RogueWave SOFTWARE

Accelerating Great Code

PROTOTYPE TO PRODUCTION WITH IMSL NUMERICAL LIBRARIES

In the development of software that requires advanced math, statistics, or analytics, there is often a disconnect early in the development process. This occurs at the transition from algorithm selection and testing to the beginning of coding in the actual compiled language. We refer to this as the prototype to production transition.

Typically, mathematicians, statisticians, and data scientists do the initial algorithm development. Often their preferred tool is a scripting language like R or Matlab where you don't need to worry about memory allocation, data types, or exception handling. After algorithm selection and validation is complete, the algorithm functionality must be implemented and integrated into the production code. Even seemingly trivial functions in the scripting languages can take significant effort to implement in compiled code. The very things that make scripting tools so easy to use leads to potential pitfalls when writing production code.

Finally, there is the critical issue of validating the compiled version. With two implementations of complex algorithms, script and compiled code, there are often differences in results. Compounding the issue, the differences vary depending on the input data. Resolving these issues may lead to modification of the algorithm, not just the implementation. This bogs down the development process.

To address these issues during prototype to production, we are presenting a method to run IMSL Numerical Libraries routines in R or Matlab. The goal is not to replace the algorithm developer's tool of choice but to run a compiled version of the code in parallel. Pitfalls can be caught early, and data discrepancies can be resolved quickly by running the script version and compiled version side by side.

In this paper, we describe how to call IMSL routines directly from R and Matlab. In general, use of IMSL reduces software production time and costs. Moreover, the technique we describe streamlines the process further by easing the transition from algorithm development to production code. We show how to integrate IMSL with common tools used by applied mathematicians, data scientists, and advanced algorithm groups.

IMSL

IMSL Libraries save development time by providing optimized mathematical and statistical algorithms that can be embedded into C, C++, .NET, Java, and Fortran applications, including many databases. IMSL enhances application performance, reliability, portability, scalability, and maintainability as well as developer productivity. IMSL Libraries support a wide range of languages as well as hardware and operating system environments including Windows, Linux, and many UNIX platforms.

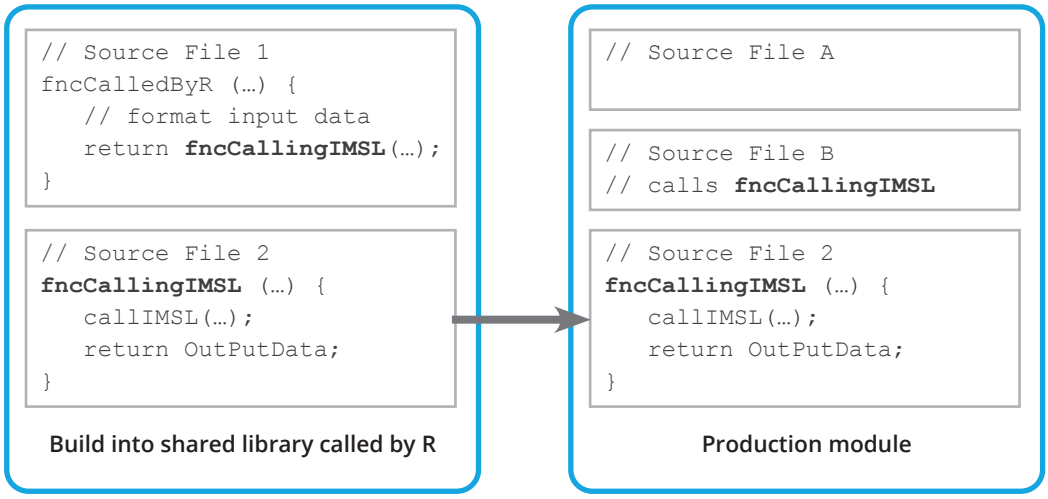

Figure 1. Prototype to production schematic
Both prototype and production versions are built by linking to the same IMSL library.

Figure 1 illustrates the prototype to production process using R and IMSL. In the first phase, a library is built that exports a function that will be called in R. To facilitate the later transition to a production module, it's useful to separate the implementation of this first stage into two source files. The first source file acts as the interface to R and as a surrogate for the production code that gets and prepares data to be called with IMSL. The first source file implements the function exported to R and the second source file implements the function that calls IMSL routines. After testing is complete in the prototype phase, the code in "Source File 2" can be used without modification in the production module. Any numerical issues or output discrepancies that arise during unit testing or beyond can quickly be assessed by comparison with the R version. This eliminates bottlenecks in the development process.

IMPLEMENTATION

IMSL is available in C, C#, Fortran, and Java language versions. The first three of these languages can be grouped together due to their ability to implement shared libraries with C style linkage (no name mangling). In Java, there is no equivalent to a shared library that's callable by any operating system process. The requirement to run within a Java Virtual Machine (JVM) qualitatively separates Java from the other languages. Strictly speaking, you could use the Java Native Interface (JNI) to call Java in a similar manner as shared libraries. However, this approach is fraught with difficulties, and as we show below there is a much easier alternative to the JNI with R and Matlab.

At a high level, the C style and Java implementations are quite similar. The goal is to develop methods to call IMSL from R or Matlab and to smoothly transition to production testing. Towards that end, it's useful to have one source file for the function exported to R/Matlab. A second source file implements the calls to IMSL. To be clear the first source file implements functions called by R/Matlab and the second source file implements functions that make calls to IMSL. After prototyping is complete, this second file can be moved with little or no modification to the production code.

	C style	Java

Source File 1
(R/Matlab interface)
	`extern "C" __declspec(dllexport) fncName1`
	`extern "C" __declspec(dllexport) fncName2`
	`//parameter input/output`
		`public static fncName1`
		`public static fncName2`

Source File 2
(Later moved to production)
	`#include "imsl.h"`
	`#include "imsls.h"`
	`//  actual calculations`
		`import com.imsl.<algo>`
		
		`// calculation class`

Figure 2. C and Java comparison

In Figure 2, we outline the design in C and Java. For the C case on UNIX variants, the declspec is omitted. For the Java case, the first source file could be a class with public static methods to handle calls from R/Matlab. Then a second class implements the calculations.

In the examples shown here, our goal is to implement shared libraries and Java classes so that users can call IMSL and JMSL (IMSL for Java) directly from R/Matlab. For that reason, after the first example we implement all functionality in a single source file. Our emphasis with these examples is to demonstrate how to pass data, in particular 2D arrays, between R/Matlab and IMSL/JMSL. Towards that end, we make repeated use of IMSL Singular Value Decomposition (SVD).

C style call

As mentioned above, C, C#, and Fortran code can be packaged into shared libraries with C style linkage for exported functions. This is also true for C++. We'll describe particular details for actual C code but the differences when implementing C style libraries for the other languages is not significant.

R

R has a few ways to call external code. One of those ways uses the `.Call()` function. This function targets building R extensions and enables calling R from the external code. In this paper, we'll use the simpler `.C()` function. It is limited but well suited for our particular purpose. For example, the external functions don't need to know anything about R and all function parameters are primitive types. With actual C code, all parameters must be pointers, even scalar input.

Below is an example that calculates the least-squares solution to the equation **Ax**=**b** using IMSL SVD. Here **A** is an **m*n** matrix, **x** is an n-element vector, and **b** is an m-element vector. This is on a Windows system so the result is a dynamic link library (DLL).

In this example, the exported function called by R is `rwSvdLsq`, defined in source file svdLsq.cpp. This function handles parameters passed from and to R, and it calls a second function that does the actual calculations. This second function is called `svdLsq` and is defined in source file svdLsq.cpp.

```
// declaration of function defined in a separate source file
void svdLsq(double*, int, int, double*, double*, double*);

extern "C" {
    __declspec(dllexport) void rwSvdLsq(double *A, int *nm, int *nn,
                                        double *b, double *x, double *s)
    {
          int m = *nm;
          int n = *nn;

          svdLsq(A, m, n, b, x, s);
    }
    // + other functions
}// IMSL_R64.cpp
```

```
#include <cstring>
#include "imsl.h"
#include "imsls.h"

void svdLsq(double *A, int m, int n, double *b, double *x, double *s)
{
    int p = n < m ? n : m;

    double* pInv = new double[m*p];

    // get the singular values and the pseudo-inverse in one go
    imsl_d_lin_svd_gen(m, n, A,
          IMSL_RETURN_USER, s,
          IMSL_INVERSE_USER, pInv,
          IMSL_INV_COL_DIM, m,
          0);

    // x = pInv * b
    double *xloc = imsl_d_mat_mul_rect("A*x",
          IMSL_RETURN_USER, x,
          IMSL_A_MATRIX, n, m, pInv,
          IMSL_X_VECTOR, m, b,
          0);

    delete[] pInv;

}// svdLsq.cpp
```

The SVD example is useful because it demonstrates an important difference in how matrices are stored differently in R/Matlab (column major) compared to C/C++ (row major). The ambiguity arises from representing a two dimensional object in one dimension. In the actual call from R to our function, we pass in the transpose but specify the correct rows and columns. See the IMSL documentation for further details on the functions used in these examples.

In the R workspace, we need to create a wrapper function for each function from a loaded library that will be called. The R `.C()` function needs the imported function name, followed by the parameter list.

R commands	R output

`# load the DLL`
`dyn.load("IMSL_R64.dll")`
 	
`# create a wrapper to IMSL SVD based routine`
`svdLsq <- function(A,m,n,b){`
`  S<-.C("rwSvdLsq",`
`  as.double(A),`
`  as.integer(m),`
`  as.integer(n),`
`  as.double(b),`
`  x = double(n),`
`  s = double(min(m,n)))`
`  list(x=S$x,s=S$s)`
`}`
 	
`# create a test matrix`
`a<-c(1,3,4,2,1,1,2,2,3,1,5,2,1,1,1,3,2,2,4,3`
`,4,1,2,3)`
`A<-matrix(a,6,4)`
 	
`b <- c(11,6,8,1,5,8)`
`# solve for x in Ax=b`
 	
`# output is a list of two vectors: $x, $s`
`Res<-svdLsq2(t(A),6,4,b)`
	`> A`
	`  [,1] [,2] [,3] [,4]`
	`[1,] 1 2 1 4`
	`[2,] 3 2 1 3`
	`[3,] 4 3 1 4`
	`[4,] 2 1 3 1`
	`[5,] 1 5 2 2`
	`[6,] 1 2 2 3`
	
	`> Res$s # singular values`
	`[1] 11.485 3.270 2.653 2.089`
	
	`> Res$x # exact solution: -1,0,0,3`
	`[1] -1.00e+00 8.88e-16`
	`-6.66e-16 3.00e+00`
	
	`> A %*% Res$x # test: does Ax=b?`
	`[,1]`
	`[1,] 11`
	`[2,] 6`
	`[3,] 8`
	`[4,] 1`
	`[5,] 5`
	`[6,] 8`

Matlab

In Matlab, there are two options for calling external routines, using MEX-Files or the loadlibrary utility. The MEX-File API is better documented and is probably more familiar to Matlab users than loadlibrary. For these reasons, we won't discuss loadlibrary here.

With MEX-Files, you create a specialized shared library that must implement a function called `mexFunction()` that is the entry point for Matlab to the library. The shared libraries that you create each have their unique name but their interface to Matlab is always `mexFunction()`.

Again, we use the SVD but here we return the matrix decomposition and don't use it for least squares. We also have to deal with the ambiguity of representing matrices as 1D arrays. Matlab is column major and IMSL C is row major, the easiest way to deal with this is to pass in transposed arrays as input. Any output arrays then need to be transposed in the Matlab workspace.

```
#include "mex.h"
#include "imsl.h"

void mSVD(double* A, int m, int n, double* s, double* U, double* V)
{
    imsl_d_lin_svd_gen(m, n, pData,
          IMSL_RETURN_USER, s,
          IMSL_U_USER, U,
          IMSL_V_USER, V, 0);
}

void mexFunction(int nlhs, mxArray *plhs[], int nrhs, const mxArray *prhs[])
{
          // A = U S V
          // this analysis corresponds to what Matlab calls the "economy size"
          // [U,S,V]=svd(A,0);

          // this is the contiguous length
          int nCols = mxGetM(prhs[0]);
          int nRows = mxGetN(prhs[0]);

          int nn = nCols < nRows ? nCols : nRows;

          // The user needs to pass in the transpose of the Matlab array
          double* A = mxGetPr(prhs[0]);

          double* U = 0;
          double* V = 0;
          double* s = 0;

          mwSize dims[2] = { nCols, 1 };
          plhs[0] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
          s = mxGetPr(plhs[0]);

          dims[0] = nn; dims[1] = nRows;
          plhs[1] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
          U = mxGetPr(plhs[1]);

          dims[0] = nCols; dims[1] = nCols;
          plhs[2] = mxCreateNumericArray(2, dims, mxDOUBLE_CLASS, mxREAL);
          V = mxGetPr(plhs[2]);

          /* Compute SVD */
          mSVD(A, nRows, nCols, s, U, V);

}
```

In this case, we save the built DLL as mextest.mexw32. Matlab requires that extension and complete details on building mexFunctions on Windows are in the appendix. An explicit call in the workspace can be thought of as following this mapping:

[s,UT,VT]=mextest(A'); **->** mexFunction(3, { s,UT,VT }, 1, {A'})

Below we show a workspace session using the same input array as in the R example.

```
>> A
A =
     1     2     1     4
     3     2     1     3
     4     3     1     4
     2     1     3     1
     1     5     2     2
     1     2     2     3

>> [s,UT,VT]=mextest(A');   % pass in the transpose of A
>> U=UT'; V=VT';    % transpose the 2D array output
>> U,s,V
U =
     -0.38048      0.11967      0.43908      -0.5654
     -0.40375      0.34511    -0.056576      0.21478
     -0.54512      0.42926     0.051393      0.43214
     -0.26478     -0.06832     -0.88386     -0.21525
     -0.44631     -0.81683       0.1419      0.32127
     -0.35463     -0.10215   -0.0043184      -0.5458
s =
       11.485
       3.2698
       2.6534
       2.0887
V =
     -0.44429      0.55553     -0.43538      0.55175
     -0.55807      -0.6543      0.27746      0.42834
     -0.32439     -0.35136      -0.7321     -0.48513
     -0.62124      0.37393       0.4444     -0.52607

>> [U,s,V]=svd(A,0)   % Call Matlab's SVD
U =
     -0.38048      0.11967      0.43908       0.5654
     -0.40375      0.34511    -0.056576     -0.21478
     -0.54512      0.42926     0.051393     -0.43214
     -0.26478     -0.06832     -0.88386      0.21525
     -0.44631     -0.81683       0.1419     -0.32127
     -0.35463     -0.10215   -0.0043184       0.5458
s =
       11.485            0            0            0
            0       3.2698            0            0
            0            0       2.6534            0
            0            0            0       2.0887
```

```
V =
     -0.44429      0.55553     -0.43538     -0.55175
     -0.55807      -0.6543      0.27746     -0.42834
     -0.32439     -0.35136      -0.7321      0.48513
     -0.62124      0.37393       0.4444      0.52607
```

If you examine the output of the two versions of the SVD, you'll notice that the singular values agree. This is good as the singular values of a matrix are unique. However, the U and V matrices are not unique. They each need to form a basis of two different vector subspaces but their matrix representations are not unique, and we see that here.

Java

For Java applications, prototyping with R/Matlab and JMSL is slightly different. Matlab implements its own JVM, as does R with the rJava package. In both cases, an additional run-time layer is added to the environment such that the Java byte code, in particular Java class files, can be loaded into interactive terminal sessions. With this capability, you can call static methods directly. For non-static methods, you first need to instantiate a class object and then call any of its public methods.

R

In the first example, we implement a very thin wrapper to the JMSL SVD routine. Again, our emphasis is on demonstrating how to pass 1D and 2D arrays as input and output. In a proper prototype that's planned for use in software development, the parameter input/output would be handled by one class (probably a static method), and the actual calculations performed in a second class.

```
package rJav;

import com.imsl.math.SVD;

public class Rsvd {
    SVD svdobj; // the one data member

    Rsvd() { svdobj = null; }

    Rsvd( double[][] A ) {
          try {
                svdobj = new SVD(A);
          } catch (DidNotConvergeException e) {
                System.err.println(e);
                e.printStackTrace();
          }
    }

    double[] getS() { return svdobj.getS(); }
    double[][] getU() { return svdobj.getU(); }
    double[][] getV() { return svdobj.getV(); }
}
```

Below are the contents of an R script file and its output. With the rJava package, Java classes are created with `.jnew()` and methods are called by `.jcall()`. 2D arrays passed as input to Java methods and classes need to be "cast" using the function `.jarray()`.

R commands

```
library("rJava", lib.loc="~/R/win-library/3.2")    # load the rJava library

.jinit()                                           # initialize the JVM

.jaddClassPath("C:/RogueWave/IMSL/jmsl700/lib/jmsl.jar")   # update the classpath
.jaddClassPath("~/rJav/bin/")

# create a matrix
a<-c(1,3,4,2,1,1,2,2,3,1,5,2,1,1,1,3,2,2,4,3,4,1,2,3)
A<-matrix(a,6,4)
#put array in form to be passed from R to Java as double[][]
jA<- .jarray(A,dispatch=TRUE)


# create a new Java object
svdobj = .jnew("rJav/Rsvd",jA)

s<-.jcall(svdobj,"[D","getS")
UU<-.jcall(svdobj,"[[D","getU")
VV<-.jcall(svdobj,"[[D","getV")

# 2D arrays of java objects confuse rJava, here's how to unpack
U <- do.call(rbind, lapply(UU, .jevalArray))
V <- do.call(rbind, lapply(VV, .jevalArray))
s
U
V
```

R output

```
> s
[1] 11.485018  3.269751  2.653356  2.088730

> U
             [,1]          [,2]          [,3]         [,4]          [,5]          [,6]
[1,]   -0.3804756    0.11967099    0.439082824   -0.5653996    0.02431152   -0.57258686
[2,]   -0.4037537    0.34511084   -0.056576185    0.2147756    0.80890059    0.11929742
[3,]   -0.5451205    0.42926489    0.051392693    0.4321442   -0.57232765    0.04033092
[4,]   -0.2647843   -0.06831953   -0.883860867   -0.2152537   -0.06252092   -0.30621670
[5,]   -0.4463101   -0.81682762    0.141899675    0.3212696    0.06213378   -0.07993527
[6,]   -0.3546287   -0.10214740   -0.004318444   -0.5458002   -0.09879463    0.74573958

> V
             [,1]         [,2]         [,3]         [,4]
[1,]   -0.4442941    0.5555313   -0.4353790    0.5517544
[2,]   -0.5580672   -0.6542987    0.2774569    0.4283361
[3,]   -0.3243861   -0.3513606   -0.7320995   -0.4851285
[4,]   -0.6212386    0.3739303    0.4444020   -0.5260662
```

You'll notice that the input 2D array doesn't need to be transposed as was the case for the C/C++ implementation. Also, JMSL calculates the full SVD, and in this case the four singular values would be the diagonal elements of an otherwise all-zeros 6x4 matrix.

Matlab

Using Matlab to call Java is very similar to R but the implementation is actually cleaner with regards to passing 2D arrays. We use the same Java wrapper to the JMSL SVD function.

Matlab session

```
>> A
A =
     1     2     1     4
     3     2     1     3
     4     3     1     4
     2     1     3     1
     1     5     2     2
     1     2     2     3

>> svdobj = javaObject('rJav.Rsvd',A);   % create a new Java object
>> U = javaMethod('getU', svdobj)
```

```
U =
     -0.38048      0.11967      0.43908      -0.5654      0.024312      -0.57259
     -0.40375      0.34511    -0.056576      0.21478        0.8089        0.1193
     -0.54512      0.42926     0.051393      0.43214      -0.57233       0.04033
     -0.26478     -0.06832     -0.88386     -0.21525      -0.06252      -0.30622
     -0.44631     -0.81683       0.1419      0.32127      0.062134     -0.079935
     -0.35463     -0.10215   -0.0043184      -0.5458      -0.09879       0.74574

>> s = javaMethod('getS', svdobj)
s =
       11.485
       3.2698
       2.6534
       2.0887

>> V = javaMethod('getV', svdobj)
V =
     -0.44429      0.55553     -0.43538      0.55175
     -0.55807      -0.6543      0.27746      0.42834
     -0.32439     -0.35136      -0.7321     -0.48513
     -0.62124      0.37393       0.4444     -0.52607
```

Consult the Matlab documentation for instructions on configuring the internal Java classpath. In our case, we added the imsl.jar file and our own rJav/Rsvd package to the Matlab static classpath.

SUMMARY

This paper described a set of practices to speed the development of software that requires sophisticated math, statistics, or analytics. In particular, precise details were provided on how to incorporate IMSL Numerical Libraries into very early prototyping with tools such as R or Matlab. This method makes the transition to production development faster and easier.

APPENDIX

Building DLLs in Windows

Since Microsoft Visual Studio is such a common tool for Windows developers and users, we'll show explicit steps for building DLLs. It's important to remember that we're creating DLLs that require the IMSL DLLs. Windows searches for DLLs in this order:

1. The directory where the executable module for the current process is located
2. The current directory
3. The Windows system directory
4. The Windows directory
5. The directories listed in the PATH environment variable

R

Below is a procedure for 64-bit IMSL with Visual Studio 2012 that creates a DLL that can be loaded by R. This is modified from the file README_winms120x64.txt that is in <VNI_DIR>\imsl\cnl850\winms120x64\notes

1) Start the Microsoft Visual Studio 2012 Developer Environment

Start →
All Programs →
Microsoft Visual Studio 2012 →
Visual Studio 2012

2) Create a New Project

File →
New →
Project

- Choose the template Visual C++ → Win32 Console Application
- Change the name and location of the project as needed
- Verify that the "Create directory for solution" box is checked

Click OK

From the "Win32 Application Wizard" window that is created click on "Application Settings"

Choose "**DLL**" and "Empty Project", then click "Finish"

3) Add the Include Files Directory

From Solution Explorer right click on the project name

Choose Properties

Choose Configuration Properties →
C/C++ →
General

To the right of "Additional Include Directories" add the pathname to the appropriate include files
<VNI_DIR>\imsl\cnl850\winms120x64\include

Click OK

4) Add the Libraries to the Project

From Solution Explorer right click on the project name

Choose Add →
Existing Item

Browse to the directory
<VNI_DIR>\imsl\cnl850\winms120x64\lib

Choose the files imslcmath_imsl_dll.lib and imslcstat_imsl_dll.lib

Click "Add"

5) Copy Win32 project settings into x64 project configuration

From Solution Explorer right click on the project name

Choose Properties

Click Configuration Manager to open the Configuration Manager dialog box

Click the Active Solution Platform list, and then select the <New...> option to open the New Solution Platform dialog box

Click the Type or select the new platform drop-down arrow, and then select x-64 platform

Click OK. The platform you selected in the preceding step will appear under Active Solution Platform in the Configuration Manager dialog box

Click Close in the Configuration Manager dialog box, and then click OK in the Property Pages dialog box

6) Add your source file(s)

From Solution Explorer right click on the project name
Choose Add →
New Item

Enter the name of your source file and click OK

As a simple test, below is source code for a function that gets the IMSL version string from the IMSL libraries. You must specify `extern "C"` and `__declspec(dllexport)`.

```
#include "imsl.h"
#include "imsls.h"

extern "C" __declspec(dllexport) void getVersion(char **version_string) {
          *version_string = imsl_version(IMSL_LIBRARY_VERSION);
}
```

Once your DLL is built (in this case **IMSL_R64.dll**) you need to load it in to R

```
> dyn.load("IMSL_R64.dll")
```

In R we create a wrapper function

```
imsl_ver <- function() {
  result <- .C("getVersion",  imslversion="")
  return(result$imslversion)
}
```

And then call the function

```
> imslver<-imsl_ver()
> imslver
[1] "IMSL C Math Library Version 8.5.0"
>
```

All parameters passed to functions in the DLL must be of pointer type, even input scalars. The SVD example described above illustrates this.

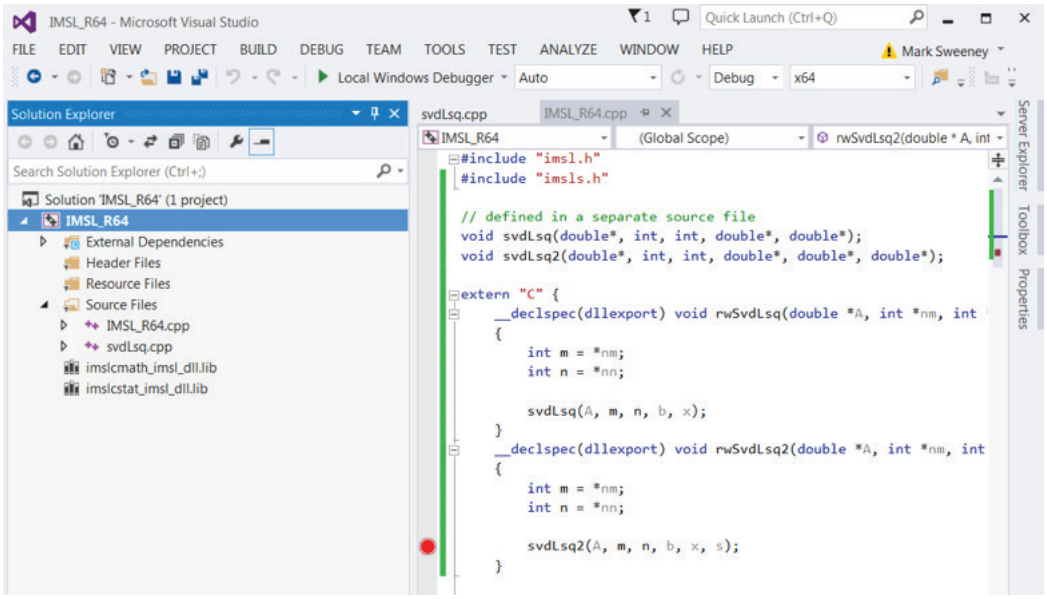

Figure 3. Screenshot from a Visual Studio DLL project

Matlab

The instructions below are for 32-bit IMSL and 32-bit Matlab (they must be the same). Steps for 64-bit versions are nearly identical (the differences shown in square brackets) with the additional requirement that a final step of creating an x64 platform build in Visual Studio.

1) Start the Microsoft Visual Studio 2012 Developer Environment
Start →
All Programs →
Microsoft Visual Studio 2012 →
Visual Studio 2012

2) Create a New Project
File →
New →
Project

- Choose the template Visual C++ → Win32 Console Application
- Change the name and location of the project as needed
- Verify that the "Create directory for solution" box is checked

Click OK

From the "Win32 Application Wizard" window that is created click on "Application Settings"

Choose "DLL" and "Empty Project" and then click "Finish"

3) Add the Include Files Directory

From Solution Explorer right click on the project name

Choose Properties

Choose Configuration Properties →
C/C++ →
General

To the right of "Additional Include Directories" add the pathname to the appropriate include files
<VNI_DIR>\imsl\cnl850\winms120i32\include [<VNI_DIR>\imsl\cnl850\ winms120x64\include]
<MATLAB_ROOT>\extern\include;

Click OK

4) Add the Libraries to the Project

From Solution Explorer right click on the project name

Choose Add →
Existing Item

Browse to the directory
<VNI_DIR>\imsl\cnl850\winms120i32\lib [<VNI_DIR>\imsl\cnl850\winms120x64\lib for 64 bit]

Choose the files imslcmath_imsl_dll.lib and imslcstat_imsl_dll.lib

Click "Add"

Choose Add →
Existing Item

Browse to the directory
<MATLAB_ROOT>\extern\lib\win32\microsoft [<MATLAB_ROOT>\extern\lib\win64\microsoft]

Choose the files libmx.lib, libmex.lib, and libmat.lib

Click "Add"

5) Change output extension

From Solution Explorer right click on the project name

Choose Configuration Properties →
General

Change "Target Extension" from .dll to .mexw32 [.mexw64 for 64bit]

6) Specify export name

From Solution Explorer right click on the project name

Choose Configuration Properties →
Linker →
Command Line →
Additional Options

Add `/export:mexFunction`

7) Create a mexFunction() in a source file

The project must implement a function that is the entry point
from Matlab to the DLL

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

8) Add specific function calls

It is useful for transitioning to stand alone applications
to add IMSL specific calls in separate source files from the
one that implements mexFunction.

Your mexFunction() should
1) get float or double arrays (and scalars) from the input mxArray pointer prhs
2) pass these to the functions that call IMSL
3) copy any output data to the output mxArray pointer plhs

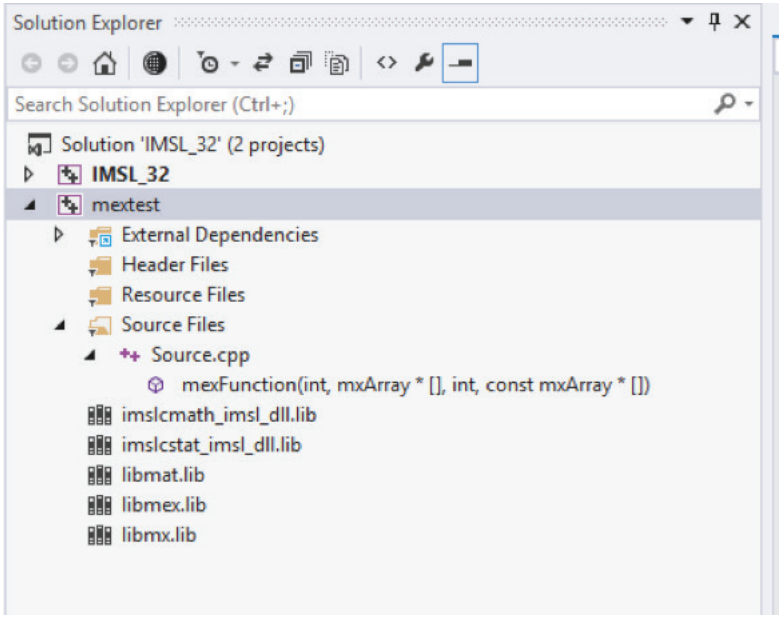

Figure 4. mexFunction() IMSL project in Visual Studio

Java Reflection

Both R and Matlab make use of the Java reflection API to create and modify objects at runtime. The reader may have noticed that our Java example class was a very thin wrapper to the JMSL SVD class. In fact, you can instantiate JMSL classes and call their methods. In addition, you can also directly call static methods. The usefulness of this for rapid prototype to production is dubious but it does allow for direct instantiation of JMSL objects, and method calls, in R or Matlab.

REFERENCES

Matlab.

The R Project for Statistical Computing.

rJava.

Rogue Wave provides software development tools for mission-critical applications. Our trusted solutions address the growing complexity of building great software and accelerates the value gained from code across the enterprise. The Rogue Wave portfolio of complementary, cross-platform tools helps developers quickly build applications for strategic software initiatives. With Rogue Wave, customers improve software quality and ensure code integrity, while shortening development cycle times.