Science & Technology Facilities Council

Technical Report

Guidelines for development of Matlab interfaces for HSL packages

M. Arioli, I. S. Duff, N. I. M. Gould,
J. D. Hogg, and H. S. Thorne

April 19, 2010

RAL-TR-2010-13

Library and Information Services
SFTC Rutherford Appleton Laboratory
Harwell Science and Innovation Campus
Didcot
OX11 0QX
UK
Tel: +44 (0)1235 445384
Fax: +44(0)1235 446403
Email: library@rl.ac.uk

The STFC ePublication archive (epubs), recording the scientific output of the Chilbolton, Daresbury, and Rutherford Appleton Laboratories is available online at:
http://epubs.cclrc.ac.uk/

ISSN 1358-6254

RAL-TR-2010-013

Guidelines for development of Matlab interfaces for HSL packages

M. Arioli, I. S. Duff, N. I. M. Gould, J. D. Hogg, and H. S. Thorne,

ABSTRACT
In this report we describe the modus operandi for providing Matlab interfaces for HSL codes. We discuss the file structure for the HSL-Matlab interface and how the mex file should be constructed. We also provide details of an hsl_matlab package that is designed to facilitate the interface and discuss how the user can install the resulting software on LINUX platforms.

Keywords: MATLAB interfaces, HSL, Fortran

Current reports available by anonymous ftp to ftp.numerical.rl.ac.uk in directory pub/reports.
The work was supported by EPSRC grant GR/S42170/01.

Computational Science and Engineering Department
Atlas Centre
Rutherford Appleton Laboratory
Oxon OX11 0QX

April 19, 2010

Contents

1 Introduction

These guidelines are a simple indication of a safe and consistent approach to the design of interfaces between HSL or GALAHAD Fortran codes and the Matlab environment. We will restrict ourselves to LINUX platforms and rely on the presence of the `g95` open source Fortran compiler. Although developers can maintain reasonable freedom, we strongly encourage them to follow these guidelines when designing and coding the interface.

Matlab interfaces provide a mechanism for using HSL or GALAHAD packages within the Matlab environment. Matlab is widely used so the provision of such an interface can be of great benefit to current HSL or GALAHAD users and should also increase the number of users of this software.

A Matlab interface is distributed with selected HSL or GALAHAD packages and consists of the following, in addition to the standard components of a package:

- a `mex` file for the HSL or GALAHAD package;
- a `mex` file containing the `hsl_matlab` or `galahad_matlab` package;
- an installation file: `<packagename>_install.m`;
- a file `README` containing general information about using the interface;
- a file `INSTALL` containing installation instructions for the interface;
- for some HSL codes, additional Matlab mex files may be present if other interfaces are required. For example, for the HSL packages `MA57` and `ME57`, there are interfaces that return the factors rather than just solve a set of equations.

We will describe these components in the following sections as well as discussing the directory structure, documentation, and testing. We focus in what follows on HSL codes: essentially identical ideas are used within GALAHAD.

2 Fortran codes for the HSL package

When we develop a Matlab interface, our starting point will be the HSL Fortran package. **As the interface will be distributed with the Fortran package as part of the library, any changes required to the main HSL Fortran package must be reflected in and tested with the Matlab interface and vice-versa.**

There may be occasions when minor changes need to be carried out within the Fortran code to enable the provision of a Matlab interface. Accordingly, the version should be updated and passed to the HSL librarian. For example, in the Fortran 95 HSL codes, it is common practice to use a derived data type that contains components that the user must not alter. To ensure this, the components are declared as `private`. However, within our Matlab interface, if we need access to these components, we will need to remove the `private` declaration. Similarly, if any of the dependencies need modification, the associated HSL packages should be formally updated.

3 The `mex` file

The main component of the Matlab interface is its `mex` file: `mex` stands for *Matlab executable*. We have chosen to construct the `mex` file based on Fortran 95. The `mex` file is a way to call Fortran (or C) routines from Matlab as if they were built-in Matlab functions. A major complication with these files is that the interface between Fortran and Matlab is far from straightforward, in particular Matlab pointers are quite different from those used in Fortran and there is no direct equivalent of the derived data types used in Fortran. The `mex` file should always have the extension `.F90` and its name determines how the function is called within Matlab. For example, the `mex` file `factorlu.F90` will produce a Matlab function that can be called as follows

```
[output_arguments] = factorlu(input_arguments)
```

Remark 3.1 *It is necessary that the suffix* `.F90` *be in upper case format. The* `mex` *command does not accept a suffix* `.f90`*. The capital F90 indicates that a preprocessor must be run to obtain standard-compliant Fortran code.*

All `mex` files for interfacing with our codes should include three components:

- initial statement: `#include <fintrf.h>`
- the gateway subroutine
- the statement: `use hsl_matlab`

In Listing 1, we provide the skeleton of a `mex` file. We will now describe the components of the `mex` file in more detail.

Listing 1: Skeleton of mex file

```
#include <fintrf.h>

! The gateway routine
  SUBROUTINE mexFunction(nlhs,plhs,nrhs,prhs)
  use hsl_matlab
  ! Other 'use' statements

  implicit none

  ! mex declarations
  integer*4 :: nlhs, nrhs
  mwPointer :: plhs(*), prhs(*)

  ! Other variable declarations

  ! Body of the subroutine (in Fortran)

  END SUBROUTINE mexFunction
```

3.1 `fintrf.h`

The initial statement `#include <fintrf.h>` is necessary to provide the underlying interface routines from Matlab. For further information on this see Matlab "help".

3.2 The gateway routine

The *gateway routine* is the entry point to the `mex` interface. It is through this routine that Matlab accesses the subroutines from the HSL package. The name of the gateway routine must be `mexFunction` and it must contain the parameters `nlhs`, `plhs`, `nrhs` and `prhs`, where `nrhs` and `nlhs` are the number of input and output arguments, respectively, and `prhs` and `plhs` are arrays of Matlab pointers to the input and output arguments, respectively.

It is good practice to give alias names to the input and output variables even if it is not necessary. In Listing 2, we provide an example.

Listing 2: Alias in mex file

```
#include <fintrf.h>

! The gateway routine
  SUBROUTINE mexFunction(nlhs,plhs,nrhs,prhs)
  use hsl_matlab
  ! Other 'use' statements

  implicit none

  ! mex declarations
  integer*4 :: nlhs, nrhs
  mwPointer :: plhs(*), prhs(*)

  ! Alias example
  ! We assume that we have one input argument
  ! and two output arguments
  ! i.e. nlhs = 1 and nrhs = 2
  ! (a matrix on input and two matrices on output)

  mwSize :: A_inp, L_out, U_out

  A_inp = 1
  L_out = 1
  U_out = 2

  ! prhs(1) is then equal to prhs(A_inp) and
  ! plhs(1) is equal to plhs(L_out) and
  ! plhs(2) is equal to plhs(U_out)

  ! Body of the subroutine (in Fortran)
  ......

  END SUBROUTINE mexFunction
```

3.3 `hsl_matlab.F90`

The `hsl_matlab.F90` package is a Fortran-based module that provides interfaces to commonly used routines that are provided by `fintrf.h`. This enables portability of the interfaces between

32 and 64-bit machines and also allows for better maintainability of the code. For example, any changes to these underlying routines will only result in an alteration to the hsl_matlab package. If you find that you need to use a mex function that is not available via an interface within hsl_matlab, then you should update the hsl_matlab package to include an interface to this function.

The hsl_matlab package is available with the distribution and provides interfaces to a wide range of mex functions and subroutines. The documentation is in the file:

/numerical/num/hsl2007/packages/hsl_matlab/hsl_matlab.ral.pdf .

3.4 Variable Declarations

The variable declarations are split into two parts: those associated with mex declarations and those associated with standard Fortran declarations.

To enable cross-platform flexibility, Matlab contains several preprocessor macros (we call them Matlab data types) that are used by hsl_matlab functions:

- mwSize is a data type for sizes, such as array dimensions and number of elements; default Fortran integers should be declared as such.
- mwIndex is a data type for index values, needed to identify components of arrays; integer arguments to hsl_matlab functions should be as such.
- mwPointer: is a data type for a Matlab pointer. Matlab uses a unique data type, the mxArray. Because there is no way to create such a data type in Fortran, Matlab passes a special identifier, created by the mwPointer preprocessor macro, to a Fortran program.

The Fortran preprocessor hsl_matlab converts mwPointer variables to integer*4 variables when building binary mex-files on 32-bit platforms and to integer*8 variables when building on 64-bit platforms.

After the mex declarations, we have the standard Fortran declarations including any allocatable arrays that are required to run the Fortran package and any other functionality that is required.

3.5 Body of the mex code

3.5.1 Initial Check

The first check is on the arguments nrhs and nlhs of the gateway function. These must be checked to ensure that the correct number of arguments have been passed to the function. In the following example, there must be between 1 and 3 input arguments and between 1 and 5 output arguments.

```
IF (nrhs < 1 .or. nrhs > 3) THEN
  CALL MATLAB_error( "Wrong # of input arguments" )
END IF
IF (nlhs < 1 .or. nlhs > 5) THEN
  CALL MATLAB_error( "Wrong # of output arguments" )
END IF
```

The subroutine MATLAB_error prints an error message and returns the user to the Matlab prompt.

3.5.2 Matlab structures

In the Fortran 95 HSL codes, it is common practice to define data types to hold the control parameters, information details, and data that needs to be stored from one call to the next. In Matlab, we can use *structures* to carry out the same functionality. Unfortunately, we have discovered that there is currently a restriction on the use of array components within structures (namely, you can define and assign arrays within a Matlab structure but they cannot be read back from the structure within the interface). Hence, arrays need to be passed separately.

3.5.3 Allocation and deallocation

Having declared the local variables, storage must be allocated for each array to be used in the `mex` file. The command

```
ALLOCATE( name( n+1 ), STAT = err  )
```

will allocate the array `name` of dimension `n+1`. If the allocation is successful `err` will be equal to zero. Otherwise, an error has occurred (normally we have run out of storage). Thus, it is necessary to check if the allocation was successful:

```
IF (err .ne. 0) THEN
  CALL MATLAB_error( " Insufficient memory " )
END IF
```

Before the final return to Matlab it is necessary to deallocate all the arrays that have been used. Input or output variables must not be deallocated. To avoid an error return we always check whether an array is allocated before deallocating it viz.

```
IF ( ALLOCATED( name ) ) DEALLOCATE( name, STAT = err)
```

4 Installation files

4.1 Directory structure

The Matlab interface is treated as part of the library package and sits in its own `matlab` subdirectory of the main package directory.

The files in this directory must all be prefixed with the package name, except for the README and INSTALL files. This ensures that multiple HSL Matlab interfaces can be stored in the same directory.

For distribution purposes, the name of the parent directory is included in the name of most of the following files:

README information on the installation of the package for use with Fortran, including external libraries and a pointer to the matlab subdirectory.

`<packagename>.pdf` user documentation for the Fortran interface.

`<packagename>.f/f90` the source code for the main Fortran routine (for Fortran 77/Fortran 90, respectively).

`ddeps.f` the source code for Fortran 77 style dependencies, if any. Will not be present if none exist.

`ddeps90.f90` the source code for Fortran 90 style dependencies, if any. Will not be present if none exist.

Typical files inside the `matlab` directory:

`<packagename>_install.m` installation script

`<packagename>_test.m` test script

`<packagename>_test_data1.mat` matlab data for test, file 1

`<packagename>_test_data2.mat` matlab data for test, file 2

`<packagename>_full_test.m` comprehensive test of interface

`<filename>.output` sample output from running the test

The matlab directory should not contain any subdirectories except the optional `examples` subdirectory containing several Matlab files and data files that give examples of how to use the interface (supplementary to the `<packagename>_test.m` file).

It is our convention to use the extension `.output` for example output files, and `.log` for output files generated when the user runs the code. This prevents example outputs being overwritten accidentally.

4.2 The `<packagename>_install.m` file

Installation is performed using the Matlab file `<packagename>_install.m`. It can have several input arguments passed using the `varargin` parameter:

`<packagename>_install()` installs `<packagename>` and its Matlab Interface. It is assumed that the `BLAS` and `LAPACK` routines provided with the interface are used, the relevant version of `g95` can be called by the command "**g95**" and that the default libraries (`libf95.a` and `libgcc.a`) are available for use. The test example is not run.

`<packagename>_install`**(TEST)** installs `<packagename>` and its Matlab Interface. It is assumed that the `BLAS` and `LAPACK` routines provided with the interface are used, the relevant version of `g95` can be called by the command "**g95**", and that the default libraries are available for use. If TEST <= 0, the test example is not run; if TEST > 0, the test example is run.

`<packagename>_install`**(TEST, BLAS)** installs `<packagename>` and its Matlab Interface. It is assumed that the relevant version of `g95` can be called by the command "**g95**" and that the default compiler libraries are available for use. If BLAS <= 0, the `BLAS` and `LAPACK` routines that are provided with the interface are used. If BLAS > 0, the default `BLAS` and `LAPACK` routines provided by Matlab are used. If the installed `BLAS/LAPACK` have any bugs, then this may not be robust and BLAS <= 0 should be used.

`<packagename>_install`**(TEST, BLAS, COMPILER)** installs `<packagename>` and its Matlab Interface. COMPILER is a string and it must contain the path for the relevant `g95` compiler. It is assumed that the default compiler libraries are available for use.

`<packagename>_install`**(TEST, BLAS, COMPILER, LIBF95)** installs `<packagename>` and its Matlab Interface. LIBF95 is a string and must contain the path where the desired version of `libf95.a` is located, for example, '/usr/local/g95-install/lib/gcc-lib/i686-pc-linux-gnu/4.0.3'. The default gcc library is used.

`<packagename>_install`**(TEST, BLAS, COMPILER, LIBF95, LIBGCC)** attempts to install
`<packagename>` and its Matlab Interface. LIBGCC is a string and must contain the path where the desired version of `libgcc.a` is located, for example, '/usr/lib/gcc/i386-redhat-linux/3.4.6'.

The `<packagename>_install.m` file has four parts:

1. Initially, it checks whether it is on a `Linux` system and, if so, whether the version of Matlab is recent enough for the installation of the interface. It also checks whether the `g95` and the `gcc` libraries are correctly installed in the directories given as input. Finally, it determines whether it is on a 32 or 64-bit machine.
2. It sets all the paths in accordance to the directory of sub-section 4.1 and all the libraries that must be linked. Then it compiles all of the dependences and the Fortran code using `g95`. Finally, it executes the `mex` command.
3. It adds the current path where the `<packagename>.mexglc` (the object code) is has been generated.
4. It runs `test.m` if required.

5 Matlab codes to enable simple call of subroutines

The compiled version of the `mex` file can be used either standalone or through a short Matlab file. We recommend the second strategy for two reasons:

- We can design a simpler user interface giving the naive user some simple predetermined options but also allowing a more aware user the possibility of using more sophisticated options.
- We can use the header comments in the Matlab files for a quick on-line help that can be called directly in a Matlab session by the `help` command.

The short Matlab file will have the name `<packagename>.m`.

6 Interface testing

As described in Section 4.1, the developer must supply a `<packagename>_test.m` file that can be used either to check the installation and/or to supply useful examples of execution with the different input and output arguments.

In addition, an automated exhaustive testing of the interface should be performed by the script `<packagename>_full_test.m`. To test errors such as the wrong number of arguments, a try-catch structure should be used. An example that tests for no arguments is shown in Listing 3. This mechanism enables all the interface tests to be included in a single script with each test similar to that shown in Listing 3.

Listing 3: Catching exceptions

```
% No inputs
try
    x = hsl_mi20();
catch
    errstr = lasterror;
    str=strtrim(errstr.message);
    str1=strtrim(['hsl_mi20 requires at least 1' ...
        'input argument']);
    if (size(strfind(str,str1),2)==0)
        error('Failure at error test 1')
    end
end
```

The newer version of the `try-catch` construct that replaces `catch; errstr = lasterror` with the simpler `catch errstr` version **must not** be used as this is incompatible with older versions of matlab. Before an update is accepted to a library code this test and the Fortran comprehensive test must both complete successfully.

7 Pitfalls

For those used to Fortran programming and the good diagnostics available from most Fortran compilers, the Matlab environment can come as a shock, in particular the debugging of mex files is not always a pleasurable business. In this section, we highlight some of the problems we have already encountered and request that you report any other issues to us so that we can expand this section accordingly.

- One of the main issues is that Matlab structures and pointers do not really have anything in common with Fortran derived data types or pointers.
- Variables in mexfiles are case sensitive so that variable jimmy is quite different from Jimmy, for example.
- If an array is allocated both within a mexfile and in a Fortran subroutine called by the mexfile, then no warning or error flag is raised but the program will fail badly, usually with a segmentation fault.
- There is very little helpful diagnostic messages when a Matlab program fails, a return is always made back to the Matlab calling environment and very often the Matlab window is closed.

8 Documentation

8.1 The `README` file

The `README` file contains details about the installation requirements and the use of the Matlab interface. Some of this information will be repeated in the online documentation (Section 8.2). The `README` file should contain the following sections:

1. A short introduction that describes what the package is used for, any assumption made, and references.
2. The requirements for installing the interface
   - version of Matlab and g95;
   - the Matlab environment variables point to the correct place.
3. The directory structure with description of the files and subdirectories.
4. Reference to installation instructions contained in the INSTALL file.
5. A description of the Matlab interface `<packagename>`.m and of the test examples. Some Matlab interfaces will be only a way to give on-line information. Other interfaces will be more sophisticated and will allow the user to use more options.
6. A description of any control substructures and their components.
7. A description of any information structures and their components.

8.2 On-line documentation

The documentation for the code is included in the `<packagename>`.m file so that it can be viewed using the Matlab `help <packagename>` command. Analogously, with the command `help hsl_install` the documentation of the installation can be seen on-line.

A Example of the I/O for a sparse matrix

In this section, we will firstly show how to take a sparse Matlab matrix (given as the first input to the Matlab function) and copy it to a matrix of `zd11_type` with compressed column format storage.

```
! Store matrix%m, matrix%n and matrix%ne
      m = MATLAB_get_m(prhs(1))
      matrix%m = m
      n = MATLAB_get_m(prhs(1))
      matrix%n = n
      ne = MATLAB_get_nzmax(prhs(1))
      matrix%ne = ne

! Allocate arrays
      ALLOCATE( matrix%row(ne), matrix%val(ne), matrix%ptr(n+1), STAT=err )
      IF (err .ne. 0) THEN
        CALL MATLAB_error( " Insufficient memory " )
      END IF

! Copy pointers to column start
      cpr_pr = MATLAB_get_jc(prhs(1))
      CALL MATLAB_copy_from_ptr( cpr_pr, matrix%ptr, n+1 )
! Use Fortran indexing
      matrix%ptr(1:n+1) = matrix%ptr(1:n+1)+1
```

```
! Copy row indices
      row_pr = MATLAB_get_ir(prhs(1))
      CALL MATLAB_copy_from_ptr( row_pr, matrix%row, ne )
! Use Fortran indexing
      matrix%row(1:ne) = matrix%row(1:ne)+1

! Copy nonzero entries
      val_pr = MATLAB_get_ptr(prhs(1))
      CALL MATLAB_copy_from_ptr( val_pr, matrix%val, ne )

! Core of Fortran routine
       CALL dummy(matrix,matrix_out)

! Extract information for Matlab

! plhs(1) = matrix_out stored by rows

       CALL_MATLAB_CreateSparse(matrix_out%m, matrix_out%n, matrix_out%ne, 0, plhs(1))

        jc_pr  = MATLAB_get_jc(plhs(1))
        ir_pr   = MATLAB_get_ir(plhs(1))
        ptr_pr = MATLAB_get_ptr(plhs(1))

         DO i= 1,n+1
           jc(i) = matrix_out%ptr(i)-1
         END DO

        CALL MATLAB_copy_from_ptr(jc, jc_pr, n+1)

        ALLOCATE ( ir( matrix_out%ne ), STAT = err)
        IF (err .ne. 0) THEN
          CALL MATLAB_error( " Insufficient memory " )
        END IF

          DO i=1,matrix_out%ne
             ir(i) = matrix_out%rowl(i)-1
          END DO

          CALL MATLAB_copy_from_ptr(ir, ir_pr, matrix_out%ne)
          CALL MATLAB_copy_from_ptr(matrix_out%val, ptr_pr, matrix_out%ne)

! Free workspace

          IF ( ALLOCATED( ir ) ) DEALLOCATE( ir, STAT = err)
          IF ( ALLOCATED( jc ) ) DEALLOCATE( jc, STAT = err)
```