

Extending MATLAB[®] with Numerical Recipes

Numerical Recipes code, or for that matter any other C++ code, can easily be invoked from within MATLAB, from the console, or from MATLAB functions written in m-code. For some tasks, C++ executes *hugely* faster than MATLAB m-code, and you can also access features in Numerical Recipes (or elsewhere) that are not available in MATLAB. You can code some parts of your project in m-code and other parts in C++, and control the whole project from the MATLAB console or command line.

This document shows how this is done, with examples of two different coding methodologies: (1) "API", using the MATLAB [C API function calls](#), and (2) "NR3", using the special Numerical Recipes include file [nr3matlab.h](#). We think that "NR3" is the best choice (it's simpler to code); but we also show "API" examples so that you can understand the underlying MATLAB interface.

Contents

[Conventions and Overview](#)

["Hello, world!" Using the Matlab C API](#)

[Do Just Once: Introduce Matlab to Your Compiler](#)

[The NR3 Coding Method](#)

["Hello, world!" Using nr3matlab.h](#)

[Accessing Matlab Vectors with nr3matlab.h](#)

[Living Dangerously](#)

[Living Safely](#)

[Accessing Matlab Scalars with nr3matlab.h](#)

[Matrices: "Through the Looking Glass" \(TTLG\)](#)

[Accessing Matlab Matrices with nr3matlab.h](#)

[Wrapper Functions That Access Multiple Member Functions in a Class](#)

[The API Coding Method](#)

[Tips on Matlab C API Programming](#)

[Numerical Recipes Using the Matlab C API](#)

[Appendix: Using Microsoft Visual Studio](#)

Conventions and Overview

MATLAB m-code, or text from the MATLAB console is shown on a red background:

```
% MATLAB code
[a b] = someroutine(c,d)
```

C++ code is shown on a green background:

```
/* C++ code */
#include "mex.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
// don't worry yet about these weird types!
}
```

A compiled, executable file that is invoked by a MATLAB command is called a "mex file". Mex files are written as source code outside of the MATLAB console, in a text editor such as vi or Emacs or the MATLAB editor, or in an integrated development environment (IDE) such

as Microsoft Visual Studio.

Mex source code files must be compiled and linked. This can be done either from the MATLAB console using the command "mex", or else from the IDE. The final step is, if necessary, to move the compiled mex file into your MATLAB working directory or path.

A mex file, as one compilation unit, always codes exactly one MATLAB-callable function. In fact, the **name** of the compiled mex-function **is** the MATLAB function name used to access it. (That name typically doesn't even appear in the source code, unless you happen to include it in a comment!)

Of course, within your mex file you can do any number of C++ function calls, memory allocations or deallocations, input-output, and so forth, before returning control (either with or without return values) to the MATLAB console or calling m-file. You can also access existing or create new named variables in MATLAB's workspace, independent of whether they are input arguments to your mex function.

"Hello, world!" Using the MATLAB C API

Here is source code for a mex function that receives an input array of arbitrary size and shape and returns the squares of each of its elements as a row vector. Oh, yes, it also prints "Hello, world!" to the MATLAB console.

```
/* helloworld.cpp */
#include "mex.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    int i, n = mxGetNumberOfElements(prhs[0]);
    double *indata = (double *)mxGetData(prhs[0]);
    plhs[0] = mxCreateNumericMatrix(1,n,mxDOUBLE_CLASS,mxREAL);
    double *outdata = (double *)mxGetData(plhs[0]);
    for (i=0;i<n;i++) outdata[i] = indata[i]*indata[i];
    printf("Hello, world!\n");
}
```

We compile it from the MATLAB console,

```
>> mex helloworld.cpp
>>
```

An example of using our new mex function is

```
>> b = [1 2 3; 4 5 6]
b =
     1     2     3
     4     5     6
>> a = helloworld(b)
Hello, world!
a =
     1    16     4    25     9    36
>>
```

Do Just Once: Introduce MATLAB to Your Compiler

You might wonder how we got the mex command to compile C++ source, since MATLAB comes only with a built-in, rather brain dead, C compiler. The answer is that you need to

have a C++ compiler already on your machine and, just once, you need to introduce MATLAB to it. For Linux, g++ and Intel C++ are likely compilers; for Windows either Microsoft Visual C++ or Intel C++.

To bind the "mex" command to a compiler, do something like the following:

```
>> mex -setup
Please choose your compiler for building external interface (MEX) files:

would you like mex to locate installed compilers [y]/n? y

Select a compiler:
[1] Intel C++ 9.1 (with Microsoft Visual C++ 2005 linker)
    in C:\Program Files\Intel\Compiler\C++\9.1
[2] Lcc-win32 C 2.4.1 in C:\PROGRA~1\MATLAB\R2007A~1\sys\lcc
[3] Microsoft Visual C++ 2005 in C:\Program Files\Microsoft Visual Studio 8
[4] Microsoft Visual C++ .NET 2003
    in C:\Program Files\Microsoft Visual Studio .NET 2003
[5] Microsoft Visual C++ 6.0 in C:\Program Files\Microsoft Visual Studio

[0] None

Compiler: 3

Please verify your choices:

Compiler: Microsoft Visual C++ 2005
Location: C:\Program Files\Microsoft Visual Studio 8

Are these correct?([y]/n): y

Trying to update options file:
C:\Documents and Settings\...\MathWorks\MATLAB\R2007a\mexopts.bat
From template: C:\PROGRA~1\MATLAB\R2007A~1\bin\win32\mexopts\msvc80opts.bat

Done . . .
>>
```

The above shows a Windows machine with several installed compilers. Your choices may be fewer. Options 1 or 3 above would be good choices. Option 2, the brain-dead C compiler, would not be.

The NR3 Coding Method

The NR3 coding method uses wrapper functions that are in the include file [nr3matlab.h](#) instead of the MATLAB C API. Of course, you can also use any of the API functions directly. But you should rarely or never need to.

Mex source files in the NR3 coding method **always** have **exactly** the framework

```
#include "nr3matlab.h"
/* you may put other stuff here */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
/* you must put something here */
}
```

The `void mexFunction(...)` line is somewhat equivalent to the `int main(...)` in ordinary C++ programming. (You *don't* include a `main` routine in mex files.)

Within your mex code, you are given the integer number of arguments, `nrhs`, and the

expected number of returned values `nlhs`. Both arguments and return values may of course be arrays. Below, we'll see how to use the input arguments `plhs` and `prhs` with the NR3 coding method.

Although the order of presentation might at first seem odd, we'll discuss vectors, then scalars, and finally matrices. But, before any of these, we must of course repeat the "Hello, world!" example, now in NR3 coding style:

"Hello, world!" Using `nr3matlab.h`

Using the NR3 coding method, our "Hello, world!" mex function is rather more readable than before.

```
/* helloworld2.cpp */
#include "nr3matlab.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    MatDoub indata(prhs[0]);
    Int i, j, k=0, m=indata.nrows(), n=indata.ncols();
    VecDoub outdata(m*n,plhs[0]);
    for (i=0;i<m;i++) for (j=0;j<n;j++) outdata[k++] = SQR(indata[i][j]);
    printf("Hello, world!\n");
}
```

Execution is no different from before.

```
>> mex helloworld2.cpp
>> b = [1 2 3; 4 5 6]
b =
     1     2     3
     4     5     6
>> a = helloworld2(b)
Hello, world!
a =
     1    16     4    25     9    36
```

Accessing MATLAB Vectors with `nr3matlab.h`

The include file `nr3matlab.h` is mostly the same as the standard NR3 include file `nr3.h`. What it adds are some new vector constructors and a new vector member function. These make it easy to interact with MATLAB vectors in a natural way, without needing to use the API mx-functions. (That is, natural for C++ programmers who have a copy of Numerical Recipes!)

For reference, the additions to the templated class `NRvector` are:

```
template <class T> class NRvector {
    /* ... */
    NRvector(const mxArray *prhs); // map Matlab rhs to vector
    NRvector(int n, mxArray* &plhs); // create Matlab lhs and map to vector
    NRvector(const char *varname); // map Matlab variable by name
    void put(const char *varname); // copy NRvector to a named Matlab variable
}
```

You don't really have to understand the above declarations, or their definitions, in detail. Here are some usage examples:

```

/* NRvectorDemo.cpp */
#include "nr3matlab.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {

    Int i,n;

    // vector rhs arguments and lhs return values

    VecDoub b(prhs[0]); // bind the first rhs argument
    n = b.size(); // get its number of elements
    VecDoub c(n,plhs[0]); // create a lhs return value of same length
    for (i=0;i<n;i++) c[i] = SQR(b[i]);

    // get and put values for named variables in MATLAB space

    VecDoub d("dd"); // bind the variable dd (throws error if not exist)
    n = d.size(); // get its number of elements
    VecDoub e(2*n); // make a local vector twice as long
    for (i=0;i<n;i++) e[2*i] = e[2*i+1] = d[i]; // duplicate each value
    e.put("ee"); // copy e to MATLAB variable ee

    // advanced usage (OK, but dangerous! see section "Living Dangerously")

    b[0] = 999.; // modify a rhs argument in MATLAB space
    d[0] = 999.; // modify a named variable directly (not using put())
}

```

See, no mx-functions! Except, of course, `mexFunction` itself, which is always required. Although the above example shows only `VecDoub` exemplars, everything would work fine for other types of vectors, e.g., `VecInt`. The constructors check for type consistency as necessary, and report an error if you have done something type-inconsistent. Let's see it work:

```

>> bb = [1 2 3]
bb =
    1     2     3
>> dd = [4 5 6]
dd =
    4     5     6
>> cc = NRvectorDemo(bb)
cc =
    1     4     9
>> ee
ee =
    4     4     5     5     6     6
>> bb
bb =
    999     2     3
>> dd
dd =
    999     5     6
>>

```

The `nr3matlab.h` vector constructors can access any numerical MATLAB array, flattening it to a vector in the MATLAB storage order ("by columns" or "Fortran-like" or "reverse of odometer"). So, for example,

```

>> bb = [1 2; 3 4]
bb =
    1     2

```

```

    3     4
>> dd = [5 6; 7 8]
dd =
    5     6
    7     8
>> cc = NRvectorDemo(bb)
cc =
    1     9     4    16
>> bb
bb =
  999     2
    3     4
>> dd
dd =
  999     6
    7     8
>>

```

Notice that the "dangerous" operations, because they write directly into MATLAB's workspace, don't flatten the matrices to vectors, since the matrices stay in MATLAB space.

Living Dangerously

Officially, we have to say, "don't do it!". But since the reason you are programming C++ within MATLAB is probably for performance on big data sets, you will probably want to use the operations that we label above as "dangerous". We use them all the time.

What is dangerous about these operations is *not* that they will crash MATLAB or void your warranty (if there were one, always doubtful for software). The danger is that they can have unintended computational side-effects. If you understand these, you can manage them; but if you don't, you might be rather surprised:

```

>> bb = [1 2 3]
bb =
    1     2     3
>> bbsave = bb
bbsave =
    1     2     3
>> dd = [4 5 6]
dd =
    4     5     6
>> cc = NRvectorDemo(bb)
cc =
    1     4     9
>> bb
bb =
  999     2     3
>> bbsave
bbsave =
  999     2     3
>>

```

Knowing that `bb` was going to be modified by `NRvectorDemo`, you carefully saved a copy `bbsave`. But look: *it got modified too*. The reason is that MATLAB's clever memory management uses a "copy on modify" strategy. Had you modified `bb` inside MATLAB, it would have copied `bbsave`'s values at that point. But since we modified it with a "dangerous" method, undetectable by MATLAB, the saved copy never got made.

Moral: If you are going to use `nr3matlab.h`'s ability to modify values directly in MATLAB space, be sure that there are no exact copies of the variable that you care about. If

necessary, you can enforce uniqueness by trickery, such as

```
bbsave = bb;  
bb = bb .* 0.5; bb = bb .* 2;
```

but this should almost never be necessary if you just keep in mind the copy history of your large-data variables.

So, shouldn't this scare you away from living dangerously? Maybe. But suppose you have a MATLAB data vector of length 10^8 , and you intend to apply a series of C++ mex functions to some specific components of the vector. You can *read* those components using non-dangerous `nr3matlab.h` methods. But the only non-dangerous way to *write* them is by the `.put()` method, which copies back the whole vector. If that is just too slow, then the "dangerous" operations are just what you need.

Living Safely

If you worry that you might *accidentally* write into MATLAB space, you should declare vectors derived from rhs arguments or MATLAB named variables with `const`. Then, the compiler will catch any such accidents. Changed lines from the example above are:

```
const VecDoub b(prhs[0]); // bind a rhs argument  
const VecDoub d("dd"); // bind the variable dd  
b[0] = 999.; /* caught as compile time error */  
d[0] = 999.; /* caught as compile time error */
```

If you later decide that you do want to write into MATLAB space after all, you can do something like this:

```
VecDoub &bx = const_cast<VecDoub&>(b);  
bx[0] = 999.; // modifies b, no compile time error
```

Accessing MATLAB Scalars with `nr3matlab.h`

Turn now to MATLAB scalars.

Since scalars like `double` and `int` are fundamental types, we can't overload functionality into their constructors, as we did for vectors. Instead, within `nr3matlab.h` several overloaded versions of a templated function `mxScalar` are defined. (Note that, despite the naming convention, this is a Numerical Recipes, not a MathWorks, function.) These functions let you grab scalar values, either from your mex function's arguments, or from variables in the MATLAB workspace. They also let you bind a C++ variable to any of your function's return values, and copy scalar values into named variables (existing or new) in MATLAB's workspace.

For reference, the functions are declared as follows:

```
template <class T> const T& mxScalar(const mxArray *prhs);  
template <class T> T& mxScalar(mxArray* &plhs);  
template <class T> const T& mxScalar(const char *varname);  
template <class T> void mxScalar(T val, const char *varname);
```

Easier to understand is sample code. (Note that we are now programming in NR3 style,

using typedef'd types like `Doub`. You don't have to do this if you don't want to.)

```
/* mxScalarDemo.cpp */
#include "nr3matlab.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
// normal usage of mxScalar:

    // get a rhs argument by value
    Doub a = mxScalar<Doub>(prhs[0]);

    // bind a reference to a lhs returned value
    Doub &b = mxScalar<Doub>(plhs[0]);
    b = SQR(a); // set a return value, e.g.

    // get a named variable in MATLAB space by value
    Doub c = mxScalar<Doub>("cc");

    // set a named variable in MATLAB space
    Doub d = SQR(c); // e.g.
    mxScalar<Doub>(d,"dd");

// abnormal usage (OK but dangerous! -- see "Living Dangerously")

    // get a non-const reference to a rhs argument
    Doub &e = const_cast<Doub&>( mxScalar<Doub>(prhs[1]) );
    e = SQR(e); // overwrite its value in MATLAB space

    // get a non-const reference to a variable by name
    Doub &f = const_cast<Doub&>( mxScalar<Doub>("ff") );
    f = SQR(f); // overwrite its value in MATLAB space
}
```

The action of this function is to return the square of its first argument and set the named variable `dd` to be the square of the named variable `cc`. It then does two "dangerous" operations: It changes the value of its second right-hand-side argument to its square, and it overwrites a non-const reference to the named variable `ff` with its square. Let's watch it go:

```
>> aa=2; bb=3; cc=4; dd=5; ee=6; ff=7;
>> [aa bb cc dd ee ff]
ans =
     2     3     4     5     6     7
>> bb = mxScalarDemo(aa,ee)
bb =
     4
>> [aa bb cc dd ee ff]
ans =
     2     4     4    16    36    49
>>
```

You can verify that the "after" values are what we expect.

Although the above example shows only `Doub` scalars, everything would work fine for other types (e.g., `Int`). The `mxScalar` functions check for type consistency as necessary, and report an error if you have done something type-inconsistent.

Notice that the `mxScalar` functions are templated, and always require an explicit template argument like `mxScalar<double>()` or `mxScalar<int>()`. This is so that type checking against MATLAB's type can be made automatic, with an error thrown if you try to do something bad.

Also note that most MATLAB scalars are `double`, even things like array sizes that are more logically an integer. So you will often find yourself importing arguments like this:

```
int n = int(mxScalar<Dou>(prhs[0]));
```

Matrices: "Through the Looking Glass" (TTLG)

Matrices are handled pretty much like vectors, above, except for one fly in the ointment, the fact that MATLAB and C++ store the elements of a matrix in different orders. You **must** understand this before using `nr3matlab.h` with `NRmatrix` data types, or you will be rather unhappy: your matrices will sometimes be the transpose of what you expected! `NRmatrix` data types include `NR3` types like `MatDou`, `MatInt`, etc.

The goal is performance: We want to be able to work with potentially large amounts of data from both the MATLAB side and the Numerical Recipes side of the mex interface. What we must **avoid** is unnecessary copying back and forth, not only because of the actual copying, but also because of the overhead involved in rapidly allocating and deallocating large chunks of memory (thus possibly driving MATLAB's memory manager into poor performance).

Therefore, we want to point directly into MATLAB's memory, both to read data and also (with some restrictions) to write it. For scalar values and vectors, we saw that this posed no special problems except understanding the implications of "dangerous" operations and avoiding them, or not, according to your taste. Matrices have the additional problem of storage order: MATLAB is hard-wired to store by columns ("Fortran order" or "reverse of odometer"), while C++ is hard-wired to store by rows ("C order" or "odometer order").

We adopt a "Through the Looking Glass" (TTLG) approach: In our interface, when a MATLAB matrix crosses the interface into C++, what appears on the other side as its **transpose**. When it (or any other C++ matrix) is sent back across the interface to MATLAB, what appears is also the transpose of what was sent. So, for example, a matrix that makes a round trip into C++ and back again (without being modified by C++ code) is unchanged.

The good news about the TLGC approach is that it makes for a very clean and efficient interface. The bad news is that you have to remember that you always receive the transpose of what was sent. Usually you can just code for this on-the-fly: Where you might have written `mymatrix[i][j]`, you instead write `mymatrix[j][i]`, and so forth. In many cases, such as doing something in parallel to all the elements of a matrix, there is nothing to remember at all. Occasionally, when you absolutely need to process an un-transposed matrix, you have to take an explicit transpose. This of course will use memory allocation and add overhead.

It is important to understand that each side of the TTLG interface is self consistent according to its own usual conventions, both as to size and as to subscript numbering (1-based in MATLAB vs. 0-based in C++). So, if `mymatrix` is 3x5 in MATLAB (3 rows, 5 columns), and thus 5x3 in C++ (5 rows, 3 columns), then the following quantities are numerically equal:

```
size(mymatrix,1) = 3 = mymatrix.ncols()
size(mymatrix,2) = 5 = mymatrix.nrows()
mymatrix(1,1) = mymatrix[0][0]
mymatrix(2,4) = mymatrix[3][1]
```

Accessing MATLAB Matrices with nr3matlab.h

For reference, the additions to the templated class `NRmatrix` are:

```
template <class T> class NRmatrix {
  /* ... */
  NRmatrix(const mxArray *prhs); // map Matlab rhs to matrix
  NRmatrix(int n, int m, mxArray* &plhs); // create Matlab lhs and map to matrix
  NRmatrix(const char *varname); // import Matlab variable by name
  void put(const char *varname); // copy NRmatrix to a named Matlab variable
}
```

Easier to understand are some usage examples:

```
/* NRmatrixDemo.cpp */
#include "nr3matlab.h"

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    Int i,j,m,n;

    // matrix rhs arguments and lhs return values

    MatDoub b(prhs[0]); // bind a rhs argument
    m = b.nrows(); // get shape
    n = b.ncols();
    MatDoub c(m,n,plhs[0]); // create a lhs return value of same shape
    for (i=0;i<=m;i++) for (j=0;j<=n;j++) c[i][j] = SQR(b[i][j]);

    // get and put values for named variables in MATLAB space

    MatDoub d("dd"); // bind the variable dd
    m = b.nrows(); // get shape
    n = b.ncols();
    MatDoub e(m,2*n); // make a local vector with twice as many cols
    // duplicate each value (duplicating cols in C++, rows in Matlab!)
    for (i=0;i<=m;i++) for (j=0;j<=n;j++) e[i][2*j] = e[i][2*j+1] = d[i][j];
    e.put("ee"); // copy e to MATLAB variable ee

    // advanced usage (OK, but dangerous! see section "Living Dangerously")

    MatDoub &bx = const_cast<MatDoub&>(b);
    MatDoub &dx = const_cast<MatDoub&>(d);
    bx[0][1] = 999.; // modify a rhs argument in MATLAB space
    dx[1][0] = 999.; // modify a named variable with no copying
}
```

From the MATLAB side, we can use the mex function `NRmatrixDemo` like this.

```
>> mex NRmatrixDemo.cpp
>> bb = [1 2 3; 4 5 6]
bb =
     1     2     3
     4     5     6
>> dd = [7 8 9; 10 11 12]
dd =
     7     8     9
    10    11    12
>> cc = NRmatrixDemo(bb)
cc =
     1     4     9
    16    25    36
>> ee
```

```

ee =
    7     8     9
    7     8     9
   10    11    12
   10    11    12
>> bb
bb =
    1     2     3
   999     5     6
>> dd
dd =
    7    999     9
   10    11    12

```

Notice that the modified element is `b[0][1]` on the C++ side, but `b(2,1)` on the MATLAB side, because of the TTLG approach.

Wrapper Functions That Access Multiple Member Functions in a Class

If you want a single mex file to perform multiple functions (for example, call different member functions within a single class), you'll need to figure out on the C++ side what is intended from the number and type of right-hand side arguments. You can also include arguments of character type that can function as keywords.

For example, a simple wrapper for the Numerical Recipes class `Gaumixmod` (Gaussian mixture model) might have these possible calls on the MATLAB side:

```

>> gmm('construct',data,means) % construct the model from data and means
>> loglike = gmm('step',nsteps) % step the model and return log-likelihood
>> [mean sig] = gmm(k) % return the mean and covariance of the kth component
>> resp = gmm('response') % return the response matrix
>> gmm('delete') % delete the model

```

The corresponding C++ looks like this. Note the use of the functions `mxT()` and `mxT<>()` to test the type of MATLAB arguments. These functions are in `nr3matlab.h`.

```

/* gmm.cpp */
#include "nr3matlab.h"
#include "cholesky.h"
#include "gaumixmod.h"

Gaumixmod *gmm = NULL;

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    int i,j,nn,kk,mm;
    if (gmm) {nn=gmm->nn; kk=gmm->kk; mm=gmm->mm;}
    if (gmm && nrhs == 1 && mxT(prhs[0]) == mxT<Doub>()) {
        // [mean sig] = gmm(k)
        Int k = Int(mxScalar<Doub>(prhs[0]));
        if (nlhs > 0) {
            VecDoub mean(mm,plhs[0]);
            for (i=0;i<mm;i++) mean[i] = gmm->means[k-1][i];
        }
        if (nlhs > 1) {
            MatDoub sig(mm,mm,plhs[1]);

```

```

        for (i=0;i<mm;i++) for (j=0;j<mm;j++) sig[i][j] = gmm->sig[k-1][i][j];
    }
} else if (nrhs == 1 && mxScalar<char>(prhs[0]) == 'd') {
    // gmm('delete')
    delete gmm;
} else if (gmm && nrhs == 1 && mxScalar<char>(prhs[0]) == 'r') {
    // gmm('response')
    if (nlhs > 0) {
        MatDoub resp(nn,kk,plhs[0]);
        for (i=0;i<nn;i++) for (j=0;j<kk;j++) resp[i][j] = gmm->resp[i][j];
    }
} else if (gmm && nrhs == 2 && mxT(prhs[1]) == mxT<Doub>()) {
    // delta loglike = gmm('step',nsteps)
    Int nstep = Int(mxScalar<Doub>(prhs[1]));
    Doub tmp;
    for (i=0;i<nstep;i++) {
        tmp = gmm->estep();
        gmm->mstep();
    }
    if (nlhs > 0) {
        Doub &delta loglike = mxScalar<Doub>(plhs[0]);
        delta loglike = tmp;
    }
} else if (nrhs == 3 && mxT(prhs[0]) == mxT<char>()) {
    // gmm('construct',data,means)
    MatDoub data(prhs[1]), means(prhs[2]);
    if (means.ncols() != data.ncols()) throw("wrong dims in gmm 1");
    if (means.nrows() >= data.nrows()) throw("wrong dims in gmm 2");
    if (gmm) delete gmm;
    gmm = new Gaumixmod(data,means);
} else {
    throw("bad call to gmm");
}
return;
}
}

```

Note that, once instantiated, the pointer `*gmm` is persistent between calls until we explicitly delete it. You'd need a more complicated scheme to instantiate more than one `Gaumixmod` object at a time.

The API Coding Method

You need to read the following sections only if you want to go beyond the capabilities provided by `nr3matlab.h` and use MATLAB C API calls directly.

Tips on MATLAB C API Programming

Mex file programming using the C API interface is well documented on the MathWorks web site. A good starting page is the chapter on [external interfaces](#), as well as the [C API Reference](#) page. Nevertheless, we will make a few additional points here.

- Mex source files **always** have **exactly** the framework

```

#include "mex.h"
/* you may put other stuff here */
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    /* you must put something here */
}

```

The `void mexFunction(...)` line is somewhat equivalent to the `int main(...)`

in ordinary C++ programming. (You *don't* include a `main` routine in `mex` files.)

- Within your `mex` code, you are given the number of arguments, `nrhs`, the expected number of returned values `nlhs`, which may of course be arrays, an array of pointers to the input structures (`mxArrays`), and an array of pointers that **you** populate with output `mxArrays`, as shown in the example.
- Although data in MATLAB is (as the name implies) organized into subscripted matrices and arrays, the C API always presents MATLAB data as a one-dimensional array (above, `indata`). The ordering **is always by columns, not rows**, or, more generally, with the last subscript changing least rapidly ("reverse of odometer" or "Fortran order"). It is easy to find out MATLAB's view of the internal matrix arrangement by API calls like `mxGetM`, `mxGetN`, `mxGetNumberOfDimensions`, `mxGetDimensions`, and a few others. But, once you do this, it is up to you to locate by subscript(s) any desired elements. (This is basically what the `nr3matlab` interface is designed to make easier.) MATLAB's Fortran storage ordering has other implications that are discussed in the section [Matrices: "Through the Looking Glass"](#), above.
- The `mxGetData` function returns a void pointer to the data. It is up to **you** to cast it to the intended C++ type. Above, we trusted the user to send us `double` data, which is MATLAB's default type (even for things that you might otherwise expect to be integers). That is, of course, bad programming. We should have used API functions like `mxIsDouble` or `mxGetClassID` to get MATLAB's opinion as to the data type, and then aborted with `mexErrMsgTxt` if it is not the expected enum value `mxDOUBLE_CLASS`.

If you plan to use the API interface, you should peruse the file `matrix.h` located in the `/extern/include` directory of your MATLAB installation. Among other things, this file defines the class names like `mxDOUBLE_CLASS`.

Numerical Recipes Using the MATLAB C API

Although we prefer to use the NR3 coding method, above, you can perfectly well use Numerical Recipes Third Edition (NR3) code, and the standard `nr3.h` include file, with the MATLAB C API interface.

For example, here is a `mex` file that makes NR3's generalized Fermi-Dirac integral routine available to MATLAB. (The NR3 routine does a fancy quadrature, using the DE rule.)

```

/* fermidirac.cpp */
#include "mex.h"

#include "nr3.h" // see http://www.nr.com/dependencies
#include "quadrature.h"
#include "derule.h"
#include "fermi.h"

Fermi *fermi = NULL;

// usage: f = fermidirac(k, eta, theta)

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    double k = mxGetScalar(prhs[0]);
    double eta = mxGetScalar(prhs[1]);
    double theta = mxGetScalar(prhs[2]);
    double f;
    if (fermi == NULL) fermi = new Fermi();
    if (theta < 0. || k <= -1.) {

```

```

    delete fermi;
    mexErrMsgTxt("bad args to fermidirac (also kludge destructor)");
} else {
    f = fermi->val(k,eta,theta);
}
plhs[0] = mxCreateDoubleScalar(f);
}

```

Compile this with the mex command, then put it in your MATLAB working directory.

```

>> f = fermidirac(3/2, 0.7, 0.2)
f =
    2.3622

```

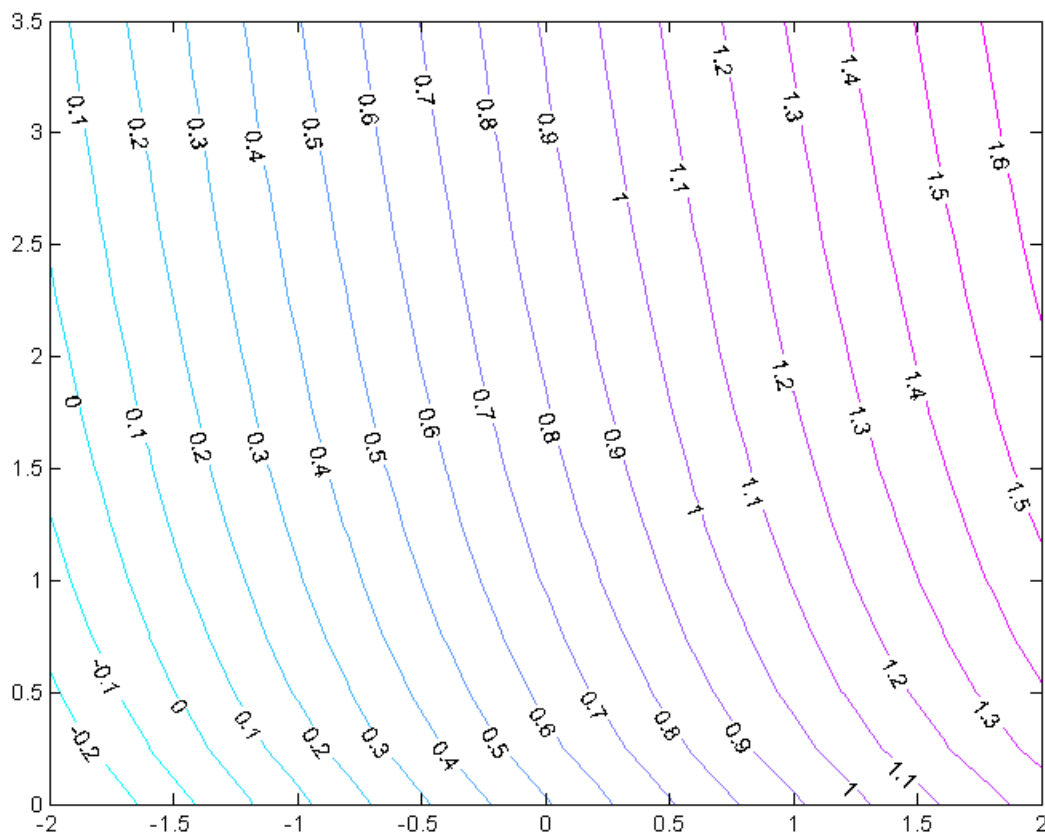
Now you can also do powerful MATLAB things, like making a contour plot of the logarithm of the function.

```

% fermidirac demo
[eta theta] = meshgrid(-2:.2:2,0:.25:3.5);
f = arrayfun(@(e,t) fermidirac(5/2,e,t), eta, theta);
flog = log10(f);
[C,h] = contour(eta,theta,flog,-0.2:.1:1.7);
set(h,'ShowText','on','TextStep',get(h,'LevelStep')*2)
colormap cool

```

(see MATLAB help for "contour" to understand this) which produces



Because this example passes and returns scalars only, the file `fermidirac.cpp` would not look too different in the NR3 coding method: `mxScalar` would replace `mxGetScalar`,

throw would replace `mexErrMsgTxt`, and `mxCreateDoubleScalar` would disappear (since `f` could be declared directly as the return value `plhs[0]`).

Appendix: Using Microsoft Visual Studio

In all sections above we've done the compilation and linking from the MATLAB console, using the `mex` command. We saw in the section [Do Just Once: Introduce Matlab to Your Compiler](#) that we could cause the `mex` command to use our favorite compiler. But we were not getting the convenience of compiling within an integrated development environment (IDE) such as Microsoft Visual Studio. This section explains how to do this for MATLAB 7.1+ and Visual Studio 2005. Other versions should be similar.

We follow the useful [article](#) by "abn9c". We suppose that your function is called `MyFunc` (make the obvious changes below for whatever it is actually named).

Follow these steps:

1. Create a new C++ Win32 console application named `MyFunc`. If you are offered a choice of application type, choose `DLL`.
2. In the menu `Project/Add Existing Item...` add the file `mexversion.rc`, located in your Matlab install directory (e.g., at `C:\Program Files\MATLAB\R2007b\extern\include`). You'll need to show files of type "Resource Files" to see this file.
3. Create a `.def` file with this text

```
LIBRARY MyFunc.mexw32
EXPORTS mexFunction
```

and add it to the Solution. You can create it with the menu `File/New/File...`, choosing "Header File `.h`" type. But after you enter the above text, save the file as `MyFunc.def`, not `MyFunc.h`. Add it to the Solution via menu `Project/Add Existing Item...`

Now go to the menu `Project/MyFunc Properties for the following steps:`

4. Select Configuration: All Configurations
5. Under Configuration Properties/General/Configuration Type, choose `.dll` if it is not already selected.
6. Under Configuration Properties/C++/General/Additional Include Directories, add the MATLAB `extern\include` directory (e.g., `C:\Program Files\MATLAB\R2007b\extern\include`).
7. Under Configuration Properties/C++/Preprocessor/Preprocessor Definitions, add `MATLAB_MEX_FILE`
8. Under Configuration Properties/Linker/General/Output File, change `.dll` to `.mexw32`. You could also here change `$(OutDir)` to the full path of your MATLAB working directory. (This will save you from having to recopy the output file every time you recompile.)
9. Under Configuration Properties/Linker/General/Additional Library Dependencies, add `extern\lib\win32\microsoft` (e.g., `C:\Program Files\MATLAB\R2007b\extern\lib\win32\microsoft`).

10. Under Configuration Properties/Linker/Input/Additional Dependencies add libmx.lib, libmex.lib, and libmat.lib. You don't need paths, just the names.
11. Under Configuration Properties/Linker/Input/Module Definition File add MyFunc.def (that is, the .def file that you already created). You don't need a path, just the file name.

You can now "OK" out of "MyFunc Property Pages".

12. In the main MyFunc.cpp window, you can now erase whatever is "helpfully" provided, and enter your mex code. A minimal skeleton is

```
#include "stdafx.h"
#include "nr3matlab.h"
void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[]) {
    printf("Hello, world!\n"); // remove this line after testing
}
```

13. Be sure that [nr3matlab.h](#) is in your project directory or include path.
14. If you now do the menu Build/Build Solution, your file should compile and link. If necessary, move the file MyFunc.mexw32 to your MATLAB working directory. Test from MATLAB:

```
>> MyFunc()
Hello, world!
```

If you execute MyFunc from within MATLAB between compilations, you might need to do

```
>> clear MyFunc
```

before the linker can overwrite a new file, or before your new executable is recognized by MATLAB.

Microsoft Visual Studio doesn't make it easy to copy all the above settings from one project to another. There are various commercial tools for this, so that you don't have to keep going through all the above steps for each new mex function that you write. (We use one called [CopyWiz](#).)