Getting CUTE with Matlab

Mary Ann Branch[†]
Department of Computer Science
Cornell University, Ithaca, NY 14853
branch@cs.cornell.edu

Report # CTC94TR194
September 1994

Abstract

CUTE is a testing environment for nonlinear programming algorithms developed by Bongartz, Conn, Gould and Toint ([CUTE: *Constrained and unconstrained testing environment*, Research Report RC 18860, IBM T.J. Watson Research Center, Yorktown Heights, USA, 1993]). The test problems in this environment are encoded in standard input format (SIF) and accessed by a set of FORTRAN subroutines. An extension to this environment was developed to allow fast access to the CUTE test problems from Matlab. This report describes this new Matlab interface to CUTE and how to use it.

[†]Research partially supported by the Applied Mathematical Sciences Research Program (KC-04-02) of the Office of Energy Research of the U.S. Department of Energy under grant DE-FG02-86ER25013.A000.

Contents

1 Introduction

The CUTE testing environment includes a database of test problems to aid developers of nonlinear optimization codes. The test problems are encoded in standard input format (SIF). The testing environment to select and access these problems is a combination of FORTRAN code and shell scripts. Once a test problem is selected, decoded, and compiled, an optimization code accesses the test problem by calling the CUTE tools which are FORTRAN subroutines.

To provide access to these same test problems for codes developed in Matlab ([The Mathworks, Inc., 1992]), this interface was created using the Matlab MEX-file facility. The interface is a combination of Matlab M-files and MEX-files and FORTRAN code. The result is a set of M-files that call the CUTE FORTRAN tools.

The main steps to use CUTE with this interface are

- select the problems using the shell script `select`
- decode and compile the problem into a MEX-file using the shell scripts `sdmatmex` and `matmex`
- call the CUTE tools from Matlab using the M-files discussed in §5

The emphasis of this user's guide is on the Matlab interface to CUTE. References to other CUTE documentation are given when appropriate. Also, the current description assumes a UNIX environment. Adjustments may be necessary for other systems.

2 Software requirements

The following software is needed to use the interface described in this document.

- The double precision version of CUTE (see [Bongartz *et al.*, 1993] for more details)

- The Matlab software package [The Mathworks, Inc., 1992], version 4.0 or higher
- A FORTRAN compiler that supports the `%VAL()` construct, an extension to standard FORTRAN-77 (see the comments in §6)

3 Preparing your environment

This section describes what to add or change in your environment. If you are using the C-shell, this amounts to updating your **.cshrc** file.

The rest of this document assumes that the double precision version of CUTE and the Matlab interface to CUTE have been installed.

- Two environment variables are needed. Set the variable `CUTEDIR` to the main directory where CUTE is installed. This is done in your `.cshrc` file with a line such as

  ```
  setenv CUTEDIR /vol/software/cute_files
  ```

  if `CUTE` were installed in the directory `/vol/software/cute_files`.

  The second variable specifies the directory of `.SIF` files. The following line added to your `.cshrc` file would set the variable `MASTSIF` to the main test file directory

  ```
  setenv MASTSIF $CUTEDIR/small_mastsif
  ```

  The two main directories of `.SIF` files are `$CUTEDIR/small_mastsif` and `$CUTEDIR/large_mastsif`; the second contains problems whose `.SIF` files are large and so are compressed (this means the problem takes many lines to specify but is not necessarily a "large" problem in terms of number of variables). Most of the problems are in `small_mastsif`.

- To access the M-files which call the CUTE FORTRAN tools, set the `MATLABPATH` environment variable to include the directory where the interface was installed, which should be a subdirectory of `$CUTEDIR`.

- Some of the CUTE directories must be added to your `path` variable. Two to include are

  ```
  $CUTEDIR/interfaces
  $CUTEDIR/select
  ```

  These can be added in your `.cshrc` file with the line

  ```
  set path=($path $CUTEDIR/interfaces $CUTEDIR/select)
  ```

- You will need access to the Matlab command `matlab`, the Matlab shell script `fmex`, and the FORTRAN compiler `f77`. Check that the appropriate directories are on your `path`.

4 Decoding and compiling test problems

For information on selecting test problems, see Appendix B. Once you have selected a test problem, the next step is to decode and compile.

Copy the desired `.SIF` file from the `MASTSIF` directory into your own local directory. Edit the file if necessary. See Appendix C.

To decode an *unconstrained* or *bound constrained* problem named `PROBNAME.SIF`, from the local directory execute the command

```
sdmatmex -u PROBNAME
```

(note that the `.SIF` is omitted). If the problem has *general constraints* omit the `-u` flag as in

```
sdmatmex PROBNAME
```

Both these commands will decode the `.SIF` file into four (or five) FORTRAN files and a data file. These files are

```
ELFUNS.f GROUPS.f RANGES.f SETTYP.f EXTERN.f OUTSDIF.d
```

(`EXTERN.f` may or may not be present). These files have these same names no matter what problem you decode, so you can only have one decoded problem per directory.

The `sdmatmex` command then calls `matmex`, which compiles the FORTRAN files above into object files and creates a Matlab MEX-file for accessing the given test problem. If running on a Sun SPARCstation under Solaris, this MEX-file will have the name `ucute.mexsol`, or `ccute.mexsol`, and will be written to the directory you issued the `sdmatmex` command from. On a Sun-4 SPARCstation running under SunOS, the file names are `ucute.mex4` or `ccute.mex4`.

If the decoding and compiling stages execute successfully, output from `sdmatmex -u PROBNAME` will be generated similar to

```
Problem name:  PROBNAME

Double precision version will be formed.

The objective function uses 9 nonlinear groups

There are 10 variables bounded from below and above

The Unconstrained CUTE tools are ready to run from Matlab
on the current test problem ...
```

If you wish to have more output about the problem and the decoding than shown above, you may use the `-o` flag as in

```
sdmatmex -o 1 PROBNAME
```

(`1` means verbose output; the default is `0` and generates minimal output). Beware: if the problem has many variables or groups, the output will be very long!

The `sdmatmex` command also has a help option as in

```
sdmatmex -h
```

The MEX-files generated are quite large as they link in many files and libraries. You will want to delete them when not being used actively. Once you have decoded a problem, but deleted the MEX-file, you can regenerate the MEX-file by executing

```
matmex -u
```

for unconstrained and bound constrained problems, or

```
matmex
```

for constrained problems (both commands use the exact same flags). Note that since only one decoded problem may be in any directory, you do not need to specify the problem name to `matmex` (in fact, it will not execute if you do).

It is necessary to run `sdmatmex` and `matmex` on the same machine you intend to run Matlab so that the correct compiler is used.

5 Matlab M-files to access the CUTE tools

Below are descriptions of the M-files. These files are located in the subdirectory of `$CUTEDIR` where the interface was installed. After a test problem has been decoded and compiled, these M-files may be called from Matlab from the directory where the compiled MEX-file resides (or anywhere else if that directory is on your `MATLABPATH`). The unconstrained tools *must* be used separate from the constrained tools (this restriction is inherited from CUTE).

Any calls to the unconstrained or bound constrained tools must be preceded by a call to `usetup.m`; likewise, calls to the constrained tools must be preceded by a call to `csetup.m`. It is recommended that you call the function `uclean.m` or `cclean.m` at the end. This closes and removes any files opened for error messages.

5.1 Unconstrained tools

For each matlab M-file, we show the first part of the M-file that includes the description. Thus the following descriptions are available on-line within Matlab using the `help` command.

- USETUP Set up the correct data structures for subsequent computations.

```
function [x, bl, bu]= usetup(nmax)
%        [x, bl, bu] = USETUP(nmax) initializes the problem.
%        x is the starting value (by default, CUTE uses the origin
%          as the starting point if not specified in the .SIF file).
%        bl is the lower bounds on the variables.
%        bu is the upper bounds on the variables.
%        x, bl, and bu are vectors of length n (the number of variables).
%
%        nmax is an upper bound on n.
%        NOTE: if nmax is less than n, an error will occur and a
%          message will be written to the file CUTE.ERR specifying
%          how much to increase nmax.
```

- UNAME Obtain the name of the problem.

```
function [pname]= uname
%       pname = UNAME returns the name of the problem.
```

- UFN Evaluate the function value at x.

```
function [f]= ufn(x)
%        f = UFN(x) computes the objective function f at the
%          point x.
```

- UGR Evaluate the gradient at x.

```
function [g]= ugr(x)
%        g = UGR(x) computes the gradient of the objective function
%          at the point x.
```

- UOFG Evaluate the objective function and possibly its gradient.

```
function [f, g]= uofg(x)
%        f = UOFG(x) computes the objective function f at x.
%
%        [f, g] = UOFG(x) computes the objective function value f and
%        the gradient g at x.
```

- UDH Evaluate the Hessian matrix. The matrix is stored as a dense matrix.

```
function [H]= udh(x)
%       H = UDH(x) computes the Hessian H at the point x.
```

- UGRDH Evaluate both the gradient and Hessian matrix. The matrix is stored as a dense matrix. Calling this routine is more efficient than separate calls to UGR and UDH.

```
function [g, H]= ugrdh(x)
%       [g, H] = UGRDH(x) computes the Hessian H and gradient g
%         of the objective function at the point x.
```

- USH Evaluate the Hessian matrix. The matrix is sparse.

```
function [Hs]= ush(x, maxnnzh)
%       Hs = USH(x) computes the sparse Hessian of the objective function
%         at the point x using the default value for maxnnzh, which specifies
%         the maximum number of nonzeros in Hs.  Currently the default
%         value is 6*n, where n is the length of x.
%
%       Hs = USH(x, maxnnzh) computes the sparse Hessian at x using
%         maxnnzh to override the default value.
%       NOTE: if maxnnzh is not large enough, an error will occur and
%         a message will be written to the file CUTE.ERR requesting
%         an increase in the size of IWK (an internal CUTE variable).
```

- UGRSH Evaluate both the gradient and Hessian matrix. The matrix is stored as a sparse matrix. Calling this routine is more efficient than separate calls to UGR and USH.

```
function [g, Hs]= ugrsh(x, maxnnzh)
%       [g, Hs] = UGRSH(x) computes the gradient and sparse Hessian
%         of the objective function at x using the default value of
```

```
%           maxnnzh, which specifies the maximum number of nonzeros in Hs.
%           Currently the default value is 6*n where n is the length of x.
%
%         [g, Hs] = UGRSH(x, maxnnzh) computes the gradient and
%           sparse Hessian at the point x using maxnnzh to override the
%           default value.
%         NOTE: if maxnnzh is not large enough, an error will occur and
%           a message will be written to the file CUTE.ERR requesting
%           an increase in the size of IWK (an internal CUTE variable).
%
```

- UPROD Form the product of a vector with the Hessian matrix.

```
function [q]= uprod(x, p, goth)
%         q = UPROD(x, p, goth) computes the product of p with the Hessian
%           evaluated at x, and stores the result in q.
%         goth specifies whether the second derivatives of the groups
%           and elements have already been set (goth nonzero) or if they
%           should be computed (goth = 0).  goth should be nonzero whenever
%           a previous call has been made to UDH, USH, UGRDH, or UGRSH at
%           the current point, or whenever a previous call with goth = 0
%           has been made to UPROD or UBANDH at the current point x.
%           Otherwise set goth = 0.
%
%         q = UPROD(x, p) computes the product with goth = 0.
```

- UBANDH Obtain the elements of the Hessian that lie within a given semi-bandwidth of its diagonal.

```
function [bandh]= ubandh(x, nsemib, goth)
%         bandh = UBANDH(x, nsemib, goth) computes the bands of the
%           Hessian evaluated at x that lie within nsemib of the
%           diagonal.
%         nsemib = 0 retrieves only the main diagonal of the Hessian.
%         bandh is a two-dimensional array with row i holding
```

```
%           the i-1 diagonal of the Hessian in columns 1:n-i+1 (where
%           diagonal 0 is the main diagonal).  Thus the diagonal entry
%           in column i of the Hessian is returned in location bandh(1,i),
%           while the entry j places below the diagonal in column i may be
%           found in location bandh(j+1, i).
%         goth specifies whether the second derivatives of the groups
%           and elements have already been set (goth nonzero) or if they
%           should be computed (goth = 0).  goth should be nonzero whenever
%           a previous call has been made to UDH, USH, UGRDH, or UGRSH at
%           the current point, or whenever a previous call with goth = 0
%           has been made to UPROD or UBANDH at the current point x.
%           Otherwise set goth = 0.
%
%         bandh = UBANDH(x, nsemib) computes the bands of the Hessian
%           with goth = 0.
```

- UCLEAN Clean up output files.

```
function uclean
%         UCLEAN will close and delete the file CUTE.ERR that USETUP
%           opened for potential error messages.
```

5.2 Constrained tools

- CSETUP Set up the correct data structures for subsequent computations.

```
function [x, bl, bu, equatn, linear, v, cl, cu] ...
     = csetup(nmax, mmax, efirst, lfirst, nvfirst)
%         [x,bl,bu,equatn,linear,v,cl,cu] = ...
%           CSETUP(nmax, mmax, efirst, lfirst, nvfirst)
%         x is the starting value (by default, CUTE uses the origin
%           as the starting point if not specified in the .SIF file).
%         bl is the lower bounds on the variables.
%         bu is the upper bounds on the variables.
```

```
%         equatn is a vector of 1's and 0's where equatn(i)==1 indicates
%           constraint i is an equation and equatn(i)==0 indicates
%           constraint i is an inequality.
%         linear is a vector of 1's and 0's where linear(i)==1 indicates
%           constraint i is linear or affine and linear(i)==0 indicates
%           constraint i is nonlinear.
%         v is the initial estimates of the Lagrange multipliers at the
%           solution to the problem.
%         cl is the lower bounds on the inequality constraints.
%         cu is the upper bounds on the inequality constraints.
%
%         nmax is an upper bound on n (the number of variables).
%         mmax is an upper bound on m (the number of general constraints).
%         efirst nonzero causes the general equations to occur before
%           the general inequalities in the list of constraints;
%           efirst=0 indicates the order is unimportant.
%         lfirst nonzero causes the general linear (or affine) constraints
%           to occur before the general nonlinear ones in the list of
%           constraints. lfirst=0 indicates order is unimportant.
%         Both efirst and lfirst nonzero cause the linear constraints to
%           occur before the nonlinear constraints.  The linear constraints
%           will have the linear equations before the linear inequalities.
%           And the nonlinear equations will appear before the nonlinear
%           inequalities.
%         nvfirst nonzero causes the nonlinear variables to appear before
%           the linear variables.  Within the nonlinear variables, the
%           smaller set of either the nonlinear objective or nonlinear
%           Jacobian variables appears first.
%         [x,bl,bu,equatn,linear,v,cl,cu] = ...
%           CSETUP(nmax, mmax, efirst, lfirst) calls CSETUP with
%           nvfirst=0.
%         [x,bl,bu,equatn,linear,v,cl,cu] = ...
%           CSETUP(nmax, mmax, efirst) calls CSETUP with lfirst=0
%           and nvfirst=0.
```

```
%        [x,bl,bu,equatn,linear,v,cl,cu] = ...
%          CSETUP(nmax, mmax) calls CSETUP with efirst=0, lfirst=0,
%          and nvfirst=0.
%        NOTE: if nmax is less than n or if mmax is less than m, an error
%          will occur and a message will be written to the file CUTE.ERR
%          specifying how much to increase nmax or mmax.
```

- CNAME Obtain the name of the problem.

```
function [pname]= cname
%        pname = CNAME
```

- CFN Evaluate the objective and general constraint functions at the point x.

```
function [f, c]= cfn(x)
%        [f, c] = CFN(x) computes the objective function f and the
%          general constraint functions c at x.
```

- CGR Evaluate the gradients of the general constraint functions and the gradient of either the objective function or the Lagrangian function.

```
function [g, cjac]= cgr(x, jtrans, v, grlagf)
%        [g, cjac] = CGR(x, jtrans, v, grlagf) computes g as the gradient of
%          the objective function if grlagf==0 and the gradient of the
%          Lagrangian if grlagf is nonzero.
%        v is the Lagrange multipliers used when grlagf is nonzero.
%        cjac is the Jacobian matrix of the constraint functions if
%          jtrans==0 and its transpose if jtrans is nonzero.
%
%        [g, cjac] = CGR(x, jtrans) computes g as the gradient of the
%          objective function (grlagf==0).
%
%        [g, cjac] = CGR(x) computes g as the gradient of the objective
%          function (grlagf==0) and cjac as the Jacobian of the
%          constraint functions (jtrans==0).
```

- COFG Evaluate the objective function and possibly its gradient.

```
function [f, g]= cofg(x)
%       f = COFG(x) computes the objective function f at x.
%
%       [f, g] = COFG(x) computes the objective function value f and
%       the gradient g at x.
```

- CSGR Evaluate the gradients of the general constraint functions. Also obtain the gradient of either the objective function or the Lagrangian function. The gradients are sparse matrices.

```
function [gs, csjac]= csgr(x, jtrans, v, grlagf, nnzcj)
%       [gs, csjac] = CSGR(x, jtrans, v, grlagf, nnzcj) computes gs as
%         the sparse gradient of the objective function if grlagf==0
%         and the sparse gradient of the Lagrangian if grlagf is nonzero.
%       v is the Lagrange multipliers used when grlagf is nonzero.
%       csjac is the sparse Jacobian matrix of the constraint functions
%         if jtrans==0 and its transpose if jtrans is nonzero.
%       nnzcj is the maximum number of nonzeros in csjac and overrides
%         the default value.
%
%       [gs, csjac] = CSGR(x, jtrans, v, grlagf) uses the default
%         value of nnzcj.
%
%       [gs, csjac] = CSGR(x, jtrans) computes gs as the gradient of the
%         objective function (grlagf==0) and uses the default for nnzcj.
%
%       [gs, csjac] = CSGR(x) computes gs as the gradient of the objective
%         function (grlagf==0) and csjac as the Jacobian of the
%         constraint functions (jtrans==0) and uses the default for nnzcj.
```

- CCFG Evaluate the constraint functions and possibly their gradients. The gradients are dense matrices.

```
function [c, cjac]= ccfg(x, jtrans)
%        c = CCFG(x) computes the values of the general constraint
%          functions c at the point x.
%
%        [c, cjac] = CCFG(x, jtrans) computes the general constraint
%          functions c and the Jacobian matrix of the constraint
%          functions cjac if jtrans==0 and its transpose if
%          jtrans is nonzero (at the point x).
%
%        [c, cjac] = CCFG(x) computes the general constraint
%          functions c and the Jacobian matrix of the constraint
%          functions cjac (jtrans==0).
```

- CSCFG Evaluate the constraint functions and possibly their gradients. The gradients are sparse matrices.

```
function [c, csjac]= cscfg(x, jtrans, nnzcj)
%        c = CSCFG(x) computes the values of the general constraint
%          functions c at the point x.
%
%        [c, csjac] = CSCFG(x, jtrans, nnzcj) computes the general
%          constraint functions c and the sparse Jacobian matrix of the
%          constraint functions csjac if jtrans==0 and its transpose if
%          jtrans is nonzero (at the point x).
%        nnzcj is the maximum number of nonzeros in csjac and overrides
%          the default value.
%
%        [c, csjac] = CSCFG(x, jtrans) computes the general constraint
%          functions c and the Jacobian matrix of the constraint
%          functions cjac if jtrans==0 and its transpose if
%          jtrans is nonzero (at the point x).  The default for
%          nnzcj is used.
%
%        [c, csjac] = CCFG(x) computes the general constraint
%          functions c and the Jacobian matrix of the constraint
```

```
%          functions cjac (jtrans==0).  The default for nnzcj is
%          used.
```

- CDH Evaluate the Hessian matrix of the Lagrangian function. The matrix is stored as a dense matrix.

```
function [H]= cdh(x,v)
%        H = CDH(x,v) computes the Hessian of the Lagrangian H, with
           respect to x, evaluated at x and v.
```

- CGRDH Evaluate both the gradients of the general constraint functions and the Hessian matrix of the Lagrangian function. Also obtain the gradient of either the objective function or the Lagrangian function. The matrices are dense. Calling this routine is more efficient than separate calls to CGR and CDH.

```
function [g, cjac, H]= cgrdh(x, v, jtrans, grlagf)
%        [g, cjac, H] = CGRDH(x, v, jtrans, grlagf) computes the Hessian
%         of the Lagrangian H (with respect to x) at x and v.
%        v is the Lagrange multipliers.
%        g is the gradient of the objective function if grlagf==0
%          and the gradient of the Lagrangian if grlagf is nonzero.
%        cjac is the Jacobian matrix of the constraint functions if
%          jtrans==0 and its transpose if jtrans is nonzero.
%
%        [g, cjac, H] = CGRDH(x, v, jtrans) computes g as the gradient of
%          the objective function (grlagf==0).
%
%        [g, cjac, H] = CGRDH(x, v) computes g as the gradient of the
%          objective function (grlagf==0) and cjac as the Jacobian of the
%          constraint functions (jtrans==0).
```

- CSH Evaluate the Hessian matrix of the Lagrangian function. The matrix is sparse.

```
function [Hs]= csh(x, v, maxnnzh)
%        Hs = CSH(x,v) computes the sparse Hessian of the Lagrangian
%          function, with respect to x,  evaluated at x and v using
%          the default value for maxnnzh, which specifies
%          the maximum number of nonzeros in Hs.  Currently the default
%          value is 6*n, where n is the length of x.
%        v is the Lagrange multipliers.
%
%        Hs = CSH(x, v, maxnnzh) computes the sparse Hessian of the
%          Lagrangian at x using maxnnzh to override the default value.
%        NOTE: if maxnnzh is not large enough, an error will occur and
%          a message will be written to the file CUTE.ERR requesting
%          an increase in the size of IWK (an internal CUTE variable).
```

- CSGRSH Evaluate both the gradients of the general constraint functions and the Hessian of the Lagrangian function. Also obtain the gradient of either the objective function or the Lagrangian function. The gradients and Hessian are sparse matrices. Calling this routine is more efficient than separate calls to CSGR and CSH.

```
function [gs, csjac, Hs]= csgrsh(x, v, jtrans, grlagf, nnzcj, maxnnzh)
%        [gs, csjac, Hs] = CSGRSH(x, v, jtrans, grlagf, nnzcj, maxnnzh)
%          computes gs as the sparse gradient of the objective function
%          if grlagf==0 and the sparse gradient of the Lagrangian if
%          grlagf is nonzero.
%        Hs is the sparse Hessian of the Lagrangian, with respect to
%        x, evaluated at x and v.
%        v is the Lagrange multipliers.
%        csjac is the sparse Jacobian matrix of the constraint functions if
%          jtrans==0 and its transpose if jtrans is nonzero.
%        nnzcj is the maximum number of nonzeros in csjac and overrides
%          the default value.
%        maxnnzh overrides the default value for the maximum number of
%          nonzeros in Hs.  Currently the default value is 6*n, where n
%          is the number of variables.
```

```
%
%        [gs, csjac, Hs] = CSGRSH(x, v, jtrans, grlagf, nnzcj) uses the
%          default value of maxnnzh.
%
%        [gs, csjac, Hs] = CSGRSH(x, v, jtrans, grlagf) uses the default
%          values of nnzcj and maxnnzh.
%
%        [gs, csjac, Hs] = CSGRSH(x, v, jtrans) computes gs as the gradient
%          of the objective function (grlagf==0) using the defaults for nnzcj
%          and maxnnzh.
%
%        [gs, csjac, Hs] = CSGRSH(x, v) computes gs as the gradient of the
%          objective function (grlagf==0) and csjac as the Jacobian of the
%          constraint functions (jtrans==0) and the defaults for nnzcj and
%          maxnnzh.
```

- CPROD Form the product of a vector with the Hessian matrix of the Lagrangian function.

```
function [q]= cprod(x, p, v, goth)
%       q = CPROD(x, p, v, goth) computes the product of p with the
%         Hessian of the Lagrangian evaluated at x, and stores the
%         result in q.
%       v is the Lagrange multipliers.
%       goth specifies whether the second derivatives of the groups
%         and elements have already been set (goth nonzero) or if they
%         should be computed (goth = 0).  goth should be nonzero whenever
%         a previous call has been made to CDH, CSH, CGRDH, or CSGRSH at
%         the current point, or whenever a previous call with goth = 0
%         has been made to CPROD at the current point x.
%         Otherwise set goth = 0.
%
%       q = CPROD(x, p, v) computes the product of p with goth = 0.
```

- CCLEAN Clean up output files.

```
function cclean
%       CCLEAN will close and delete the file CUTE.ERR that CSETUP
%         opened for potential error messages.
```

6 Comments

1. To call `usetup.m` or `csetup.m`, you need to know an upper bound on the number of variables and the number of constraints. This information is printed when the problem is decoded no matter what level of output you request. For efficiency reasons, the smallest upper bound is preferred to reduce the amount of memory Matlab uses.

2. For some of the sparse routines, you may also have to give an upper bound on the number of nonzeros in the Hessian or the Jacobian of the constraints. It is best to give the smallest possible upper bound.

3. Since the problems are generated with FORTRAN, there is no value `Inf` as in Matlab. Thus, for example, the vectors of upper and lower bounds `bl` and `bu` will not be set to `-Inf` and `Inf` when no bounds are specified. Rather, they will be set to `-1e20` and `1e20`. (The exact number they are set to may differ on your system.) You may want to check this and convert to `-Inf` and `Inf`.

4. If an error is detected in the Matlab half of the interface, an error message is sent to the Matlab environment. If an error is detected in the one of the FORTRAN CUTE tools, an error message will be sent to the Matlab environment stating that further error information can be found in the file `CUTE.ERR` in the current working directory. This file can be viewed within Matlab using the command

   ```
   type CUTE.ERR
   ```

   Runtime errors in the CUTE tools usually occur when one of the parameters such as `NNZSH`, `NMAX`, or `MMAX` is too small.

5. The CUTE tools that are being accessed by the Matlab interface are a series of FORTRAN files with large COMMON blocks. Many of inconveniences in this Matlab interface are inherited from the FORTRAN tools (such as specifying NMAX and NNZSH before you know them).

6. Because these COMMON blocks persist between calls to the CUTE tools, it is necessary to clear the MEX-file between runs of the same problem using "clear mex" at the Matlab prompt. Furthermore, after running `sdmatmex` or `matmex`, you should quit and restart matlab; failure to do so may result in a segmentation error by Matlab.

7. This interface could have been written without requiring a FORTRAN compiler that supports the `%VAL` construct. However, this would have substantially increased the run time of each call to the CUTE tools.

8. Please send bugs and comments to `branch@cs.cornell.edu`.

7 Common problems

- If an error occurs when executing `sdmatmex` or `matmex`, rerun in verbose mode (with the flag `-o 1`) to determine what stage the error is occurring.

  If it is happening in the decoding stage, it is likely that you have exceeded the space limitations of the FORTRAN code that does the decoding. You will have to re-install the CUTE SIF decoder and CUTE tools with larger parameters. See the file `$CUTEDIR/doc/install.rdm`.

  If the error occurs in the compile stage, it is likely the error is happening when `fmex` is called. One possibility is that the `f77` compiler on your system does not support the `%VAL()` construct which is an extension to standard FORTRAN-77. It is supported by most, but not all, FORTRAN compilers.

  A second possibility is `fmex` is not being called with the correct flags and libraries. To ensure the correct compiler options are used when generating MEX-files, an appropriate `.mexrc.sh` file may be needed in your home directory. See the file `README.mex` in the directory where the Matlab shell script `fmex` is located for information about `.mexrc.sh`.

8 Obtaining the Matlab interface to CUTE

This interface is available via anonymous ftp from `ftp.cs.cornell.edu` `[128.84.154.10]` in the directory `/pub/branch/interface`. Get the files

```
README
mat_cute.tar.Z
```

These files are also accessible using Mosaic at the URL

```
ftp://ftp.cs.cornell.edu/pub/branch/interface
```

9 Acknowledgements

I thank Tom Coleman for testing the interface and commenting on this report, and Ingrid Bongartz for some helpful communication during the writing of this interface.

Appendix

A Obtaining CUTE

The following description for obtaining CUTE is taken from [Bongartz *et al.*, 1993]:

> The package may be obtained in one of two ways. Firstly, the reader can obtain CUTE electronically via an anonymous ftp call to the account at the Rutherford Appleton Laboratory *camelot.cc.rl.ac.uk* (Internet i.d. 130.246.8.61, in directory pub/cute), or at Facultés Universitaires Notre-Dame de la Paix (Namur) *thales.math.fundp.ac.be* (Internet i.d. 138.48.4.14, in directory cute). We request that the userid be given as the password. This will serve to identify those who have obtained a copy via ftp.

B Selecting Test Problems

The classification scheme and the tools to select problems are documented in [Bongartz *et al.*, 1993]. A latex and postscript version of this paper is found in the directory `$CUTEDIR/doc` in the files `paper.tex` and `paper.ps`. See §2, 3, 4 and appendix B.

The main tool is the shell script `select`. We include only a brief description and refer the reader to [Bongartz *et al.*, 1993] for more details. To run the `select` script issue the command

```
select
```

This script will probe .DB files contained in the directory pointed to by the shell variable `MASTSIF`. The script then prompts the user for various problem characteristics. It finishes by listing the problems in the `MASTSIF` directory that meet those characteristics. It will also allow you to save these names to a file. Fully specify the path name for the file or it will try to save it to the `MASTSIF` directory. If you do not have write permission on the `MASTSIF` directory, it will abort. The complete path and directory name cannot exceed 32 characters.

If you want to select problems from another directory, remember to update the `MASTSIF` variable in your `.cshrc` file.

C Changing the size of the problem in SIF

The size of the problem is, in general, specified at the beginning of the `.SIF` file in some documented manner. The only way to change the number of variables (or constraints) is to edit the `.SIF` file directly.

The files usually have some different problem sizes already in the file, with all but one commented out. (A comment is denoted by a asterisk (*) in column 1 of the file). For example, an excerpt of a `.SIF` file may look like:

```
*   N is the number of variables

 IE N                   10
*IE N                   50
*IE N                   100
*IE N                   500
*IE N                   1000
*IE N                   5000
*IE N                   10000
*IE N                   50000
```

In the file, `N` is 10 since the other lines are commented out.

The `.SIF` files are dependent on fields, that is, certain parts of each line *must* be within certain columns. Make sure you do not change the format when removing and adding *'s.

Once you have changed the desired variables, rerun `sdmatmex`.

More information about `.SIF` files may be found in ([Bongartz *et al.*, 1993]) or ([Conn *et al.*, 1992a]).

References

[Bongartz *et al.*, 1993] I. Bongartz, A. R. Conn, N. I. M. Gould, and Ph. L. Toint. CUTE*: Constrained and unconstrained testing environment.* Research Report RC 18860, IBM T.J. Watson Research Center, Yorktown Heights, USA, 1993.

[Conn *et al.*, 1992a] A. R. Conn, N. I. M. Gould, and Ph. L. Toint. LANCELOT*: a Fortran package for large-scale nonlinear optimization (Release A).* Number 17 in Springer Series in Computational Mathematics. Springer Verlag, Heidelberg, Berlin, New York, 1992.

[The Mathworks, Inc., 1992] The MathWorks, Inc. *Matlab User's Guide.* The MathWorks, Natick, MA, 1992.