The GAMS Callable Program Library for Variational and Complementarity Solvers

Steven P. Dirkse * Michael C. Ferris * Paul V. Preckel †
Thomas Rutherford ‡

July 19, 1994

Abstract

The GAMS modeling language has recently been extended to enable the formulation of Mixed Complementarity Problems (MCP). The GAMS Callable Program Library (CPLIB) is a set of Fortran subroutines developed as an extension for the GAMS I/O library and designed to provide a simple and convenient interface to the MCP defined by a GAMS model. This paper provides technical documentation for CPLIB for use by those who are developing or have developed algorithms for MCP, in order that their solvers may be made available as GAMS subsystems.

1 Introduction

This paper provides technical documentation for the GAMS Callable Program Library (CPLIB) which has been developed as an extension of the GAMS I/O library (Kalvelagen 1992). CPLIB is a set of routines which simplify the implementation of GAMS solution systems for linear and nonlinear equations, complementarity problems and variational inequalities. The GAMS language has only recently been extended to accomodate this class of problems. Brooke, Kendrick & Meeraus (1988) provide an introduction to the GAMS language for optimization. The MCP extension is described by Rutherford (1994). The present paper is written for algorithm developers rather than modelers. The library detailed here has been constructed to encourage researchers who have algorithms for the MCP problem types to make them available to others as part of the GAMS language.

---

*Computer Sciences Department, University of Wisconsin, Madison, WI 53706. Research supported by the National Science Foundation grant CCR–9157632.

†Agricultural Economics Department, Purdue University, West Lafayette, IN 47907.

‡Department of Economics, University of Colorado, Boulder, CO 80309. Research supported by the Canadian Natural Science and Engineering Research Council grant T306A1.

A large class of problems including systems of equations, complementarity problems, constrained nonlinear optimization problems and finite dimensional variational inequalities can be characterized as special cases of the Mixed Complementarity Problem (MCP):

Definition 1 (MCP) *Given a function $F : \mathbb{R}^n \to \mathbb{R}^n$ and bounds $\ell, u \in \overline{\mathbb{R}}^n$,*

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

$$\text{s. t.} \quad \begin{aligned} F(z) &= w - v && \text{(1a)} \\ \ell \leq z &\leq u && \text{(1b)} \\ (z - \ell)^\top w &= 0 && \text{(1c)} \\ (u - z)^\top v &= 0 && \text{(1d)} \end{aligned}$$

Not all solvers may be capable of solving all problems in this class. Certain algorithms will, for example, only apply to systems of equations ($\ell = -\infty, u = +\infty$), while others may apply only to the nonlinear complementarity problem ($\ell = 0, u = +\infty$). It will be the modeler's job to choose an appropriate solution algorithm for a particular application.

A number of economists currently use GAMS for economic equilibrium analysis. These models typically use a nonlinear optimizer such as MINOS (Murtagh & Saunders 1983) or CONOPT (Drud 1985) to solve square systems of smooth nonlinear equations. We believe that there will be significant interest among these practitioners in solvers for nonsmooth systems of equations, provided the algorithms are sufficiently robust and efficient. It is likely that the complementarity facility within GAMS will find applications in other disciplines as well.

While it is apparent that modelers will benefit from this extension, we feel that the algorithmic community can benefit as well. This is because the GAMS modeling system provides an excellent framework through which researchers interested in exploring the properties of new algorithms can interact with people who are building "real" models. Clearly, the availability of many easily accessed, large scale test problems will provide better insight into the relative strengths of new algorithms.

We have developed CPLIB to simplify the process of connecting a solver to GAMS. Until now the coding required to link a solver to GAMS may have deterred those with non-commercial solvers from making them available for testing on true applications. The CPLIB interface significantly reduces the set-up cost. We believe that connecting a solver to GAMS using CPLIB represents less work than programming the input-output and function evaluation routines for a single large-scale test problem.

The GAMS I/O library is available in both C and Fortran. CPLIB is only available in Fortran, although its use with solvers written in C is relatively easy under operating systems where compatible C and Fortran compilers are available. CPLIB is designed to be used together with the linear and nonlinear components of the GAMS I/O library. Using these tools, it is possible to produce solution systems which are truly portable because all system-specific functions can be performed by the (platform-specific) GAMS library routines. The GAMS routines are currently available on many machines, including PCs, many workstations, mainframes and supercomputers.

From the perspective of an MCP algorithm, CPLIB is simply a function and derivative evaluation facility. Before connecting a solver to GAMS, it should be debugged to a reasonable extent using small examples, but it need not be "bullet-proof". The GAMS interface can be used from an early stage in algorithm development to generate simple test problems and to help with debugging.

A schematic representation of the relationship between the developer-written routines, CPLIB, and the GAMS I/O library is presented in Figure 1. The developer-written routines are indicated by dashed boxes. Referring to the top of this diagram, we begin with the

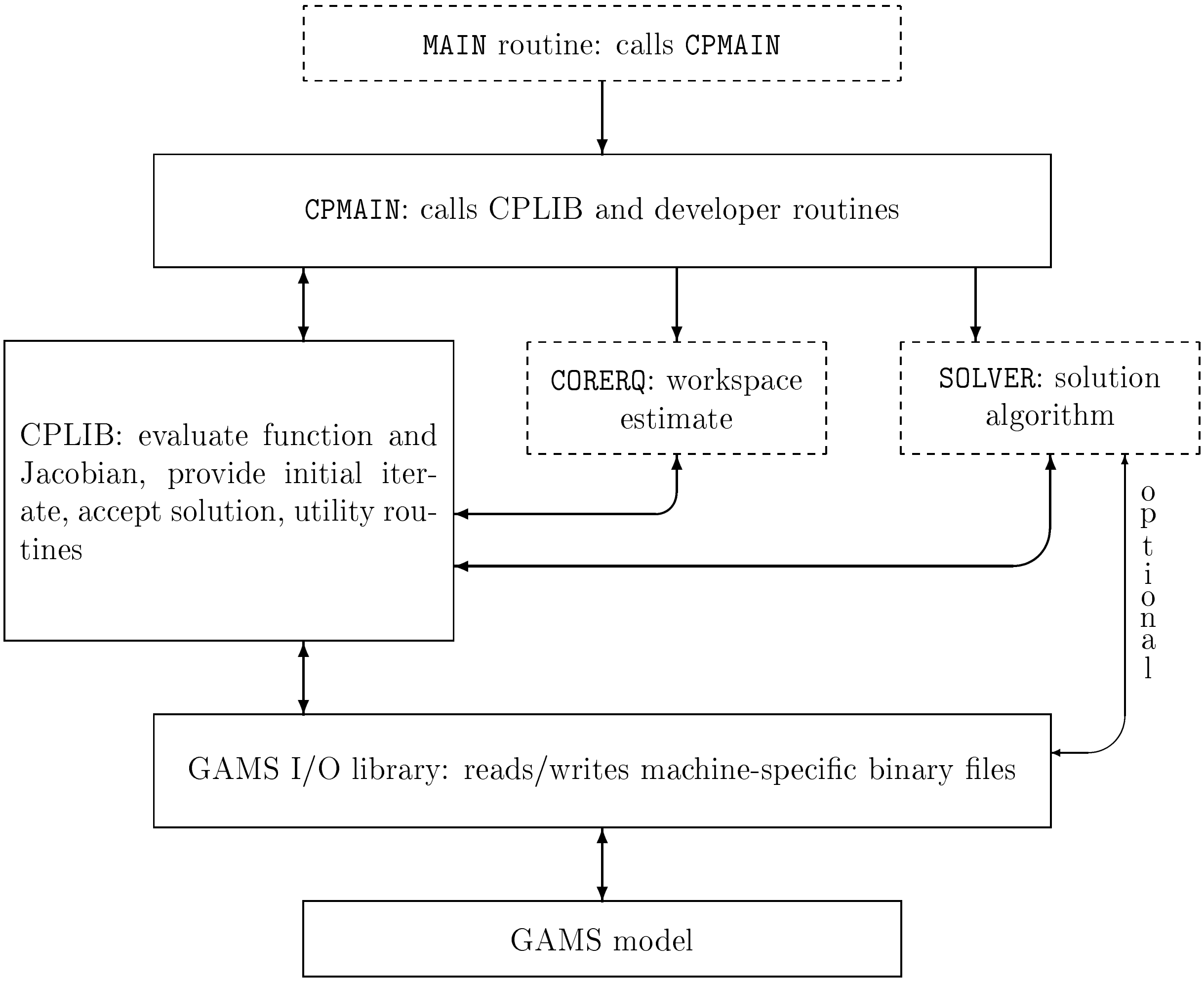

Figure 1: Interrelationship of Developer Code and CPLIB

developer's main program, the sole function of which is to call the `CPMAIN` subroutine. The `CPMAIN` routine, which has no arguments, is the top-level CPLIB routine, from which calls are made to routines from the GAMS I/O library, CPLIB, and the developer-written routines. `CPMAIN` first calls I/O library routines to read information from the GAMS compiler summarizing the model structure. After forming the data structures which associate variables with equations, it calls the developer-written `CORERQ` (core-requirments) subroutine. The purpose of `CORERQ` is to communicate the following to the CPLIB system:

- an indication of the solver's capabilities,
- a request for workspace (i.e., dynamically allocated memory) if needed, and
- optionally, special values to install, such as those which are to be used to represent plus and minus infinity.

In order to simplify development and maintenance of CPLIB, parameter values are generally not passed as subroutine arguments. Instead, communication between developer code and the library takes place through calls to the "scalar interrogation" and "scalar return" routines `CPGET*` and `CPPUT*`. (Details are provided below.) We have avoided (where possible) subroutine argument lists because we feel that this will make it easier to provide upward compatability in future revisions of the library. Furthermore, this approach simplifies cross-language linking.

`CPMAIN` allocates workspace and then calls the developer-written `SOLVER` subroutine. This routine does the bulk of the problem solving. In the process of solving the problem, the `SOLVER` routine will typically call CPLIB utility routines to

- query the GAMS I/O library regarding the values of machine dependent parameters;
- obtain values of problem-dependent parameters such as variable initial values and bounds;
- evaluate the nonlinear function and its Jacobian at a given point; and
- return the computed solution, or an indication of why the solution process has terminated without finding a solution.

For the most part, the CPLIB system provides all necessary utilities for an MCP algorithm. The developer may, however, call the GAMS I/O library directly to perform certain input-output functions or string manipulations. At the end of the solution process, control returns to the `CPMAIN` routine where additional housekeeping activities are performed to close down the GAMS I/O library.

A great advantage of CPLIB (with the GAMS I/O library) is that all machine or installation dependent code may be removed from a solution system. In writing a solver, a developer should be careful to employ library routines to perform platform-specific tasks in order that the algorithmic code remains fully portable. Using the interface routines in this way, the algorithm developer may concentrate on the mathematics of problem solution rather than the quirks of porting Fortran or C to particular machines and operating systems.

Reserved names in the CPLIB and GAMS I/O library modules begin with the letters CP and GF. It is good policy to avoid subroutine and common block names which begin with either of these pairs of letters.

In Section 2, we introduce the developer-written subroutines and their functions. Section 3 provides a description of the CPLIB subroutines and their use, Section 4 a description of how a solver communicates with a user, and Section 5 a discussion of how a solver is installed as a GAMS subsystem. Appendix A discusses the use of solvers written in the C language. Appendix B contains Fortran code for a Newton algorithm from Press, Flannery, Teukolsky & Vetterling (1988). Appendix C contains a sample solver coded in C.

2 Developer-Written Subroutines

2.1 Workspace estimation: `CORERQ`

The first of the developer-written subroutines is `CORERQ`. This routine is used to communicate to CPLIB the capabilities of the solver (to detect incompatible problem/solver combinations). It is also the place to compute an estimate of the amount of memory required to process the problem. This routine may also install special values for plus and minus infinity.

The `CORERQ` subroutine has no arguments. The communication between CPLIB and subroutine `CORERQ` is accomplished through the parameter interface routines `CPGETI`, `CPPUTI` and `CPPUTD` documented below. The integer parameters to be set are 'ISTYPE' (solver type indicator) and 'NWUCOR' (the memory required, expressed in number of "words" - double-precision real equivalents). In estimating 'NWUCOR', `CPGETI` is typically used to determine problem size and density. (See Section 3.5 for details.) The double precision values which may be returned correspond to the strings 'PLINFY' and 'MNINFY' and are subsequently used to represent plus and minus infinity. (If these values are not specified, they assume default values equal to $+/-10^{20}$.)

2.2 Solution control: `SOLVER`

The second of the developer-written subroutines is `SOLVER`. This is the developer's top-level routine for problem solution. If memory is dynamically allocated, it is typically in this routine that the requested workspace is "partitioned" and passed along as separate arrays. The structure of this subroutine is:

```
SUBROUTINE SOLVER(WORK, NWUCOR)
INTEGER NWUCOR
DOUBLE PRECISION WORK(NWUCOR)
```

`NWUCOR`	input	number of words (double-precision reals) of memory reserved for the solver in the previous call to `CORERQ`
`WORK`	input	workspace array of `NWUCOR` words

In the process of solving a problem, this routine and its subsidiaries will call routines from CPLIB and (possibly) routines from the GAMS I/O library. The CPLIB routines it will need to call are described in the following section.

3 CPLIB Subroutines Called by Solvers

The CPLIB subroutines include one routine which returns initial values and bounds, two routines to evaluate functions and derivatives, one routine for reporting the solution vector, one routine which triggers a system interrupt, and six routines for passing scalar values between the solver and the library. This section will introduce these routines.

3.1 Initial levels and bounds: `CPBNDS`

GAMS/CPLIB passes three values for each variable to the solver. These are the initial level, the lower bound and the upper bound. The solver obtains these values from CPLIB using the subroutine `CPBNDS`, whose structure is:

```
SUBROUTINE CPBNDS(Z, BL, BU, N)
INTEGER N
DOUBLE PRECISION Z(N), BL(N), BU(N)
```

`Z`	output	initial values of the problem variables
`BL`	output	lower bounds
`BU`	output	upper bounds
`N`	input	problem dimension.

The representations of plus and minus infinity used in `BU` and `BL` should be obtained via calls to `CPGETD` with the strings 'PLINFY' and 'MNINFY'.

3.2 Solution value return: `CPSOLN`

Before the `SOLVER` routine returns control to `CPMAIN`, the level values for the problem variables may be returned through the CPLIB routine `CPSOLN`. The structure of this routine is:

```
SUBROUTINE CPSOLN(Z,N)
INTEGER N
DOUBLE PRECISION Z(N)
```

`Z`	input	solution estimate at solver termination
`N`	input	problem dimension.

This routine is optional. If it is not called, CPLIB will return level values to GAMS which are the "best" values encountered in the solution process, based on $\|\epsilon\|_\infty = \max_i |\epsilon_i(z)|$ (the box norm of ϵ). The vector ϵ is defined by

$$\epsilon_i(z) := \max\ \{F_i(z)_+ \min[1, z_i - \ell_i],\ (-F_i(z))_+ \min[1, u_i - z_i],\ (l_i - z_i)_+,\ (z_i - u_i)_+\},$$

where $x_+ := \max\ (0, x)$. An algorithm is not required to use this norm for determining convergence. The CPLIB calculation is provided only in order that a "good" solution can be returned in the event of an abnormal exit.

3.3 Abnormal interrupt: CPPUNT

In the normal program sequence, a solver processes the problem, returns the solution values and status indicators and then returns control to CPMAIN which closes down the GAMS I/O library and returns control to the developer's main program. In certain circumstances, particularly with solvers which are under development, it is helpful to be able to abort directly from a lower level routine in the solver. In these cases, the CPPUNT routine should be used rather than a Fortran STOP statement. CPPUNT will assure that the GAMS I/O library is properly closed so that the GAMS program exits "gracefully". CPPUNT has no arguments.

The CPPUNT routine will set MODSTA = 12 (error unknown) and it will trigger a "SYSOUT" (i.e., the entire status file will be copied onto the listing file, as though the GAMS program had specified "OPTION SYSOUT = ON;".) In addition, the current solution (either based on the CPLIB merit function or installed by the solver through a prior call to CPSOLN) will be passed to the listing file.

3.4 Function evaluation: CPFUNF, CPSPRJ

Two routines provide function evaluations. One evaluates only the vector function F, while the second evaluates both F and its Jacobian J, returning J in a sparse matrix data structure.

CPFUNF evaluates the nonlinear function F at a given point, Z. It does not return the Jacobian of F. The structure of this subroutine is

```
SUBROUTINE CPFUNF(Z,F,N)
INTEGER N
DOUBLE PRECISION Z(N), F(N)
```

Z	input	point at which to evaluate the function F
F	output	value of F evaluated at Z
N	input	problem dimension.

CPSPRJ evaluates the function F and its Jacobian J, the matrix of first partial derivatives of F with respect to its arguments. The Jacobian is returned in the well-known row index, column pointer, column length format. The subroutine structure is:

```
SUBROUTINE CPSPRJ(Z, F, J, JROW, JCOL, JLEN, N, NADIM)
INTEGER N, NADIM, JROW(NADIM), JCOL(N), JLEN(N)
DOUBLE PRECISION Z(N), F(N), J(NADIM)
```

Z	input	point at which to evaluate the function F and Jacobian J
F	output	value of F evaluated at Z
J	output	nonzero coefficients of the matrix J evaluated at Z
JROW	output	row indices of the coefficients stored in J
JCOL	output	pointers to columns starts in J
JLEN	output	lengths of the columns in J
N	input	problem dimension
NADIM	input	number of nonzero components in J.

The coefficients for the nonzero entries of the k'th column of J are stored in the vector `J`, in positions `JCOL(k)`, `JCOL(k)+1`, ..., `JCOL(k)+JLEN(k)-1`. The row indices for these coefficents are stored in the corresponding positions of `JROW`.

3.5 Scalar interrogation: CPGETD, CPGETI, CPGETL

Three routines are provided to pass scalar values from CPLIB to the solver. The routines which "get" parameter values from CPLIB are `CPGETD`, `CPGETI` and `CPGETL`. These return double precision (real), integer and logical parameters, respectively. The structure of these routines is:

```
SUBROUTINE CPGETD(NAME, DPARAM)
CHARACTER*(*) NAME
DOUBLE PRECISION DPARAM

SUBROUTINE CPGETI(NAME, IPARAM)
CHARACTER*(*) NAME
INTEGER IPARAM

SUBROUTINE CPGETL(NAME, LPARAM)
CHARACTER*(*) NAME
LOGICAL LPARAM
```

`NAME`	input	the name of the parameter to be returned
`DPARAM`	output	real parameter returned
`IPARAM`	output	integer parameter returned
`LPARAM`	output	logical parameter returned

Character string identifiers for which these subroutines produce useful values are listed in Tables 1, 2, and 3, along with definitions of the results.

3.6 Scalar return: CPPUTD, CPPUTI

Two utility routines similar in design to the `CPGET*` routines are provided to pass scalar values from the solver to CPLIB. The structure of these routines is:

```
SUBROUTINE CPPUTD(NAME, DPARAM)
CHARACTER*(*) NAME
DOUBLE PRECISION DPARAM

SUBROUTINE CPPUTI(NAME, IPARAM)
CHARACTER*(*) NAME
INTEGER IPARAM
```

Table 1: `CPGETD` Arguments

String	Result returned in double precision argument
'CLOCK'	Current elapsed time (for checking RESLIM).
'CONTOL'	Convergence tolerance (default value = 1.e-6)
'EPS'	The smallest positive number that can be added to 1.0 to obtain a result different from 1.0.
'GMINF'	GAMS bit pattern for $-\infty$ (a logical indicator not to be treated as a number).
'GPINF'	GAMS bit pattern for $+\infty$ (a logical indicator not to be treated as a number).
'HUGE'	The largest positive number representable on the machine.
'MAXEXP'	The largest positive decimal exponent representable on the machine.
'MINEXP'	The largest negative decimal exponent representable on the machine.
'MNINFY'	Value currently used for $-\infty$.
'OBJ'	Merit function associated with the most recent function evaluation.
'PLINFY'	Value currently used for $+\infty$.
'PRECIS'	The number of significant decimal digits.
'REAL1' – 'REAL5'	Five real values can be set in a user's GAMS program using option statements of the form: `option REAL3 = 0.1;` these should be used only during solver development.
'RESLIM'	The resource limit in CPU seconds.
'TINY'	The smallest positive number representable on the machine.

Table 2: `CPGETI` Arguments

String	Result returned in integer argument
'DOMERR'	The number of domain errors which have been encountered.
'DOMLIM'	Maximum number of domain errors allowed before the iterations are terminated.
'INTEGER1'- 'INTEGER5'	Five integer values can be set in a user's GAMS program using option statements of the form: `option INTEGER3 = 10;` these should be used only during solver development.
'INTW'	The number of long integers per "word" (1 word = 1 double precision real).
'IOLOG'	The I/O unit number of the log file.
'IOOPT'	The I/O unit number of the options file (cf. USEOPT from `CPGETL` to see if an options file has been provided).
'IOSTA'	The I/O unit number of the status file.
'ITERLIM'	An iteration limit set via the GAMS ITERLIM option; default = 1000.
'LLOGW'	The number of long logicals per "word".
'MAXCOL'	The maximum number of nonzeros in any column of the matrix.
'N'	The number of equations/structural variables in the condensed problem (after fixed variables are removed).
'NADIM'	The number of nonzeros in the Jacobian matrix of the condensed problem.
'NUMCOL'	The number of structural variables in the original problem.
'NUMNNZ'	The number of nonzeros in the Jacobian matrix of the original problem.
'NUMROW'	The number of rows in the original problem.
'SCREEN'	The I/O unit number of the screen.
'SHORTW'	The number of short integers per "word". 'INTW' is returned if no short integers exist. (If possible, avoid using short integers).
'SLOGW'	The number of short logicals per "word". 'LLOGW' is returned if no short logicals exist. (If possible, avoid using short logicals).

Table 3: `CPGETL` Arguments

String	Result returned in logical argument
'USEBAS'	If true, use basis information implied by the bounds and initial values.
'USEOPT'	If true, attempt to read user's options file, whose format and syntax are solver-defined.
'SYSOUT'	If true, GAMS will copy the complete status file to the listing file.

`NAME`	input	the name of the parameter to be passed
`DPARAM`	input	real parameter passed
`IPARAM`	input	integer parameter passed

Character string identifiers used as input to these routines are listed in Tables 4 and 5, along with definitions of the results.

Table 4: `CPPUTD` Arguments

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

4 Communication and Control

This section describes how a solver communicates with the user. It also indicates how a GAMS user can affect the solution algorithm, either through the use of iteration or resource limits or through the provision of a solver-specific "option" file with tolerances and switches. For most applications, the CPLIB routines provide all the necessary hooks. Some developers may choose to call utility routines from the GAMS I/O library directly. Potentially relevant utilities are listed at the end of this section.

4.1 Communicating with the User

The solver communicates with the GAMS user through the status file, the log file, and the screen. The unit numbers for these devices are accessed through calls to `CPGETI` using the identifiers 'IOSTA', 'IOLOG' and 'SCREEN'.

The status file contains two classes of information – information which is always be copied to the GAMS listing file and information which is copied to the listing file only when the

Table 5: `CPPUTI` Arguments

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

GAMS user specifies the option SYSOUT = ON. The first type of output is identified by first calling `CPPUTI` with the string 'STARTC' (the integer argument is ignored). Subsequent solver output to the status file will then appear in the GAMS listing. To stop copying to the listing file, call `CPPUTI` with the string 'STOPC' (again, the integer argument is ignored).

The log file and the screen are typically the same unit. On interactive platforms, the solver may send messages to the log file to indicate progress. This is particularly reassuring to the user when the system is slow. It is possible, however, for the user to redirect this output. (A user might do this if he is operating over a slow phone line.) Only when information (such as a copyright notice) is always to be displayed on the screen should the SCREEN unit be used.

4.2 Iteration limits

It is up to the solver to see to it that iteration limits set by the user (through the ITERLIM option) are not exceeded. By querying CPLIB for the value of 'ITERLIM' (using `CPGETI`), a solver can obtain the iterations limit. It is a GAMS convention that for algorithms with major and minor iterations, 'ITERLIM' refers to the cumulative minor iterations performed. Other iteration limits may be specified in the solver-specific options file. The solver should see that these limits are not exceeded as well.

4.3 CPU time limits

GAMS also provides a way to limit the amount of CPU resources used by a solver. This resource limit is set using the RESLIM option, and is obtained from CPLIB via a call to `CPGETD`, using the string 'RESLIM'. This limit is expressed in units (typically CPU seconds) which are consistent with the value returned by `CPGETD` through the string 'CLOCK'. The value of 'CLOCK' should be evaluated at various times during problem solution and compared with the CPU limit 'RESLIM'. The run should be terminated when 'RESLIM' is exceeded.

4.4 Options files

Some algorithms may need tuning for particular problems. Tuning parameters are typically specified in an options file, the format and contents of which are the choice of the algorithm developer. Typically, these are free-format files containing optional key words and parameter values. CPLIB routine `CPGETL` returns a logical value for 'OPTFIL' which is true if an options file exists. When a file is provided, CPLIB routine `CPGETI` returns the unit number of the options file (an integer value) when passed the string 'IOOPT'.

4.5 GAMS I/O utilities

The CPLIB package has been designed so that direct calls to the GAMS I/O library are not absolutely necessary. Developers may, however, wish to use some of the I/O utilities in order to improve portability of their code or to perform special functions. The names and short descriptions of potentially relevant utilities are presented here. Interested readers should consult the I/O library manual for calling sequences (Kalvelagen 1992).

Input library routines:

`GFOPTI`	Return options file name
`GFTIME`	Return GAMS time.
`GFWDIR`	Return GAMS working directory.
`GFSDIR`	Return GAMS system directory.
`GFCDIR`	Return GAMS scratch directory.

Resident library routines:

`GFMSG`	Write a message.
`MEMINF`	Print memory statistics.
`GFUOPN`	Open a file.
`GFFRST`	Return position of first non-blank in a string.
`GFINDX`	Return length of a string excluding trailing blanks.
`GFFRMT`	Return a string with a “good” format for printing a real number.

5 Introducing a Solver to GAMS

Once a solver has been linked to CPLIB and the GAMS I/O library, it must be introduced as a GAMS subsystem before it is used by a GAMS program. This section describes the various files which must be written and modified for this purpose. In this discussion, file names and batch command language for the DOS version of GAMS are used. Different file name extensions etc. may be used on other platforms.

5.1 Updating `gamscomp.txt`

To introduce a new solver, the `gamscomp.txt` file in the GAMS directory must be updated. In this file there are entries for each of the solvers currently connected to the GAMS system. For example, if the system is licensed for MINOS, there will be an entry such as:

```
minos5 2 0 LP RMIP NLP DNLP RMINLP
gamsmns3
```

The first line indicates the name of the solver (MINOS5 in this case). Two integers follow the solver name. They signify the data file type and the dictionary file type, respectively. The capability list for the solver follows these integers. This is the class of problems for which this solver may be employed. The line immediately following the solver capability list indicates the name of the batch control file in the GAMS system directory which invokes your solver. In this case, the batch file is named `gamsmns3.bat`.

The data file types include 0 for flat (ASCII), 1 for stream (binary with no header or footer), 2 for binary with four byte header and footer, and 3 for “special”. When introducing a new solver, use the file type which is appropriate for your compiler. File type 0 works for all compilers but it introduces a significant performance penalty for large models. On the PC, file type 2 is used for programs compiled with NDP Fortran and type 1 is used for Lahey Fortran.

The dictionary file types include 0 for none, 1 for a flat file with no quotes, and 2 for a flat file with quoted strings. When using CPLIB, the library file type must be of type 2. Unlike other optimization models, the dictionary file is always needed for MCP models; it is read using Fortran free-format (list-directed) input.

To introduce a new solver, give it a name and then make the appropriate two-line entry in `gamscomp.txt`. For example:

```
newton 0 2 mcp
runnwt
```

If you want this to become the default MCP solver, you enter the solver name after MCP at the bottom of the `gamscomp.txt` file.

5.2 The solver batch command file

After introducing a solver in `gamscomp.txt`, the batch command file which is named in the solver definition must be written. A batch file is used in order that the executable image can be stored in a directory other than the GAMS system directory. For example, the NEWTON solver invocation file (`runnwt.bat`) is:

```
@echo off
C:\gams\mnewt\newton %4
gamscmex %3
gamsnext %3
```

Here (in the DOS version), %3 and %4 are the third and fourth arguments in the batch file invocation. %4 is the control file name which is obtained from the command line by the I/O library routines. The control file is read by the I/O library, providing model dimensions, control parameters and file locations.

5.3 Shortening the debug cycle

After linking a solver to GAMS it can be expected that one or more test problems will prove to be particularly recalcitrant. When using GAMS test problems to debug a solver, it is not necessary to recompile the GAMS input file each time the solver is invoked. Instead, temporarily modify the GAMS batch files in order to skip the scratch file clean-up step. In the PC version of GAMS, this simply involves omitting the following line from `gamsexit.bat`:

```
if exist %1*.scr erase %1*.scr
```

(This can be done by simply placing a comment flag (":") in the first column of the record.)

Having done this, run GAMS on the input file once to generate the data and instructions. These will be saved in a set of intermediate files with the `.scr` extension on the GAMS scratch directory. (See parameter SCRDIR in `gamsparm.txt` for the scratch directory location.) As long as you do not delete these files, you can then invoke the solver directly without running GAMS. For example, typing `newton D:gamscntr.scr` at the DOS prompt (when SCRDIR=`D:\`) will run the NEWTON solver from a saved model. `gamscntr.scr` is the control file in which the location of all other files is passed to the I/O library.

Acknowledgements

The authors are indebted to Alex Meeraus and Erwin Kalvelagen for their suggestions on the design of this library. We assume responsibility for errors remaining in the code and documentation.

References

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

Drud, A. (1985), 'CONOPT: A GRG code for large sparse dynamic nonlinear optimization problems', *Mathematical Programming* **31**, 153–191.

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

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

Press, W. H., Flannery, B. P., Teukolsky, S. A. & Vetterling, W. T. (1988), *Numerical Recipes : the Art of Scientific Computing*, Cambridge University Press.

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

A The C Interface

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

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

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

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

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

```
/* defines, etc for the C solver interface to CPLIB
   by
   Steven Dirkse
   Computer Sciences Department
   UW-Madison
   */
/* we want INT to correspond in size to Fortran's integer, etc */
#define BOOLEAN int
#define CHAR char
#define DREAL double
#define INT int

#define CPBUF_LEN 8
#define OPTFLN_LEN 80

/* defines for modsta, solsta return values */
#define MODEL_SOLVED        1
#define MODEL_NOT_SOLVED    7
#define MODEL_ERROR         13
#define SOLU_NORMAL         1
#define SOLU_ITERATION      2
#define SOLU_RESLIM         3
#define SOLU_KILLED         4
#define SOLU_EVAL_LIMIT     5
#define SOLU_ERROR          11

INT c_cpgeti (CHAR *string);
DREAL c_cpgetd (CHAR *string);
BOOLEAN c_cpgetl (CHAR *string);
void c_cpputi (CHAR *string, INT ivalue);
void c_cpputd (CHAR *string, DREAL dvalue);
void c_cpputl (CHAR *string, BOOLEAN lvalue);
void c_print_msg (INT unit_no, CHAR *msg);
CHAR *get_row_name (INT index, CHAR *s, INT len);
CHAR *get_variable_name (INT index, CHAR *s, INT len);

/*   compile with -DPOSTUC  when trailing underscores are needed
     This is what user-written C routines should look like   */
#ifdef POSTUC
```

```
void corerq_ (void);
void solver_ (DREAL *z, INT *nwucor);
#else
void corerq (void);
void solver (DREAL *z, INT *nwucor);
#endif

/* headers for Fortran CPLIB routines called direct from C solver */
#ifdef POSTUC
void cpbnds_ (DREAL initial_point_z[], DREAL lower_bound[],
              DREAL upper_bound[], INT *n);
void cpsoln_ (DREAL solution_point_z[], INT *n);
void cppunt_ (void);
void cpfunf_ (DREAL z[], DREAL f[], INT *n);
void cpsprj_ (DREAL z[], DREAL f[], DREAL J[], INT rowindex[],
              INT colptr[], INT collen[], INT *n, INT *nnz);
#else
void cpbnds (DREAL initial_point_z[], DREAL lower_bound[],
             DREAL upper_bound[], INT *n);
void cpsoln (DREAL solution_point_z[], INT *n);
void cppunt (void);
void cpfunf (DREAL z[], DREAL f[], INT *n);
void cpsprj (DREAL z[], DREAL f[], DREAL J[], INT rowindex[],
             INT colptr[], INT collen[], INT *n, INT *nnz);
#endif
```

B Sample Solver Coded in Fortran

```
C       -------------------------------------------------------------------

        PROGRAM NEWTON
C
C       A SIMPLE NEWTON ALGORITHM FOR NONLINEAR
C       EQUATIONS USING THE GAMS CALLABLE PROGRAM LIBRARY
C
C       THIS FILE CONTAINS AN ALGORITHM FROM PRESS,
C       FLANNERY, TEUKOLSKY AND VETTERLING: "NUMERICAL RECIPIES";
C       CAMBRIDGE UNIVERITY PRESS (1986).
C
C       THOMAS RUTHERFORD
C       DEPARTMENT OF ECONOMICS
C       UNIVERSITY OF WESTERN ONTARIO
C
C
C       THE MAIN PROGRAM SIMPLY CALLS THE CPLIB MAIN PROGRAM:
C
        CALL CPMAIN
        END

C       -------------------------------------------------------------------

        SUBROUTINE CORERQ
C
C       CORE REQUIREMENTS ROUTINE
C
        IMPLICIT NONE
C
C       ISTYPE = SOLVER TYPE INDICATOR
C       N      = DIMENSION OF THE NONLINEAR SYSTEM
C       NADIM  = # OF NONZEROES IN JACOBIAN
C       INTW   = NUMBER OF INTEGERS PER "WORD" (DOUBLE PRECISION REAL)
C       NWUCOR = CORE REQUIRMENT
C
        INTEGER ISTYPE, N, NADIM, INTW, NWUCOR
C
C       CONTROL PARAMETERS FOR THE NEWTON ALGORITHM:
C
C       NTRIAL  = NUMBER OF NEWTON STEPS
C       TOLX    = X ITERATE TOLERANCE (NOT USED HERE)
C       TOLF    = FUNCTION CONVERGENCE TOLERANCE (CONTOL)
C
        INTEGER NTRIAL
        DOUBLE PRECISION TOLX, TOLF
```

```
      COMMON /NWTCOM/ TOLX, TOLF, NTRIAL
C
C     INDICATE THAT WE CAN ONLY SOLVE SMOOTH NONLINEAR EQUATIONS
C
      ISTYPE = 1
      CALL CPPUTI('ISTYPE',ISTYPE)
C
C     READ DIMENSION:
C
      CALL CPGETI('N',N)
      CALL CPGETI('NADIM',NADIM)
C
C     READ COUNT OF INTEGERS PER DOUBLE PRECISION ON PRESENT PLATFORM:
C
      CALL CPGETI('INTW',INTW)
C
C     DETERMINE WORKSPACE REQUIREMENT:
C
C     2 N-DIMENSION DOUBLE PRECISION VECTORS
C     1 NADIM-DIMENSIONAL DOUBLE PRECISION VECTOR
C     1 NADIM-DIMENSIONAL INTEGER VECTOR
C     2 N-DIMENSIONAL INTEGER VECTORS
C     (ADD ONE WORD IN CASE N/INTW IS NOT INTEGRAL)
C
      NWUCOR = 2 * N + NADIM + NADIM/INTW + 1 + 2 * (N/INTW + 1)
      CALL CPPUTI('NWUCOR',NWUCOR)
C
C     QUERY FOR THE CONVERGENCE TOLERANCES AND ITERATION LIMIT:
C
      TOLX   = 0.0
      CALL CPGETD('CONTOL',TOLF)
      CALL CPGETI('ITERLIM',NTRIAL)
C
      RETURN
      END

C     ------------------------------------------------------------------

      SUBROUTINE SOLVER(Z, NWUCOR)
      IMPLICIT NONE
      INTEGER NWUCOR
      DOUBLE PRECISION Z(NWUCOR)
C
      INTEGER NTRIAL
      DOUBLE PRECISION TOLX, TOLF
      COMMON /NWTCOM/ TOLX, TOLF, NTRIAL
      INTEGER IOLOG, IOSTA
```

```
C
C       THE FOLLOWING ARE POINTERS TO SUBVECTORS OF Z:
C
        INTEGER LX, LBETA, LGRAD, LJROW, LJCOL, LJLEN
        INTEGER N, NADIM, INTW
C
C       ANNOUNCE THE PROGRAM
C
        CALL CPGETI('IOSTA',IOSTA)
        CALL CPGETI('IOLOG',IOLOG)
        CALL CPGETI('STARTC',IOSTA)
        WRITE(IOLOG,110)
        WRITE(IOSTA,110)
110     FORMAT(//,' Newton-Raphson Algorithm MNEWT'/
     *    ' ==============================='//
     *  ' from Numerical Recipes (The Art of Scientific Computing)'/
     *  ' by Press, Flannery, Teukolsky, Vetterling'/
     *  ' Cambridge University Press'//)
        CALL CPGETI('STOPC',IOSTA)
C
C       PARTITION THE WORKSPACE HERE (COMPUTE POINTERS TO THE FIRST
C       ELEMENT OF EACH OF THE ARRAYS WHICH WILL BE STORED IN Z)
C
        CALL CPGETI('N',N)
        CALL CPGETI('NADIM',NADIM)
        CALL CPGETI('INTW',INTW)
        LX    = 1
        LBETA = LX    + N
        LGRAD = LBETA + N
        LJROW = LGRAD + NADIM
        LJCOL = LJROW + (NADIM/INTW)+1
        LJLEN = LJCOL + (N/INTW)+1
C
C       LOAD THE INITIAL VALUE (THE SECOND AND THIRD ARGUMENTS
C       RETURN THE BOUNDS WHICH WILL NOT BE USED HERE, SO WE PASS
C       BETA AND GRAD AS PLACE-HOLDERS):
C
C       WE "PARTITION" THE WORKSPACE ARRAY Z BY PASSING LOCATIONS
C       ACROSS THE CALL.  (NOTICE THE DECLARATIONS FOR THE FIRST THREE
C       ARGUMENTS IN CPBNDS.)
C
        CALL CPBNDS(Z(LX), Z(LBETA), Z(LGRAD), N)
C
C       INVOKE THE ALGORITHM, ONCE AGAIN PARTITIONING THE WORKSPACE:
C
        CALL MNEWT(Z(LX), N, NADIM, Z(LBETA), Z(LGRAD), Z(LJROW),
     $            Z(LJCOL), Z(LJLEN))
```

```
C
        RETURN
        END


      SUBROUTINE MNEWT(X, N, NADIM, BETA, GRAD, JROW, JCOL, JLEN)
      IMPLICIT DOUBLE PRECISION (A-H,O-Z)
      DIMENSION X(N), ALPHA(N,N), BETA(N), INDX(N), VV(N)
      DOUBLE PRECISION OBJ
      INTEGER IOLOG
C
C       THIS CODE IS MORE OR LESS STRAIGHT FROM PRESS ET AL.
C
        INTEGER NTRIAL
        DOUBLE PRECISION TOLX, TOLF
        COMMON /NWTCOM/ TOLX, TOLF, NTRIAL
C
C       GENERATE SOME OUTPUT TO THE LOG FILE:
C
        CALL CPGETI('IOLOG',IOLOG)
C
        WRITE(IOLOG,'(A)')
     *  ' ITER         OBJ        ERRF        ERRX'

        ERRX = 0.

      DO 13  K=1,NTRIAL
        CALL CPSPRJ(X,BETA,GRAD,JROW,JCOL,JLEN,N,NADIM)
        ERRF=0.
        DO 11 I=1,N
C
C       CPLIB RETURNS +F, BUT BETA MUST EQUAL -F, SO WE
C       REVERSE THE SIGN HERE:
C
          BETA(I) = -BETA(I)
          ERRF=ERRF+DABS(BETA(I))
11      CONTINUE
        CALL CPGETD('OBJ',OBJ)
        WRITE(IOLOG,100) K,OBJ,ERRF,ERRX
        IF(ERRF.LE.TOLF)RETURN
C
C       call to factorization routine to solve J x = beta,
C       overwriting beta with x, is omitted
C
        ERRX=0.
        DO 12 I=1,N
          ERRX=ERRX+DABS(BETA(I))
```

```
          X(I)=X(I)+BETA(I)
12      CONTINUE
        IF(ERRX.LE.TOLX)RETURN
13    CONTINUE
      RETURN
100   FORMAT(1H ,I4,1P3E12.2)
      END
```

C Sample Solver Coded in C

```
/* projgrad.c ********************************************
   Steven Dirkse
   Computer Science Department, UW-Madison
   Source for a (simple) projected-gradient solver for MCP
   *******************************************************/

#include <stdio.h>
#include <malloc.h>
#include <math.h>
#include "c_cplib.h"

#define MEMALLOC(type,num) ((type *) mymalloc(sizeof(type)*(num)))
#define PROJECT(l,z,u) ( (z) < (l) ? (l) : ((z) > (u) ? (u) : (z)) )

void projected_gradient (INT n);
DREAL nonsmooth_norm (INT n, DREAL z[], DREAL f[], DREAL lb[], DREAL ub[]);
void *mymalloc(long Len);
/* code for nonsmooth_norm and mymalloc not included */

INT iterlimit, iosta, iolog, screen;
DREAL *z, *lower, *upper, *f;
CHAR msgbuf[256];

#ifdef POSTUC
void corerq_ (void)
#else
void corerq (void)
#endif
{
  c_cpputi ("nwucor", 1);       /* ask for one "word", it won't be used */
  c_cpputi ("istype", 2);       /* istype == 2 means we solve general MCP's */
  return;
}

#ifdef POSTUC
void solver_ (DREAL *work, INT *nwucor)
#else
void solver (DREAL *work, INT *nwucor)
#endif
{
  INT n;

  /* get n, unit numbers, iterations limit */
  n = c_cpgeti ("n");
  iosta = c_cpgeti ("iosta");
```

```
  iolog = c_cpgeti ("iolog");
  iterlimit = c_cpgeti ("iterlim");

  c_cpputi ("startc", 0);
  sprintf (msgbuf, "Sample solver programmed by Steve Dirkse");
  c_print_msg (screen, msgbuf);
  c_print_msg (iolog, msgbuf);
  c_print_msg (iosta, msgbuf);
  c_cpputi ("stopc", 0);

  projected_gradient (n);
  return;
}

/* Not a complete implementation, just a model */
void projected_gradient (INT n)
{
  int iteration = 0,
  i;
  CHAR *s;
  DREAL metric, obj,
  stepsize = 0.5;

  lower = MEMALLOC (DREAL,n);
  upper = MEMALLOC (DREAL,n);
  z = MEMALLOC (DREAL,n);
  f = MEMALLOC (DREAL,n);

  /* get initial iterate and lower, upper bounds */
  cpbnds_ (z, lower, upper, &n);

  cpfunf_ (z, f, &n);
  s = msgbuf;
  sprintf (s, "\n iterate\t  residual norm\t     CPLIB norm\n");
  while (*s) s++;
  sprintf (s,   " -------\t  -------------\t     ----------");
  c_print_msg (iolog, msgbuf);
  metric = nonsmooth_norm (n, z, f, lower, upper);
  obj = c_cpgetd ("obj");
  sprintf (msgbuf, "%6d\t%15.7f\t%15.7f", iteration, metric, obj);
  c_print_msg (iolog, msgbuf);

  while (iteration < iterlimit) {
    if (metric < 1e-6) {        /* convergence! */
      cpsoln_ (z, &n);
      c_cpputi ("modsta", MODEL_SOLVED);
      c_cpputi ("solsta", SOLU_NORMAL);
```

```
      return;
    }
    for (i = 0;  i < n;  i++) {
      z[i] -= stepsize * f[i];
      z[i] = PROJECT(lower[i], z[i], upper[i]);
    }
    iteration++;
    cpfunf_ (z, f, &n);
    metric = nonsmooth_norm (n, z, f, lower, upper);
    obj = c_cpgetd ("obj");
    sprintf (msgbuf, "%6d\t%15.7f\t%15.7f", iteration, metric, obj);
    c_print_msg (iolog, msgbuf);
  }
  c_cpputi ("modsta", MODEL_NOT_SOLVED);
  c_cpputi ("solsta", SOLU_ITERATION);
  return;
}
```