ehl_kost 1	fail	5	5	fail	10	8
		ehl_kost 2	fail	7	7	fail	41	37
		ehl_kost 3	fail	4	4	fail	5	5
		ehl_kost 4	fail	4	4	fail	4	4
		ehl_kost 5	fail	5	5	fail	5	5
5	25	gafni 1	3	3	3	12	6	5
		gafni 2	3	3	3	12	5	5
		gafni 3	4	4	4	17	10	6
14	116	hanskoop 1	15	4	15	164	20	29
		hanskoop 3	15	6	15	164	26	29
		hanskoop 5	5	5	5	59	21	19
		hanskoop 7	6	5	6	70	21	20
		hanskoop 9	13	fail	13	133	fail	25
99	740	hydroc20 1		fail	13*		fail	21*

Table 17: Comparison of Func/Jac. Evals & Solution Times - MCPLIB

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
bertsekas 1	5	5	5	5	0.06	1.00	0.03
bertsekas 2	5	5	5	5	0.04	0.07	0.04
bertsekas 3	13	13	13	13	0.10	0.17	0.08
choi 1	5	5	5	5	1.20	2.21	2.10
colvdual 1	5	5	5	5	0.05	0.07	0.05
colvdual 2	5	5	5	5	0.05	0.07	0.04
colvnlp 1	5	5	5	5	0.04	0.06	0.03
colvnlp 2	5	5	5	5	0.03	0.06	0.03
ehl_kost 1	fail	6	fail	6	fail	4.74	4.21
ehl_kost 2	fail	8	fail	8	fail	6.95	6.05
ehl_kost 3	fail	5	fail	5	fail	3.92	3.50
ehl_kost 4	fail	5	fail	5	fail	3.90	3.50
ehl_kost 5	fail	6	fail	6	fail	4.82	4.24
gafni 1	4	4	4	4	0.02	0.04	0.02
gafni 2	4	4	4	4	0.03	0.04	0.03
gafni 3	5	5	5	5	0.03	0.06	0.03
hanskoop 1	16	16	16	16	0.10	0.07	0.07
hanskoop 3	16	16	16	16	0.10	0.10	0.08
hanskoop 5	6	6	6	6	0.04	0.08	0.03
hanskoop 7	7	7	7	7	0.05	0.08	0.03
hanskoop 9	14	14	14	14	0.08	fail	0.07
hydroc20 1		16*		14*			0.62*

Table 18: Comparison of Major and Minor Iteration Counts - MCPLIB

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
4	16	josephy 1	6	fail	6	16	fail	7
		josephy 2	4	fail	10	12	fail	16
		josephy 3	10	10	21	30	12	30
		josephy 4	3	fail	3	9	fail	4
		josephy 5	3	3	3	9	4	3
		josephy 6	4	4	14	12	6	33
4	16	kojshin 1	5	fail	5	14	fail	6
		kojshin 2	4	4	4	12	6	6
		kojshin 3	10	10	53	30	12	86
		kojshin 4	1	1	3	3	1	3
		kojshin 5	1	1	3	3	2	3
		kojshin 6	5	6	8	15	14	18
3	9	mathinum 1	fail	5	6	fail	7	7
		mathinum 2	4	4	4	16	4	4
		mathinum 3	fail	7	11	fail	11	17
		mathinum 4	5	5	5	20	5	5
4	11	mathisum 1	6	4	6	24	4	6
		mathisum 2	4	4	4	16	4	4
		mathisum 3	fail	7	9	fail	12	23
		mathisum 4	5	5	5	20	5	5
31	195	methan08 1	fail	4	4	fail	4	4
10	100	nash 1	6	6	6	66	6	6
		nash 2	6	6	6	66	6	6

Table 19: Comparison of Func/Jac. Evals & Solution Times - MCPLIB

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
josephy 1	7	7	7	7	0.02	fail	0.03
josephy 2	5	15	5	11	0.02	fail	0.03
josephy 3	11	22	11	22	0.04	0.10	0.06
josephy 4	4	4	4	4	0.01	fail	0.01
josephy 5	4	4	4	4	0.01	0.03	0.02
josephy 6	5	15	5	15	0.02	0.04	0.04
kojshin 1	6	6	6	6	0.02	fail	0.02
kojshin 2	5	5	5	5	0.02	0.04	0.02
kojshin 3	11	59	11	54	0.03	0.09	0.11
kojshin 4	2	4	2	4	0.02	0.02	0.01
kojshin 5	2	4	2	4	0.02	0.01	0.02
kojshin 6	6	9	6	9	0.03	0.08	0.03
mathinum 1	fail	10	fail	7	fail	0.05	0.02
mathinum 2	5	5	5	5	0.02	0.03	0.02
mathinum 3	fail	20	fail	12	fail	0.07	0.04
mathinum 4	6	6	6	6	0.03	0.04	0.01
mathisum 1	7	7	7	7	0.02	0.04	0.02
mathisum 2	5	5	5	5	0.02	0.03	0.02
mathisum 3	fail	24	fail	10	fail	0.07	0.05
mathisum 4	6	6	6	6	0.03	0.05	0.02
methan08 1	fail	5	fail	5	fail	0.11	0.06
nash 1	7	7	7	7	0.07	0.12	0.08
nash 2	7	7	7	7	0.07	0.11	0.07

Table 20: Comparison of Major and Minor Iteration Counts - MCPLIB

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
42	142	pies 1	2	2	2	126	5	64
16	188	powell 1	8	7	8	122	7	12
		powell 2	5	5	5	29	13	17
		powell 3	7	7	7	51	7	13
		powell 4	6	6	6	34	6	18
8	47	powell_mcp 1		6	6		6	6
		powell_mcp 2		7	7		7	7
		powell_mcp 3		8	8		8	8
		powell_mcp 4		7	7		7	7
13	86	scarfanum 1	4	4	4	74	7	21
		scarfanum 2	5	5	5	93	15	29
		scarfanum 3	4	4	4	75	16	9
14	96	scarfasum 1	4	4	4	71	9	9
		scarfasum 2	3	7	3	52	12	9
		scarfasum 3	4	4	4	77	16	13
39	323	scarfbnum 1	4	4	4	147	39	37
		scarfbnum 2	4	4	4	147	39	37
40	575	scarfbsum 1	3	3	3	102	34	36
		scarfbsum 2	3	3	3	106	34	40
27	84	sppe 1	7	7	7	185	147	16
		sppe 2	5	5	5	134	90	7
42	202	tobin 1	9	7	9	174	51	31
		tobin 2	9	fail	21	170	fail	5338

Table 21: Comparison of Func/Jac. Evals & Solution Times - MCPLIB

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
pies 1	3	3	3	3	0.11	0.06	0.05
powell 1	9	9	9	9	0.10	0.15	0.07
powell 2	6	6	6	6	0.04	0.14	0.05
powell 3	8	8	8	8	0.06	0.17	0.07
powell 4	7	7	7	7	0.04	0.15	0.07
powell_mcp 1	fail	7	fail	7	fail	0.07	0.03
powell_mcp 2	fail	8	fail	8	fail	0.09	0.04
powell_mcp 3	fail	9	fail	9	fail	0.09	0.04
powell_mcp 4	fail	8	fail	8	fail	0.08	0.04
scarfanum 1	5	5	5	5	0.06	0.08	0.04
scarfanum 2	6	6	6	6	0.05	0.10	0.06
scarfanum 3	5	5	5	5	0.05	0.09	0.04
scarfasum 1	5	5	5	5	0.06	0.08	0.04
scarfasum 2	4	4	4	4	0.03	0.14	0.04
scarfasum 3	5	5	5	5	0.07	0.09	0.04
scarfbnum 1	5	5	5	5	0.20	0.15	0.06
scarfbnum 2	5	5	5	5	0.17	0.16	0.06
scarfbsum 1	4	4	4	4	0.28	0.16	0.09
scarfbsum 2	4	4	4	4	0.23	0.16	0.08
sppe 1	8	8	8	8	0.13	0.22	0.04
sppe 2	6	6	6	6	0.09	0.14	0.03
tobin 1	10	10	10	10	0.16	0.20	0.08
tobin 2	10	24	10	22	0.18	fail	3.03

Table 22: Comparison of Major and Minor Iteration Counts - MCPLIB

size		problem	major			minor		
n	nnz		J-N	MILES	PATH	J-N	MILES	PATH
5000	16992	bert_oc 1	1	12	1	1190	74	554
		bert_oc 2	1	13	1	1190	115	591
		bert_oc 3	1	fail	1	1671	fail	671
		bert_oc 4	1	1	1	1671	331	671
5625	28125	bratu 1	fail	fail	6	fail	fail	6
5625	28125	obstacle 1	fail	fail	1	fail	fail	1329
		obstacle 2	fail	fail	2	fail	fail	3505
		obstacle 3	fail	fail	2	fail	fail	3365
		obstacle 4	fail	fail	2	fail	fail	6014
		obstacle 5	fail	fail	2	fail	fail	1456
		obstacle 6	fail	fail	1	fail	fail	2077
		obstacle 7	fail	fail	2	fail	fail	4255
		obstacle 8	fail	fail	1	fail	fail	1942
1024	17152	opt_cont	1	1	1	703	727	375
4096	69376	opt_cont	1	2	1	2815	2565	1527
8192	139008	opt_cont	1	fail	1	5635	fail	3063
16384	278272	opt_cont	1	fail	2	11265	fail	6135

Table 23: Comparison of Func/Jac. Evals & Solution Times - MCPLIB

problem	func evals		Jac. evals		time (sec)		
	J-N	PATH	J-N	PATH	J-N	MILES	PATH
bert_oc 1	2	2	2	2	222.10	185.90	32.98
bert_oc 2	2	2	2	2	219.36	200.60	46.71
bert_oc 3	2	2	2	2	365.09	fail	36.65
bert_oc 4	2	2	2	2	370.23	136.40	48.71
bratu 1	fail	7	fail	7	fail	fail	77.33
obstacle 1	fail	2	fail	2	fail	fail	84.29
obstacle 2	fail	3	fail	3	fail	fail	1087.49
obstacle 3	fail	3	fail	3	fail	fail	570.35
obstacle 4	fail	3	fail	3	fail	fail	537.11
obstacle 5	fail	3	fail	3	fail	fail	488.73
obstacle 6	fail	2	fail	2	fail	fail	815.08
obstacle 7	fail	3	fail	3	fail	fail	1021.87
obstacle 8	fail	2	fail	2	fail	fail	742.92
opt_cont31	2	2	2	2	33.00	44.40	9.34
opt_cont127	2	2	2	2	725.01	2573.00	196.88
opt_cont255	2	2	2	2	4052.23	fail	1084.31
opt_cont511	2	3	2	3	25686.77	fail	6348.34

less solution time in many cases, due to the smaller number of pivots it performs. This is the result of the warm start taken by the PATH solver on the subproblems; in most cases, the optimal basis remains the same over the last few subproblems, so that only one pivot step is required for each. There are a number of problems, however, for which the PATH solver performs no better than Josephy-Newton's method, especially on some of the smaller problems in which the pivots are very inexpensive. One of the design goals of the PATH solver was to always perform at least as well as the Josephy-Newton method; there are a number of instances in which an improvement on it is not possible.

The difference in performance between the PATH solver and MILES is not a great one, especially if the results from the larger problems in Tables 22 and 23 are discounted. A comparison of iteration counts reveals much similarity; neither solver consistently outperforms the other. The robustness of the two solvers is quite similar as well, as neither of them fail on many of the problems tested. However, the solution time required by the PATH solver is frequently less than that required by MILES, although there are exceptions to this. For the larger problems, it is quite clear that the PATH solver is the method of choice. MILES failed on a large number of these runs, while its solution times compare poorly with those of the PATH solver when both algorithms compute a solution.

5.2 Comparison of PATH to B-DIFF and NE/SQP

Since the B-DIFF algorithm of Harker & Xiao (1990) and the NE/SQP algorithm of Pang & Gabriel (1993) are not publicly available, it is not possible to obtain a meaningful comparison between the solution times for these methods and those for the PATH solver. However, the above references do contain results regarding the number of major iterations required for the solution of several problems. These problems have been coded in GAMS, and the models solved using the PATH solver, thus allowing a comparison between the number of major iterations required for solution, as shown in Table 24. The voids in this table indicate that data for a particular problem and start point were not available. Start points for these problems can be obtained from Harker & Xiao (1990) and Pang & Gabriel (1993), respectively.

The results of Table 24 indicate that the PATH solver compares favorably with the B-DIFF and NE/SQP algorithms, although there are instances where the latter methods

Table 24: Comparison of Major Iteration Counts - PATH, B-DIFF & NESQP

problem		Major		
		PATH	B-DIFF	NE/SQP
josephy	1	6		7
	2	10	10	
	3	21	15	
	4	3	7	
	5	3	7	
	6	14	9	
hanskoop	1	15		10
	3	15		11
mathiesen	1	5		11
	2	5		3
	3	6	5	
	4	4	4	
	5	10	6	
nash	1	6	10	
	2	6	13	
scarf	1	4	5	6
	2	3	5	
	3	4		28
	4	3		3
sppe	1	7	10	
	2	5	10	
tobin	1	9	16	20
	2	21	15	20

require fewer major iterations. We note once again that the code for the PATH solver was not modified in order to solve any of these problems, and was run with default parameters except where indicated. This does not appear to be the case for B-DIFF and NE/SQP, as Harker & Xiao (1990) and Pang & Gabriel (1993) indicate that certain modifications to their codes as applied to some of the problems were used in order to achieve the results given.

5.3 Comparison of PATH to Other Techniques

Geiger & Kanzow (1994) have implemented an algorithm which solves NCP via a reformulation as an unconstrained minimization problem and present computational results on finding the KKT points of 4 constrained optimization problems. In Table 25, we compare results using the PATH solver to their results, obtained from Tables 3 through 6 of (Geiger & Kanzow 1994). We have taken the results for m=5 from each of these tables. The column in Table 25 headed G-K contains results obtained by Geiger and Kanzow using their reformulation, while the column headed M-S contains results obtained by the same authors, using the same code and a different but similar reformulation of the NCP due to Mangasarian & Solodov (1993). The asterisks (*) indicate convergence to a stationary point that is not a solution for the NCP.

The results in Table 25 show that the PATH solver requires many fewer iterations to solve these problems than does the minimization approach described above. The results reported by Geiger & Kanzow are for a limited-memory BFGS scheme, so that one would expect their method to require more major iterations than a Newton method such as the PATH solver. However, the minor iteration counts for the PATH solver are also very low, so that we can conclude that the PATH solver has outperformed Kanzow's technique for the problems included in Table 25. Due to the small size and limited number of these problems, it is not possible to compare the two methods conclusively, although the PATH solver appears to be more robust than Kanzow's technique.

In his Ph.D. thesis, Sellami (1994) gives results for a continuation method for normal maps as applied to a number of complementarity problems. In Table 26, we compare the results from his thesis with those obtained from the PATH solver. Again, it is only possible to compare the major iteration counts, although the minor iterations required by the path solver are also given.

Table 25: Iteration Counts - PATH and Kanzow's technique

problem		Minor	Major		
		PATH	PATH	G-K	M-S
hs34	1	12	6	114	101
	2	6	5	107	106
	3	12	6	101	100
	4	19	6	110	98
	5	12	6	114	112
hs35	1	1	1	30	30
	2	1	1	43	9*
	3	1	1	44	10*
	4	2	1	43	14*
	5	5	1	53	17*
hs66	1	15	8	39	35
	2	10	5	64	44
	3	10	5	43	45
	4	14	7	61	62
	5	22	10	62	41
hs76	1	5	1	47	42
	2	5	1	48	33
	3	4	1	102	27*
	4	4	1	41	40
	5	4	1	50	42

Table 26: Iteration Counts - PATH and Sellami's technique

problem		Minor	Major	
		PATH	PATH	Sellami
prob1	1	23	18	8
	2	15	13	55
	3	4	3	16
	4	15	13	116
	5	4	4	20
prob2	1	5	5	10
	2	12	12	23
	3	13	13	36
prob3	1	9	9	18
	2	9	9	47
prob4	1	7	6	12
	2	41	40	24
prob5	1	6	5	25
	2	6	5	33
prob6	1	6	5	69
	2	6	5	30
	3	6	5	33
nash-10	1	6	6	200
	2	6	6	175

Almost without exception, the PATH solver requires fewer iterations to solve the problems in Table 26 than does the continuation method of Sellami. Since each major iteration of Sellami's method involves a QR factorization, we can expect the PATH solver to solve these problems much more quickly. The difference in speed will become more pronounced as problems of a larger size are solved, especially if these problems are sparse, due to the sparse matrix routines used by the PATH solver. Both techniques appear to be equally robust.

5.4 Conclusions

We have designed the PATH solver to be both fast and robust, in order minimize both solution time and the number of failures encountered in problem solution. The data presented in this chapter indicate that we have achieved these twin objectives. The stabilization techniques incorporated into the PATH solver have resulted in a significant reduction in the failure rate as compared to the undamped method of Josephy. At the same time, these techniques have not slowed down the solver on problems for which Josephy-Newton's method performs well. In fact, the parameterized path construction method serves to decrease the number of minor iterations required, thereby speeding the solution process. A comparison to the MILES solver does not yield as dramatic a difference, especially in robustness, but we can conclude that the PATH solver is somewhat faster, in general, and both faster and more reliable for the larger problems solved. A comparison to the algorithms considered in Section 5.3 yields the same conclusion.

Chapter 6

Preprocessing and Other Extensions

In the preceding chapters, we have described the core of a system for effectively formulating and solving the MCP. In this chapter, we describe the results of our attempts at improving the computational results achieved and indicate directions for future research based on the content of this thesis.

6.1 Preprocessing

We noted at the close of Chapter 5 that the PATH solver compares favorably with the other methods considered there. However, the data from Tables 22 and 23 indicate that the PATH solver performs a large number of pivot steps when solving large problems. This is to be expected: the pivotal techniques employed by the PATH solver place it among those QP solvers which use an active set strategy. For solvers that add or subtract one constraint at a time from the active set, the number of pivots required is bounded below by the difference in size between the initial and optimal set of active constraints. This bound can be expected to grow with the size of the problem, as is seen from the iteration counts given in Table 22. In order to reduce the number of pivots required, the initial iterate can be adjusted so that it corresponds more closely to the active set at the solution to the problem. We will call such an adjustment a preprocessing step. In this section, we will consider a number of different preprocessing techniques.

A simple approach to this problem is to use the power of the GAMS language to compute an initial point satisfying as many of the model constraints as possible. For example, the variables in the optimal control models are of two types, control variables and state variables. The state variable are completely determined by the control variables, but are not substituted out of the model, since doing so would result in a completely dense problem. Given the initial values for the control variables, it is a simple matter to compute values for the state variables such that all the equality constraints are satisfied. This was done, and resulted in a decrease in the residual norm at the initial iterate. However, there was no corresponding reduction in the number of pivot steps required to solve the problem. Since the variables corresponding to the equality constraints of the model are all free, they remain in the basis regardless of whether their corresponding constraints are satisfied as equalities or not.

A more algorithmic approach to preprocessing involves the projected gradient techniques studied by Bertsekas & Gafni (1982) and Calamai & Moré (1987). Sufficient conditions for the convergence of such a method to a solution of an MCP are given by Bertsekas & Tsitsiklis (1989, Proposition 5.4) and include the Lipschitz continuity and strong monotonicity of F. Convergence results for a number of projection methods based on a gap function derived from the variational inequality problem are given by Fukushima (1992) and Larsson & Patriksson (1994).

We have implemented a projected gradient type preprocessing step for the initial point supplied to the PATH solver. In this method, an iterate $z^k \in B$ is replaced by a new point $z^{k+1} := z(\alpha)$, where

$$z(\alpha) := \pi_B(z^k - \alpha D^{-1} F(z^k)), \tag{6.1}$$

D is a diagonal positive definite matrix chosen to approximate the diagonal of the Jacobian, and α is chosen via an Armijo linesearch technique so as to reduce $\|F_B(\cdot)\|$. At each trial step α, a simple projection step is required to obtain $z(\alpha)$. Note that in order to gauge the acceptability of the projected points $z(\alpha)$, we are using $\|F_B\|$, a mapping defined over all of $\mathbb{R}^n$. To do so, we evaluate $\|F_B\|$ at the points

$$x(\alpha) := \arg\min_x \{\|F_B(x)\| \mid z(\alpha) = \pi_B(x)\}, \tag{6.2}$$

where $z(\alpha)$ is given in (6.1). The computation of $x(\alpha)$ in (6.2) is trivial and can be done in a simple loop, the same loop used to calculate $\|F_B(x(\alpha))\|$. The preprocessing is terminated when either the active set is not changed from one iteration to the next or the decrease in $\|F_B(x)\|$ becomes less than a fixed fraction of the maximum decrease achieved over all previous linesearch steps. The results obtained using this technique are mixed. Table 27 compares the number of pivots and function evaluations and the solution time required to solve some of the larger models via the PATH solver both with and without the projected gradient preprocessing step. While this technique was useful in reducing the number of pivots required to solve some of the obstacle and optimal control problems, it was less helpful in solving the Bertsekas optimal control or bratu problems. In many cases, the projection step failed to reduce $\|F_B(x)\|$, resulting in no basis change and no reduction in the number of pivot steps required for solution. This is consistent with the theoretical results for this type of method; the functions F for the optimal control problems are not strongly monotone, so that $F(z^k)$ is not necessarily a descent direction.

Table 27: Performance Metrics - PATH & Projected Gradient Preprocessing

size		problem	pivots		func. evals		time (sec)	
n	nnz		PATH	PGP	PATH	PGP	PATH	PGP
5000	16992	bert_oc 1	554	554	2	16	32.98	35.77
		bert_oc 2	591	811	2	16	46.71	67.85
		bert_oc 3	671	671	2	16	36.65	45.02
		bert_oc 4	671	331	2	16	48.71	21.80
5625	28125	bratu 1	6	10	7	22	77.33	95.16
5625	28125	obstacle 1	1329	991	2	21	84.29	77.28
		obstacle 2	3505	3505	3	24	1087.49	1082.3
		obstacle 3	3365	2626	3	25	570.35	508.39
		obstacle 4	6014	1859	3	18	537.11	303.17
		obstacle 5	1456	358	3	32	488.73	87.99
		obstacle 6	2077	1804	2	65	815.08	683.36
		obstacle 7	4255	2264	3	9	1021.87	385.05
		obstacle 8	1942	1855	2	14	742.92	690.79
1024	17152	opt_cont	375	94	2	19	9.34	1.93
4096	69376	opt_cont	1527	1271	2	24	196.88	170.31
8192	139008	opt_cont	3063	2551	2	41	1084.31	925.13
16384	278272	opt_cont	6135	5114	3	42	6348.34	5466.9

The failure of the projected gradient technique discussed above has led to the implementation of a projected Newton technique. Motivated by the work of Bertsekas (1982), this technique computes a Newton direction for a reduced system and uses this direction in a linesearch similar to that of (6.1). The Newton direction for the reduced system is essentially the same direction used in the initial pivot step of the path construction phase of the PATH solver. While the path construction algorithm stops at a boundary and recomputes the direction (i.e. performs a pivot step), the projected Newton technique takes a damped step and projects back onto the feasible set. Our projected Newton step is also similar to the step taken by the B-DIFF algorithm of Harker & Xiao (1990).

We use the index sets $\mathcal{A}$ and $\mathcal{I}$ to indicate box constraints active and inactive at the solution of an approximation defined at z^k. These index sets are defined as follows:

$$\begin{aligned}\mathcal{A} &:= \{i \mid \ell_i = z_i,\ F_i(z^k) \geq 0\} \bigcup \{i \mid u_i = z_i,\ F_i(z^k) \leq 0\} \\ \mathcal{I} &:= \{i \mid i \notin \mathcal{A}\}\end{aligned}$$

The reduced system

$$F'_{\mathcal{I}\mathcal{I}}(z^k) d_{\mathcal{I}} = F_{\mathcal{I}}(z^k) \tag{6.3}$$

computes the nonzero part of the search direction d. Assuming a reordering of the variables, the new iterate $z^{k+1} := z(\alpha)$, where

$$z(\alpha) := \pi_B\left(\begin{bmatrix} z^k_{\mathcal{I}} \\ z^k_{\mathcal{A}} \end{bmatrix} - \alpha \begin{bmatrix} d_{\mathcal{I}} \\ 0 \end{bmatrix}\right). \tag{6.4}$$

Again, α is chosen via a linesearch to reduce $\|F_B(x(\alpha))\|$, where $x(\alpha)$ is chosen as in (6.2). The new iterate z^{k+1} leads to a new choice of index sets $\mathcal{A}$ and $\mathcal{I}$.

In computational tests of the projected Newton preprocessor, the preprocessing phase was terminated when the membership of the index sets for successive iterations differed by less than 10 indices. At this point, the most recent iterate was used as the initial iterate for the PATH solver. For problems with fewer than 10 variables, no preprocessing step was attempted, since pivots for these problems are relatively inexpensive. Also, the preprocessing phase was terminated if the Newton direction did not result in a decrease in $\|F_B\|$ or if the Newton step could not be computed due to rank deficiency in the reduced system (6.4).

The data in Tables 28 through 34 give the time and number of projected Newton steps, major and minor iterations, and function and gradient evaluations required for problem solution via the PATH solver with projected Newton preprocessing. These tables show a significant decrease in the solution time required for the larger models over that required by the pivotal methods considered in Chapter 5, without sacrificing the robustness of the PATH solver on the smaller, more complex problems. The difference in solution time required on the larger problems is illustrated in Figure 14.

Our projected Newton preprocessor is similar to the B-differentiable equations approach for solving $F_{\mathbb{R}^n_+} = 0$ taken by Harker & Xiao (1990). In the latter approach, an iterate x is used to define the index sets

$$\begin{aligned}\mathcal{P} &:= \{i \mid x_i > 0\}\\ \mathcal{D} &:= \{i \mid x_i = 0\}\\ \mathcal{N} &:= \{i \mid x_i < 0\}\end{aligned}$$

In their implementation, Harker & Xiao chose the point x so that the set of degenerate indices $\mathcal{D} = \emptyset$. In this case, the Newton direction for $F_{\mathbb{R}^n_+}$ is given by

$$\begin{bmatrix} F'_{\mathcal{P}\mathcal{P}}(z^k) & 0 \\ F'_{\mathcal{P}\mathcal{N}}(z^k) & -I \end{bmatrix} \begin{bmatrix} d_{\mathcal{P}} \\ d_{\mathcal{N}} \end{bmatrix} = \begin{bmatrix} r_{\mathcal{P}} \\ r_{\mathcal{N}} \end{bmatrix}, \tag{6.5}$$

where $r := F_{\mathbb{R}^n_+}(x^k)$ is the residual at the current point. This search direction is used in a linesearch step which seeks to reduce $\left\| F_{\mathbb{R}^n_+}(x - \alpha d) \right\|^2$. We note that our projected Newton method differs from the above method both in the choice of search direction and the manner of search performed. The search directions for each method can each be calculated by solving a reduced system of linear equations, but the composition of these systems may well differ. More importantly, the linesearch performed by B-DIFF updates the vector x, while our projected Newton step modifies only the components of z corresponding to inactive constraints and then computes the best x consistent with z. This type of search assures that a good choice of steplength α and potential iterate $z(\alpha)$ is not rejected due to a poor update of the slack variables corresponding to the active constraints.

The similarity of the projected Newton preprocessing step to the major iteration of the B-DIFF algorithm of Harker & Xiao (1990), together with the much-improved results of Tables 28 through 34, led us to implement an algorithm that performed only projected Newton steps. The results were very disappointing, and are not included here. We found that a pure projected Newton algorithm suffered from a marked lack of robustness. Many

Table 28: Projected Newton Performance Metrics - GAMSLIB

size		problem	PATH (projected Newton preprocessing)					
n	nnz		proj-N	major	minor	func	Jac	time
242	1380	cammcp 1	2	3	3	6	6	0.36
232	1321	ers82mcp 1	1	4	4	6	6	0.36
262	2532	gemmcp 1	1	0	0	2	2	0.20
262	2532	gemmcp 2	0	0	0	1	1	0.05
262	2536	gemmcp 3	1	5	5	7	7	0.96
262	2532	gemmcp 4	1	4	4	6	6	0.78
262	2532	gemmcp 5	1	3	3	5	5	0.63
43	356	hansmcp 1	0	27	97	28	28	0.26
32	100	harkmcp 1	0	9	35	10	10	0.06
32	103	harkmcp 2	1	3	3	5	5	0.03
32	103	harkmcp 3	1	3	3	5	5	0.03
92	329	harkmcp 4	1	5	39	13	7	0.10
78	346	kormcp 1	1	2	2	4	4	0.07
350	1338	mr5mcp 1	1	6	6	8	8	0.81
6	16	oligomcp 1	0	6	6	7	7	0.02
11	24	transmcp 1	0	1	9	2	2	0.01
11	27	transmcp 2	0	0	0	1	1	0.02
11	24	transmcp 3	1	0	0	2	2	0.01
11	27	transmcp 4	0	6	16	7	7	0.03
6	24	two3mcp 1	0	6	6	7	7	0.04
		two3mcp 2	0	4	4	5	5	0.03
125	636	vonthmcp 1	0	12	185	13	13	0.45
6	20	wallmcp 1	0	2	2	3	3	0.01

Table 29: Projected Newton Performance Metrics - MPSGE

size		problem	PATH (projected Newton preprocessing)					
n	nnz		proj-N	major	minor	func	Jac	time
47	316	cafemge 2	0	7	9	21	8	1.67
129	1731	cammge 1	1	3	3	5	5	0.43
219	1601	co2mge 1	1	17	55	21	19	1.63
170	1728	dmcmge* 1	1	14	270	258	16	8.10
194	1532	dmcmge 2	1	5	5	7	7	0.76
114	941	etamge 1	1	16	55	36	18	1.26
183	2871	finmge 2	0	4	212	5	5	1.72
153	2806	finmge 3	1	3	3	5	5	0.91
183	2853	finmge 4	1	4	5	6	6	1.05
153	2796	finmge 5	1	3	3	5	5	0.93
323	7036	gemmge 2	1	4	346	6	6	4.37
323	7012	gemmge 3	1	9	10	11	11	3.67
323	7012	gemmge 4	1	10	13	12	12	3.92
323	6962	gemmge 5	0	6	693	7	7	7.18
43	793	hansmge 1	0	3	40	4	4	0.16
9	81	harmge 2	0	4	4	5	5	0.04
		harmge 3	0	6	6	7	7	0.06
		harmge 4	0	5	5	6	6	0.04
9	81	kehomge 1	0	8	8	9	9	0.09
		kehomge 2	0	7	7	8	8	0.08
		kehomge 3	0	6	6	7	7	0.07

Table 30: Projected Newton Performance Metrics - MPSGE

size		problem	PATH (projected Newton preprocessing)					
n	nnz		proj-N	major	minor	func	Jac	time
14	148	sammge 2	1	3	3	5	5	0.05
14	150	sammge 3	1	5	5	7	7	0.06
14	170	sammge 4	1	4	4	6	6	0.06
18	324	scarfmge 1	0	7	25	8	8	0.13
20	348	scarfmge 2	1	5	5	7	7	0.10
20	348	scarfmge 3	1	7	9	9	9	0.15
20	348	scarfmge 4	1	9	13	17	11	0.22
		shovmge 2	0	4	4	5	5	0.04
10	100	shovmge 3	0	5	5	6	6	0.05
		shovmge 4	0	5	5	6	6	0.05
5	25	unstmge 1	0	13	16	17	14	0.08
80	842	vonthmge 1	0	13	74	14	14	0.54

Table 31: Projected Newton Performance Metrics - MCPLIB

size		problem	PATH (projected Newton preprocessing)					
n	nnz		proj-N	major	minor	func	Jac	time
15	60	bertsekas 1	0	4	17	5	5	0.04
		bertsekas 2	1	4	6	6	6	0.05
		bertsekas 3	1	10	22	12	12	0.07
13	169	choi 1	1	4	4	6	6	2.58
20	149	colvdual 1	0	4	28	5	5	0.04
		colvdual 2	0	4	26	5	5	0.04
15	99	colvnlp 1	0	4	15	5	5	0.03
		colvnlp 2	1	3	5	5	5	0.02
101	10200	ehl_kost 1	1	5	7	7	7	5.24
		ehl_kost 2	1	17	56	19	19	14.66
		ehl_kost 3	1	5	5	12	7	6.28
		ehl_kost 4	1	4	4	6	6	4.26
		ehl_kost 5	1	4	4	8	6	4.82
5	25	gafni 1	0	4	6	5	5	0.03
		gafni 2	0	4	6	5	5	0.02
		gafni 3	0	4	6	5	5	0.02
14	116	hanskoop 1	0	15	29	16	16	0.07
		hanskoop 3	0	15	29	16	16	0.07
		hanskoop 5	0	6	20	7	7	0.04
		hanskoop 7	0	6	20	7	7	0.04
		hanskoop 9	0	13	25	14	14	0.07
99	740	hydroc20 1	1	8	8	11	10	0.42

Table 32: Projected Newton Performance Metrics - MCPLIB

size n	size nnz	problem	proj-N	major	minor	func	Jac	time
			PATH (projected Newton preprocessing)					
4	16	josephy 1	0	6	7	7	7	0.02
		josephy 2	0	10	16	15	11	0.03
		josephy 3	0	21	30	22	22	0.05
		josephy 4	0	4	6	5	5	0.01
		josephy 5	0	4	4	5	5	0.02
		josephy 6	0	14	33	15	15	0.04
4	16	kojshin 1	0	6	7	7	7	0.02
		kojshin 2	0	5	7	6	6	0.02
		kojshin 3	0	48	67	53	49	0.12
		kojshin 4	0	1	1	2	2	0.01
		kojshin 5	0	4	4	5	5	0.02
		kojshin 6	0	9	19	10	10	0.03
3	9	mathinum 1	0	6	7	10	7	0.02
		mathinum 2	0	5	5	6	6	0.02
		mathinum 3	0	12	18	21	13	0.04
		mathinum 4	0	6	6	7	7	0.02
4	11	mathisum 1	0	7	7	8	8	0.03
		mathisum 2	0	5	5	6	6	0.01
		mathisum 3	0	9	23	24	10	0.04
		mathisum 4	0	5	5	6	6	0.02
31	195	methan08 1	1	3	3	5	5	0.07
10	100	nash 1	0	7	7	8	8	0.09
		nash 2	0	6	6	7	7	0.07

Table 33: Projected Newton Performance Metrics - MCPLIB

size n	size nnz	problem	proj-N	major	minor	func	Jac	time
			PATH (projected Newton preprocessing)					
42	142	pies 1	0	2	64	3	3	0.06
16	188	powell 1	1	7	7	10	9	0.07
		powell 2	0	5	17	6	6	0.05
		powell 3	0	7	13	8	8	0.07
		powell 4	0	6	18	7	7	0.06
8	47	powell_mcp 1	0	6	6	7	7	0.03
		powell_mcp 2	0	7	7	8	8	0.03
		powell_mcp 3	0	9	9	10	10	0.05
		powell_mcp 4	0	8	8	9	9	0.04
13	86	scarfanum 1	0	4	21	5	5	0.05
		scarfanum 2	0	6	30	7	7	0.06
		scarfanum 3	1	4	11	7	6	0.06
14	96	scarfasum 1	1	4	5	7	6	0.06
		scarfasum 2	1	4	6	6	6	0.05
		scarfasum 3	1	4	11	7	6	0.05
39	323	scarfbnum 1	0	5	38	6	6	0.08
		scarfbnum 2	0	5	38	6	6	0.07
40	575	scarfbsum 1	0	4	37	5	5	0.12
		scarfbsum 2	0	4	41	5	5	0.10
27	84	sppe 1	1	7	16	16	9	0.05
		sppe 2	1	5	7	7	7	0.04
42	202	tobin 1	1	7	37	12	9	0.08
		tobin 2	2	6	10	12	9	0.07

Table 34: Projected Newton Performance Metrics - MCPLIB

size		problem	PATH (projected Newton preprocessing)					
n	nnz		proj-N	major	minor	func	Jac	time
5000	16992	bert_oc 1	3	0	0	4	4	2.39
		bert_oc 2	3	0	0	4	4	2.40
		bert_oc 3	3	1	1	5	5	2.58
		bert_oc 4	3	1	1	5	5	2.55
5625	28125	bratu	1	6	6	8	8	87.36
5625	28125	obstacle 1	14	1	1	16	16	11.41
		obstacle 2	15	1	1	17	17	24.02
		obstacle 3	13	0	0	22	14	22.03
		obstacle 4	13	0	0	23	14	21.65
		obstacle 5	7	0	0	8	8	24.24
		obstacle 6	12	0	0	25	13	43.65
		obstacle 7	13	0	0	26	14	35.60
		obstacle 8	13	0	0	25	14	51.41
1024	17152	opt_cont	4	1	1	6	6	1.47
4096	69376	opt_cont	4	1	1	6	6	6.70
8192	139008	opt_cont	4	1	1	6	6	14.38
16384	278272	opt_cont	4	1	1	6	6	32.92

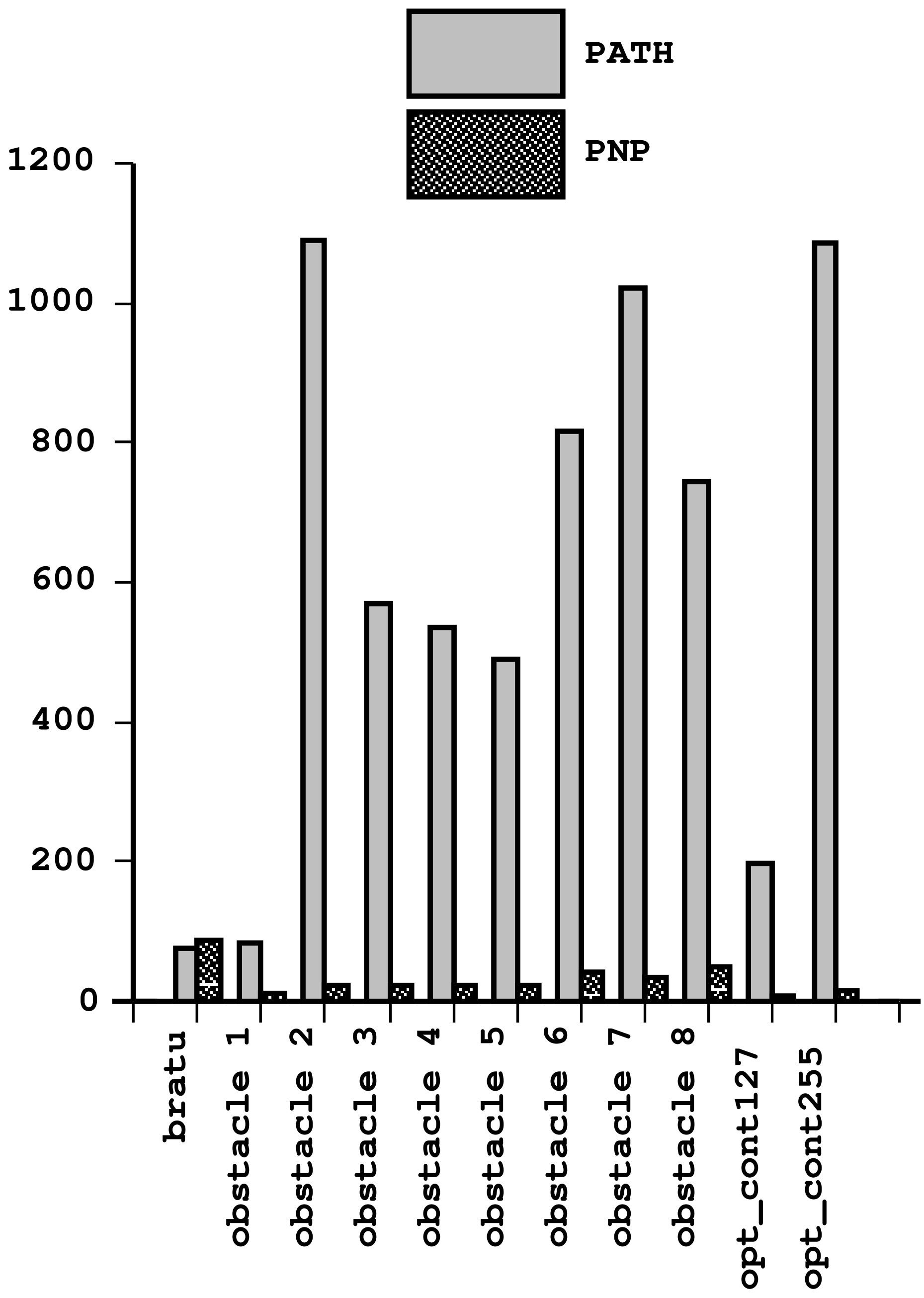


Figure 14: PATH Solution Times - Original / Projected Newton Preprocessing

of the difficulties reported by Harker & Xiao (1990) were evidenced in our computational tests as well. These authors report obtaining convergence in some cases only through the use of heuristics such as proximal point. This explains some of the differences between PATH and B-DIFF in Table 24 (page 119), particularly for the Nash problem. The use of these heuristics is a likely explanation for the difference in major iterations required by these two methods, since without the heuristics the two methods should perform identically on this model.

The excellent performance of the PATH solver with projected Newton preprocessing has resulted in a new release of the PATH solver, version 2.5. This latest version is available from the author.

6.2 Other Extensions

The complementarity interface to AMPL is a very recent development, and will need further testing and development before it can be distributed along with the rest of the AMPL solver interface library. In addition, it may be possible to extend its functionality as well. At present, problems can only be expressed as MCP's. The library could be extended to allow the expression of side constraints, so that variational inequalities over more general sets could be formulated explicitly, rather than having to be reformulated as a MCP before being written down in the AMPL model. With this extension, the reformulation as a MCP could be done automatically by the interface library for MCP solvers, or the side constraints could be furnished directly to a solver able to handle feasible sets more general than the rectangular set of MCP.

It may also be possible to narrow rather than broaden the focus of the AMPL interface, in a manner similar to that taken in the GAMS/MPSGE system designed by Rutherford (1994a) for the formulation of general equilibrium models. The MPSGE system speeds and simplifies the formulation of Arrow-Debreu economic equilibrium models by allowing a description of the model at a higher level than is possible using GAMS/MCP alone. The development of a similar AMPL system for formulating these or other types of models may be quite useful. However, such a project is best left until the latest revision of the AMPL language, due out in the latter half of 1994, is available.

Of course, the model library is easily improved through the addition of more models, especially those from fields outside of which the current models are drawn. Currently, there are many more small and medium size models in the library than there are larger models. The library would benefit greatly from additional large, nonlinear models. It is to be expected that such models will be made available as more people begin to use the complementarity facilities now available as part of the GAMS and AMPL modeling languages.

6.3 Conclusions

In this thesis, we have focused on algorithms and software for effectively solving MCP. In Chapter 2, we discuss the design fundamentals for complementarity interfaces to modeling languages and give details for two such interfaces, the GAMS Callable Program Library and the AMPL MCP interface library developed by the author. In Chapter 3, we present a library of complementarity models written in the GAMS and AMPL modeling languages, while Chapter 4 contains a description of the PATH solver, a novel application of a stabilization technique to a Newton method for nonsmooth equations. In Chapter 5 we present computational results for the PATH solver and other available solvers for MCP, results obtained using the interface and model libraries discussed in the previous chapters. This combination of model library, interface library, and solver has worked well and has helped greatly in the development and testing of the PATH solver. Finally, in this chapter, we have discussed preprocessing techniques used to improve the performance of the PATH solver and have indicated promising directions of future research based the interface libraries we have described.

Bibliography

Armijo, L. (1966), 'Minimization of functions having Lipschitz-continuous first partial derivatives', *Pacific Journal of Mathematics* **16**, 1–3.

Bertsekas, D. P. (1982), 'Projected Newton methods for optimization problems with simple constraints', *SIAM Journal on Control and Optimization* **20**, 221–246.

Bertsekas, D. P. & Gafni, E. M. (1982), 'Projection methods for variational inequalities with application to the traffic assignment problem', *Mathematical Programming Study* **17**, 139–159.

Bertsekas, D. P. & Tsitsiklis, J. N. (1989), *Parallel and Distributed Computation*, Prentice–Hall, Inc, Englewood Cliffs, New Jersey.

Bongartz, I., Conn, A. R., Gould, N. & Toint, P. (1993), CUTE: Constrained and unconstrained testing environment, Publications du Départment de Mathématique Report 93/10, Facultés Universitaires De Namur.

Brooke, A., Kendrick, D. & Meeraus, A. (1988), *GAMS: A User's Guide*, The Scientific Press, South San Francisco, CA.

Calamai, P. H. & Moré, J. J. (1987), 'Projected gradient methods for linearly constrained problems', *Mathematical Programming* **39**, 93–116.

Cao, M. & Ferris, M. C. (1992), A pivotal method for affine variational inequalities, Technical Report 1114, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. To appear in Mathematics of Operations Research.

Cao, M. & Ferris, M. C. (1994), Lineality removal for copositive–plus normal maps, Mathematical Programming Technical Report 94-02, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin. Submitted to Communications on Applied Nonlinear Analysis.

Chamberlain, R. M., Powell, M. J. D. & Lemaréchal, C. (1982), 'The watchdog technique for forcing convergence in algorithms for constrained optimization', *Mathematical Programming Study* **16**, 1–17.

Choi, S. C., DeSarbo, W. S. & Harker, P. T. (1990), 'Product positioning under price competition', *Management Science* **36**, 175–199.

Chvátal, V. (1983), *Linear Programming*, W. H. Freeman and Company, New York.

Ciarlet, P. G. (1978), *The Finite Element Method for Elliptic Problems*, North-Holland, New York.

Colville, A. R. (1968), A comparative study on nonlinear programming codes, Technical Report 320–2949, IBM New York Scientific Center.

Conn, A. R., Gould, N. I. M. & Toint, P. (1992), *LANCELOT: A Fortran package for Large–Scale Nonlinear Optimization (Release A)*, number 17 *in* 'Springer Series in Computational Mathematics', Springer Verlag, Heidelberg, Berlin.

Cottle, R. W. & Dantzig, G. (1968), 'Complementary pivot theory of mathematical programming', *Linear Algebra and Its Applications* **1**, 103–125.

Cottle, R. W., Pang, J. S. & Stone, R. E. (1992), *The Linear Complementarity Problem*, Academic Press, Boston.

Cryer, C. W. & Dempster, M. A. H. (1980), 'Equivalence of linear complementarity problems and linear programs in vector lattice Hilbert spaces', *SIAM Journal on Control and Optimization* **18**, 76–89.

Dennis, J. E. & Schnabel, R. B. (1983), *Numerical Methods for Unconstrained Optimization and Nonlinear Equations*, Prentice–Hall, Inc, Englewood Cliffs, New Jersey.

Dirkse, S. P. & Ferris, M. C. (1994), 'The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems', *Optimization Methods & Software*. To appear.

Dirkse, S. P., Ferris, M. C., Preckel, P. V. & Rutherford, T. (1994), The GAMS callable program library for variational and complementarity solvers, Mathematical Programming Technical Report 94-07, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.

Eaves, B. C. (1971), 'On the basic theorem of complementarity', *Mathematical Programming* **1**, 68–87.

Ferris, M. C. & Lucidi, S. (1991), Globally convergent methods for nonlinear equations, Technical Report 1030, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin.

Ferris, M. C. & Lucidi, S. (1994), 'Nonmonotone stabilization methods for nonlinear equations', *Journal of Optimization Theory and Applications* **81**, 53–74.

Fletcher, R. (1987), *Practical Methods of Optimization*, second ed., John Wiley & Sons, New York.

Fourer, R., Gay, D. & Kernighan, B. (1993), *AMPL*, The Scientific Press, South San Francisco, CA.

Friesz, T. L., Tobin, R. L., Smith, T. E. & Harker, P. T. (1983), 'A nonlinear complementarity formulation and solution procedure for the general derived demand network equilibrium problem', *Journal of Regional Science* **23**, 337–359.

Fukushima, M. (1992), 'Equivalent differentiable optimization problems and descent methods for asymmetric variational inequality problems', *Mathematical Programming* **53**, 99–110.

Gay, D. M. (1993), 'Hooking your solver to AMPL', Numerical Analysis Manuscript 93–10. AT&T Bell Laboratories, Murray Hill, New Jersey.

Geiger, C. & Kanzow, C. (1994), 'On the resolution of monotone complementarity problems', Preprint 82, Institute of Applied Mathematics, University of Hamburg. Bundesstrasse 55, D-20146 Hamburg Germany.

Goldstein, A. A. (1967), *Constructive Real Analysis*, Harper and Row, New York.

Grippo, L., Lampariello, F. & Lucidi, S. (1986), 'A nonmonotone line search technique for Newton's method', *SIAM Journal on Numerical Analysis* **23**, 707–716.

Grippo, L., Lampariello, F. & Lucidi, S. (1991), 'A class of nonmonotone stabilization methods in unconstrained optimization', *Numerische Mathematik* **59**, 779–805.

Hansen, T. & Koopmans, T. C. (1972), 'On the definition and computation of a capital stock invariant under optimization', *Journal of Economic Theory* **5**, 487–523.

Harker, P. T. (1986), 'Alternative models of spatial competition', *Operations Research* **34**, 410–425.

Harker, P. T. (1988), 'Accelerating the convergence of the diagonalization and projection algorithms for finite–dimensional variational inequalities', *Mathematical Programming* **41**, 29–50.

Harker, P. T. & Xiao, B. (1990), 'Newton's method for the nonlinear complementarity problem: A B–differentiable equation approach', *Mathematical Programming* **48**, 339–358.

Hearn, D. W. (1982), 'The gap function of a convex program', *Operations Research Letters* **1**, 67–71.

Hiriart-Urruty, J.-B. & Lemaréchal, C. (1993), *Convex Analysis and Minimization Algorithms I*, Vol. 305 of *Grundlehren der mathematischen Wissenschaften*, Springer Verlag, Berlin.

Hogan, W. W. (1975), 'Energy policy models for Project Independence', *Computers & Operations Research* **2**, 251–271.

Hoppe, R. H. W. & Mittelmann, H. D. (1989), 'A multi-grid continuation strategy for parameter-dependent variational inequalities', *Journal of Computational and Applied Mathematics* **26**, 35–46.

Josephy, N. H. (1979*a*), Newton's method for generalized equations, Technical Summary Report 1965, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin.

Josephy, N. H. (1979*b*), Newton's Method for Generalized Equations and the PIES Energy Model, PhD thesis, Department of Industrial Engineering, University of Wisconsin–Madison.

Kalvelagen, E. (1992), 'The GAMS I/O library', mimeo, GAMS Development Corporation. Preliminary Version.

Kostreva, M. M. (1984), 'Elasto-hydrodynamic lubrication: A non-linear complementarity problem', *International Journal for Numerical Methods in Fluids* **4**, 377–397.

Larsson, T. & Patriksson, M. (1994), 'A class of gap functions for variational inequalities', *Mathematical Programming* **64**, 53–79.

Lemke, C. E. (1965), 'Bimatrix equilibrium points and mathematical programming', *Management Science* **11**, 681–689.

Mangasarian, O. L. (1969), *Nonlinear Programming*, McGraw–Hill, New York.

Mangasarian, O. L. (1976), 'Equivalence of the complementarity problem to a system of nonlinear equations', *SIAM Journal of Applied Mathematics* **31**, 89–92.

Mangasarian, O. L. & Solodov, M. V. (1993), 'Nonlinear complementarity as unconstrained and constrained minimization', *Mathematical Programming* **62**, 277–298.

Mathiesen, L. (1987), 'An algorithm based on a sequence of linear complementarity problems applied to a Walrasian equilibrium model: An example', *Mathematical Programming* **37**, 1–18.

Miersemann, E. & Mittelmann, H. D. (1989), 'Continuation for parametrized nonlinear variational inequalities', *Journal of Computational and Applied Mathematics* **26**, 23–34.

Minty, G. J. (1962), 'Monotone (nonlinear) operators in Hilbert space', *Duke Mathematics Journal* **29**, 341–346.

Moré, J. J. & Toraldo, G. (1991), 'On the solution of large quadratic programming problems with bound constraints', *SIAM Journal on Optimization* **1**, 93–113.

Murphy, F. H., Sherali, H. D. & Soyster, A. L. (1982), 'A mathematical programming approach for determining oligopolistic market equilibrium', *Mathematical Programming* **24**, 92–106.

Murtagh, B. A. & Saunders, M. A. (1983), MINOS 5.0 user's guide, Technical Report SOL 83.20, Stanford University.

Ortega, J. M. & Rheinboldt, W. C. (1970), *Iterative Solution of Nonlinear Equations in Several Variables*, Academic Press, San Diego, California.

Pang, J. S. (1990), 'Newton's method for B–differentiable equations', *Mathematics of Operations Research* **15**, 311–341.

Pang, J.-S. & Gabriel, S. A. (1993), 'NE/SQP: A robust algorithm for the nonlinear complementarity problem', *Mathematical Programming* **60**, 295–338.

Ralph, D. (1994), 'Global convergence of damped Newton's method for nonsmooth equations, via the path search', *Mathematics of Operations Research* **19**, 352–389.

Robinson, S. M. (1979), 'Generalized equations and their solution: Part I: Basic theory', *Mathematical Programming Study* **10**, 128–141.

Robinson, S. M. (1980), 'Strongly regular generalized equations', *Mathematics of Operations Research* **5**, 43–62.

Robinson, S. M. (1990), 'Mathematical foundations of nonsmooth embedding methods', *Mathematical Programming* **48**, 221–229.

Robinson, S. M. (1992), 'Normal maps induced by linear transformations', *Mathematics of Operations Research* **17**, 691–714.

Robinson, S. M. (1993), 'Newton's method for a class of nonsmooth functions', *Set Valued Analysis*. To appear.

Rockafellar, R. T. (1970), *Convex Analysis*, Princeton University Press, Princeton, New Jersey.

Rockafellar, R. T. (1987), 'Linear-quadratic programming and optimal control', *SIAM Journal on Control and Optimization* **25**, 781–814.

Rockafellar, R. T. (1988), 'Multistage convex programming and discrete-time optimal control', *Control and Cybernetics* **17**(2-3), 225–245.

Rockafellar, R. T. (1990), 'Computational schemes for large–scale problems in extended linear–quadratic programming', *Mathematical Programming* **48**, 447–474.

Rockafellar, R. T. (1991), Large-scale extended linear-quadratic programming and multistage optimization, *in* S. Gomez, J. P. Hennart & R. A. Tapia, eds, 'Advances in Numerical Partial Differential Equations and Optimization', SIAM Publications, chapter 15, pp. 247–261.

Rockafellar, R. T. & Wets, R. J.-B. (1986*a*), 'A Lagrangian finite generation technique for solving linear-quadratic problems in stochastic programming', *Mathematical Programming Study* **28**, 63–93.

Rockafellar, R. T. & Wets, R. J.-B. (1986*b*), Linear-quadratic programming problems with stochastic penalties: the finite generation algorithm, *in* V. I. Arkin, A. Shiraer & R. J.-B. Wets, eds, 'Stochastic Optimization', Lecture Notes in Control and Information Sciences, IIASA Series No. 81, Springer-Verlag, New York, Berlin, pp. 545–560.

Rutherford, T. F. (1993), MILES: A mixed inequality and nonlinear equation solver, Working Paper, Department of Economics, University of Colorado, Boulder.

Rutherford, T. F. (1994*a*), Applied general equilibrium modeling with MPSGE as a GAMS subsystem, Manuscript, Department of Economics, University of Colorado, Boulder.

Rutherford, T. F. (1994*b*), Extensions of GAMS for complementarity problems arising in applied economic analysis, Manuscript, Department of Economics, University of Colorado, Boulder.

Scarf, H. E. (1973), *The Computation of Economic Equilibria*, Yale University Press, New Haven, Conneticut.

Sellami, H. (1994), A Continuation Method for Normal Maps, PhD thesis, University of Wisconsin – Madison, Madison, Wisconsin.

Tobin, R. L. (1988), 'A variable dimension solution approach for the general spatial equilibrium problem', *Mathematical Programming* **40**, 33–51.

Varian, H. R. (1978), *Microeconomic Analysis*, W.W. Norton & Company, New York, New York.