Interfaces to PATH 3.0: Design, Implementation and Usage *

Michael C. Ferris   Todd S. Munson†

May 5, 1998

Abstract

Several new interfaces have recently been developed requiring PATH to solve a mixed complementarity problem. To overcome the necessity of maintaining a different version of PATH for each interface, the code was reorganized using object-oriented design techniques. At the same time, robustness issues were considered and enhancements made to the algorithm. In this paper, we document the external interfaces to the PATH code and describe some of the new utilities using PATH. We then discuss the enhancements made and compare the results obtained from PATH 2.9 to the new version.

1 Introduction

The PATH solver [12] for mixed complementarity problems (MCPs) was introduced in 1995 and has since become the standard against which new MCP solvers are compared. However, the main user group for PATH continues to be economists using the MPSGE preprocessor [36]. While developing the new PATH implementation, we had two goals: to make the solver accessible to a broad audience and to improve the effectiveness of the code on large, complex problems. Therefore, this paper is split into two main parts, each discussing one of these issues.

We completely redesigned the PATH implementation using object-oriented design techniques in order to easily maintain the code, rapidly build interfaces to the solver, and quickly test new ideas regarding the algorithm. The relevant components of the PATH library necessary to add new interfaces and port the code to different architectures are the subject of Section 2. A discussion of some of the interfaces built using this functionality and how they influenced design decisions can be found in Section 3. Some of the interfaces now supported include

*This material is based on research supported by National Science Foundation Grant CCR-9619765 and GAMS Corporation.

†Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton Street, Madison, Wisconsin 53706 (ferris,tmunson@cs.wisc.edu)

links to AMPL [23], GAMS [6], MATLAB, and NEOS [9], as well as a callable subroutine library version. To further broaden the user base, implementations for some of the interfaces presented are available online at

`http://www.cs.wisc.edu/cpnet/path.html`

and the PATH library can be freely obtained by contacting one of the authors.

Two significant changes were previously made to the code described in [12, 13] for improved reliability. One was the introduction of a crashing technique [14] to quickly identify an active set from the user-supplied starting point. The other was the addition of a proximal perturbation scheme [1, 2] to overcome problems with a singular basis matrix. The new implementation has further improved numerical properties and restarts when a stationary point of the merit function is found. These changes have led to a more effective version of the code. Section 4 describes the new extensions made to the algorithm for improved robustness. Finally, Section 5 reports on a comparison made between PATH 3.0 and PATH 2.9 on a suite of test problems.

Some notation and definitions are in order before proceeding. The mixed complementarity problem (MCP) is defined using lower bounds, $l \in \{\Re \cup \{-\infty\}\}^n$, and upper bounds, $u \in \{\Re \cup \{\infty\}\}^n$, such that $l < u$. Let $B := \{z \in \Re^n \mid l \leq z \leq u\}$, and $F : B \rightarrow \Re^n$. The vector $z \in B$ is a solution to MCP(F,B), if and only if exactly one of the following holds for each $i \in \{1, \ldots, n\}$:

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

The core PATH algorithm [12] uses a nonsmooth Newton method [34] to find a zero of the normal map [33] associated with the MCP. The normal map for the MCP is given by

$$F(\pi(x)) + x - \pi(x)$$

where $\pi(x)$ represents the projection of x onto B in the Euclidean norm. It is well known that if x is a zero of the normal map, then $\pi(x)$ solves the MCP. A non-monotone pathsearch [13, 20] using the residual of the normal map

$$||F(\pi(x)) + x - \pi(x)||$$

as a merit function is used to improve robustness. A proof of convergence and rate of convergence results can be found in [32]. The enhancements described in this paper do not change the basic properties of the algorithm; they only modify the implementation. Hence, the theory already developed [32, 12] still applies.

2 Component Interfaces

Object-oriented design techniques were used to completely restructure the PATH code. The basic premise is to encapsulate data and functions together into objects. The implementation for each of the objects is hidden from the user. Virtual classes are used for system-dependent code and the basis package. The exact implementation for each of these objects is chosen at link time without making any modifications to PATH. C was chosen as the implementation medium so that we can easily port the code to any platform. In this section, we discuss the external interfaces required in order link the code with new utilities.

The overriding concern faced when developing the component interfaces was flexibility. The goal was to easily support any environment, architecture, or programming language using the same interfaces. In order to achieve this goal, the component interfaces are split into two parts: the system-dependent components, which include memory allocation, timing mechanisms, error reporting, and output, and the interface-specific functions, which provide information about the problem and mechanisms to perform function and Jacobian evaluations. The interface writer also provides a driver program that performs initializations and calls the main PATH routine. We describe the system-dependent objects required, mechanisms needed to provide problem data, and the driver in the following subsections.

2.1 System-Dependent Objects

The system-dependent objects provide an abstract view of the machine. Whenever the PATH code is ported to a new architecture, only implementations for these virtual base classes need to be coded. The four objects comprising the system-dependent interface, memory, time, error, and output, are now described and the required functionality given.

2.1.1 Memory

The memory object consists of all necessary functions to allocate and relinquish memory. We distinguish between two different types of memory allocation: general memory allocation and the allocation necessary for the basis factors. The general memory allocation routine is frequently called upon to allocate relatively small pieces of memory. The factor allocation requires a single large section of memory to be obtained. Mechanisms optimized for these differing types of memory request patterns can be coded. Table 1 describes the functionality required of the memory object.

We guarantee that within the core PATH algorithm only one set of factors will be allocated at a time. However, the sequence Memory_AllocateFactors(), Memory_FreeFactors() may be repeated within the code.

A standard implementation for this object would use the routines malloc() and free(). More sophisticated implementations are possible. For example, the GAMS implementation of these routines places the factors on the GAMS heap, a portion of memory previously allocated as workspace for the MPSGE preprocessor.

Function	Description
void *Memory_Allocate(long int)	Allocate the specified number of bytes from memory and return a pointer to the allocated memory.
void *Memory_AllocateFactors(long int)	Allocate the specified number of bytes of memory and return a pointer to the allocated memory. The amount of memory requested in Memory_AllocateFactors() is typically much greater than that requested in Memory_Allocate().
void Memory_Free(void *)	Free the indicated memory allocated with Memory_Allocate().
void Memory_FreeFactors(void *)	Free the indicated memory allocated with Memory_AllocateFactors().

Table 1: Memory Object Functions

Function	Description
Time *Time_Create()	Allocate and return a time structure.
void Time_Destroy(Time *)	Free the indicated time structure.
void Time_Start(Time *)	Place the correct value for the current time in the indicated structure.
double Time_Query(Time *)	Return the number of seconds elapsed since the indicated structure has been started.

Table 2: Time Object Functions

2.1.2 Time

The time object measures the amount of time spent in particular sections of the PATH code. The object requires the definition of a structure, struct _Time, containing the private data members for the class. An implementation using the clock() standard function might define the structure as follows:

```
struct _Time { clock_t time; };
```

where clock_t is declared in the standard header file time.h. The necessary functions for the time object are found in Table 2.

System-dependent routines such as rusage() on a UNIX platform can be used to implement the time object. On the DOS/Windows platform the clock() function can be used. We refer the reader to the code that is available online for these implementations.

2.1.3 Error

The functions contained in this package provide the user with information concerning warnings and errors. A warning tells the user about difficulties or non-standard events encountered. At the end of the warning, control should be returned to the PATH algorithm. An error on the other hand is fatal. Execution should stop at the end of the error routine. The requisite functions are in Table 3.

An interesting contrast in the error function implementation can be found in the online codes by comparing the standard implementation using exit() with the MATLAB interface error routine, which uses mexErrMsgTxt(). When exit() is used, the operating system frees all previously allocated memory and the program terminates. In the case of MATLAB, we need to relinquish all allocated memory before returning control back to MATLAB, otherwise, the MATLAB session leaks memory. The mexErrMsgTxt() routine deallocates the necessary memory for us before returning control back to MATLAB. The use of exit() is entirely inappropriate within the MATLAB session.

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

Table 3: Error Object Functions

2.1.4 Output

Output is a key object because support for output from both C and FORTRAN is required. We have an enumerated type, Output_Mode, that indicates the necessary type of output. This enumerated type has three elements:

Output_Log The log is used to demonstrate that PATH is making progress.

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

Output_Copy Output specified for copy is placed into both the log file and the status file.

Our use of these modes is motivated by GAMS standards. A brief description of the required functionality can be found in Table 4.

A typical C implementation uses the standard functions vprintf(), vfprintf(), and vsprintf(). FORTRAN output is considerably more difficult and depends upon the system and compiler. Details on FORTRAN output can be found in the online standalone FORTRAN code. However, the PATH code remains unchanged even if FORTRAN output is used.

Function	Description
Output_Printf(Output_Mode, char *, ...)	Output the indicated message of type Output_Mode to the correct place.
Output_VPrintf(Output_Mode, char *, va_list)	Output the indicated message to the correct file.

Table 4: Output Object Functions

2.2 Problem-Specific Interface

To accommodate all of the available interfaces, we have abstracted problem-specific information from the PATH algorithm. Five functions are required in the problem-specific interface implementation and are given in Table 5. The function_evaluation() and jacobian_evaluation() functions should return the number of domain violations encountered in the evaluation. A domain violation occurs, for example, when we attempt to take the log of a negative number or we encounter division by zero.

For the sake of compatibility, indices are in the FORTRAN format, i.e. an index goes from one to the number of indices. The sparse matrix is stored in compressed column format. Furthermore, the structure of the sparse matrix need only be determined and allocated once. We guarantee that the PATH code will not alter the structure. Finally, we also note that contrary to previous versions of the code, the addition of a diagonal element to the sparse Jacobian is no longer required.

2.3 Driver

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

```
1. Initialize system-dependent parts.
2. Call Options_Default().
3. Create an initial Path structure using Path_Create().
4. Invoke Path_Solve().
5. Do something with the results.
6. Finish system-dependent parts.
```

If special setups need to be performed for the system-dependent parts of the code, they are done at the beginning. Options_Default() ensures that all defaults

Function	Description
void problem_size(int *n, int *nnz)	Give the size of the problem and number of nonzeros in the Jacobian.
void bounds(int n, double *z, double *lb, double *ub)	Give the lower and upper bounds and a starting point for the problem.
int function_evaluation(int n, double *z, double *f)	Evaluate the function for a point $z \in B$. Return the number of domain violations.
int jacobian_evaluation(int n, double *z, int wantf, double *f, int *nnz, int *col, int *len, int *row, double *data)	Evaluates the function and Jacobian for a point $z \in B$. Return the number of domain violations. The Jacobian is stored in a compressed column format. Initially nnz is the allocated size for the row and data arrays. The actual number of nonzeros in the Jacobian should be supplied before returning.
void variable_name(int var, char *buf, int bufSize)	Provide a name for the specified variable.
void constraint_name(int con, char *buf, int bufSize)	Provide a name for the specified constraint.

Table 5: Problem-Specific Interface Functions

have been set and must be issued before solving the problem. An options file may be read using the command Options_Read(). Users who repeatedly invoke the algorithm can directly allocate required workspaces and reuse that workspace from one PATH solve to the next. This is a key point for applications solving a sequence of MCP's, as is done in [15]. The argument lists for the aforementioned routines can be found in the online version of the PATH header files.

These component interfaces have been successfully used to construct links to the GAMS and AMPL modeling languages. We have also developed MATLAB and NEOS hookups using the same library. Details on these interfaces and how to invoke PATH directly from C and FORTRAN codes are now given.

3 Supported Interfaces

A number of new interfaces are now supported by the PATH implementation. We briefly discuss some of them and describe how their unique characteristics influenced the component interface design.

3.1 MATLAB

The MATLAB interface (for MATLAB versions 5.0 and above) to PATH consists of a single mex file. Function and Jacobian evaluations are provided by the user as MATLAB m-functions. Hence, the PATH interface routines function_evaluation() and jacobian_evaluation() are implemented via the mexCallMATLAB() function. The problem size and bounds are passed directly to the mex function. The memory and error routines are implemented using the mxCalloc() and mexErrMsgTxt() routines provided by the MATLAB API. The abstract view of the machine provided by the new PATH library permits this and gives distinct advantages over previous implementations when a user interrupts the PATH code from within MATLAB. A MATLAB user invokes PATH with the following syntax:

```
z = pathsol(z, l, u, cpfunjac)
```

The name pathsol is used to avoid conflicts with the MATLAB defined variable path. Here, l and u are the bounds on the variables, and cpfunjac is the name of an m-file for evaluating the function F and its Jacobian J. The corresponding file cpfunjac.m contains the definition of

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

that computes the function F and if jacflag=1 the sparse Jacobian J at the point z. domerr returns the number of domain violations encountered during the evaluation.

If the fourth input argument to pathsol is omitted, the code takes a function cpfunjac as its default. If l, u are also omitted, z is assumed non-negative. If PATH fails to solve the problem, then a MATLAB error is generated. This is part of the pathsol.m file, so a user could modify the termination behavior if desired.

3.2 Callable subroutines

To encourage other users to invoke PATH from directly within their applications we have developed two interfaces allowing the code to be called from either a C or FORTRAN program. The aim of our design was to be as simple as possible in order to eliminate most of the errors that may occur while coding a particular application to use PATH. Therefore, the standalone interface is even simpler than that described in Section 2.2.

The C interface consists of three routines:

```
void pathMain(int n, int nnz, int *status,
              double *z, double *F,
              double *l, double *u);
int funcEval(int n, double *z, double *F)
int jacEval(int n, int nnz, double *z, int *col,
            int *len, int *row, double *data);
```

The user has the responsibility of writing the latter two routines and linking them with the library of PATH routines. The first evaluates the nonlinear function F at $z \in B$. Contrary to many codes for MCP, we guarantee that the function F will only be evaluated within the box, B. The second fills in the relevant information about the Jacobian of F at z in compressed column format. The data regarding bounds and size are passed to the pathMain() routine, which returns the solution z and $F(z)$ along with a termination criterion status. The values for status are available in the header file 'Path.h'; a value of 1 indicates successful completion.

The FORTRAN interface is almost identical:

```
subroutine pathmain(n, nnz, status, z, F, l, u)
integer function funceval(n, z, F)
integer function jaceval(n, nnz, z, col, len, row, data)
```

The PATH libraries for certain machine architectures are now freely available by contacting one of the authors.

3.3 NEOS

NEOS [9] enables users to submit optimization problems across the Internet to the NEOS server. The server contacts a client, transmits the problem information to the client, which then attempts to solve the problem. Results are sent back to the original submitter. The PATH solver is hooked up to NEOS via a suite of interface routines [21].

When a user submits an MCP problem to NEOS, the user specifies FORTRAN functions initpt(), xbounds(), and fcn(), along with an integer to represent n. The interface uses ADIFOR [5] to compute the Jacobian for the FORTRAN function representing the problem. The sparse structure and number of nonzeros in the Jacobian are also generated. The PATH problem-specific interface routines of Section 2.2 are coded to use the sparse Jacobian and the data supplied by the other routines passed to NEOS. All these routines are linked with the PATH library to produce an executable that is run on one of the machines in the Condor pool [30] available at Wisconsin.

Current work involves generating a new C interface using ADOL-C [27] and allowing the PATH solver to be called across the Internet as a subroutine in a manner similar to that outlined in Section 3.2.

3.4 GAMS

The major users of PATH continue to be economists, many of whom use the code [13] for solving MCP models generated with the MPSGE preprocessor [36, 37] of the GAMS modeling language [6]. CPLIB [16], a suite of routines giving a solver access to problem-specific information, including function and Jacobian evaluations, facilitates the linkage of PATH and other solvers [1, 4, 7, 35] to GAMS. The GAMS/MCP interface to PATH [16] remains essentially unchanged. However, several new uses of the code from within GAMS were made possible because of the added flexibility provided by the object-oriented design, the reuse of memory, and access to the workspace. Two examples are the GAMS/CNS interface, and the bundle method for solving MPEC, which uses PATH as a subroutine to solve MCP subproblems. The latter usage is described in [15].

Recently, GAMS added the constrained nonlinear system, CNS, model type to their language. The constrained nonlinear system is defined by a set of bounds, B and a function $F : B \to \Re^n$. A solution to CNS(F, B) is such that $x \in B$ and $F(x) = 0$. The most popular approach in GAMS to solving this problem has been to set up a dummy objective function and solve

$$\min 0 \text{ subject to } F(x) = 0, \quad x \in B. \tag{1}$$

The new CNS model type allows solvers such as CONOPT [17] and MINOS [31] to set up alternative internal models to solve (1) such as

$$\min \|F(x)\|_2^2 \text{ subject to } x \in B. \tag{2}$$

Our work internally reformulates the CNS model as an MCP. One choice would be to solve the Karush-Kuhn-Tucker conditions of (2) as an MCP. This is essentially a Gauss-Newton approach (involving a function $\nabla F(x)^T F(x)$) to the problem and appears to be computationally less effective than the following Newton approach. The particular MCP that our implementation uses, is defined by the following functions and bounds:

$$G(x, y) = \begin{bmatrix} -y \\ F(x) \end{bmatrix}, L = \begin{bmatrix} l \\ -\infty \end{bmatrix}, U = \begin{bmatrix} u \\ \infty \end{bmatrix} \tag{3}$$

Let $\bar{B} := \{z \in \Re^{2n} \mid L \leq z \leq U\}$. If $\bar{x} \in \Re^n$ solves CNS(F, B), then $(\bar{x}, 0)$ solves MCP(G, $\bar{B}$). Furthermore, if $(\bar{x}, \bar{y})$ solves MCP(G, $\bar{B}$), then $\bar{x}$ solves CNS(F, B). These implications following directly from the definitions of the MCP and CNS. The implementation of the CNS solver reuses the memory, time, error, and output packages from the GAMS/MCP interface. However, the problem-specific interface routines were modified to construct (3) and the Jacobian of G from the GAMS provided routines for CNS(F, B).

3.5 AMPL

New syntax for expressing complementarity relationships has been added to the AMPL modeling language [19]. In order to test this syntax, the PATH solver has been hooked up to AMPL [23] using the AMPL library routines described in [24]. These routines allow easy implementation of the PATH problem-specific routines outlined in Section 2.2. The AMPL interface routines exploit the fact that a function evaluation may have already been carried out (thus providing partial derivative information) when a call to the Jacobian evaluator is made. We have updated the PATH code to allow for efficiencies to be exploited when this occurs (see the parameters to jacobian_evaluation in Section 2.2).

4 Enhancements

During the reconstruction of the PATH implementation, robustness issues were once again addressed. Utilizing experience gained from solving practical problems, three areas were targeted for improvement. These areas, the linear model, pathsearch, and restarts, are the subject of subsequent subsections.

When the PATH implementation was rewritten, the names of several options were changed. We give a list of the user-available options along with their defaults and meaning in Table 6.

In particular, PATH can emulate Lemke's method [8, 29] for LCP with the following options:

```
crash_method none;
major_iteration_limit 1;
lemke_start first;
nms no;
```

If instead, PATH is to imitate the Josephy Newton method [28] for NCP with an Armijo style linesearch on the normal map residual, then the options to use are:

```
crash_method none;
lemke_start always;
nms_initial_reference_factor 1;
nms_memory size 1;
nms_mstep_frequency 1;
nms_searchtype line;
```

Note that `nms_memory_size 1` and `nms_initial_reference_factor 1` turn off the non-monotone linesearch, while `nms_mstep_frequency 1` turns off watchdoging. `nms_searchtype line` forces PATH to search the line segment between the initial point and the solution to the linear model, as opposed to the default pathsearch (see Section 4.3).

4.1 Basis Package

The core PATH code has been recoded in an object-oriented fashion as well. One key feature of the new design is the notion of a basis package. PATH requires a basis package to provide certain functionality expressed via the following functions:

Basis_Factor() Factor the given basis matrix.

Basis_Solve() Use the computed factors to solve a linear system of equations.

Basis_Replace() Replace the indicated column of the basis matrix.

Basis_SingularVariable() Tell whether the indicated variable is linearly dependent.

Option	Default	Explanation
convergence_tolerance	1e-6	Stopping criterion
crash_iteration_limit	50	Maximum iterations allowed in crash
crash_method	pnewton	pnewton or none
crash_minimum_dimension	1	Minimum problem dimension to perform crash
crash_nbchange_limit	1	Number of changes to the basis allowed
cumulative_iteration_limit	10000	Maximum minor iterations allowed
lemke_start	automatic	Frequency of lemke starts (automatic, first, always)
major_iteration_limit	500	Maximum major iterations allowed
minor_iteration_limit	1000	Maximum minor iterations allowed in each major iteration
nms	yes	Allow line/path searching, watchdoging, and non-monotone descent
nms_initial_reference_factor	20	Controls size of initial reference value
nms_memory_size	10	Number of reference values kept
nms_mstep_frequency	10	Frequency at which m steps are performed
nms_searchtype	path	Search type to use (path or line)
output	yes	Turn output on or off. If output is turned off, selected parts can be turned back on.
output_crash_iterations	yes	Output information on crash iterations
output_crash_iterations_frequency	1	Frequency at which crash iteration log is printed
output_errors	yes	Output error messages
output_linear_model	no	Output linear model each major iteration
output_major_iterations	yes	Output information on major iterations
output_major_iterations_frequency	1	Frequency at which major iteration log is printed
output_minor_iterations	yes	Output information on minor iterations
output_minor_iterations_frequency	500	Frequency at which minor iteration log is printed
output_options	no	Output all options and their values
output_warnings	no	Output warning messages
proximal_perturbation	0	Initial perturbation
restart_limit	3	Maximum number of restarts (0 - 3)
time_limit	3600	Maximum number of seconds algorithm is allowed to run

Table 6: PATH Options

Currently, two interchangeable basis packages have been implemented. The choice of which package to use is done at link time without making any modifications to the code PATH code. One package uses a FORTRAN version of LUSOL [25] (based on a Markovitz factorization and Bartels-Golub updates). These routines are a key component of the MINOS [31] nonlinear programming package. The second is a dense matrix implementation, written in C that uses the Fletcher-Matthews [22] updating procedure. A freely distributed version of the PATH library contains the dense factorization. A library without any factorization software is also available. In this case, the user can obtain source code for MINOS from Stanford University and incorporate the appropriate object code into the library.

A future basis object for the MATLAB implementation will use mexCallMATLAB() to invoke the MATLAB LU routine, along with an update procedure provided via the Schur-Complement described in [18].

4.2 The Linear Model

Let $M \in \Re^{n \times n}$, $q \in \Re^n$, and $B = [l, u]$ be given. $(\bar{z}, \bar{w}, \bar{v})$ solves the linear mixed complementarity problem defined by M, q, and B if and only if it satisfies the following constrained system of equations:

$$
\begin{aligned}
Mz - w + v + q &= 0 && (4)\\
w^T(z - l) &= 0 && (5)\\
v^T(u - z) &= 0 && (6)\\
z \in B, w \in \Re^n_+, v \in \Re^n_+, & && (7)
\end{aligned}
$$

where $x + \infty = \infty$ for all $x \in \Re$ and $0 \cdot \infty = 0$ by convention. A triple, $(\hat{z}, \hat{w}, \hat{v})$, satisfying equations (3) - (5) is called a complementary triple.

The objective of the linear model solver is to construct a path from a given complementary triple $(\hat{z}, \hat{w}, \hat{v})$ to a solution $(\bar{z}, \bar{w}, \bar{v})$. The algorithm used to solve the linear problem is identical to that given in [10]; however, artificial variables are incorporated into the model. The augmented model is then:

$$
\begin{aligned}
Mz - w + v + a + (1 - t)r + q &= 0 && (8)\\
w^T(z - l) &= 0 && (9)\\
v^T(u - z) &= 0 && (10)\\
z \in B, w \in \Re^n_+, v \in \Re^n_+, a \equiv 0, t &\in [0, 1] && (11)
\end{aligned}
$$

where r is the residual, t is the path parameter, and a is a vector of artificial variables. The residual is scaled in the code to improve numerical stability.

The addition of artificial variables enables us to construct an initial invertible basis consistent with the given starting point even under rank deficiency. The procedure consists of two parts: constructing an initial guess as to the basis and then recovering from rank deficiency to obtain an invertible basis. The crash technique gives a good approximation to the active set. The first phase of the algorithm uses this information to construct a basis by partitioning the variables into three sets:

1. $W = \{i \in \{1, \ldots, n\} \mid \hat{z}_i = l_i \text{ and } \hat{w}_i > 0\}$
2. $V = \{i \in \{1, \ldots, n\} \mid \hat{z}_i = u_i \text{ and } \hat{v}_i > 0\}$
3. $Z = \{1, \ldots, n\} \setminus W \cup V$

Since $(\hat{z}, \hat{w}, \hat{v})$ is a complementary triple, $Z \cap W \cap V = \emptyset$ and $Z \cup W \cup V = \{1, \ldots, n\}$.

Using the above guess, we can recover an invertible basis consistent with the starting point using the following procedure:

1. Let $A = \{\emptyset\}$.
2. While $M_{Z,Z}^{-1}$ does not exist:
   (a) Choose $k \in Z$ such that $M_{Z,k}$ is a linearly dependent column of $M_{Z,Z}$.
   (b) If $\hat{z}_k = l_k$, then $W = W \cup \{k\}$ and $Z = Z \setminus \{k\}$.
   (c) If $\hat{z}_k = u_k$, then $V = V \cup \{k\}$ and $Z = Z \setminus \{k\}$.
   (d) Otherwise, $A = A \cup \{k\}$ and $Z = Z \setminus \{k\}$.

The technique relies upon the factorization to tell the linearly dependent columns of $M_{Z,Z}$. LUSOL [25] and our dense factorization provide this information (see section 4.1). Some of the variables may be nonbasic, but not at their bounds. For such variables, the corresponding artificial will be basic. The above procedure is guaranteed to terminate because Z is a finite set and we remove one element from it each iteration.

The invertible basis constructed is then:

$$H = \begin{bmatrix} M_{\cdot,Z} & -I_{\cdot,W} & I_{\cdot,V} & I_{\cdot,A} \end{bmatrix}$$

To demonstrate that an inverse exists, we rearrange the rows and columns to obtain the following matrix, which is easily shown to be invertible because $M_{Z,Z}^{-1}$ is known to exist:

$$\hat{H} = \begin{bmatrix} -I_{W,W} & & & M_{W,Z} \\ & I_{V,V} & & M_{V,Z} \\ & & I_{A,A} & M_{A,Z} \\ & & & M_{Z,Z} \end{bmatrix}$$

We use a modified version of EXPAND [26] to perform the ratio test. Variables are prioritized as follows:

1. t leaving at its upper bound.
2. Any artificial variable.

3. Any z, w, or v variable.

If a choice as to the leaving variable can be made while maintaining numerical stability and sparsity, we choose the variable with the highest priority (lowest number above).

When an artificial variable leaves the basis and a z-type variable enters, we have the choice of either increasing or decreasing that entering variable because it is nonbasic but not at a bound. The determination is made such that t increases and stability is preserved.

If the code is forced to use a ray start at each iteration (`lemke_start always`), then the code carries out Lemke's method, which is known [8] not to cycle. However, by default, we use a regular start to guarantee that the generated path emanates from the current iterate. Under appropriate conditions, this guarantees a decrease in the nonlinear residual. However, it is then possible for the pivot sequence in the linear model to cycle. To prevent this undesirable outcome, PATH 2.9 uses a minor iteration limit. In PATH 3.0, we attempt to detect the formation of a cycle with the heuristic that if a variable enters the basis more that a given number of times, we are cycling. If this occurs, we terminate the linear solver at the largest value of t and return to the nonlinear pathsearch code.

Another heuristic is added when the linear code terminates on a ray. The returned point in this case is not the base of the ray. We move a slight distance up the ray and return this new point. If we fail to solve the linear subproblem five times in a row, a Lemke ray start will be performed in an attempt to solve the linear subproblem. Computational experience has shown this to be an effective heuristic and generally results in solving the linear model. Using a Lemke ray start is not the default mode, since typically many more pivots are required.

The resulting linear solver as modified above is robust and has the desired property that we start from $(\hat{z}, \hat{w}, \hat{v})$ and construct a path to a solution. This path is used by the nonlinear code for the implementation of pathsearching.

4.3 Pathsearching

An interpolating pathsearch that preserves stability replaces the pathsearch found in PATH 2.9. Instead of only using breakpoints, as outlined in [10], we allow points between the breakpoints to be used.

The EXPAND rules enforce a minimum stepsize. Therefore, some steps, and the first in particular, can be quite small. The breakpoint method has the undesirable property that a very small step may be taken in the backtrack. Therefore, we implemented a pathsearch that performs interpolation. We determine the section of the path where we want to find a point. We stably construct the two endpoints and interpolate between them to find the desired point.

Stability is achieved by using checkpoints, pivots where the basis matrix was refactorized and the basic variable values recomputed. At such a checkpoint, we know the basic variables, can reconstruct the basis matrix, factorize that matrix, and compute exactly the same basic variable values found during the original solve. Then we can perform the pivot sequence in order from the checkpoint. Once we have determined the two desired endpoints, we go to the appropriate checkpoint and regenerate the path followed to the desired endpoint, obtaining the exact values computed by the linear solve for those points. Careful bookkeeping minimizes the number of factorizations performed.

We note that difficulties can arise because t is not guaranteed to be monotonically increasing. In fact, it can decrease. To overcome this difficulty, we select a subsequence of the path over which t is monotonically increasing. We use this as the path over which we perform our nonlinear pathsearch.

In contrast, PATH 2.9 carried out backtracking by performing the sequence of pivots in reverse. This technique can generate additional numerical rounding error, leading on occasion to a failure in the backtracking procedure. The interpolating pathsearch removes these errors, is more stable, and furthermore checks a more varied sequence of iterates along the path.

4.4 Restarts

The PATH code attempts to fully utilize the resources provided by the user to solve the problem. PATH 3.0 is much more aggressive in its determination that a stationary point of the residual of the normal map has been encountered. When we determine that no progress is being made, we restart the problem from the user-supplied initial point with a different set of parameters. The three sets of parameter choices used during restarts are given in Table 7.

These restarts give us the flexibility to change the algorithm in the hopes that the modified algorithm leads to a solution. The ordering and nature of the restarts were determined by empirical evidence based upon tests performed on real-world problems.

This technique can be contrasted with that used PATH 2.9, in which significant efforts were made in an attempt to move away from a stationary point by modifying algorithm parameters within a major iteration and only restarting if these modifications failed to make progress. In some cases, the heuristics used were successful. More often they simply used resources that could have been better spent by restarting. We believe that finding a stationary point and restarting, as done in PATH 3.0, is a significantly better strategy than the ad-hoc heuristics of PATH 2.9.

At each iteration of the algorithm, several different step types can be taken. In order to help the PATH 3.0 user, we have added a code letter indicating the step type to the log file. Table 8 explains the meaning of each code letter.

5 Computational Results

In this section we compare the results obtained from PATH 2.9 and PATH 3.0 using their standard default options on the problems comprising MCPLIB [11]. We do not compare PATH 3.0 to other codes since this was carried out in [3].

Restart Number	Parameter Values
1	crash_method none nms_initial_reference_factor 2 proximal_perturbation 1e-2*initial_residual
2	crash_method none proximal_perturbation 0
3	crash_method pnewton crash_nbchange_limit 10 nms_initial_reference_factor 2 nms_searchtype line

Table 7: Restart Definitions

Code	Meaning
B	A Backtracking search was performed from the current iterate to the Newton point in order to obtain sufficient decrease in the merit function.
D	The step was accepted because the Distance between the current iterate and the Newton point was small.
I	Initial information concerning the problem is displayed.
M	The step was accepted because the Merit function value is smaller than the non-monotone reference value.
O	A step that satisfies both the distance and merit function tests.
R	A Restart was carried out.
W	A Watchdog step was performed in which we returned to the last point encountered with a better merit function value than the non-monotone reference value (M, O, or B step), regenerated the Newton point, and performed a backtracking search.

Table 8: Step Type Codes

However, the results of [3] seem to indicate that PATH was the most robust and efficient code at that time. The problems are available online (as GAMS files) as documented in [11], along with some recent additions. The computations were carried out on a Sun UltraSparc 300 MHz processor with 256MB RAM. We report the numbers of successes (Succ), failures (Fail), function evaluations (F), and total time (Time) in seconds used by each algorithm on a given problem in the accompanying tables. The times reported are rounded to the nearest 1/10 second, resulting in several problems having solution times of 0.0 seconds. The size of the first instance of each problem is also given in these tables; note that several of the problems are modified within the GAMS code to include more variables or equations on subsequent runs. The termination criterion for both codes is identical, requiring the norm of the residual in the normal map to be less than 10^{-6}. We do not report results for GAMSLIB [6], since both algorithms solve all of the problems in the suite on defaults. The results displayed in Tables 9 and 10 indicate that PATH 3.0 is significantly more robust than PATH 2.9. We believe the main reasons for improved robustness are increased numerical accuracy (see comments in Section 4.3) and more aggressive restart procedures (see remarks in Section 4.4).

The CPU times in the tables show that in general, the increase in robustness is not due to more resources being expended, but by using the given resources more effectively. In general, PATH 3.0 takes about the same time as PATH 2.9, with some substantial improvements in CPU time on several problems. Because of the significant differences between the two codes, PATH 3.0 is sometimes slower than PATH 2.9. In some of these cases, PATH 2.9 exits prematurely because of numerical problems, i.e. the basis matrix becomes singular while backtracking. At other times, the extra basis factorizations performed by PATH 3.0 to determine an initial feasible basis and to backtrack stably account for the discrepancy. We also note that the aggressive restarts in PATH 3.0 can cause increased time because we restart the problem from the user-supplied starting point while PATH 2.9 may succeed quickly with one of its ad-hoc heuristics.

We firmly believe that computational times are the best indication of the efficiency of the code. However, we have also included the number of function evaluations in the tables. For most of the problems the costs associated with linear algebra dominate the costs of function evaluations. Of particular note in this class is the hanson problem, which due to our backtracking procedure takes more function evaluations, but spends less time on linear algebra leading to a faster computation. Our more careful backtracking procedure, while increasing robustness of the code, can lead to more function evaluations being performed. In particular, on the ehl_k* problems, in which function evaluations dominate linear algebra costs, we perform significantly more function evaluations than PATH 2.9. However, in summary we point out that PATH 3.0 is more robust, takes less time, and performs fewer function evaluations when we average over the entire test set.

Problem	Size	PATH 3.0			PATH 2.9		
		Succ(Fail)	F	Time	Succ(Fail)	F	Time
asean9a	10199	1 (0)	4	2.9	1 (0)	4	2.4
badfree	5	1 (0)	4	0.0	0 (1)	4080	0.9
bert_oc	5000	4 (0)	16	1.9	4 (0)	16	1.5
bertsekas	15	6 (0)	79	0.0	6 (0)	94	0.1
billups	1	0 (3)	557	0.1	0 (3)	13584	1.8
bishop	1645	1 (0)	43	51.3	1 (0)	40	15.5
box	44	355 (6)	7875	10.4	351 (10)	31298	43.6
bratu	5625	1 (0)	48	18.8	1 (0)	48	18.1
choi	13	1 (0)	5	0.5	1 (0)	5	0.4
colvdual	20	4 (0)	91	0.1	4 (0)	57	0.2
colvnlp	15	6 (0)	44	0.0	6 (0)	43	0.0
cycle	1	1 (0)	5	0.0	1 (0)	60	0.0
degen	2	1 (0)	7	0.0	1 (0)	2	0.0
denmark	2093	40 (0)	1556	532.3	36 (4)	1504	836.1
duopoly	63	1 (0)	340	0.9	0 (1)	575	2.4
ehl_k40	41	2 (1)	2346	25.1	2 (1)	6194	60.0
ehl_k60	61	3 (0)	1009	19.9	3 (0)	108	7.5
ehl_k80	81	3 (0)	449	19.9	3 (0)	99	11.9
ehl_kost	101	3 (0)	36	4.4	3 (0)	36	4.6
electric	158	1 (0)	578	1.3	1 (0)	297	1.2
eppa	1269	8 (0)	45	11.9	8 (0)	45	13.4
eta2100	296	1 (0)	16	0.4	1 (0)	30	0.6
explcp	16	1 (0)	6	0.0	1 (0)	8	0.0
forcebsm	184	1 (0)	11	0.1	1 (0)	11	0.1
forcedsa	186	1 (0)	11	0.1	1 (0)	11	0.1
freebert	15	7 (0)	63	0.0	7 (0)	73	0.1
gafni	5	3 (0)	15	0.0	3 (0)	13	0.0
games	16	23 (2)	1970	5.6	22 (3)	9205	31.9
hanskoop	14	10 (0)	128	0.1	10 (0)	146	0.1
hanson	487	2 (0)	231	1.6	2 (0)	118	4.7
hydroc06	29	1 (0)	7	0.0	1 (0)	8	0.0
hydroc20	99	1 (0)	10	0.1	1 (0)	11	0.1
jel	6	2 (0)	11	0.0	2 (0)	11	0.0
jmu	2253	1 (0)	38	15.9	1 (0)	40	14.0
josephy	4	8 (0)	96	0.0	8 (0)	87	0.1
kojshin	4	8 (0)	132	0.0	8 (0)	54	0.1
lincont	419	1 (0)	15	1.3	1 (0)	16	1.3
mathinum	3	6 (0)	59	0.0	6 (0)	60	0.1
mathisum	4	7 (0)	512	0.1	7 (0)	73	0.1

Table 9: Comparison of PATH 3.0 to PATH 2.9

Problem	Size	PATH 3.0			PATH 2.9		
		Succ(Fail)	F	Time	Succ(Fail)	F	Time
methan08	31	1 (0)	5	0.0	1 (0)	5	0.0
multi-v	48	3 (0)	48	0.2	2 (1)	1581	3.2
nash	10	4 (0)	24	0.0	4 (0)	24	0.1
ne-hard	3	1 (0)	37	0.0	0 (1)	4	0.0
obstacle	2500	8 (0)	104	8.7	8 (0)	104	7.5
olg	249	1 (0)	1396	10.5	0 (1)	2574	16.3
opt_cont	288	1 (0)	6	0.1	1 (0)	6	0.1
opt_cont127	4096	1 (0)	6	1.7	1 (0)	6	1.2
opt_cont255	8192	1 (0)	6	3.7	1 (0)	6	2.6
opt_cont31	1024	1 (0)	6	0.3	1 (0)	6	0.2
opt_cont511	16384	1 (0)	6	8.3	1 (0)	6	5.2
pgvon105	105	3 (1)	181	1.8	3 (1)	323	2.8
pgvon106	106	3 (3)	4447	22.6	1 (5)	5763	38.7
pies	42	1 (0)	15	0.0	1 (0)	13	0.0
powell	16	6 (0)	57	0.1	5 (1)	6561	4.7
powell_mcp	8	6 (0)	94	0.1	6 (0)	51	0.1
qp	4	1 (0)	3	0.0	1 (0)	2	0.0
romer	214	0 (1)	325	1.9	0 (1)	142	4.1
scarfanum	13	4 (0)	30	0.0	4 (0)	33	0.1
scarfasum	14	4 (0)	21	0.0	4 (0)	23	0.0
scarfbnum	39	2 (0)	47	0.1	2 (0)	49	0.1
scarfbsum	40	2 (0)	128	0.7	1 (1)	216	2.3
shubik	30	48 (0)	6241	4.2	40 (8)	13064	11.9
simple-ex	17	0 (1)	732	0.6	0 (1)	370	0.7
simple-red	13	1 (0)	11	0.0	1 (0)	11	0.0
sppe	27	3 (0)	27	0.0	3 (0)	29	0.0
tinloi	146	63 (1)	1638	12.2	56 (8)	9632	77.2
tobin	42	4 (0)	41	0.1	4 (0)	44	0.1
trade12	600	2 (0)	26	3.0	2 (0)	30	3.9
trafelas	2376	2 (0)	141	16.9	2 (0)	146	15.2
uruguay	2281	7 (0)	27	796.0	7 (0)	27	1477.29
Total		712 (19)	34344	1568.6	679 (52)	109015	2694.1

Table 10: Comparison of PATH 3.0 to PATH 2.9

6 Conclusion

We have described a redesign of the PATH solver that facilitates easier interfacing to a variety of computing environments and shown how such interfacing has been effected. The redesign has also improved robustness of the code, without a corresponding increase in computational resources used. This paper has carefully described the major enhancements to the implementation and provided some numerical justification of improved robustness.

References

[1] S. C. Billups. *Algorithms for Complementarity Problems and Generalized Equations*. PhD thesis, University of Wisconsin–Madison, Madison, Wisconsin, August 1995.

[2] S. C. Billups. Improving the robustness of complementarity solvers using proximal perturbations. Technical report, Center for Computational Mathematics, University of Colorado, Denver, Colorado, 1997.

[3] S. C. Billups, S. P. Dirkse, and M. C. Ferris. A comparison of large scale mixed complementarity problem solvers. *Computational Optimization and Applications*, 7:3–25, 1997.

[4] S. C. Billups and M. C. Ferris. QPCOMP: A quadratic program based solver for mixed complementarity problems. *Mathematical Programming*, 76:533–562, 1997.

[5] C. Bischof, A. Carle, P. Khademi, A. Mauer, and P. Hovland. ADIFOR 2.0 user's guide. Mathematics and Computer Science Division Report ANL/MCS–TM–192, Argonne National Laboratory, Argonne, Illinois, 1995.

[6] A. Brooke, D. Kendrick, and A. Meeraus. *GAMS: A User's Guide*. The Scientific Press, South San Francisco, CA, 1988.

[7] Chunhui Chen and O. L. Mangasarian. A class of smoothing functions for nonlinear and mixed complementarity problems. *Computational Optimization and Applications*, 5:97–138, 1996.

[8] R. W. Cottle and G. B. Dantzig. Complementary pivot theory of mathematical programming. *Linear Algebra and Its Applications*, 1:103–125, 1968.

[9] J. Czyzyk, M. P. Mesnier, and J. J. Moré. The Network-Enabled Optimization Server. Preprint MCS-P615-0996, Argonne National Laboratory, Argonne, Illinois, 1996.

[10] S. P. Dirkse. *Robust Solution of Mixed Complementarity Problems.* PhD thesis, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1994. Available from ftp://ftp.cs.wisc.edu/math-prog/tech-reports/.

[11] S. P. Dirkse and M. C. Ferris. MCPLIB: A collection of nonlinear mixed complementarity problems. *Optimization Methods and Software*, 5:319–345, 1995.

[12] S. P. Dirkse and M. C. Ferris. The PATH solver: A non-monotone stabilization scheme for mixed complementarity problems. *Optimization Methods and Software*, 5:123–156, 1995.

[13] S. P. Dirkse and M. C. Ferris. A pathsearch damped Newton method for computing general equilibria. *Annals of Operations Research*, 1996.

[14] S. P. Dirkse and M. C. Ferris. Crash techniques for large-scale complementarity problems. In M. C. Ferris and J. S. Pang, editors, *Complementarity and Variational Problems: State of the Art*, Philadelphia, Pennsylvania, 1997. SIAM Publications.

[15] S. P. Dirkse and M. C. Ferris. Modeling and solution environments for MPEC: GAMS & MATLAB. In M. Fukushima and L. Qi, editors, *Nonsmooth, Piecewise Smooth, Semismooth and Smoothing Methods.* Kluwer Academic Publishers, 1998, forthcoming.

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

[17] A. Drud. CONOPT: A GRG code for large sparse dynamic nonlinear optimization problems. *Mathematical Programming*, 31:153–191, 1985.

[18] S. K. Eldersveld and M. A. Saunders. A block-LU update for large–scale linear programming. *SIAM Journal on Matrix Analysis*, 13:191–201, 1992.

[19] M. C. Ferris, R. Fourer, and D. M. Gay. Expressing complementarity problems and communicating them to solvers. Mathematical Programming Technical Report 98-02, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1998.

[20] M. C. Ferris and S. Lucidi. Nonmonotone stabilization methods for nonlinear equations. *Journal of Optimization Theory and Applications*, 81:53–71, 1994.

[21] M. C. Ferris, M. P. Mesnier, and J. Moré. NEOS and condor: Solving nonlinear optimization problems over the Internet. Mathematical Programming Technical Report 96-08, Computer Sciences Department, University of Wisconsin, Madison, Wisconsin, 1996 (Revised 1998). Also available as

MCS-P616-1096, Mathematics and Computer Science Division, Argonne National Laboratory.

[22] R. Fletcher and S. P. J. Matthews. Stable modifications of explicit LU factors for simplex updates. *Mathematical Programming*, 30:267–284, 1984.

[23] R. Fourer, D.M. Gay, and B.W. Kernighan. *AMPL: A Modeling Language for Mathematical Programming*. Duxbury Press, 1993.

[24] D. M. Gay. Hooking your solver to AMPL. Technical report, Bell Laboratories, Murray Hill, New Jersey, 1997. Revised 1994, 1997.

[25] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. Maintaining LU factors of a general sparse matrix. *Linear Algebra and Its Applications*, 88/89:239–270, 1987.

[26] P. E. Gill, W. Murray, M. A. Saunders, and M. H. Wright. A practical anti-cycling procedure for linearly constrained optimization. *Mathematical Programming*, 45:437–474, 1989.

[27] A. Griewank, D. Juedes, and J. Utke. ADOL-C: A package for the automatic differentiation of algorithms written in C/C++. *ACM Transactions on Mathematical Software*, 1996.

[28] N. H. Josephy. Newton's method for generalized equations. Technical Summary Report 1965, Mathematics Research Center, University of Wisconsin, Madison, Wisconsin, 1979.

[29] C. E. Lemke and J. T. Howson. Equilibrium points of bimatrix games. *SIAM Journal on Applied Mathematics*, 12:413–423, 1964.

[30] M. J. Litzkow, M. Livny, and M. W. Mutka. Condor: A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 104–111, June 1988.

[31] B. A. Murtagh and M. A. Saunders. MINOS 5.0 user's guide. Technical Report SOL 83.20, Stanford University, Stanford, California, 1983.

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

[33] S. M. Robinson. Normal maps induced by linear transformations. *Mathematics of Operations Research*, 17:691–714, 1992.

[34] S. M. Robinson. Newton's method for a class of nonsmooth functions. *Set Valued Analysis*, 2:291–305, 1994.

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

[36] T. F. Rutherford. Extensions of GAMS for complementarity problems arising in applied economic analysis. *Journal of Economic Dynamics and Control*, 19:1299–1324, 1995.

[37] T. F. Rutherford. Applied general equilibrium modeling with MPSGE as a GAMS subsystem: An overview of the modeling framework and syntax. *Computational Economics, forthcoming*, 1997.