How to Interface Fortran with Matlab

Claudia Sagastizábal, Guillaume Vige

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

How to Interface Fortran with Matlab

Claudia SAGASTIZÁBAL
Guillaume VIGÉ

N° 176
Juillet 1995

PROGRAMME 5

How to interface Fortran with Matlab

Claudia Sagastizábal , Guillaume Vigé

Programme 5 — Traitement du signal, automatique et productique
Projet Promath

Rapport technique n ° 0176 — Juillet 1995 — 15 pages

Abstract: We describe the general procedure for interfacing Fortran routines with Matlab. We explain how to write a mex-file and the associated gateway function. In particular, each different type of argument is considered in detail. We finish with an illustrative example.

Key-words: Fortran MEX-Files, Matlab, Gateway functions

(Résumé : tsvp)

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau, Rocquencourt, BP 105, 78153 LE CHESNAY Cedex (France)
Téléphone : (33 1) 39 63 55 11 – Télécopie : (33 1) 39 63 53 30

Comment faire une interface Fortran-Matlab

Résumé : Nous décrivons le procédé générale pour faire des interfaces Fortran-Matlab. Nous expliquons comment écrire un fichier mex et la fonction passerelle associée. Nous considérons en profondeur les differents types d'argument. Nous finissons avec un exemple illustratif.

1 Introduction

When interfacing Fortran routines with Matlab, it is important to properly transmit the data to avoid compatibility problems. Indeed, when we wrote our interface, most of the bugs we had to dealt with were originated by such incompatible structures. In this report we give some hints to avoid or to quickly identify these kind of errors. This apparently trivial goal is not that easy to achieve: many times we have been confronted to obscure messages like:

MATLAB segmentation violation detected.
Please report this problem to MathWorks if you can replicate it.
Send mail to bugs@mathworks.com or FAX to 508-653-2997.
Try to save your workspace and quit.
Error in ==> .... On line nn ==> ....

or other equally vague diagnostics. Although these problems only appeared during execution time, what they really meant was that some transmission error occured in the interface.

An interface can be more or less complicated; its complexity generally falls in one of two distinct levels of difficulty. A first, simpler one (the only one properly explained in the Matlab manual) is when the Fortran code is a main program or a routine only using fixed-length arrays. A different matter, much more complicated, is when the Fortran code uses variable-length arrays. In this case, it is very important to clearly distinguish inputs, outputs and inputs/outputs in the arguments list, as well as their respective input- and output-lengths.

This report is organized as follows. We start with a general description of the interface procedure, then, in § 3 we detail the structure of gateway functions. After a summary section, we finish with an example. In the appendix we give a fast recall of the External Interface Library as well as some useful routines. For more information on their syntax, we refer to the Matlab External Interface Guide.

2 External Interfaces

2.1 What is a mex-file?

A mex-file is a matlab external interface to call a Fortran code from Matlab as if it was a built-in function. In other words, a mex-file is a subroutine dynamically linked that can be automatically loaded and executed from Matlab. It is created from the compilation of a gateway function, the Fortran code and all its depending subroutines and functions.

For example, consider a Fortran program "main.f" which calls two subroutines "subr1.f" and "subr2.f". Suppose you have already written a suitable gateway function: "mainmex.f" (see § 3 for details on how to write this function). To compile and link these files, you should type

```
fmex mainmex.f main.o subr1.o subr2.o        (1)
```

You can include as well libraries ("*.a") or Fortran source codes ("*.f"). The fmex command carries out all the necessary steps to create a mex-file "mainmex.mexaxp", executable from Matlab. As for the mex extension, `mexaxp` identifies the platform DEC-OSF, `mex4` corresponds to a Sun-4, an so on. Once in Matlab, the mex-file is called without any extension, "mainmex" in the example.

The mex-file only interfaces Fortran *subprograms*, so main programs should first be transformed into subroutines. Also, every Fortran instruction `common` should be eliminated before writing the interface.

2.2 Calling a mex-file

Inside Matlab, the mex-file is invoked with the general syntax for built-in functions:

$$[arg_{out_1}, ..., arg_{out_{nlhs}}] = name(arg_{inp_1}, ..., arg_{inp_{nrhs}}),$$

that is, *name* is a function with *nrhs* input arguments (data) and *nlhs* outputs (results).

For the example (1) above, "name" is precisely `mainmex`. If you enter this single-line command directly in Matlab, it will be processed immediatly. This means that "name" will be compiled and placed into memory every time you invoke it. If you wantto have it available for subsequent use without recompilation, you need to write and "M-file". M-files are ASCII text-files containing a sequence of normal Matlab statements and whose extension is always `.m` (mainmex.m in the example). For the M-file to be considered a built-in function, the first line must declare the function name as well as the input and output arguments:

$$\texttt{function}[\texttt{arg}_{\texttt{out}_1}, ..., \texttt{arg}_{\texttt{out}_{\texttt{nlhs}}}] = \texttt{name}(\texttt{arg}_{\texttt{inp}_1}, ..., \texttt{arg}_{\texttt{inp}_{\texttt{nrhs}}}).$$

In § 5.1 we give an example of M-function.

3 Gateway functions

Matlab is written in C and it uses extensively one of its data types: pointers. Pointers are integer variables that refer to an object. In Fortran they are not used, because the transmission of arrays to a subroutine is done by address. Thus, instead of pointing to the object, the object itself is transmitted. A gateway function is a knack that allows Fortran compilers to treat addresses as pointers. It uses a set of routines in the External Interface Library (a list is included in the Appendix) and is written in Fortran according to a fixed pattern. This pattern has four main sections:

1. A heading with all the declarations;
2. the transformation of Matlab arguments into Fortran variables;
3. the call to the Fortran subroutine;
4. the transformation of Fortran outputs into Matlab objects.

3.1 Heading

3.1.1 Structure

Every gateway routine must start with

```
#include <fintrf.h>
subroutine mexFunction(nlhs, plhs, nrhs, prhs)
pointer plhs(*), prhs(*)
integer nlhs, nrhs
```

These names follow the mnemonics **right**hand side for input arguments and **left**hand side for output arguments. Their number is respectively "nrhs" and "nlhs"; accordingly, "prhs" and "plhs" are arrays of pointers to the corresponding parameters.

After this fixed set of lines, follows the declaration of variables, pointers and functions to be used. It is advisable to declare everything, even objects already responding to Fortran pre-definitions.

3.1.2 Error detection

An optional section to check if the arguments are properly transmitted can be included. Error messages can be displayed using the `mexErrMsgTxt` function. To validate the number of arguments, for example, write

```
if (nrhs.ne.3) then
  call mexErrMsgTxt('Three input arguments required')
else if (nlhs.ne.4) then
  call mexErrMsgTxt('Four output arguments required')
endif
```

3.2 Conversion Matlab⇒Fortran

We have already said that Matlab input arguments must be turned into Fortran-like variables before calling the Fortran subroutine and that the inverse process must be done for output arguments before returning to Matlab. These conversions vary with the type of the arguments. We start first with the righthand side arguments

3.2.1 Real RHS

Take the jth rhs Matlab argument, call it "argj"; and suppose it is double precision in the calling list of the Fortran routine. The corresponding pointer "iargj" must first be declared as a *pointer* variable, and then be defined using the `mxGetPr` function:

```
iargj = mxGetPr(prhs(j))
```

We also need to identify the address to be transmitted to the Fortran code. However, there is an extension to standard Fortran-77 which permits to transfer the value of the pointer instead. It is the double precision `%VAL` function, supported by many Fortran compilers (DEC-OSF's compiler included) and it has the pointer iargj as unique argument.

The function mxGetPr is of integer type; by the way, in the Matlab manual, page 2-57, there is an error in the *Fortran Synopsis* describing this function. The arguments pm, pr, pi must be declared as pointers and not as integer*4 elements, otherwise there is an error.

3.2.2 Integer RHS

When `argj` happens to be an integer argument for the Fortran routine, it is necessary to transform %VAL(iargj) into an integer variable before using it in the Fortran list. This can be done with a Fortran routine `dblint`, which Ulf Brännlund, from the *KTH*, Stockholm, kindkly sent to us (in § 6 we include a listing). Assume argj is an array of length `lj`, then write

```
call dblint(lj,%VAL(iargj),%VAL(iargj))
```

to obtain an integer %VAL(iargj). [1]

Beware of not mixing up three different concepts related to argj:

- Its pointer `iargj`, referring to the Matlab object.
- Its address, necessary for the Fortran call, but "skipped" by the somewhat equivalent construct `%VAL(iargj)`.
- Its actual value, which can be recovered by a call to an appropriate `mxcopyPtrto...` subroutine. For an integer argj this comes to write the sentence

```
call mxcopyPtrtoInteger4(iargj,argj,lj).
```

3.2.3 Function names. Strings RHS

We have not been able to transmit function names which are `external` to the Fortran routine. We decided to use abstract names associated to real functions in the compilation time. When `argj` is a character string, the integer function `mxGetString` catches `lchar` characters in argj and assigns them to the character variable `char`: `mxGetString(prhs(j),char, lchar)`

3.2.4 Error detection

The type and size of the rhs arguments can also be checked. We strongly recommend to do it: properly fixing this point can be crucial for debugging. For instance, to inquire if the jth-element in prhs is in double precision, use the function `mxIsDouble`: it returns 1 if true. Since we always had either double precision or integer variables, we actually used this function to detect if an argument was integer:

```
if(mxIsDouble(iargj).eq.1) call mexErrMsgTxt('argj not integer')
```

The integer functions `mxGetM` and `mxGetN` give the row and column dimensions of a Matlab argument. They can be used to check the arguments size:

```
if(max(mxGetM(iargj),mxGetN(iargj)).ne.lj) then
  call mexErrMsgTxt('Improper dimensions')
endif
```

In the example, we are checking if the array is lj $\times$ k or k$\times$ lj, for k$\leq$lj.

[1] As an alternative, we could declare first `argj` as integer, then call `dblint`: `call dblint(lj,%VAL(iargj),argj)`, and finally use `argj` alone in the Fortran list.

3.2.5 Lefthand side arguments. Generalities

Since Matlab has an object-oriented approach, it handles all the internal data structures using objects of the same type: a matrix. A matrix object may represent scalars, vectors or text as well as matrices themselves. This gives Matlab a lot of flexibility for manipulating its elements. It may be convenient, however, to keep some coherence when initializing the data to be used by the mex-file; for instance, all vectors should be 1-column or 1-row matrices. The treatment of lhs arguments depends essentially on whether the argument is only O or both I/O.

3.2.6 Pure Output LHS

The integer functions `mxCreateFull` and `mxCreateString` create matrices (numeric and string respectively) of required sizes in which results from the lhs list are returned. Consider for instance a numeric `argk`, the kth-element in the lhs list, and assume its length `m`×`n` is available. Its space allocation can be done by typing

```
plhs(k)=mxCreateFull(m,n,ComplexFlag)
```

where `ComplexFlag` is set to 0 if it is a real variable. Then the pointer `iargk` can be recovered as usual:

```
iargk=mxGetPr(plhs(k))
```

3.2.7 Input-Output LHS

If the I/O `argk` has fixed length, there is no need to allocate space with one of the `mxCreate...` functions. Actually, if argk is the kth- lhs element as well as the jth-righthand side, you can simply type

```
plhs(k)=prhs(j)
```

When `argk` has an output-length bigger than its input-length, an appropriate matrix must be created. Suppose argk is a vector of dimension `m`×`1`, with `m` an output itself, say the kmth- element. Then the following commands should be done

```
plhs(km) = mxCreateFull(1,1,0)
m=nint(mxGetScalar(plhs(km)))
plhs(k)=mxCreateFull(m,1,0)
```

Clearly, the alternative previously described in footnote can also be used.

3.2.8 Error detection

Similarly to what has been done for righthand sides, but putting special attention to checking different I/O lengths.

3.3 Calling the Fortran subroutine

This step consists in a sentence of the type

```
call name(fort-arg1, fort-arg2,...)
```

As it has already been explained, "fort-argj" is generally replaced by "%val(iargj)", the exception being the alternative mentioned in footnote.

3.4 Conversion Fortran⇒Matlab

After the Fortran subprogram has been called, its outputs need to be adapted to Matlab standards. Moreover, every integer variable, say `iarg`, must be transformed into real, independently of its status (input, output, input/output). This is done with a call to `intdbl`:

```
call intdbl(l,iarg,arg),
```

where `l` is the length of the integer variable `iarg` to be converted into the real `arg`.

If the alternative described in footnote has been used, this step of reconversion can be omitted.

4 Summary

1. Analyze the structure of your Fortran code and its dependencies (subroutines and functions), to decide at which level you will need the interface. If it is a main program, then rewrite it as a Fortran subroutine. Sometimes it may be more convenient to rewrite the main Fortran program in Matlab and interface the subroutines originally called by this program.

2. Eliminate commons by adding I/O variables to the calling list. Recompile to verify that you have not added any bug during this process.

3. Examine closely each variable of the argument list to identify its

   (a) status: is it an Input, Output or Input/Output variable?

   (b) type: integer, real, character?

   (c) size: fixed or variable length? If I/O, identify input and output dimensions.

4. Write the Gateway file following the instructions in § 3.

5. Compile, debug and enjoy Matlab!

5 Example

5.1 M-file

We give here the M-file we use for invoking inside Matlab the Fortran subroutine "fqprox":

```
function [ahat,delta,solut,alamqp,kqp,jsqp,workqp] = fqprox(modeqp,...
t,bundle,weight,imp,io,ahat,gram,alamqp,kqp,jsqp,workqp,memax)
[nbun,n] = size(bundle);
bundle = reshape(bundle,nbun*n,1);
z=[];
for j=1:nbun; z=[z gram(j:nbun,j)]; end;
gram=z;
[ahat,delta,solut,alamqp,kqp,jsqp,workqp] = fqproxmex(modeqp,...
t,bundle,weight,imp,io,ahat,gram,alamqp,kqp,jsqp,workqp,memax);
end
```

With this M-file we define a function `fqprox`, with 13 arguments in input and 7 in output; note that some of them are I/O. In lines 3 to 7 we convert the matrices `bundle` and `gram`, of sizes `nbun`×`n` and `nbun`×`nbun` respectively, into vectors. This is done because the Fortran code works with vectors; in particular, since the matrix gram is symmetric, we only store its lower triangle. Finally, there is the call to the mex-file.

5.2 Gateway function

Heading

```
      #include <fintrf.h>
c     Gateway function fqprox, to be run from the matlab file fqprox.m
      subroutine mexFunction(nlhs, plhs, nrhs, prhs)
      pointer plhs(*), prhs(*)
      integer nlhs, nrhs
      pointer mxCreateFull, mxGetPr, mxGetPi, mexGetMatrixPtr,
      & mxCalloc, mexPutMatrix,mexGetMatrix,mxGetM,mxGetN
      parameter (dblsiz=8, intsiz=4)
      double precision mxGetScalar
      pointer ibundle,iweight,isolut,igram,ialamqp,ijsqp,iworkqp
      pointer ikqp,iimp,iio,it,iahat,imodeqp,idelta,inbun
      integer nbun,n,n1,memax
      external fqprox
```

Error detection

```
      if (nrhs.ne.20) then
      call mexErrMsgTxt('20 input arguments required in fqproxmex')
      endif
      if (nlhs.ne.14) then
      call mexErrMsgTxt('14 ouput arguments required in fqproxmex')
      endif
```

CONVERSION Matlab ⇒ Fortran - RHS

```
      imodeqp = mxGetPr(prhs(1))
      call dblint(1,%val(imodeqp),%val(imodeqp))
      it = mxGetPr(prhs(2))
      ibundle = mxGetPr(prhs(3))
      iweight = mxGetPr(prhs(4))
      iimp = mxGetPr(prhs(5))
      call dblint(1,%val(iimp),%val(iimp))
      iio = mxGetPr(prhs(6))
      call dblint(1,%val(iio),%val(iio))
      iahat = mxGetPr(prhs(7))
      igram = mxGetPr(prhs(8))
      ialamqp = mxGetPr(prhs(9))
```

```
ikqp = mxGetPr(prhs(10))
call dblint(1,%val(ikqp),%val(ikqp))
ijsqp = mxGetPr(prhs(11))
memax=mxGetM(prhs(11))
call dblint(memax,%val(ijsqp),%val(ijsqp))
iworkqp = mxGetPr(prhs(12))
istopqp = mxGetPr(prhs(13))
iconqp = mxGetPr(prhs(14))
iitmqp = mxGetPr(prhs(15))
call dblint(1,%val(iitmqp),%val(iitmqp))
ikmaxqp = mxGetPr(prhs(16))
call dblint(1,%val(ikmaxqp),%val(ikmaxqp))
ilenqp = mxGetPr(prhs(17))
call dblint(1,%val(ilenqp),%val(ilenqp))
iidmqp = mxGetPr(prhs(18))
call dblint(1,%val(iidmqp),%val(iidmqp))
ilworqp = mxGetPr(prhs(19))
call dblint(1,%val(ilworqp),%val(ilworqp))
inbun = mxGetPr(prhs(20))
call dblint(1,%val(inbun),nbun)
n1 = max(mxGetM(prhs(3)),mxGetN(prhs(3)))
n=n1/nbun
```

CONVERSION Matlab ⇒ Fortran - LHS

```
c  Fixed-lenght I/O
   plhs(1) = prhs(7)
c  Pure Output LHS
   plhs(2) = mxCreateFull(1,1,0)
   idelta = mxGetPr(plhs(2))
   plhs(3) = mxCreateFull(n,1,0)
   isolut = mxGetPr(plhs(3))
   plhs(4) = prhs(9)
   plhs(5) = prhs(10)
   plhs(6) = prhs(11)
   plhs(7) = prhs(12)
```

```
      plhs(8) = prhs(13)
      plhs(9) = prhs(14)
      plhs(10) = prhs(15)
      plhs(11) = prhs(16)
      plhs(12) = prhs(17)
      plhs(13) = prhs(18)
      plhs(14) = prhs(19)
Call to the Fortran subroutine
      call fqprox(%val(imodeqp),nbun,n,%val(it),%val(ibundle),
     & %val(iweight), %val(iimp),%val(iio),%val(iahat),
     & %val(idelta),%val(isolut),%val(igram),%val(ialamqp),
     & %val(ikqp),%val(ijsqp),%val(iworkqp),
     & %val(istopqp), %val(iconqp), %val(iitmqp), %val(ikmaxqp),
     & %val(ilenqp), %val(iidmqp), %val(ilworqp))
CONVERSION Fortran ⇒Matlab
      call intdbl(1,%val(imodeqp),%val(imodeqp))
      call intdbl(1,%val(ikqp),%val(ikqp))
      call intdbl(memax,%val(ijsqp),%val(ijsqp))
      call intdbl(1,%val(iitmqp),%val(iitmqp))
      call intdbl(1,%val(ikmaxqp),%val(ikmaxqp))
      call intdbl(1,%val(ilenqp),%val(ilenqp))
      call intdbl(1,%val(iidmqp),%val(iidmqp))
      call intdbl(1,%val(ilworqp),%val(ilworqp))
      call intdbl(1,%val(iimp),%val(iimp))
      call intdbl(1,%val(iio),%val(iio))
      return
      end
```

6 Appendix

6.1 External Interface Library

We just give here a list of all the available functions, for a complete description of each routine, see the Matlab External Interface Guide, section 2.

6.1.1 Matrix access routines

- `mxCreate/Full/Sparse, mxFreeMatrix` (here we have shortened two commands: mxCreateFull and mxCreateSparse).
- `mxCreateString, mxGetString`
- `mxGet/Ir/Jc, mxSet/Ir/Jc`
- `mxGet/M/N, mxSet/M/N`
- `mxGetName, mxSetName`
- `mxGetNzmax, mxSetNzmax`
- `mxGet/Pi/Pr, mxSet/Pi/Pr`
- `mxGetScalar`
- `mxIs/Complex/Double/Full/Sparse/Numeric/String`

6.1.2 Memory allocation routines

- `mxCalloc, mxFree`

6.1.3 Fortran conversion routines

- `mxCopyPtrto/Character/Integer4/Real8/Complex16`, and the respective inverses:
- `mxCopy/Character/Integer4/Real8/Complex16/ToPtr`

6.1.4 Mex-file routines

- `mexAtExit`
- `mexCallMATLAB`
- `mexErrMsgTxt`
- `mexEvalString`
- `mexFunction`

- `mex/Get/Put/Full`
- `mex/Get/Put/Matrix`
- `mexGetMatrixPtr`
- `mexPrintf`
- `mexSetTrapFlag`

6.2 Swedish routines

```
subroutine dblint( n, dblev, intv )
* In order to transform a matlab double precision vector to a Fortran
* integer vector, call this routine with intv and dblev equal.
* Written 21-June-1989 by Anders Forsgren (andersf@math.kth.se)
integer n, intv(n)
double precision dblev(n)
integer i
do 2000 i = 1, n
intv(i) = nint(dblev(i))
2000 continue
return
end

subroutine intdbl( n, intv, dblev )
* In order to retransform an integer vector to a double precision
* vector, call this routine with intv and dblev equal.
* Written 21-June-1989 by Anders Forsgren
integer n, intv(n)
double precision dblev(n)
integer i
do 2000 i = n, 1, -1
dblev(i) = dble(intv(i))
2000 continue
return
end
```

Unité de recherche INRIA Rocquencourt
Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

Unité de recherche INRIA Lorraine - Technopôle de Nancy Brabois - Campus scientifique
615, rue du Jardin Botanique - B.P. 101 - 54602 Villers lès Nancy Cedex (France)
Unité de recherche INRIA Rennes - IRISA, Campus universitaire de Beaulieu - 35042 Rennes Cedex (France)
Unité de recherche INRIA Rhône-Alpes 46, avenue Félix Viallet - 38031 Grenoble Cedex 1 (France)
Unité de recherche INRIA Sophia Antipolis - 2004, route des Lucioles - B.P. 93 - 06902 Sophia Antipolis Cedex (France)

Éditeur
INRIA - Domaine de Voluceau - Rocquencourt - B.P. 105 - 78153 Le Chesnay Cedex (France)

ISSN 0249 - 0803