Modeling and Solution Environments for MPEC: GAMS & MATLAB *

Steven P. Dirkse[†] Michael C. Ferris[‡]

October 1997, Revised January 1998

Abstract

We describe several new tools for modeling MPEC problems that are built around the introduction of an `MPEC` model type into the GAMS language. We develop subroutines that allow such models to be communicated directly to MPEC solvers. This library of interface routines, written in the C language, provides algorithmic developers with access to relevant problem data, including for example, function and Jacobian evaluations. A MATLAB interface to the GAMS `MPEC` model type has been designed using the interface routines. Existing MPEC models from the literature have been written in GAMS, and computational results are given that were obtained using all the tools described.

Keywords

Complementarity, Algorithm, MPEC, Modeling

1 Introduction

The Mathematical Program with Equilibrium Constraints (MPEC) arises when one seeks to optimize an objective function subject to equilibrium contraints. These equilibrium constraints may take the form of a variational inequality or complementarity problem, or they may be implicitly defined by a second-level optimization problem, the so-called bilevel programming

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

[†]GAMS Development Corporation, 1217 Potomac Street NW, Washington, D.C. 20007 (`steve@gams.com`).

[‡]Computer Sciences Department, University of Wisconsin – Madison, 1210 West Dayton Street, Madison, Wisconsin 53706 (`ferris@cs.wisc.edu`).

problem. Problems of this form have existed for quite some time but have recently become the subject of increased interest, as evidenced by the monograph [23] describing the theoretical and algorithmic state of the art for these problems. In this paper, we briefly examine the currently available modeling and solution environments for MPEC problems. We then describe an extension to the GAMS [4] modeling language that aids effective modeling of the MPEC problem and software libraries that allow MPEC solvers written in C, Fortran, and MATLAB to access these MPEC models.

Currently, a person wishing to solve an MPEC problem might start by choosing a solver designed for this purpose, most likely one written in C, Fortran or MATLAB. They would then formulate their problem to interface correctly with that particular solver. This would require them to code the functions defining the problem, and perhaps first or second derivatives as well. They would need to provide this information in the (non)sparse format required by the solver, and satisfy any other requirements peculiar to their algorithm of choice. Such an approach is undesirable from a modeling standpoint simply because it ignores the organizational complexity of correctly assembling (and reassembling) data from many different sources into a single model. In addition, such an approach limits the modeler to using the target solver. If this solver fails, major changes may be required to put the problem into a form suitable for another solver.

These problems (and others) are exactly those that led to the development and use of modeling languages, first for linear and then nonlinear programming. These modeling languages help organize and automate the modeling task and allow this to be done in a machine- and solver-independent manner. In addition to GAMS, other modeling languages exist (e.g. AMPL [14], AIMMS [3]) that have a great deal in common. However, GAMS is currently the only one to support the mixed complementarity, or `MCP` [10], model type, although a similar AMPL interface [12] is currently under construction. The addition of the complementarity model to GAMS has been well received, becoming widely used in economics, structural engineering and other fields. In addition, the collection of models MCPLIB [6] and its MATLAB interface [13] has been very useful to the community of algorithm developers for the complementarity problem. It has been a convenient source of test problems, a basis for computational comparison between algorithms, and a factor in advancing the computational state of the art. We hope that the tools described in this paper can be similarly useful to those modeling and developing algorithms for MPEC problems.

The rest of the paper is organized as follows. In Section 2, we introduce terminology and provide a definition of the MPEC problem, as well as an example. Our definition is intimately related to the definition of the `MPEC` model type in GAMS, the subject of Section 3. Once a problem is formulated using the `MPEC` model type, the problem is made available to a solver via the MPECIO software library described in Section 4. It is also possible to access the problem from within MATLAB using an additional interface layer, the subject of Section 5.

As an example of how all these tools can be used, we have formulated several MPEC models from the literature in GAMS and have solved them using C, Fortran, and MATLAB solvers. Preliminary computational results are given in Section 6.

2 MPEC definition

An MPEC is a constrained nonlinear programming problem in which some or all of the constraints are formulated in terms of the solution of a second level problem. A popular form in which such constraints arise is as equilibrium constraints. A simple example is that of maximizing revenue generated from the tolling of roads on a traffic system subject to that traffic system being in equilibrium. Another example involves generating optimal taxation policies under the assumption that an economy is in equilibrium.

A crucial part of the MPEC problem definition is the system of equilibrium constraints. These can be defined in a number of ways. For example, they can arise as the solution to an optimization problem or using generalized equations [30, 29], variational inequalities [17], the min operator [17], or a complementarity problem [17]. In our case, we will assume that the equilibrium system always takes the form of a mixed complementarity problem, or MCP, defined in terms of some (possibly infinite) lower and upper bounds $\ell \in (\mathbf{R} \cup \{-\infty\})^n$ and $u \in (\mathbf{R} \cup \{+\infty\})^n$ satisfying $-\infty \leq \ell_i < u_i \leq +\infty$ and a nonlinear function $F\colon \mathbf{B} \to \mathbf{R}^n$. Throughout the paper, we will use the notation $\mathbf{B}$ to represent the box $\mathbf{B} := [\ell, u] = \{y \in \mathbf{R}^n \colon \ell_i \leq y_i \leq u_i\}$. The variable $y \in \mathbf{R}^n$ solves $MCP(F, \mathbf{B})$ if the following holds:

$$\begin{array}{llll} & F_i(y) = 0 & \text{and} & \ell_i < y_i < u_i \\ \text{or} & F_i(y) \geq 0 & \text{and} & y_i = \ell_i \\ \text{or} & F_i(y) \leq 0 & \text{and} & y_i = u_i. \end{array} \tag{1}$$

The MCP definition is quite compact; only F and $\mathbf{B}$ need be specified. It is entirely equivalent to the box-constrained or rectangular VI. This allows a certain simplicity in formulation and solution. However, the formulation is general enough to include as special cases the nonlinear complementarity problem (NCP) where $\mathbf{B} := [0, +\infty]$ and nonlinear systems of equations where $\mathbf{B} := [-\infty, +\infty]$. The box $\mathbf{B}$ allows both free variables and variables with one or two finite bounds, hence the mixed nature of the problem. This results in improved efficiency of modeling and problem solution, as compared with the fixed box $[0, +\infty]$ of the NCP. We will use the shorthand notation $F(y) \perp y \in \mathbf{B}$ for (1) as a generalization of the orthogonality that holds in the case of the NCP.

We define the MPEC over the design variables $x \in \mathbf{X} \subseteq \mathbf{R}^n$ and state variables $y \in \mathbf{Y} \subseteq \mathbf{R}^m$. There is an objective function $\theta: \mathbf{R}^{n+m} \to \mathbf{R}$ to be optimized, subject to two types of constraints. The first type of constraints require (possibly joint) feasibility of the variables (x, y) and are determined by the functions $h: \mathbf{R}^n \to \mathbf{R}^k$ and $g: \mathbf{R}^{n+m} \to \mathbf{R}^p$ and the box $\mathbf{X}$. The equilibrium constraints are defined by the box $\mathbf{Y}$ and the function $F: \mathbf{R}^{n+m} \to \mathbf{R}^m$. The latter constraints require the state variables y to solve the MCP defined by $\mathbf{Y}$ and (parametrically) by $F(x, \cdot)$. Put succinctly, we have

$$
\begin{aligned}
\text{minimize} \quad & \theta(x, y) && \text{(2a)} \\
\text{subject to} \quad & h(x) \in H && \text{(2b)} \\
& x \in \mathbf{X} && \\
& g(x, y) \in G && \text{(2c)} \\
\text{and} \quad & y \text{ solves } \mathrm{MCP}(F(x, \cdot), \mathbf{Y}) && \text{(2d)}
\end{aligned}
$$

where the sets G and H are used to signify that the given constraint could either be an inequality or an equation.

Note that the definition above is quite general in allowing for feasibility constraints of the form (2b) and (2c). The models from the literature do not require constraints of the form (2c); many omit the constraints (2b) as well. This is due primarily to the scarcity of techniques applicable to problems with the joint feasibility constraint (2c). We include this constraint in the hope that solvers for this type of model will soon become available, and because it is a programmatically trivial task to generalize from (2b) to (2c) in the GAMS model definition and in the solver interface routines of Section 4.

As an example, the following problem exercises the full generality of our definition:

$$
\begin{array}{ll}
\text{minimize} & \theta(x,y) := (x-1-y)^2 \\
\text{subject to} & x^2 \le 2 \\
 & \ell \le x \le u \\
 & (x-1)^2 + (y-1)^2 \le 3 \\
 & y - x^2 + 1 \ \perp \ y \ge 0
\end{array}
\tag{3}
$$

The final line above represents the fact that y solves the NCP with $F(x,y) = y - x^2 + 1$. The MPEC model as we have defined it generalizes both the nonlinear program and the MCP. Those doing modeling work in complementarity are eager to adapt existing MCP models to take advantage of this framework. For example, one can optimize revenue or congestion in traffic equilibrium problems over a set of feasible tolls [9], or convert a Nash equilibrium model to a Stackelberg (leader-follower) game [33]. Typically, the optimization problem has relatively few design variables and many more state variables. This approach has implications for algorithm design, which can exploit the fact that the number of design variables and "side constraints" h and g is very small.

One can also view the problem as an NLP generalized to include some equilibrium constraints. This approach is taken by chemical process engineers, who have equations that are valid only for certain states of the system [28]. In these models, the contraints g and h may dominate, requiring a different kind of algorithm. In either case, the problem fits well into the MPEC framework, and into the GAMS `MPEC` model, which we now describe.

3 The `MPEC` model type

The MPEC definition (2) combines components of NLP and MCP definitions. In the same way, the GAMS `MPEC` model type borrows from both the `NLP` and `MCP` model types. This makes the programming task easier, but more importantly, it allows users to move from MCP or NLP to MPEC models with little difficulty. The use of the usual GAMS constructs (e.g. sets, parameters, variables, equations, control structures) is unchanged; the only changes are those made to the GAMS `model` and `solve` statements.

The `model` statement is used to associate a model name with a list of equations and equation-variable pairs that define the model. In the MPEC case, there must be one (equality constrained) equation that defines the objective variable (one specifies an objective *variable* in GAMS, not an objective row). The objective variable must appear linearly in this equation, and must not appear in any other equation of the model, so that it can be substituted out by the interface, leaving an objective function of the form (2a). Typically, an equation "cost" is declared of the form:

```
cost.. theta =e=  some gams expression;
```

The objective equation can appear (unpaired) anywhere in the model list. In addition, other equations defining constraints to be satisfied can be included in the model list, just as in an NLP. These define the constraints (2b) and (2c); the partition occurs automatically.

Mathematically, the equilibrium constraints (2d) are defined by matching the state variables y with equations. In GAMS, this is done by including an equation-variable pair in the model list. Each pair defines a complementarity relationship between the function determined by the equation and the variable in the pair. In the case of equations and variables indexed by sets, functions and variables with matching indices are paired. These define the equilibrium constraints (2d) in exactly the same way as for the GAMS `MCP` model type. It also specifies the partition into design and state variables. Variables appearing in pairs are state variables, while those not appearing are design variables. Note a subtle difference between the MPEC model statement and the MCP model statement: in the MCP model, there can be equations that are not explicitly paired with (free) variables. In the MPEC case, any equation which is not matched is assumed to be a side constraint, while unmatched variables are deemed to be design variables. In GAMS, variable bounds are attributes of the variables, so the boxes $\mathbf{X}$ and $\mathbf{Y}$ are given as well, and the problem (2) is completely specified.

From the modeler's perspective, the `solve` statement for `MPEC` models is no different than for other model types; it simply instructs GAMS to solve the model indicated. The modeler indicates that this is an `MPEC` model, the variable to optimize, and whether to maximize or minimize. GAMS will do a number of checks (bound consistency, nonlinear functions defined at initial point, functions smooth) before writing the problem to disk as a sequence of scratch files and calling an MPEC solver. This MPEC solver will use the interface library routines of Section 4 to read and interpret the scratch files, evaluate functions and gradients, and write solution data.

As an example, we include the GAMS code for the simple example (3) in Figure 1. The file three.gms that is depicted in Figure 1 can be obtained from the MPEC Web site `http://www.gams.com/mpec/`. Other example files can also be found at this site, along with the codes that form the content of the next two sections.

```
* simple MPEC model
variables
theta,
x             'design variables',
y             'state variables';
x.lo = -1;
x.up = 2;

equations
cost,
h,
g,
F;

cost ..  theta =e= sqr(x-1-y);

h ..  sqr(x) =l= 2;

g ..  sqr(x-1) + sqr(y-1) =l= 3;

F ..  y - sqr(x) + 1 =g= 0;


model three / cost, h, g, F.y /;

option mpec=bundle;
solve three using mpec minimizing theta;
```

Figure 1: GAMS Model for (3)

4 MPECIO

In Section 3, we described how an MPEC model is processed by GAMS and sent to a solver. This communication is currently done via files, but could also be done in (virtual) memory, via the Internet, or using some other physical layer. This involves many details that depend on the operating system, the file system, the compilers used, etc. The GAMS I/O library insulates the solver from these details and allows access to the problem in a standard form. In this section, we describe the interface to an extension of the GAMS I/O library that will read and interpret the scratch files, evaluate functions and gradients for the MPEC problem, and write solution data. The reader is assumed to have some familiarity with C, since the interface we now describe is written in that language.

Interface Initialization

```
int mpecInit (char *controlFileName, int indexStart,
              int diagRequired, conType_t conType,
              mpecRec_t **mpec);
void sparseInit (mpecRec_t *mpec, int colPtrx[], int cpxdim,
                 int rowIdxx[], int rixdim,
                 int colPtry[], int cpydim,
                 int rowIdxy[], int riydim,
                 int colPtrh[], int cphdim,
                 int rowIdxh[], int rihdim,
                 int colPtrg[], int cpgdim,
                 int rowIdxg[], int rigdim);
```

The first task of the solver is to call the `mpecInit` routine to read in the scratch files and construct a problem of the form (2). If there is any inconsistency in the MPEC specification, it is detected during this initialization phase. It is here that the variables are partitioned into design and state variables, and the required maps are set up to support efficient function and gradient evaluation by other routines. Parameters to `mpecInit` exist allowing the user to specify if index ranges must begin with 0 (C style) or 1 (Fortran style), and whether or not extra space should be allocated to store all the zero elements of the diagonal of the Jacobian. Since there is some overhead in allowing for the side constraints h (2b) and g (2c) and many solvers will not be able to handle these constraints if they are present, the parameter `conType` exists to indicate whether to allow for no side constraints, those of the form (2b) or (2c) only, or both (2b) and (2c). A pointer to a record containing all information specific to this model (e.g. dimensions, nonzero estimates) is passed back to the calling routine. This pointer will be passed on all subsequent MPECIO calls.

In order to fully initialize the MPEC model, some space is required for the row indices and column pointers used to store the sparse Jacobians. Rather than allocating this space inside the library, the `mpecInit` routine returns estimates of the amount of space required for this to the calling routine. A second routine, `sparseInit`, must then be called, which passes in these arrays, as well as the number of elements actually allocated for them. This routine completes the initialization, using the space provided to it. The assumption here is that the user will not modify these arrays, as they are used by both the solver and the interface library. This assumption saves having to store two copies of the sparsity structure and copy it to the user's data structure at each derivative evaluation.

Variable Bounds and Level Values

```
void getxBounds (mpecRec_t *mpec, double lb[], double ub[]);
void getxLevels (mpecRec_t *mpec, double x[]);
void setxbar (mpecRec_t *mpec, double xbar[]);
void getyBounds (mpecRec_t *mpec, double lb[], double ub[]);
void getyLevels (mpecRec_t *mpec, double y[]);
```

The routines to obtain variable bounds and initial level values are for the most part self-explanatory. The `setxbar` routine is used to store a vector of design variables $\bar{x}$ in MPECIO for use in subsequent calls to function and gradient routines that pass only state variables y. This is useful for solvers that implement a two-level solution scheme in which the inner solver (an MCP code) has no knowledge of the variables x in the outer optimization problem. In our current implementation, the box $\mathbf{Y}$ does not depend on x, so the `getyBounds` routine would be called only once. A possible generalization is to allow $\mathbf{Y}$ to depend on x, in which case a new function with the input parameter x would be required. This function would of course be called whenever x is changed.

Function and Jacobian Evaluation

```
int getF ( mpecRec_t *mpec, double x[], double y[],
           double F[]);
int getdF (mpecRec_t *mpec, double x[], double y[],
           double F[],
           double Jx[], int colPtrx[], int rowIdxx[],
```

```
            double Jy[], int colPtry[], int rowIdxy[]);
int getFbar ( mpecRec_t *mpec, int n, double y[], double F[]);
int getdFbar (mpecRec_t *mpec, int n, int nnz, double y[],
              double F[],
              double Jy[], int colPtry[], int rowIdxy[]);
```

The routine getF takes the current point (x, y) as input and outputs the value of the function F at this point. The routine getdF computes the derivative of F as well. The derivative of F w.r.t. x and y is returned in separate matrices, both of which are stored sparsely in row index, column pointer fashion. The routines getFbar and getdFbar are similar, but in these routines, the input x is assumed to be the constant value $\bar{x}$ fixed in the previous call to setxbar. In this case, derivatives w.r.t. x and objective function values and derivatives are not passed back. These routines are designed for use by an algorithm solving an inner (MCP) problem.

```
int getObj (  mpecRec_t *mpec, double x[], double y[],
              double *obj);
int getdObj ( mpecRec_t *mpec, double x[], double y[],
              double *obj,
              double dObjDx[], double dObjDy[]);
```

In some cases, a solver may need information about the objective function θ only. This is supported efficiently in the current library via the getObj and getdObj routines. The derivative of θ w.r.t. x and y is returned as two dense vectors.

```
int getConEqSense (mpecRec_t *mpec, int hEqSense[],
                   int gEqSense[]);
int getCon ( mpecRec_t *mpec, double x[], double y[],
             double h[], int eqSense[]);
int getdCon (mpecRec_t *mpec, double x[], double y[],
             double h[], double g[],
             double dh[], int hColPtr[], int hRowIdx[],
             double dg[], int gColPtr[], int gRowIdx[]);
```

The sense of the upper-level constraints $(\geq, \leq, =)$ is determined by a call to getConEqSense. In order to evaluate the side constraints h and g, one must use the getCon and getdCon routines. The level values of h and g are returned in h[] and g[]. The derivatives of h and g, if requested, are returned in row index, column pointer format. These calls return data for constraint types consistent with the model initialization (see `mpecInit` above). For example, if the model was initialized to allow for no side constraints, it is an error to call `getCon` or `getdCon`.

Solver Termination

```
void putxLevels (mpecRec_t *mpec, double x[]);
void putyLevels (mpecRec_t *mpec, double y[]);
void putObjVal (mpecRec_t *mpec, double obj);
void putStats (mpecRec_t *mpec, solStats_t *s);
int mpecClose (mpecRec_t *mpec);
```

Once a solution has been found, the solver must pass this solution on to the interface library. This is done via the `putxLevels`, `putyLevels`, and `putObjVal` routines. The `putStats` routine is used to report the model status (e.g. local optimum found, infeasible, unbounded, intermediate nonoptimal) and solver status (e.g. normal, iteration limit, out of memory, panic termination) via integer codes, as well as information about execution times, iterations used, function evaluation errors, etc. All of this information is stored in the `mpec` data structure and written to disk when the `mpecClose` routine is called. When the solver terminates, these files are read by GAMS so that the solution information is available for reporting purposes, as data to formulate other models, and to continue the execution of the GAMS model.

5 Interfacing to MATLAB solvers

As an example of the intended use of MPECIO, we have developed an interface to the MATLAB programming environment. Our implementation is based on that developed for GAMS/MCP in [13]. We have attempted to make the interface to MPEC models more robust and portable than the MCP interface, while maintaining the ease of use of the tools within MATLAB.

The main difference between the MPEC interface and the MCP interface is in how the data is stored and accessed from within MATLAB. In the MCP interface, a large vector containing all the global data from the problem in question is copied to the user workspace. This technique is not very portable, since this large vector is stored in a machine-specific form, and requires different data files for different machine types. In addition, a special GAMS

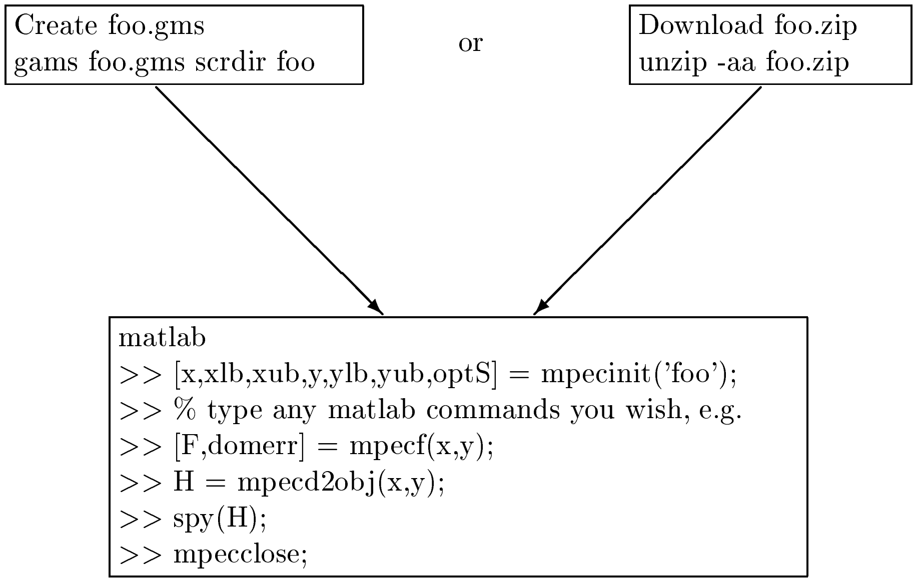

Figure 2: Two ways to run Matlab on a given problem.

"solver" is required to create this large vector. In contrast, the MPEC interface to MATLAB accesses the data in exactly the same manner that a normal solver for GAMS does, using MPECIO as a machine-independent layer between it and the GAMS data files. Therefore, the only issues are to build or obtain MPECIO for the required platform and provide the machine dependent (or independent) files that are typically accessed by a solver.

We have provided a solver "mpecdump", callable from GAMS, that creates relevant scratch files on the user's platform in binary format. In this manner, any GAMS user can write their own MPEC problems in GAMS and generate all the relevant scratch files on their own machine. The solver mpecdump is freely available for download from the MPEC Web site `http://www.gams.com/mpec/`.

For users without access to GAMS, we have created a collection of MPEC problems that can be downloaded to any machine running MATLAB and used in conjunction with the interface routines we now describe. These problems are all stored in compressed "zip" format and can be downloaded from the MPEC Web site. For portability purposes, each zip file contains the ASCII files defining a single problem. It is intended that this collection grow into a library of test problems for MPEC. Figure 2 depicts the two ways that can be used to access the data of a particular problem.

The remainder of the MATLAB interface consists of a single MEX file and several simple accompanying m-files. The idea behind using a single MEX file, mpecfunc.c, is to allow all the m-files to access a single mpec structure, namely the mpecRec_t structure provided by MPECIO, without explicitly declaring this structure in the MATLAB workspace. Furthermore, since we envision that the structure used by MPECIO may be modified in the future to enable further enhancements to the modeling format, any changes to the MATLAB interface will be transparent to the MATLAB user. The source code of the mpecfunc routine is available via the MPEC Web site `http://www.gams.com/mpec/`. Together with the object code of MPECIO, this source allows any user to build a version of the MATLAB interface for their architecture, which they can then use to read in the problem data files they download and those they create from their own GAMS models. All of this can be done independently of the authors of this paper.

Currently, mpecfunc can be called with 0, 2, or 3 input arguments. The call with a string argument and an integer allows a simple implementation of the initialization routine:

```
[x,xlb,xub,y,ylb,yub,optS[,gEq]] = mpecinit('foo');
```

The purpose of the routine is to set up the mpec structure and to assign values of the design and state variables into the MATLAB users workspace. First of all, the routine removes any previous mpec structure that is in the MATLAB workspace. It then checks whether a directory "foo" exists, containing the scratch files generated by GAMS, and aborts with an error message if they do not. The new mpec structure is created by calls to the MPECIO routines mpecInit and sparseInit. The bounds on the design and state variables are returned by this routine, using appropriate calls to the MPECIO routines get*Bounds and get*Levels. For simplicity, the MATLAB interface provides the constraints $h(x)$ and $g(x, y)$ in a single vector `g`. The optional return argument `gEq` informs the user if each of the constraints is a less than inequality (-1), an equality (0) or a greater than inequality (1). The mpecfunc routine allows the user to split these constraints into h and g respectively; interested users should examine the source code of mpecfunc.c.

We note here a nice feature of our m-file implementation, namely that the mpecinit m-file can be edited to explicitly call gams via a system call if the scratch files do not exist, but a model file "foo.gms" does exist. Other specializations of the MATLAB interface are just as easy to carry out.

In order to release the memory used by the mpec structure, a clean up routine is also provided. This function is implemented as an m-file that calls mpecfunc with no input arguments.

```
mpecclose;
```

As is to be expected, there are no return arguments for this function.

The remaining m-files pass three arguments to mpecfunc. The first argument is an internal code to indicate the function to be performed, the second passes the values of the design variables x, and the third argument passes the values of the state variables y. The particular internal codes can be seen by inspecting the source code to the m-files or mpecfunc.c.

The actual MATLAB routines currently provided are as follows:

```
[F,domerr] = mpecf(x,y);
[F,J,domerr] = mpecdf(x,y);
```

Here `x` and `y` are the values of the state and design variables respectively. The return arguments `F` and `J` contain the MCP function $F(x,y)$ and its Jacobian, while `domerr` is the number of domain errors encountered in the nonlinear function evaluator. Note that `J` is returned to the MATLAB user as a sparse matrix.

The following functions all return information about the objective function θ.

```
[theta,domerr] = mpecobj(x,y);
[theta,g,domerr] = mpecdobj(x,y);
H = mpecd2obj(x,y);
```

The first derivatives of θ are returned in the vector `g`; the second derivative matrix H as a sparse matrix. Note that `H` is calculated by finite differencing, since GAMS does not currently have the ability to provide second derivatives.

The side constraints are provided by the following routines, along with their derivatives.

```
[g,domerr] = mpeccon(x,y);
[g,Jg,domerr] = mpecdcon(x,y);
```

Details of the calling sequence for the MPECIO routines within the MATLAB interface can be obtained directly from the source code mpecfunc.c that can be found at `http://www.gams.com/mpec/`. Also, by editing the "m" files that implement `mpecdf`, `mpecdobj` and `mpecdcon` respectively, it is possible to return derivatives with respect to x and y to the MATLAB environment separately.

6 Computational results

We have linked a number of MPEC solvers to the collection of models we have put together using the MPECIO library from Section 4 and the MATLAB interface from Section 5. This has provided a thorough test of our interface routines and has helped to modify and improve their design. In addition, it allows us to perform some computational experiments using these codes and report on the results. Before doing so, a word on interpreting our results is in order. While computational work on the MPEC has not advanced to a stage where any of the solvers can be considered well known or completely robust, the techniques used in the solvers we report on are representative of the current state of the art in this area. It is hoped that the work described in this paper will allow more algorithms to be developed that robustly solve all these problems and other problems of interest to the larger optimization community.

Two of the solvers we test use an implicit programming approach in which the MPEC is reformulated as an NLP. In order to do this, our code assumes for simplicity that there are no side constraints (2b) or (2c) and further that there is a (locally) unique solution y of the equilibrium problem $\mathrm{MCP}(F(x,\cdot),\mathbf{Y})$ for each value of x. We denote this solution by $y(x)$. Under these assumptions, the problem (2) is equivalent to the implicit program:

$$\begin{array}{ll} \text{minimize} & \Theta(x) = \theta(x, y(x)) \\ \text{subject to} & x \in \mathbf{X}. \end{array} \tag{4}$$

This implicit programming formulation has the advantage of simple constraints, but a rather complex, in fact nondifferentiable objective function Θ, even though the original functions θ and F may be smooth. Nonsmoothness results from the underlying equilibrium conditions.

One solution strategy for the implicit program (4) is to apply a "bundle method", an algorithm specifically designed to solve nonsmooth optimization problems. This idea is presented in [20, 21, 25, 26]. The implementation of the bundle method we used, `btncbc` [31], is a Fortran subroutine developed for nonconvex bound constrained problems. One of the arguments to this subroutine is a routine to evaluate the objective function and a generalized gradient at an algorithm-specified point. We coded this routine in C, using the formulas developed in [26] for the MPEC formulation (2) without the side constraints (2b) and (2c). This C routine uses the MPECIO library and a version of PATH [7, 8] modified to return the basis at the solution to the inner MCP.

Another solver we have implemented also uses the implicit approach to reformulate (2) as (4). In this case, the latter problem is solved by SolvOpt 1.1 [22], a C implementation of Shor's R-algorithm [32] for unconstrained, nondifferentiable optimization problems. In order to handle the constraints $x \in \mathbf{X}$, an exterior penalty amounting to the weighted maximum constraint violation was used. In version 1.0 of SolvOpt, it was necessary to add this penalty to the objective function explicitly; version 1.1 provides some support for this, including automatic adjustment of the penalty parameter.

The SolvOpt user is required to write separate routines to compute θ and elements of its subdifferential $\partial\theta$. The constraint violations are penalized through θ, but in other respects, these routines are similar to the function / generalized gradient routine passed to the bundle code `btncbc`. Both SolvOpt and the bundle code are implemented to run as GAMS subsystems, using the MPECIO library as their interface to GAMS.

In order to test the MATLAB interface of Section 5, we obtained a prototype MATLAB implementation [19] of the penalty interior point approach (PIPA) code ([23], Chapter 6) from H. Jiang and D. Ralph. This code (that we call PIPA-N) is specialized for problems whose equilibrium constraints (2d) take the form of an NCP. The PIPA algorithm computes a search direction via a solution to a uniquely solvable QP subproblem and performs an Armijo search using this direction. In order to use the PIPA implementation with our MATLAB interface library, it was necessary to provide m-files `f.m` (to compute the objective function θ and the equilibrium function F), `df.m` (to compute derivatives of `f.m`), and `d2f.m` (to compute second derivative information for θ only). The first two files `f.m` and `df.m` are trivial, and amount to little more than wrapper functions for our interface routines. The right choice for `d2f.m` is not so obvious.

In [23], the description of and proofs of convergence for PIPA assume only that the matrix Q in the direction-finding QP is symmetric positive semidefinite; it is not necessary to use any second-order information. Computationally, this is very appealing, since this makes `d2f.m` trivial (one can use the identity matrix, for example), but our results using the identity matrix were quite discouraging. This was also the experience of the authors of the code, hence their use of the Hessian of θ. Unfortunately, second order information is not currently available in GAMS, nor consequently to MPECIO or the MATLAB interface. To improve our results, we modified the MATLAB code to compute a BFGS approximation to the Hessian of θ. This was an improvement over using an identity matrix Q, but the results were still not what we had hoped, so we implemented a simple finite difference scheme for computing the Hessian of θ, in which our only enhancement was to recognize variables appearing linearly in θ and avoid computing the corresponding columns of the Hessian.

During our computational tests, we discovered that many of the problems we wanted to solve had equilibrium constraints that did not take the form of an NCP. In some cases, problems had variables whose lower bounds could be shifted to 0, but more often there were free variables or variables with upper and lower bounds, so that a transformation into NCP form is much less appealing. Fortunately, we were able to obtain a second implementation [19] of PIPA (that we call PIPA-M) that allowed problems of the more general form

$$\begin{array}{ll} \text{minimize} & \theta(x,y,w,z) \\ \text{subject to} & Ax \leq b \\ & \ell \leq x \leq u \\ & F(x,y,w,z) = 0 \\ & 0 \leq w \ \perp \ y \geq 0 \end{array} \tag{5}$$

where $x \in \mathbf{R}^n$, $y, w \in \mathbf{R}^m$, $z \in \mathbf{R}^l$, and $F\colon \mathbf{R}^{n+m+m+l} \to \mathbf{R}^{m+l}$. The requirements for using PIPA-M are similar to using the NCP version: three MATLAB m-files `f.m`, `df.m`, and `d2f.m`. However, the transformation from (2) to (5) is more complex. Lower-bounded variables from (2) need to be shifted so that their lower bound is zero, while upper bounded variables are first negated, along with their bounds, and then shifted. Free variables require no transformation. In the most difficult case, variables from (2) with finite upper and lower bounds are converted as follows: each scalar pair $\hat{F}(z) \perp z \in [\ell, u]$ becomes

$$\begin{array}{rcl} 0 \leq w_\ell = (z - \ell) & \perp & y_\ell \geq 0 \\ 0 \leq w_u = (u - z) & \perp & y_u \geq 0 \\ \hat{F}(z) - y_\ell + y_u = 0 & \perp & z \end{array}$$

While this transformation does increase the problem size, it does so quite sparsely, adding 2 equations and 6 nonzeroes for each double-bounded variable.

Our computational tests of these solvers and the interface tools of Section 4 and Section 5 were carried out using a preliminary version of MPECLIB, a library of MPEC models formulated in the GAMS language. We do not describe all of these models here, because most of them have been described elsewhere and due to the fact that we have made the GAMS source for these models available via the Web `http://www.gams.com/mpec/`. The file fjq1.gms contains problems 1-4 from [11], desilva.gms is described in [5], hq1.gms in [18], oz3.gms in [26], mss.gms in [24], qvi.gms in [16], and bard3.gms in [2]. For bard3 and oz3, we move the constraints (2b) into the complementarity problem. In the problems where the second level problem is an optimization problem or variational inequality, we add explicit multipliers to rewrite the second level problem as an MCP.

All of our computation was done on a Sun Ultra Enterprise 2 with 256 MB of RAM and two 300MHz processors running Solaris 2.5 and equipped with the standard C, F77, and MATLAB (version 5.1.0) cmex compilers. Table 1 contains results obtained from the GAMS subsystems BUNDLE and SolvOpt. We report the number of generalized gradient evaluations, the final objective value θ for the original problem, and the CPU time in seconds. For SolvOpt, we also report the number of evaluations of the implicit objective function in the column headed k_Θ, while in the BUNDLE code, the objective and a generalized gradient are always returned together. We report the iteration count k for BUNDLE, since some of its iterations may need to compute multiple generalized gradients. Iterations that terminated with a non-optimal solution or status are marked with a † in the grad column. In some cases, the BUNDLE and SolvOpt codes failed due to an unsolved inner problem or failed internal tolerance checks; these are indicated as "fail".

It is interesting to note that the optimal value of 0 for the oz3 problem in Table 1 is quite an improvement over the value of 4.999375 reported in [11].

In our computational work, we tested 6 distinct MATLAB codes: both the general and the NCP-constrained code were tested using a constant Q matrix for the QP subproblem, using an approximate Hessian of θ obtained via BFGS updating, and using a difference approximation to the Hessian. However, the results obtained using the constant Q were disappointing, so are not included in Table 2. Since many of the models were not NCP-constrained, we do not report on the BFGS version of this algorithm either. We are left with algorithm PIPA-N (the NCP-constrained version of PIPA with a difference approximation to the Hessian), PIPA-M (the MCP-constrained version of PIPA, difference approximation), and PIPA-B (the MCP-constrained version of PIPA using BFGS updates). For each of these three techniques, we report the iteration count k (maximum 1000), the final objective value θ, and the CPU time in seconds. Each iteration involves the solution of one direction-finding QP subproblem; these subproblems are solved using the standard MATLAB QP solver.

Where a model was not NCP-constrained and PIPA-N could not be applied, this is so indicated in Table 2. We also indicate cases where the

		BUNDLE			SolvOpt			
model	k	grad	θ	secs	grad	k_Θ	θ	secs
bard3	1	1	-12.678	.01	$15^\dagger$	16	-11.80	
desilva	9	9	-1	.01	15	70	-1	.03
epowplan	fail				127	485	-451.5	6.0
fjq1 a	10	10	3.208	.02	$2^\dagger$	8	3.309	
fjq1 b	10	12	3.208	.02	$1^\dagger$	1	4.502	
fjq2 a	13	13	3.449	.01	14	42	3.450	.05
fjq2 b	12	12	3.449	.01	$24^\dagger$	28	13.60	.02
fjq3 a	8	83	4.604	.02	$15^\dagger$	19	5.552	.03
fjq3 b	9	86	4.604	.04	$24^\dagger$	27	17.70	.03
fjq4 a	13	89	6.593	.03	11	25	6.593	.03
fjq4 b	13	13	6.593	.03	$12^\dagger$	31	6.595	.03
gauvin	13	13	20	.01	14	66	20	.02
hq1	13	13	-3266.67	.01	16	89	-3266.67	.04
mss 1	14	15	-343.35	.03	13	62	-343.35	.09
mss 2	16	16	-3.16	.04	14	73	-3.16	.14
mss 3	11	180	-5.35	.21	11	43	-5.35	.05
oz3	2	2	0	.01	fail			
qvi	11	12	3.07e-9	.02	14	121	1e-18	.06
ralphmod	fail				fail			
tollmpec	7	7	-20.83	11.8	fail			

Table 1: Implicit Algorithms

	PIPA-N			PIPA-M			PIPA-B		
model	k	secs	θ	k	secs	θ	k	secs	θ
bard3	MCP			1000	$65^{\dagger}$	-12.63	fail		
desilva	218	5.9	-1	135	4.9	-.999999	145	4.6	-.999999
epowplan	MCP			stepsize			stepsize		
fjq1 a	MCP			1000	$62^{\dagger}$	3.274	1000	$62^{\dagger}$	3.2320
fjq1 b	MCP			1000	$60^{\dagger}$	3.262	1000	$63^{\dagger}$	3.267
fjq2 a	MCP			1000	$62^{\dagger}$	3.450	59	1.6	3.449
fjq2 b	MCP			1000	$51^{\dagger}$	3.450	18	.38	3.449
fjq3 a	MCP			1000	$54^{\dagger}$	4.594	35	.90	4.604
fjq3 b	MCP			1000	$53^{\dagger}$	4.594	26	.61	4.604
fjq4 a	MCP			15	.37	6.593	24	.56	6.593
fjq4 b	MCP			18	.45	6.593	52	1.5	6.593
gauvin	15	.18	20	18	.37	20	18	.34	20
hq1	12	.12	-3266.67	18	.3	-3266.67	18	.26	-3266.67
mss 1	MCP			24	1.2	-343.35	29	1.3	-343.35
mss 2	MCP			52	2.8	-3.16	52	2.8	-3.16
mss 3	MCP			21	1.0	-5.35	21	1.5	-5.35
oz3	MCP			1000	$65^{\dagger}$	57.50	1000	$104^{\dagger}$	5.172
qvi	MCP			1000	18	0	1000	14	0
ralphmod	107	138	-683.03	224	1782	-683.03	1000	$4488^{\dagger}$	-563.4
tollmpec	MCP			memory use			memory use		

Table 2: MATLAB PIPA codes

PIPA algorithms failed to converge with a † in the seconds column. There are a number of case where this occurred. For model oz3, the PIPA-B code was getting close to the optimal θ value of 4.999375 reported in [11], but the same cannot be said of PIPA-M. Both of these algorithms had difficulty with the bilevel programs in fjq1 - fjq4, hitting the iteration limit before reaching a solution. PIPA-M seemed to get quite close to the optimal value of -12.67871 from [11], but this may or may not have been at a feasible point, while PIPA-B failed completely on this model. The electric power planning model epowplan led to immediate failure in the linesearch phase of PIPA, while the memory required to solve the tollmpec model via PIPA was too much for our computing platform. The ralphmod model is a linear complementarity constrained MPEC with a quadratic objective, so the PIPA-N and PIPA-M codes use exact Hessians.

It is clear that more algorithmic development work for MPEC problems is required. New algorithms, such as those outlined in [11, 15, 19, 27] hold great potential to take advantage of the tools we have developed here.

7 Conclusions

This paper has described some enabling technology for modeling and solving MPEC problems. The cornerstone for all of these is the addition of the MPEC model type to the GAMS language. These models are constructed and passed to the solvers by MPECIO, a new library of routines to ease problem data access. To exhibit the use of these routines we have linked various MPEC solvers directly to GAMS.

As another example of the utility of the interface library, we have provided simple MATLAB functions to access the problem data from within MATLAB, thus allowing the prototyping of algorithms within this environment.

It is our hope that the interface we have created between the MPEC modeler and MPEC algorithm developer will spur development within both of these groups by fueling a powerful synergy between them.

We gratefully acknowledge those who have assisted us in this work. H. Jiang and D. Ralph provided the MATLAB codes used in our tests and were generous with comments on the MATLAB interface. A. Kuntsevich and F. Kappel have made SolvOpt publicly available. J. Outrata provided guidance on the use of bundle methods for these problems.

References

[1] A. Bachem, M. Grötchel, and B. Korte, editors. *Mathematical Programming: The State of the Art, Bonn 1982*, Berlin, 1983. Springer Verlag.

[2] J. F. Bard. Convex two-level optimization. *Mathematical Programming*, 40:15–27, 1988.

[3] J. Bisschop and R. Entriken. *AIMMS – The Modeling System*. Paragon Decision Technology, Haarlem, The Netherlands, 1993.

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

[5] A. H. DeSilva. *Sensitivity Formulas for Nonlinear Factorable Programming and their Application to the Solution of an Implicitly Defined Optimization Model of US Crude Oil Production*. PhD thesis, George Washington University, Washington, D.C., 1978.

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

[7] 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.

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

[9] S. P. Dirkse and M. C. Ferris. Traffic modeling and variational inequalities using GAMS. In Ph. L. Toint, M. Labbe, K. Tanczos, and G. Laporte, editors, *Operations Research and Decision Aid Methodologies in Traffic and Transportation Management*, NATO ASI Series F. Springer-Verlag, 1997.

[10] 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.

[11] F. Facchinei, H. Jiang, and L. Qi. A smoothing method for mathematical programs with equilibrium constraints. Technical report, Universitá di Roma “La Sapienza”, Roma, Italy, 1996.

[12] 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.

[13] M. C. Ferris and T. F. Rutherford. Accessing realistic complementarity problems within Matlab. In G. Di Pillo and F. Giannessi, editors, *Nonlinear Optimization and Applications*, pages 141–153. Plenum Press, New York, 1996.

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

[15] M. Fukushima, Z.-Q. Luo, and J. S. Pang. A globally convergent sequential quadratic programming algorithm for mathematical programs with linear complementarity constraints. *Computational Optimization and Applications, forthcoming*, 1997.

[16] P. T. Harker. Generalized Nash games and quasi-variational inequalities. *European Journal of Operations Research*, 54:81–94, 1987.

[17] P. T. Harker and J. S. Pang. Finite–dimensional variational inequality and nonlinear complementarity problems: A survey of theory, algorithms and applications. *Mathematical Programming*, 48:161–220, 1990.

[18] J. M. Henderson and R. E. Quandt. *Microeconomic Theory.* McGraw–Hill, New York, 3rd edition, 1980.

[19] H. Jiang and D. Ralph. QPECgen, a matlab generator for mathematical programs with quadratic objectives and affine variational inequality constraints. Technical report, The University of Melbourne, Department of Mathematics and Statistics, Parkville, Victoria, Australia, 1997.

[20] M. Kočvara and J. V. Outrata. On optimization of systems governed by implicit complementarity problems. *Numerical Fucntional Analysis and Optimization*, 15:869–887, 1994.

[21] M. Kočvara and J. V. Outrata. On the solution of optimum design problems with variational inequalities. In *Recent Advances in Nonsmooth Optimization*, pages 172–192. World Scientific Publishers, Singapore, 1995.

[22] A. Kuntsevich and F. Kappel. SolvOpt: The solver for local nonlinear optimization problems. Institute for Mathematics, Karl-Franzens University of Graz, 1997.

[23] Z.-Q. Luo, J. S. Pang, and D. Ralph. *Mathematical Programs with Equilibrium Constraints.* Cambridge University Press, 1996.

[24] Frederic H. Murphy, Hanif D. Sherali, and Allen L. Soyster. A mathematical programming approach for determining oligopolistic market equilibrium. *Mathematical Programming*, 24:92–106, 1982.

[25] J. V. Outrata. On optimization problems with variational inequality constraints. *SIAM Journal on Optimization*, 4:340–357, 1994.

[26] J. V. Outrata and J. Zowe. A numerical approach to optimization problems with variational inequality constraints. *Mathematical Programming*, 68:105–130, 1995.

[27] D. Ralph. A piecewise sequential quadratic programming method for mathematical programs with linear complementarity constraints. In *Proceedings of the Seventh Conference on Computational Techniques and Applications (CTAC95)*, 1996.

[28] R. Raman. *Integration of logic and heuristic knowledge in discrete optimization techniques for process systems.* PhD thesis, Department of Chemical Engineering, Carnegie Mellon University, Pittsburgh, Pennsylvania, 1993.

[29] S. M. Robinson. Strongly regular generalized equations. *Mathematics of Operations Research*, 5:43–62, 1980.

[30] S. M. Robinson. Generalized equations. In Bachem et al. [1], pages 346–367.

[31] H. Schramm and J. Zowe. A version of the bundle idea for minimizing a nonsmooth function: Conceptual idea, convergence analysis, numerical results. *SIAM Journal on Optimization*, 2:121–152, 1992.

[32] N. Z. Shor. *Minimization Methods for Nondifferentiable Functions.* Springer-Verlag, Berlin, 1985.

[33] H. Van Stackelberg. *The Theory of Market Economy.* Oxford University Press, 1952.