# CALLING FORTRAN SUBROUTINES FROM MATLAB

**FRITZ KEINERT**  
DEPARTMENT OF MATHEMATICS  
IOWA STATE UNIVERSITY  
AMES, IA 50011  
KEINERT@IASTATE.EDU

## Contents

1. Introduction 2  
2. Calling Fortran from Matlab 3  
   - Case Study 1: DGAMMA 12  
   - Case Study 2: CGAMMA 16  
   - Case Study 3: DGESVD 19  
   - Case Study 4: DQAGS 24

Copyright (C) 1997 by Fritz Keinert (keinert@iastate.edu). This document may be freely copied and distributed as long as this copyright notice is preserved. An electronic copy can be found through the author’s home page [http://www.math.iastate.edu/keinert](http://www.math.iastate.edu/keinert).
1. INTRODUCTION

Virtually all scientific computation makes use of standard ("canned") subroutines at some point. Matlab has a lot of algorithms built in, and continues to add more with each release, but there are still situations where one would like to extend its capabilities.

In this document, I have written up some general guidelines and several case studies for interfacing a Fortran library routine to Matlab version 4 or 5. Another document Calling Fortran Subroutines from Fortran, C, and C++ contains similar guidelines and examples for linking Fortran subroutines to main programs written in Fortran, C, or C++. It is assumed that you have read this other document, or that you are already familiar with calling external Fortran subroutines from a C main program.

This report is intended as a resource for research activities at Iowa State, and as a supplementary reference for numerical analysis courses.

The case studies were developed and tested on the three different hardware platforms that currently make up Project Vincent. Project Vincent is a large network of centrally administrated workstations at Iowa State University, patterned after Project Athena at MIT. With minor changes, the examples should carry over to settings outside of Project Vincent.

I have kept things simple rather than comprehensive. Some of my assertions may not even be quite correct, when I felt that a complete description would just confuse beginners. Not all aspects of the Matlab external interface are covered, only the passing of the type of arguments one would usually need for a scientific subroutine (i.e. numbers, simple strings, and subroutine addresses).

I am discussing the interface to Fortran subroutines only, but calling C routines is actually easier, and is left as an exercise for the reader.

The programming examples follow my particular tastes, which may or may not suit you. Feel free to adapt the examples to your style. Particular preferences of mine include

- I strongly believe in declaring all variables in Fortran, and in prototyping all external subroutines in C. It is not strictly necessary, but it lets the compiler spot errors.
- I use C to interface Fortran and Matlab. Part of the reason is that my Fortran is getting a little rusty, but the main reason is that the interface requires a lot of pointers, and the creation and destruction of temporary storage, and Fortran is just not set up for that. It can be done, with a lot of extra subroutine calls to manipulate the pointers, but it is much more natural to use C.

If you want to reproduce the case studies, you will need access to CMLIB, LAPACK and Matlab. On Project Vincent, the following statements should be in your .startup.tty file:

```plaintext
add math
add lapack
add matlab
```

Documentation for CMLIB and LAPACK can be found through http://www.math.iastate.edu/software.html.

CMLIB is a collection of many public domain sublibraries from Netlib, including LINPACK and EISPACK. Our version was compiled by Ronald F. Boisvert, Sally E. Howe and David K. Kahaner in April 1988. If you don’t have CMLIB, you can get the necessary pieces from Netlib, for example at http://netlib.bell-labs.com/netlib/master/readme.html.

LAPACK is a linear algebra package intended to be efficient on many types of computer architecture. It can also be found on Netlib.

Matlab is a trademark of MathWorks Inc.; you can contact the company at http://www.mathworks.com for more information.

The case studies for calling Fortran from Matlab are

- DGAMMA (double precision gamma function) from FNLIB, which is part of CMLIB. This is the simplest case: a function with one input argument, one return value.
- CGAMMA (complex gamma function). Same as DGAMMA, but for complex numbers.
Calling Fortran From Matlab

1. Calling Fortran From Matlab

This section gives a general overview of what is involved in calling a Fortran subroutine from inside Matlab. It is assumed that you fully understand how to call a Fortran subroutine from a main program written in C. You can find some information on that in my other report Calling Fortran Subroutines from Fortran, C, and C++.

I only cover the more basic aspects of interfacing external subroutines to Matlab here. This means that the only data types that get passed back and forth are double precision full matrices, strings, and subroutine names. I am skipping everything related to sparse matrices, symbolic matrices, or any of the new data types in Matlab 5.

If you want to reproduce the case studies in this section, you will need access to CMLIB, LAPACK and Matlab. See the introduction for details.

Matlab is a great program for experimenting with numerical algorithms, and there are expansion toolboxes for many things. It still happens sometimes that there is a Fortran subroutine available for a certain application, but no Matlab routine. This actually is the case for one of my case studies, the complex gamma function CGAMMA, which I could only find in a single precision version; in my opinion, this proves that Cleve Moler wasn’t able to find a public domain double precision complex gamma function, either.

Another possible reason to use a Fortran subroutine is speed. Manipulating scalars in Matlab involves a lot of overhead, and Matlab subroutines sometimes run significantly slower than Fortran code. You may not be able to use Matlab on a very large problem.

As a solution to both of these problems, Matlab 4 comes with two mex compilers (Matlab EXtension Language), one for Fortran subroutines (fmex) and one for C (cmex). Matlab 5 comes with a single compiler that can handle both Fortran and C (mex). Creating such an interface is not trivial, however. Calling Fortran from Matlab is an order of magnitude harder than calling Fortran from C. You probably also want a copy of the Matlab External Interface Guide handy when you do this.

Special Notes about the Project Vincent versions of Matlab:

1. The version 4 mex compiler on DEC Alphas is brain damaged. Some programs work, others do not. Even some subroutines that were compiled by MathWorks themselves do not work (some of the ODE solvers, specifically). When I reported this to MathWorks, the answer was: “Yes, we know about that. The solution is to use the version 5 mex compiler, not version 4.”

   The version 4 compiler on DECStations and SGIs works fine.

2. Version 5 is currently only available as a beta test version for DEC Alphas. You have to do add matlab5beta, and then invoke Matlab and mex as matlab5 and mex5. Presumably, these extra 5s at the end will go away once this version becomes the standard.

   My personal preference is to write the interface between Fortran and Matlab in C. The interface involves a lot of pointer manipulation, which Fortran cannot do, so you have to call a lot of auxiliary subroutines. It is possible to do the interface entirely in Fortran, but you will have to read the Matlab manual yourself.

   If your only concern is execution speed, you may want to investigate the Matlab compiler offered by MathWorks; it takes Matlab code, converts it to mex and compiles it. You can run this compiler from inside Matlab on Alphas and SGIs (not on DECStations); type help mcc to get started. I only played with it briefly once, and it did not do much in the way of speedup for my particular example, but you might give it a try.

The basic data type in Matlab is a two-dimensional double precision matrix.
In Matlab 4, that is actually all there is (except for symbolic arrays, about which I don’t know anything). A vector is an \((n \times 1)\) or \((1 \times n)\) matrix, a scalar is a \((1 \times 1)\) matrix, a string is a vector of double precision numbers which happen to be ASCII codes.

In Matlab 5, there is a bewildering variety of new data types: matrices can have more dimensions than 2, they can be single or double precision, one, two, or four-byte integers, logicals, strings, and so on. There are also structures and cell arrays.

For the purposes of this document, I will stick with double precision, two-dimensional full (i.e. non-sparse) matrices, and strings. For everything else, you will have to consult the *Matlab External Interface Guide*.

The internal implementation in Matlab 4 is similar to this:

```c
typedef struct {
    char name[mxMAXNAM]; /* name */
    int type; /* type (real, complex, string, sparse or full) */
    int m; /* row dimension */
    int n; /* column dimension */
    double *pr; /* pointer to real part of data */
    double *pi; /* pointer to imag part of data */
} Matrix;
```

To make future upgrades easier, it is recommended not to rely on any specifics of the `Matrix` data type. If `A` is a pointer to a matrix, you should not use `m = A->m` to access the first dimension, but `m = mxGetM(A)`, and similarly for all the other fields. Mathworks promises in the Matlab documentation that the names of these access routines will not change in future versions.

I was using a simpler version of the same setup in case study *DGESVD Revisited* in section *Calling Fortran from C* in my other report *Calling Fortran Subroutines from Fortran, C, and C++*. You may want to go back and look at that again.

Matlab 5 has a more complicated data type, but for our purposes we can think of it the same way. The only difference in practice is that this structure is now called `mxArray` instead of `Matrix`.

---

### Matlab matrices and Fortran arrays are not the same.

In all code fragments and case studies, pay attention to what is a Matlab matrix and what is a Fortran array.

A Fortran array is just a bunch of numbers. It is up to the programmer to keep track of the dimensions, avoid stepping out of bounds, etc. A C array has some slightly different conventions about numbering and ordering, but is basically the same as a Fortran array.

A Matlab matrix is a structure (see above) which contains a Fortran array, but also information about it (dimensions, type, name of the array, etc.).

In a mex program, all input and output parameters to your mex routine need to be Matlab matrices, since they get passed back and forth to Matlab; everything else can be Fortran arrays, which are easier to manipulate.

Note that each of these data types has its own set of routines to manipulate it.

---

### Every external subroutine interface has the same basic structure.

If you want your subroutine to be called `sub` inside Matlab, you need to put it into a file `sub.c`. The actual routine in that file is called `mexFunction` (same subroutine name in every case). This can be a stand-alone subroutine, or in our case it will call some Fortran routine from a library.

The file `sub.c` gets compiled into `sub.mexds`, `sub.mexxp` or `sub.mexsg`, depending on the type of machine you are using, and Matlab can load it from there.

You should also prepare a file `sub.m` which contains the documentation for `sub`. When you want to execute `sub` from Matlab, `sub.mex*` takes precedence over `sub.m`, but when you type `help sub`, Matlab will list the comments from `sub.m`.

In Matlab 4, the interface has the following structure:
void mexFunction(int nargout, Matrix *argout[], int nargin, Matrix *argin[])

Here, nargin is the number of input arguments, same as the built-in variable nargin accessible in Matlab subroutines. argin is an array of length nargin of pointers to the actual arguments. Likewise, nargout is the number of output arguments, andargout is an array of pointers to the output arguments.

In Matlab 5, the Matrix data type has been renamed to mxArray, and the input arguments have been changed to constant pointers, the rest is the same:

void mexFunction(int nargout, mxArray *argout[], int nargin, const mxArray *argin[])

Either way, it makes life easier if you assign symbolic names to the input and output arguments, using a
#define compiler directive. For example, if you are visualizing an interface like

\[ [r,s] = f(a,b,c) \]

you put in code like this:

#define a argin[0]
#define b argin[1]
#define c argin[2]
#define r argout[0]
#define s argout[1]

In Matlab 4, use fmex to compile your routine, not cmex

There are two steps involved in producing a working program: compiling and linking. Usually, the compiler will invoke the linker automatically and specify the location of the system libraries the linker needs.

For whatever internal reasons, fmex knows where the C libraries are, but cmex does not know where the Fortran libraries are. Thus, if there are any Fortran routines involved, you should use fmex for the linking, even if the main interface routine is written in C.

In Matlab 5, there is only a single compiler called mex, and this problem has disappeared.

The Conversion Process

Interfacing a library routine to Matlab is a two-stage process.

Step 1: Design The Interface

Classify the arguments of the Fortran routine into three types:

- Input arguments (further broken down into required and optional arguments)
- Output arguments (required and optional)
- Unnecessary arguments

"Unnecessary arguments" are those that are required by Fortran, but not by Matlab. This category includes array dimensions and working storage, maybe some other parameters for which you always want to assign default values.

Required input and output arguments should come first in the argument list, the optional arguments later. An optional argument is one for which the interface routine assigns a default value unless the user provides a value. For example, this could be a requested accuracy for the solution.

When you are done with the interface design step, you will have decided the look of the interface:

\[
[\text{out1, out2, \ldots}] = \text{sub(in1, in2, \ldots)}
\]

Fortran routines often use some parameters for both input and output, i.e. the output overwrites the input. In such a situation, you should make a copy of the input argument, otherwise you will overwrite some variable in your workspace. The copy can be passed to Fortran, where it gets overwritten with the result, and can then be passed back to Matlab as one of the return arguments.

For example, assume the second input argument is a matrix called A, and the first output argument is a matrix called X. X is calculated from A by a Fortran routine which writes X over the top of A. You can achieve this with
#define A argin[1]
#define X argout[0]
double *Apr, *Xpr;
int m, n;
/* create an empty Matlab array X the same size as A */
m = mxGetM(A);
n = mxGetN(A);
X = mxCreateFull(m,n,REAL);
/* copy over the data */
Apr = mxGetPr(A);
Xpr = mxGetPr(X);
for (i = 0; i < m*n; i++) Xpr[i] = Apr[i];
/* use it */
fortran_routine(Xpr,...)
/* no further copying necessary */

Step 2: Write The Code

I usually follow the following template:

- **Check and Process Input Parameters**
  - Check number and type of input parameters
  - Provide defaults for missing optional arguments
  - Convert the data to Fortran form if necessary.

Here are some typical code fragments:

Converting a real array from Matlab form to Fortran form:

```c
#define A argin[0]
double *Apr;
Apr = mexGetPr(A);
argin[0] is the first input argument. We give it an alias of A to make the program more readable. A is a pointer to a Matlab matrix. Apr points to the data area of A and can be treated like a Fortran array.
```

Converting a complex array from Matlab form to Fortran form:

```c
#define A argin[0]
typedef struct { double re, im;} double_complex;
int Am, An;
double *Apr, *Api;
double_complex *Apri;
Am = mexGetM(A);
An = mexGetN(A);
Apr = mexGetPr(A);
Api = mexGetPi(A);
Apri = mxCalloc(Am * An, sizeof(double_complex));
for (i = 0; i < Am*An; i++) {
    Apri[i].re = Apr[i];
    Apri[i].im = Api[i];
    Api++; Apr++; Apri++;
}
```

This is sort of messy, unfortunately. A complex Fortran array alternates the real and imaginary parts of the entries in a single array. A complex Matlab matrix has all real parts in one array (Apr in the sample code), all imaginary parts in another array (Api). You need to create a new Fortran array and move the data there, in proper order.

Converting a string from Matlab form to Fortran form:
#define fname argin[0]
char name[maxlen+1];
mxGetString(fname,name,maxlen);

• **Check and Create Output Parameters**
  
  – Check number of output parameters
  – Create space to hold the output if necessary

You have to explicitly create all output parameters unless you are re-using some input parameters. I tend to delay creating space as long as possible, in case an error occurs before the space is really needed.

Output parameters are Matlab matrix objects. Here is the sample creation and destruction of a 4 × 3 real Matlab matrix in Matlab 4:

Matrix *X;
X = mxCreateFull(4,3,REAL);
MxFreeMatrix(X);

Here is the corresponding code in Matlab 5:

mxArray *X:
X = mxCreateDoubleMatrix(4,3,mxREAL);
MxDestroyArray(X);

Most of the time, you don't have to explicitly destroy matrices. Matlab takes care of that automatically when your subroutine terminates.

• **Create Temporary Variables**

  – Extract array dimensions from arguments
  – Create working storage

The dimension of input arguments can be found with `mxGetM`, `mxGetN`. For example, suppose the Fortran routine is called as `sub(A,m,n,...)`, where `m` and `n` are the dimensions of `A`. In `mex`, this looks like this:

```c
int Am, An;
double *Apr;
Am = mexGetM(A);
An = mexGetN(A);
Apr = mexGetPr(A);
sub_(Apr, &Am, &An, ...)
```

Temporary variables just need to be Fortran arrays, not Matlab matrices. Here is the sample creation of a working array of length 100:

```c
double *work;
work = mxCalloc(100,sizeof(double));
```

Matlab will take care of recovering the storage automatically. If you want to free the array explicitly, use `mxFree`.

• **Invoke External Routine**

Don't forget to consider what happens if your Matlab routine is called with matrix arguments! Maybe you need to call the external Fortran routine many times, once for each matrix entry. Look at case study DGAMMA for an example.

• **Check Whether Any Errors Occurred**

Most routines have a return parameter, usually called `info` or `ier`, which contains an error code. I recommend turning off all error reporting in the subroutine itself, and translating the error code into a readable message which gets printed inside Matlab.

This is especially important when using the XERROR interface from CMLIB. Unless you intercept the calls to XERROR by writing your own, Matlab itself will terminate when a fatal error occurs.

• **Copy Output**
- Create output variables if not done earlier
- Copy output from Fortran form back to Matlab form
Table of Auxiliary Routines

This table contains the most frequently used auxiliary routines. For a complete list and more information, consult the *Matlab External Interface Guide*.

Remarks:

1. In Matlab 5, if the array has dimension higher than 2, $n$ is the product of the second and higher dimensions. Thus, the number of elements in the matrix is always $m \times n$. To access information about higher dimensions, use

   ```c
   int d, *dims;
   d = mxGetNumberOfDimensions(A);
   dims = mxGetSize(A);
   
   dims is an integer array of length $d$ containing the dimensions.
   
   2. It is not necessary to explicitly destroy storage you created, unless you run out of room. Matlab will automatically recover temporary storage when the subroutine terminates.

   3. In Matlab 5, the choices for the second argument to `mxGetArray` and `mxPutArray` are `base` (matrix is copied to or from main workspace), `caller` (matrix is copied to or from workspace of the routine which called this one), or `global` (matrix is copied to or from space of global variables).

   4. Some routines are called `mxWhatever`, some are called `MxWhatever`, and some are called `mexWhatever`. I am sure there is a logical reason somewhere, but I haven’t figured it out yet.
### Matlab 4

<table>
<thead>
<tr>
<th>Main Interface Function</th>
<th>void mexFunction(int nargout, Matrix *argout, int nargin, Matrix *argin)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Accessing Internals of a Matlab Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>Get matrix name</td>
<td>name = mxGetName(A);</td>
</tr>
<tr>
<td>Get dimensions</td>
<td>m = mxGetM(A); n = mxGetN(A);</td>
</tr>
<tr>
<td>Test matrix type</td>
<td>if (mxIsComplex(A)) {};</td>
</tr>
<tr>
<td></td>
<td>if (mxIsDouble(A)) {};</td>
</tr>
<tr>
<td></td>
<td>if (mxIsNumeric(A)) {};</td>
</tr>
<tr>
<td></td>
<td>if (mxIsString(A)) {};</td>
</tr>
<tr>
<td>Access numerical data</td>
<td>Apr = mxGetPr(A); Api = mxGetPi(A);</td>
</tr>
<tr>
<td>Access string data</td>
<td>mxGetString(A,string,max_string_length);</td>
</tr>
<tr>
<td><strong>Setting Internals of a Matlab Matrix</strong></td>
<td></td>
</tr>
<tr>
<td>Set matrix name</td>
<td>mxSetName(A,name);</td>
</tr>
<tr>
<td>Set dimensions</td>
<td>mxSetM(A,m); mxSetN(A,n);</td>
</tr>
<tr>
<td>Set matrix type</td>
<td>not possible; type can only be set at creation time</td>
</tr>
<tr>
<td>Insert numerical data</td>
<td>mxSetPr(A,Apr); mxSetPi(A,Api);</td>
</tr>
<tr>
<td>Insert string data</td>
<td>not possible; strings can only be changed at creation time</td>
</tr>
<tr>
<td><strong>Creating and Destroying Storage</strong> (see remark 2)</td>
<td></td>
</tr>
<tr>
<td>Fortran working storage</td>
<td>W = mxMalloc(m*n,sizeof(double));</td>
</tr>
<tr>
<td></td>
<td>MxFree(W);</td>
</tr>
<tr>
<td>Matlab numerical matrix</td>
<td>A = mxCreateFull(m,n,REAL); (or COMPLEX)</td>
</tr>
<tr>
<td></td>
<td>MxFreeMatrix(A);</td>
</tr>
<tr>
<td>Matlab string</td>
<td>A = mxCreateString(string);</td>
</tr>
<tr>
<td></td>
<td>MxFreeMatrix(A);</td>
</tr>
<tr>
<td><strong>Printing</strong></td>
<td></td>
</tr>
<tr>
<td>Regular messages</td>
<td>mexPrintf(&quot;value of %s is %d\n&quot;,name,value);</td>
</tr>
<tr>
<td>Error messages</td>
<td>mexErrMsgTxt(&quot;message&quot;);</td>
</tr>
<tr>
<td><strong>Accessing Matlab workspace</strong></td>
<td></td>
</tr>
<tr>
<td>Accessing variables</td>
<td>A = mexGetMatrix(&quot;A&quot;);</td>
</tr>
<tr>
<td></td>
<td>mexPutMatrix(A);</td>
</tr>
<tr>
<td>Calling functions</td>
<td>mexCallMATLAB(nargout,argout,nargin,argin,&quot;f&quot;);</td>
</tr>
<tr>
<td>Executing statements</td>
<td>mexEvalString(&quot;x = y \times z&quot;);</td>
</tr>
<tr>
<td>Main Interface Function</td>
<td>’void mexFunction(int nargout,mxArray *argout,int nargin,const mxArray *argin)”</td>
</tr>
<tr>
<td>-------------------------</td>
<td>--------------------------------------------------------------------------------</td>
</tr>
<tr>
<td>Accessing Internals of a Matlab Matrix</td>
<td>name = mxGetName(A);</td>
</tr>
<tr>
<td>Get matrix name</td>
<td>m = mxGetM(A); n = mxGetN(A); (see remark 1)</td>
</tr>
</tbody>
</table>
| Get dimensions | if (mxIsComplex(A)) {};
| Test matrix type | if (mxIsDouble(A)) {};
| if (mxIsNumeric(A)) {};
| if (mxIsChar(A)) {};
| Access numerical data | Apr = mxGetPr(A); Api = mxGetPi(A); |
| Access string data | mxGetString(A,string,max_string_length); |
| Setting Internals of a Matlab Matrix | mxSetName(A,name); |
| Set matrix name | mxSetM(A,m); mxSetN(A,n); |
| Set dimensions | not possible; type can only be set at creation time |
| Set matrix type | mxSetPr(A,Apr); mxSetPi(A,Api); |
| Insert numerical data | not possible; strings can only be changed at creation time |
| Insert string data | |
| Creating and Destroying Storage | W = mxMalloc(m*n,sizeof(double)); |
| (see remark 2) | MxFree(W); |
| Fortran working storage | A = mxCreateDoubleMatrix(m,n,mxREAL); (or mxCOMPLEX) |
| Matlab numerical matrix | MxDestroyArray(A); |
| Matlab string | A = mxCreateString(string); |
| MxDestroyArray(A); |
| Printing | mexPrintf(“value of %s is %d \n”,name,value); |
| Regular messages | mexWarnMsgTxt(“message”); |
| Warning messages | mexErrMsgTxt(“message”); |
| Error messages | |
| Accessing Matlab workspace | |
| (see remark 3) | A = mexGetArray("A","base"); |
| Accessing variables | mexPutArray(A,"base"); |
| Calling functions | mexCallMATLAB(nargout,argout,nargin,argin,"f"); |
| Executing statements | mexEvalString("x = y * z"); |
This case study illustrates the simplest case: calling a simple function with one scalar argument and one return value.

We want to interface the double precision gamma function DGAMMA from FNLIB, which is part of CMLIB, declared as

\[
\text{double precision function dgamma}(x) \\
\text{double precision } x \\
\ldots
\]

This function calculates the gamma function \( \Gamma(x) \).

The mex program `dgamma.c` is

```c
/* interface to routine DGAMMA from library FNLIB

usage: y = dgamma(x)

Fritz Keinert
November 6, 1995
*/

#include <stdio.h>
#include <math.h>
#include <mex.h>

/* shorthand names for input and output parameters */
#define x argin[0]
#define y argout[0]

/* prototype for external routine */
extern double dgamma_(double *xx);

void mexFunction(int nargout,Matrix *argout[],int nargin,Matrix *argin[]) {
    int i, m, n, nx;
    double *p, *q;

    /*--------------------------------------*
    | check and process input parameters |
    *--------------------------------------*/
    if (nargin < 1)
        mexErrMsgTxt("Missing input argument");
    if (!mxIsNumeric(x) || mxIsComplex(x))
        mexErrMsgTxt("Input argument must be real.");

    m = mxGetM(x);
    n = mxGetN(x);
    nx = m * n;

    /*----------------------*/
    | check and process input parameters |
    *----------------------*/
```

Copyright 1996 by Fritz Keinert printed February 24, 1997
| check and create output parameters |
*--------------------------------------*/

if (nargout > 1)
    mexErrMsgTxt("Too many output arguments");

y = mxCreateFull(m,n,REAL);

/*------------------------------------------------------*/
| invoke external routine |
*------------------------------------------------------*/

p = mxGetPr(x);
q = mxGetPr(y);
for (i = 0; i < nx; i++)
    q[i] = dgamma_(p+i);
}

Notes:
• Fortran routine DGAMMA is known as dgamma to the loader.
• If the interface routine is called with a matrix argument, dgamma is invoked in a loop, on every matrix entry.
• In case of error, DGAMMA calls a subroutine XERROR instead of returning an error code. XERROR is set up to terminate the program in case of fatal errors, which in this case means it will terminate Matlab. That is not usually what we want. So, I have rewritten XERROR into a mex file xerror.c, which will then print the messages inside Matlab, and abort only the subroutine.

The code for xerror.c is

/*
 This routine is a MEX translation of routine XERROR in CMLIB.

 Fritz Keinert <keinert@iastate.edu>
 December 7, 1995
 */

#include <stdio.h>
#include <mex.h>

void xerror_(char *messg, int *nmessg, int *nerr, int *level, int messg_length) {
    char message[73];
    int i;

    /* copy the message over into a C string */
    for (i = 0; i < *nmessg; i++) message[i] = messg[i];
    message[*nmessg] = '\0';

    mexPrintf("Error %d: %s\n",*nerr,message);
    switch (*level) {
    case 2: mexErrMsgTxt("Fatal Error");
    case 1: mexPrintf("Recoverable Error\n");
            return;
    default: mexPrintf("Warning\n");
    }
Calling Fortran From Matlab

return;
}
}
}

For documentation, create a file dgamma.m with only comments:

% DGAMMA -- The gamma function.
%
% y = dgamma(x)
%
% This routine calls DGAMMA from FNLIB. It was written as a
% programming exercise and duplicates the functionality of the
% built-in function GAMMA.
%
% Fritz Keinert
% January 4, 1996

On a DECStation (Matlab 4), compile the program with

% fmex dgamma.c xerror.c /home/math/lib/dec/libcm.a

When I do this, I get warning messages from the loader:

-----------------------------------------------
ld:
/lib/libm.a(floor.o): floor: multiply defined
/lib/libm.a(floor.o): ceil: multiply defined

I don't know why this happens, but the messages seem to be harmless.

On an Alpha, you could compile with the version 4 compiler as

% fmex dgamma.c xerror.c /home/math/lib/axp/libcm.a

but this is not recommended, since this compiler is known to have bugs.

On an SGI (Matlab 4), compile with

% fmex dgamma.c xerror.c /home/math/lib/sgi/libcm.a

On both Alphas and SGIs, you get a dashed line

-----------------------------------------------

but not the loader error messages.

In Matlab 5 (currently only available on DEC Alphas), you have two choices. Either, compile the same
code in version 4 compatibility mode:

% mex5 -V4 dgamma.c xerror.c /home/math/lib/axp/libcm.a

or slightly modify dgamma.c into dgamma5.c:

... void mexFunction(int nargout,mxArray *argout[],int nargin,const mxArray *argin[])
...
    y = mxCreateDoubleMatrix(m,n,mxREAL);
...

and compile like before:

% mex5 -output dgamma dgamma5.c xerror.c /home/math/lib/axp/libcm.a

Note that the new command mex supersedes both cmex and fmex. Currently, Matlab and mex in version 5
are called matlab5 and mex5, but the extra 5s will probably go away eventually.

If you plan to use mex routines, you should really make up your mind to use either Matlab 4 or Matlab
5 consistently. If you use the Matlab 5 mex compiler, you cannot load the result into Matlab 4, and vice
versa, but you can't tell the version from looking at the name.

Now you can call dgamma from inside Matlab:

Copyright 1996 by Fritz Keinert

printed February 24, 1997
>> dgamma(2.718)
an =
    1.56711274176688
>> gamma(2.718)
an =
    1.56711274176688
>> dgamma(-1)
Error 4
DGAMMA   X IS A NEGATIVE INTEGER
??? Fatal Error in Fortran library routine DGAMMA
Error in ==> /home/keinert/mex/dgamma.mexds
>> help dgamma

DGAMMA -- The gamma function.

    y = dgamma(x)

This routine calls DGAMMA from FNLIB. It was written as a
programming exercise and duplicates the functionality of the
built-in function GAMMA.
Case Study 2: CGAMMA

This case study illustrates the use of complex numbers. Routine CGAMMA is the complex equivalent of DGAMMA, that is, the gamma function $\Gamma(x)$ for complex arguments. It is declared as:

```fortran
complex function cgamma(x)
complex x
...
```

Note that CGAMMA is in single precision; I was not able to find a public domain double precision complex gamma function. Also, I have to use an extra interface routine `cgamma2.f`, because functions returning complex numbers are not compatible between Fortran and C. For details, look at case study CGAMMA in section Calling Fortran from C in my other report Calling Fortran Subroutines from Fortran, C, and C++.

The code for `cgamma2.f` is very simple:

```fortran
subroutine cgamma2(x,y)
complex x, y, cgamma
y = cgamma(x)
end
```

To illustrate some more user-friendly programming, the interface routine uses either DGAMMA and CGAMMA, depending on whether it was called with a real or complex argument.

```c
#include <stdio.h>
#include <math.h>
#include <mex.h>
/* shorthand names for input and output parameters */
#define y argout[0]
#define x argin[0]
/* prototypes for external routines */
extern double dgamma_(double *xx);
typedef struct { float re, im; } complex;
extern void cgamma2_(complex *xx, complex *yy);
void mexFunction(int nargout,Matrix *argout[],int nargin,Matrix *argin[])
{
int i, m, n, nx;
double *xpr, *xpi, *ypr, *ypi;
complex z, w;

/**********************************************************************
 | check and process input parameters |
*--------------------------------------*/
if (nargin < 1)
    mexErrMsgTxt("Missing input argument");

if (!mxIsNumeric(x))
    mexErrMsgTxt("Input argument must be numeric.");

m = mxGetM(x);
n = mxGetN(x);
nx = m * n;

/*--------------------------------------*
| check and create output parameters |
*--------------------------------------*/
if (nargout > 1)
    mexErrMsgTxt("Too many output arguments");

if (mxIsComplex(x))
    y = mxCreateFull(m,n,COMPLEX);
else
    y = mxCreateFull(m,n,REAL);

/*---------------------------*
| invoke external routine |
*---------------------------*/
xpr = mxGetPr(x);
xpi = mxGetPi(x);
ypr = mxGetPr(y);
ypi = mxGetPi(y);

for (i = 0; i < nx; i++)
    if (mxIsComplex(x)) {
        z.re = xpr[i];
z.im = xpi[i];
cgamma2_(&z,&w);
ypr[i] = w.re;
ypi[i] = w.im;
    }
else
    ypr[i] = dgamma_(xpr+i);
}

On a DECStation (Matlab 4), compile the program with
% fmex cgamma.c cgamma2.f xerror.c /home/math/lib/dec/libcm.a
On a Dec Alpha (Matlab 4), the command is
% fmex cgamma.c cgamma2.f xerror.c /home/math/lib/axp/libcm.a
On an SGI (Matlab 4), the command is
% fmex cgamma.c cgamma2.f xerror.c /home/math/lib/sgi/libcm.a

Copyright 1996 by Fritz Keinert
printed February 24, 1997
In Matlab 5 (currently available only on DEC Alphas), the command is

```
% mex5 -V4 cgamma.c cgamma2.f xerror.c /home/math/lib/axp/libcm.a
```

Alternatively, you could rewrite the code a little and compile it as Matlab 5 code, without the `-V4`. Now you can call it from inside Matlab:

```
>> cgamma(2.718)
an=
    1.56711274176688
>> cgamma(2+i)
an=
    0.65296548604965 + 0.34306597709656i
```

Of course, only about 7 of the decimals in the complex answer are correct.
Routine DGESVD calculates the singular value decomposition (SVD) of a general double precision matrix. A matrix $A$ of size $(m \times n)$, stored in array $A(lda,n)$, is decomposed into

$$A = U \Sigma V^T,$$

where $U$ is an orthogonal matrix of size $(m \times m)$, stored in array $U(ldu,m)$, $V^T$ is an orthogonal matrix of size $(n \times n)$, stored in array $VT(ldvt,n)$, and $\Sigma$ is a diagonal matrix of size $(m \times n)$, with the diagonal stored in a one-dimensional array $S(min(m,n))$.

The declaration is

```fortran
subroutine dgesvd(jobu, jobvt, m, n, A, lda, S, U, ldu, VT, ldvt,
+ work, lwork, info )
character jobu, jobvt
integer info, lda, ldu, ldvt, lwork, m, n
double precision A(lda,*), S(*), U(ldu,*),
+ VT(ldvt,*), work(*)
```

For the exact meaning of all these parameters, consult the LAPACK documentation. All you need to know here is that $jobu = jobv = 'a'$ will request that $U$, $VT$ are actually calculated. (There are applications where $U$, $VT$ are not needed, or only parts of them are needed; not calculating them saves time).

### Step 1: Design The Interface

I decided to make the interface look like that of the built-in `svd`. That is, the user can do one of the following:

```matlab
>> S = dgesvd(A)
>> [U,S,V] = dgesvd(A)
```

Input arguments: $A$ (required).
Output arguments: either $S$ or $[U,S,V]$.
Unnecessary arguments: everything else.

### Step 2: Write The Code

As I said before, this is about as bad as it gets. I will let the commented source code speak for itself. Pay attention to which variables are Fortran arrays, and which variables are Matlab matrices, and how they are created and destroyed in different ways.

Here is the program:

```c
/* interface to routine DGESVD from library LAPACK

usage: S = dgesvd(A) or [U,S,V] = dgesvd(A)

Fritz Keinert
November 16, 1995

*/

#include <stdio.h>
#include <math.h>
```

Copyright 1996 by Fritz Keinert printed February 24, 1997
```c
#include <mex.h>

#define max(a,b) ((a) > (b) ? (a) : (b))
#define min(a,b) ((a) < (b) ? (a) : (b))

/* shorthand names for input and output parameters */
#define U argout[0]
#define S argout[1]
#define V argout[2]
#define A argin[0]

/* prototype for external routine */

extern void dgesvd_(char *jobu, char *jobvt, int *m, int *n,
                    double AA[], int *lda, double SS[], double UU[],
                    int *ldu, double VV[], int *ldvt, double work[],
                    int *lwork, int *info);

void mexFunction(int nargout,Matrix *argout[],int nargin,Matrix *argin[])
{
    int m, n, nsv, i, j, lwork, info;
    Matrix *SS;
    double temp;
    char job;

    /*-------------------------------------------------------------*
     | check and process input parameters                         |
     *----------------------------------------------------------------*

    if (nargin < 1)
        mexErrMsgTxt("Missing input argument");

    if (!mxIsNumeric(A) || mxIsComplex(A))
        mexErrMsgTxt("Input argument must be real.");

    m = mxGetM(A);
    n = mxGetN(A);
    Apr = mxGetPr(A);
    nsv = min(m,n);

    /*-------------------------------------------------------------*
     | check number of output parameters                         |
     *----------------------------------------------------------------*

    if (nargout == 2 || nargout >= 4)
        mexErrMsgTxt("Incorrect number of output arguments");

    if (nargout <= 1)
        job = 'n';
    else
        job = 'a';

    ...
job = 'a';

/****************************************************************************
| create working storage and output variables for external routine |
****************************************************************************/

/* create copy of A because DGESVD will destroy it;
a Fortran array is sufficient for that */
AA = mxCalloc(m*n,sizeof(double));
for (i = 0; i < m*n; i++) AA[i] = Apr[i];

/* create working storage (Fortran arrays) */
lwork = max(3*min(m,n)+max(m,n),5*min(m,n)-4);
work = mxCalloc(lwork,sizeof(double));

/* create output arrays for DGESVD;
these need to be Matlab matrices */
U = mxCreateFull(m,m,REAL);
SS = mxCreateFull(nsv,1,REAL);
V = mxCreateFull(n,n,REAL);
Upr = mxGetPr(U);
SSpr = mxGetPr(SS);
Vpr = mxGetPr(V);

dgesvd_(&job, &job, &m, &n, AA, &m, SSpr, Upr, &m, Vpr, &n, work, &lwork, &info);

if (info > 0)
mexPrintf("\aMethod did not converge. Don’t trust the answers\n");

/* I don’t think the following code will ever get executed because
xerbla gets called first, but it doesn’t hurt to be cautious */
if (info < 0) {
    mexPrintf("DGESVD returned info = %d\n",info);
    mexErrMsgTxt("\n");
}

/****************************************************************************
| copy answer into output variables |
****************************************************************************/
if (nargout <= 1) {
    /* use SS as is */
    U = SS;
}

else {
    /* copy vector of singular values SS into matrix S */
    S = mxCreateFull(m,n,REAL);
    Spr = mxGetPr(S);
    for (i = 0; i < nsv; i++) Spr[i*m+i] = SSpr[i];
    /* transpose V in place */
    for (i = 0; i < n; i++)
        for (j = i+1; j < n; j++) {
            temp = Vpr[i*n+j];
            Vpr[i*n+j] = Vpr[j*n+i];
            Vpr[j*n+i] = temp;
        }
}

As with XERROR before, I have rewritten the error reporting routine XERBLA in mex:

```c
#include <stdio.h>
#include <mex.h>

/*
 * This routine is a MEX translation of routine XERBLA in LAPACK.
 * Fritz Keinert <keinert@iastate.edu>
 * December 7, 1995
 */

void xerbla_(char srname[6], int *info, int srname_length) {
    char name[7];
    int i;

    for (i = 0; i < 6; i++) name[i] = srname[i];
    name[6] = '\0';
    mexPrintf(" ** On entry to %6s parameter number %d had an illegal value\n", name, *info);
    mexErrMsgTxt(" ");
}
```

On a DECStation (Matlab 4), compile it with

```
% fmex dgesvd.c xerbla.c -L/home/ lapack/lib/dec -llapack -lblas
```

On a DEC Alpha (Matlab 4), compile it with

```
% fmex dgesvd.c xerbla.c -L/home/ lapack/lib/axp -llapack -lblas
```

On an SGI (Matlab 4), compile it with

```
% fmex dgesvd.c xerbla.c -L/home/ lapack/lib/sgi -llapack -lblas
```

On a DEC Alpha (Matlab 5), compile it with

```
% mex5 -V4 dgesvd.c xerbla.c -L/home/ lapack/lib/axp -llapack -lblas
```
As an example, we will calculate the SVD of the matrix

\[ A = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{pmatrix} \]

We test it from inside Matlab:

```matlab
>> [U,S,V] = svd(A)
U =
  -0.99734257743710  0.07285453473271
   0.07285453473271 -0.99734257743710
S =
  9.62285975211415    0    0
   0  0.47680461107054   0
V =
  0.99350513010790 -0.08853418260824  0.07147905259011
 -0.11359499888043 -0.80821559939778  0.57782672239993
 0.00661306877662 -0.58219347591671 -0.81302338460919
>> [U,S,V] = dgesvd(A)
U =
  -0.99734257743710  0.07285453473271
   0.07285453473271 -0.99734257743710
S =
  9.62285975211415    0    0
   0  0.47680461107054   0
V =
  0.99350513010790 -0.08853418260824 -0.07147905259011
 -0.11359499888043 -0.80821559939778  0.57782672239993
 0.00661306877662 -0.58219347591671  0.81302338460919
>> S = svd(A)
S =
  9.62323871051386
  0.57104182105954
>> S = dgesvd(A)
S =
  9.62323871051385
  0.57104182105954
```

The answer given by DGESVD has different signs than the answer returned by svd in Matlab, but it is still correct. (It is possible to shift plus and minus signs back and forth between \( U \) and \( V \)). It just shows that Matlab is not using DGESVD internally. It is probably using the corresponding routine from LINPACK.
Case Study 4: DQAGS

This case study illustrates the passing of a subroutine name as an argument, the use of strings, and the use of optional input arguments.

Routine DQAGS calculates an approximation to \( \int_a^b f(x) \, dx \). The declaration is

```fortran
subroutine dqags(f,a,b,epsabs,epsrel,result,abserr,neval,ier,
+ limit,lenw,last,iwork,work)
  integer neval, ier, limit, lenw, last, iwork(limit)
  double precision f, a, b, epsabs, epsrel, result, abserr, work(lenw)
  external f
end subroutine dqags
```

For the exact meaning of all these parameters, see the DQAGS documentation (DQAGS is part of QUADPACK, which is part of CMLIB). For our purposes it suffices to know that on input we have to specify the function \( f \), integration limits \( a, b \) and the requested absolute and/or relative accuracy \( \text{epsabs, epsrel} \). On return, we get the approximate integral \( \text{result} \), the estimated actual error \( \text{abserr} \), and the number of function evaluations done \( \text{neval} \).

Step 1: Design The Interface

I chose an interface similar to that of the Matlab built-in QUAD8:

```matlab
[result,neval] = dqags('f',a,b,tol)
```

Input arguments: required arguments \( f, a, b \); optional arguments \( \text{tol} \) with default \( 10^{-3} \).

Output arguments: \( \text{result} \), optionally \( \text{neval} \)

Unnecessary arguments: everything else.

Step 2: Write The Program

There is one tricky part to this interface: DQAGS expects the integrand as a Fortran function, but from Matlab we actually get only the name of a Matlab subroutine.

The interface works as follows:

First, we convert the name of the Matlab function to a C string called `fname`. We also set up two \( 1 \times 1 \) Matlab matrices, called \( x_\text{a} \) and \( y_\text{b} \).

Subroutine `feval` is a C function which gets called from DQAGS, as the integrand. It puts its input \( x \) (a double precision number) into a Matlab array \( x_\text{a} \) calls the Matlab routine whose name is stored in `fname`, retrieves the output from Matlab array \( y_\text{b} \) and returns it to DQAGS as a double precision number.

Don’t feel bad if this takes you a while to figure out. It took me quite a while to figure out how to write this, too.

```c
/*
   interface to routine DQAGS from library QUADPACK.
   usage: [result,neval] = dqags('f',a,b,epsrel)
*/

#include <stdio.h>
#include <math.h>
#include <mex.h>

/* shorthand names for input and output parameters */

#define result argout[0]
#define neval argout[1]
#define f argin[0]
#define a argin[1]
```
#define b argin[2]
#define tol argin[3]

/* prototype for DQAGS */

extern double dqag_(double f_(double *x_),double *a_,double *b_,
double *epsabs_,double *epsrel_,double *result_,
double *abserr_,int *neval_,int *ier_,int *limit_,
int *lenw_,int *last_,int iwork[],double work[]);

/* the following routine is callable from Fortran, as a function y = f(xarg);
it converts the Fortran number xarg into a Matlab matrix x_, does the
equivalent of "y_ = feval(fname,x_)", and converts the Matlab output
matrix y_ back into a Fortran number y. */

char *fname;
Matrix *x_, *y_

double feval(double *xarg)
{
    *(mxGetPr(x_)) = *xarg;
    if (mexCallMATLAB(1,&y_,1,&x_,fname)) {
        mexPrintf("DQAGS: function call to %s failed\n",fname);
        mexErrMsgTxt("\n");
    }
    return mxGetScalar(y_);
}

/* main program */

void mexFunction(int nargout,Matrix *argout[],int nargin,Matrix *argin[])
{
    int f_strlen, neval_, ier, limit_, lenw, last_, *iwork;
    double epsrel_, epsabs_, result_, abserr_, *work;
    Matrix *eps = mexGetMatrixPtr("eps"); /* machine epsilon */

#ifndef DQAGS

*/
    /*------------------------------------------------------------------*/
    | check and process input parameters |
    /*------------------------------------------------------------------*/

    if (nargout > 2 || nargin < 3)
        mexErrMsgTxt("usage: [result,neval] = qags('f',a,b,tol)\n");

    /*------------------------------------------------------------------*/
    | defaults |
    /*------------------------------------------------------------------*/

    epsabs_ = 0.;
    limit_ = 10;

    if (nargin == 3) { /* use default for tol */
        epsrel_ = 1.e-3;
    
    /*------------------------------------------------------------------*/
    | do the computation |
    /*------------------------------------------------------------------*/

    /*------------------------------------------------------------------*/
    | return the results |
    /*------------------------------------------------------------------*/

#endif

Copyright 1996 by Fritz Keinert
Printed February 24, 1997
} else {
    epsrel_ = mxGetScalar(tol);
}

/*********************************************************
| create temporary variables |
***********************************************************/

/* extract name of function into string */

f_strlen = mxGetN(f) + 1;
fname = mxMalloc(f_strlen,sizeof(char));
mxGetString(f,fname,f_strlen);

/* create input and output matrices for feval */

x_ = mxCreateFull(1,1,REAL);
y_ = mxCreateFull(1,1,REAL);

/* set up working storage for DQAGS */

lenw = 4 * limit_;
iwork = mxMalloc(limit_,sizeof(int));
work = mxMalloc(lenw,sizeof(double));

/*********************************************************
| invoke external routine |
***********************************************************/

dqags_(feval,mxGetPr(a),mxGetPr(b),&epsabs_,&epsrel_,
    &result_,&abserr_,&neval_,&ier,&limit_,&lenw,&last_,iwork,work);

/*********************************************************
| check whether any errors occured |
***********************************************************/

switch (ier) {
    case 0: break; /* normal termination */
    case 1: mexPrintf("DQAGS: maximum number of subdivisions exceeded\n");
        mexErrMsgTxt(" increase limit, split integral at singularity, or use a different\n");
        break;
    case 2: mexPrintf("DQAGS Warning: requested tolerance not achieved\n");
        mexPrintf(" reported result and error bound may be wrong\n");
        break;
    case 3: mexPrintf("DQAGS Warning: extremely bad integrand behavior\n");
        mexPrintf(" reported result and error bound may be wrong\n");
        break;
    case 4: mexPrintf("DQAGS Warning: algorithm does not converge\n");
        mexPrintf(" reported result and error bound may be wrong\n");
        break;
    case 5: mexPrintf("DQAGS Warning: integral divergent or very slowly convergent\n");
mexPrintf(" reported result and error bound may be wrong\n");
break;
case 6: mexPrintf("DQAGS Error: requested relative accuracy cannot be achieved\n");
mexPrintf(" the smallest allowed value is 50 eps = %e\n",
50. * mxGetScalar(eps));
mexErrMsgTxt(" ");
default: mexPrintf("DQAGS: unknown error code %d",ier);
mexErrMsgTxt(" ");
}

/*--------------------------------------*
| check and create output parameters |
*--------------------------------------*/

result = mxCreateFull(1,1,REAL);
*(mxGetPr(result)) = result_

if (nargout >= 2) {
    neval = mxCreateFull(1,1,REAL);
    *(mxGetPr(neval)) = neval_
}
}

On a DECStation (Matlab 4), compile it with
% fmex dqags.c xerror.c /home/math/lib/dec/libcm.a
On a DEC Alpha (Matlab 4), compile it with
% fmex dqags.c xerror.c -L/home/lapack/lib/axp/libcm.a
On an SGI (Matlab 4), compile it with
% fmex dqags.c xerror.c -L/home/lapack/lib/sgi/libcm.a
On a DEC Alpha (Matlab 5), compile it with
% mex5 -V4 dqags.c xerror.c -L/home/lapack/lib/axp/libcm.a

We test it from inside Matlab:
>> result = dqags('sin',0,10)
result =
    1.83907152907645
>> [result,neval] = dqags('sin',0,10)
result =
    1.83907152907645
neval =
    21
>> [result,neval] = dqags('sin',0,10,1.e-13)
result =
    1.83907152907645
neval =
    63

E-mail address: keinert@iastate.edu