Calling the IMSL C Numerical Library from Java™

Introduction

This report describes how to use the Java Native Interface (JNI) to call an IMSL C Numerical Library (CNL) function from Java. Using JNI, Java calls C interface code, which then calls the CNL function. There are three basic steps in this process: creating the Java class, creating the C interface function, and creating a dynamic library to contain the C interface.

The C interface code provides a method for the input and output arguments to be converted from Java types to the types required by the CNL function. C functions are provided as part of JNI to facilitate passing arguments from Java to C. Three examples are supplied to demonstrate the use of these functions in passing various argument types, such as arrays, strings, or functions, to CNL along with instructions for building on the Solaris and Windows platforms. Since Java code cannot be "linked" to the C object code, the C interface code must be built into a dynamic library for Java to load.

Since Java does not support optional arguments, the C interface code must have a fixed number of arguments. Optional arguments to the CNL function must be handled as required arguments to the C interface code. The examples demonstrate this method of handling optional arguments.

By default, CNL outputs error messages to the console, and will stop execution on a terminal or fatal error. To override these defaults and control error handling from within the Java code, C interface code would have to be included to call CNL function error_options to change the default action and/or error_code to return the error code number to Java.

1. C Interface Function Prototype

In Java, the "native" modifier is used to describe the method implemented as C interface code. After the Java code has been compiled, javah may be used to generate the C header file required by the C interface. This header file contains the function prototype to be used in the C interface code. Note that the name of the function and the argument list generated by javah is not the same as that declared in the Java "native" statement. The arguments added by javah are needed for JNI.

Refer to the examples or a JNI reference for more information. One Internet reference, provided by Sun, can be found at http://java.sun.com/products/jdk/1.2/docs/guide/jni/

2. Argument Passing

The input and output arguments to a CNL function must be converted from Java types to the C data type required by CNL. In most cases JNI functions are used to make this conversion. The method used is dependent on the argument type, as discussed below.

- **a scalar argument**

Java "int", "float", and "double" variables can be passed directly to the C function from the interface code.

- **an array argument**

The name of an array in C is a pointer to the first element of the array. The C pointer to a Java array can be obtained by using the Get<Type>ArrayElements functions of JNI, where <Type> is the type of array. For example, the following C code will set a C pointer to the Java double array "a".

```
jdoubleArray a;
jdouble *aptr;
aptr = (*env)->GetDoubleArrayElements(env, a, 0);
```

A two dimensional Java array is treated as an object array by JNI. JNI function GetObjectArrayElement is used to convert the Java array elements to a one dimensional C array. For example, to get row i of Java 3 X 3 array b:

```
jobjectArray b;
jdouble *rowptr;
jdoubleArray  row;
double     bc[3];

row = (*env)->GetObjectArrayElement (env, b, 1);
rowptr = (*env)->GetDoubleArrayElements(env, row, 0);

bc[0] = rowptr[0];
bc[1] = rowptr[1];
bc[2] = rowptr[2];
```

- **a character string argument**

Java characters are in Unicode format and must be translated to UTF-8 format for C. JNI function GetStringUTFChars will make this translation. For example, the following C code will convert the characters in Java String fcn to C char string fcnname.

```
jstring fcn;
char  *fcnname;
fcnname   = (*env)->GetStringUTFChars (env, fcn, 0);
```

- **a function as an argument**

Some CNL routines require a function as an argument. This function is referred to as the user-supplied function. A Java method can be used for this by adding a user-supplied C function to interface between the Java method and the CNL routine.

The name of the Java method is passed as a character string to the C native interface, which obtains the method id and stores it in a global variable. The name of the C user-supplied function is passed as an argument to the CNL routine. When CNL calls the C user-supplied function, the global variable is used to call the Java method. Example2 demonstrates this procedure.

- **an array of complex numbers**

CNL supplies two structures, f_complex and d_complex, for handling complex numbers. Since there is no complex type or structure in Java, the Java interface code must have some method for passing the real and imaginary portions of the complex number to the CNL structures. There are a number of ways that this could be done. Two of the most obvious are passing the real and imaginary portions of the complex number as two separate arguments, or

to pass the real and imaginary portions as a two dimensional array. Example 3a demonstrates the latter.

A two dimensional Java array is treated as an object by JNI. JNI function GetObjectArrayElement is used to convert the Java array elements to the C structure. For example, to get row i of Java 3 X 2 array b:

```
jobjectArray b;
jdouble *rowptr;
jdoubleArray  row;
d_complex     bc[3];

row = (*env)->GetObjectArrayElement (env, b, i);
rowptr = (*env)->GetDoubleArrayElements(env, row, 0);

bc[i].re = rowptr[0];
bc[i].im = rowptr[1];
```

A Complex class is provided with Visual Numerics products JMSL and JNL. If either of these products is available, the Complex class from these products can be used instead of the above method to define an array of complex numbers. This also will be treated as an object array in the C interface code. Example 3b demonstrates the use of the Complex class for arrays of complex numbers.

3. The Dynamic Library

In order for the Java application to find the C interface code, a dynamic library must be built from the C interface code and loaded in Java. This dynamic library is operating system dependent, i.e. one must be built for each operating system on which the application is to run.

In the examples, System.loadLibrary from the java.lang.Runtime class is used to load the dynamic library using only the library name. The actual name of the library file and locations that are searched for this file are also operating system dependent.

For example, consider this statement:

```
System.loadLibrary("jicmath");
```

On Solaris, this means: Look for the file libjicmath.so in the paths specified by environment variable LD_LIBRARY_PATH.

On Windows, this means: Look for the file named jicmath.dll in the paths specified by environment variable PATH.

If the file is not found, the following exception occurs:

Exception in thread "main" java.lang.UnsatisfiedLinkError: no jicmath in java.library.path

Each example demonstrates building a dynamic library from just the object code for the C interface for that one example. If there are multiple C interface object files, they may all be built into the same dynamic library. For example, to build one dynamic library that could be used for all three examples shown below:

Solaris command :
ld -G -o libjicmath.so Example1.o Example2.o Example3a.o $LINK_CNL_STATIC

Note: LINK_CNL_STATIC is an environment variable set by the cttsetup.csh shell script, which is supplied with the CNL product.

or

Windows command:
link /dll /out:jicmath.dll Example1.obj Example2.obj Example3a.obj % LINK_CNL_STATIC %

Note: LINK_CNL_STATIC is an environment variable set by the cnlenv.bat batch file, which is supplied with the CNL product.

4. Examples

Examples are shown for the Sun Solaris and Microsoft Windows operating systems. The following environments were used in preparing this report:

Sun JDK 1.3.1
Sun Solaris 8 with Sun Workshop 6 C 5.1
Microsoft Windows NT 4 with Microsoft Visual C++ 6.0

Example 1 - Arrays and character strings

This example demonstrates a call to imsl_d_lin_sol_gen, using a two dimensional double array for the input matrix, and double arrays for the right hand side input and the solution. A call to imsl_d_write_matrix is included to demonstrate string input.

Example1.java :

```
import java.lang.Runtime;

class Example1
{
        public void calllsg ()
        {
             double a[][] = {
                                { 1, -3,  2},
                                {-3, 10, -5},
                                { 2, -5,  6}
             };


             double  b[] = {27, -78, 64};
            double  x[] = new double [3];

            int n=3, nra=1;

          // Solve Ax = b
           // call native interface code
             imsl_ji_lin_sol_gen (n, a, b, x);

           // use IMSL utility to print solution to demonstrate passing a String as an argument
             imsl_ji_write_matrix ("solution for Example1 ", nra, n, x );

        }
```

```
        public static void main(String args[])
        {
        Example1 lsrun = new Example1();
        lsrun.calllsg();
      }

// declare native interface

      public native void imsl_ji_lin_sol_gen (int n, double [][] a, double[] b, double[] x);
      public native void imsl_ji_write_matrix (String title, int nra, int nca, double [] a);

//  load shared library containing C interface code
      static
      {
         System.loadLibrary("jicmath");
      }
}
```

Example1.c:

```
#include <stdlib.h>
#include <imsl.h>
#include <jni.h>
#include "Example1.h"

/* java interface to lin_sol_gen  */

JNIEXPORT void JNICALL Java_Example1_imsl_1ji_1lin_1sol_1gen
  (JNIEnv *env, jobject obj, jint n, jobjectArray a, jdoubleArray b, jdoubleArray x)
{
   jdouble *rowptr, *bptr, *xptr;
   jdoubleArray row;
   double *aptr;
   int  i, j, k;

/* allocate double array aptr for input converted from a   */
   aptr = malloc (n * n *  sizeof(double));

/* copy Java two dimensional array a to C double array aptr   */
   k=0;
   for (i=0; i<n; i++) {
     row = (*env)->GetObjectArrayElement (env, a, i);             /* get row i  */
     rowptr = (*env)->GetDoubleArrayElements (env, row, 0);
     for (j=0; j<n; j++) {
       aptr[k] = rowptr[j];
```

```
        k++;
      }
    }

/* convert Java double array b to C pointer bptr */

    bptr = (*env)->GetDoubleArrayElements(env, b, 0);

    xptr = imsl_d_lin_sol_gen (n, aptr, bptr,  0);

/* store result back to Java double array  */
     (*env)->ReleaseDoubleArrayElements (env, x, xptr, 0);

/* free aptr   - it is no longer needed */
     free (aptr);
}

/* java interface to write_matrix  */

JNIEXPORT void JNICALL Java_Example1_imsl_1ji_1write_1matrix
  (JNIEnv *env, jobject obj, jstring jtitle , jint nra, jint nca, jdoubleArray a)
{
    jdouble *aptr;
    char  *title;
/* convert Java double array to C pointer  */
    aptr = (*env)->GetDoubleArrayElements(env, a, 0);

/*convert title from Java string to UTF-8 format   */
    title   = (*env)->GetStringUTFChars (env, jtitle, 0);

    imsl_d_write_matrix (title, nra, nca, aptr, 0);
}
```

Compile java code:

javac Example1.java

Generate header Example1.h for Example1.c :

javah Example1

Commands to compile and build shared library on Solaris:

cc -c $CFLAGS -I/usr/j2se/include -I/usr/j2se/include/solaris Example1.c
ld -G -o libjicmath.so Example1.o $LINK_CNL_STATIC

Add path containing libjicmath.so to LD_LIBRARY_PATH :

setenv LD_LIBRARY_PATH ".:${LD_LIBRARY_PATH}"

Note: LINK_CNL_STATIC is an environment variable set by the cttsetup.csh shell script, which is supplied with the CNL product.

Command to compile and build dll on Windows:

cl -c -DANSI -Ic:\jdk1.3.1\include -Ic:\jdk1.3.1\include\win32 Example1.c
link /dll /out:jicmath.dll Example1.obj %LINK_CNL_STATIC%

Note: LINK_CNL_STATIC is an environment variable set by the cnlenv.bat batch file, which is supplied with the CNL product.

Execute Example1:

java Example1

solution for Example1
1 2 3
1 -4 7

Example 2 - A function as an argument

The following example shows how to use a Java method as the user supplied function to imsl_d_min_con_gen_lin.

Example2.java :

```
import java.lang.Runtime;

class Example2
{
      public void callmcg()
      {

            double  a[] = {-1.0, -2.0, -2.0,
                        1.0, 2.0, 2.0
            };
            double  b[] = {0.0,72.0};
            double  xlb[] = {0.0, 0.0, 0.0};
            double  xub[] = {20.0, 11.0, 42.0};
            double xguess[]={10.0, 10.0, 10.0};
            double x[] = new double [3];
            int neq=0, ncon=2, nvar=3;


//  call min_con_gen_lin interface code with name of the method to be used as
//       the user-supplied function - "fcn"

            imsl_ji_min_con_gen_lin ("fcn", nvar, ncon, neq, a, b, xlb, xub, xguess, x);
            System.out.println ("solution for Example2 ");
            System.out.println ("x[0]="+x[0] +"  x[1]=" + x[1]+"  x[2]="+ x[2]  );

                  }

//  "user-supplied function" to be minimized

      public double fcn(int n, double[] x)
      {
       double f = -x[0]*x[1]*x[2];
       return f;
      }
// main
                  public static void main(String args[])
                  {
```

```
        Example2 mcgrun = new Example2();
        mcgrun.callmcg();
    }


//  declare native interface
    public native void imsl_ji_min_con_gen_lin  ( String fcnname, int nvar, int ncon,
            int neq, double[] a, double[] b, double[] xlb, double[] xub,  double[] xguess,
           double[] x);

//  load shared library containing C interface code
    static
    {
       System.loadLibrary ("jicmath");
    }
}
```

Example2.c :

```
#include <imsl.h>
#include <jni.h>
#include <stdio.h>
#include "Example2.h"

/*  declare global variables to be used by fcnmsg  to call Java method  */
jmethodID mid;
JNIEnv *envmcg;
jobject objmcg;

/*  C  user supplied function will call Java method  to be minimized  */

void fcnmcg (int n, double *x, double *ff)
{
jdoubleArray xj ;
jsize xstart, xlen;
xlen = n;
xstart =0 ;

/*  copy x array to Java double array  xj*/
xj = (*envmcg)->NewDoubleArray (envmcg, n);
(*envmcg)->SetDoubleArrayRegion (envmcg, xj, xstart, xlen, x);

/*  call Java method  and return result in ff   */
*ff = (*envmcg)->CallDoubleMethod (envmcg, objmcg, mid, n, xj);
```

```
}

/* java interface to min_con_gen_lin  */

JNIEXPORT void JNICALL Java_Example2_imsl_1ji_1min_1con_1gen_1lin
  (JNIEnv *env, jobject obj, jstring fcn, jint nvar, jint ncon, jint neq,
   jdoubleArray a, jdoubleArray b, jdoubleArray xlb, jdoubleArray xub,
   jdoubleArray xguess, jdoubleArray x)
{
    jclass cls;
    jdouble *aptr, *bptr, *xlbptr, *xubptr, *xptr, *xguessptr;
    char  *fcnname;
    envmcg = env;
    objmcg = obj;

/* save class and method name in global variable for later use by fcnmcg  */

    cls = (*env)->GetObjectClass (env, obj);
    fcnname   = (*env)->GetStringUTFChars (env, fcn, 0);

/* get method id for fcnname (Java method fcn ) with arguments (I[D)D :
              integer I (argument n )
              double array [D  (argument x)
              return value double D    (return f)

     see JNI reference for details on GetMethodID                              */

    mid = (*env)->GetMethodID (env, cls, fcnname, "(I[D)D");

/*  convert Java double arrays to C pointers    */
    aptr = (*env)->GetDoubleArrayElements(env, a, 0);
    bptr = (*env)->GetDoubleArrayElements(env, b, 0);
    xlbptr = (*env)->GetDoubleArrayElements(env, xlb, 0);
    xubptr = (*env)->GetDoubleArrayElements(env, xub, 0);
    xguessptr = (*env)->GetDoubleArrayElements(env, xguess, 0);

/* call min_con_gen_lin with fcnmcg as user function
     fcnmcg will call the Java method to be minimized       *

    xptr = imsl_d_min_con_gen_lin (fcnmcg, nvar, ncon, neq, aptr, bptr, xlbptr, xubptr,
                        IMSL_XGUESS, xguessptr, 0);

/* copy result back to Java double array  x      */
     (*env)->ReleaseDoubleArrayElements (env, x, xptr, 0);
}
```

Compile java code:

javac Example2.java

Generate header Example2.h for Example2.c :

javah Example2

Commands to compile and build shared library on Solaris:

cc -c $CFLAGS -I/usr/j2se/include -I/usr/j2se/include/solaris Example2.c
ld -G -o libjicmath.so Example2.o $LINK_CNL_STATIC

Note: LINK_CNL_STATIC is an environment variable set by the cttsetup.csh shell script, which is supplied with the CNL product.

Add path containing libjicmath.so to LD_LIBRARY_PATH :

setenv LD_LIBRARY_PATH ".:${LD_LIBRARY_PATH}"

Command to compile and build dll on Windows:

cl -c -DANSI -Ic:\jdk1.3.1\include -Ic:\jdk1.3.1\include\win32 Example2.c
link /dll /out:jicmath.dll Example2.obj %LINK_CNL_STATIC%

Note: LINK_CNL_STATIC is an environment variable set by the cnlenv.bat batch file, which is supplied with the CNL product.

Execute Example2:

java Example2

solution for Example2
x[0]=20.0 x[1]=11.0 x[2]=15.0

Example 3 - Complex numbers

Two examples are given to show how arrays of complex numbers can be passed to imsl_z_lin_sol_gen. Example 3a shows how a two dimensional array can be used to pass complex numbers to imsl_z_lin_sol_gen. Example 3b is a variation of the same example, using the Complex class from JMSL (or JNL) in place of the double arrays.

Example3a.java :

```
import java.lang.Runtime;

class Example3a
{
      public void calllsg ()
      {
            int n=3;
             //   for a  - use n*n X 2 array to store complex numbers one per row
            double a[][] = { {1.0, 1.0}, {2.0, 3.0}, {3.0, -3.0},
                             {2.0, 1.0}, {5.0, 3.0}, {7.0, -5.0},
                             {-2.0, 1.0}, {-4.0, 4.0}, {5.0, 3.0}};
             //   for b  - use n X 2 array to store complex numbers one per row
            double b[][] = { {3.0, 5.0}, {22.0, 10.0}, {-10.0, 4.0}};
            double x[][]  = new double [3][2];

            // Solve Ax = b
            // call native interface code
            imsl_ji_c_lin_sol_gen (n, a, b, x);
            System.out.println ("solution for Example3a ");
            System.out.println ("x[0]= ("+x[0][0] + "," + x[0][1] + ")   x[1]=("
                                +x[1][0] + "," + x[1][1] + ")   x[2]=("
                                +x[2][0] + "," + x[2][1] + ")" );
     }

    public static void main(String args[])
    {
          Example3a lsrun = new Example3a();
          lsrun.calllsg();
     }

// declare native interface
    public native void imsl_ji_c_lin_sol_gen (int n, double [][] a, double[][] b, double[][] x);

//  load shared library containing C interface code
       static
```

```
        {
           System.loadLibrary("jicmath");
        }
}
```

Example3a.c :

```
#include <stdlib.h>
#include <imsl.h>
#include <jni.h>
#include "Example3a.h"

/* java interface to c_lin_sol_gen  */

JNIEXPORT void JNICALL Java_Example3a_imsl_1ji_1c_1lin_1sol_1gen
  (JNIEnv *env, jobject obj, jint n, jobjectArray a, jobjectArray b, jobjectArray x)
{
   jdouble *rowptr;
   jdoubleArray  row,  xx;
   d_complex     *ac, *bc, *xc;
   int i;
   jsize xstart, xlen;

/* allocate d_complex arrays for z_lin_sol_gen  */

   ac = malloc(n*n *sizeof(d_complex));
   bc = malloc(n * sizeof(d_complex));
   xc = malloc(n * sizeof(d_complex));

/* convert Java object array to d_complex  */

     for (i=0; i<n*n; i++) {

/* get row i   */
        row = (*env)->GetObjectArrayElement (env, a, i);
        rowptr = (*env)->GetDoubleArrayElements(env, row, 0);

        ac[i].re = rowptr[0];
        ac[i].im = rowptr[1];

     }

     for (i=0; i<n; i++) {
```

```
        row = (*env)->GetObjectArrayElement (env, b, i);
        rowptr = (*env)->GetDoubleArrayElements(env, row, 0);

        bc[i].re = rowptr[0];
        bc[i].im = rowptr[1];

      }

    imsl_z_lin_sol_gen (n, ac, bc, IMSL_RETURN_USER, xc, 0);

/* store result back to Java object array  */
     xstart = 0;
     xlen = 2;

     for (i=0; i<n; i++) {

        xx = (*env)->NewDoubleArray(env, 2);
        rowptr[0] = xc[i].re ;
        rowptr[1] = xc[i].im ;

        (*env)->SetDoubleArrayRegion (env, xx, xstart, xlen, rowptr);
        (*env)->SetObjectArrayElement (env, x, i, xx);
     }

      free (ac);
      free (bc);
      free (xc);
}
```

Compile java code

javac Example3a.java

Generate header Example3a.h for Example3a.c :

javah Example3a

Commands to compile and build shared library on Solaris:

cc -c $CFLAGS -I/usr/j2se/include -I/usr/j2se/include/solaris Example3a.c
ld -G -o libjicmath.so Example3a.o $LINK_CNL_STATIC

Note: LINK_CNL_STATIC is an environment variable set by the cttsetup.csh shell script, which is supplied with the CNL product.

Add path containing libjicmath.so to LD_LIBRARY_PATH :

setenv LD_LIBRARY_PATH ".:${LD_LIBRARY_PATH}"

Command to compile and build dll on Windows:

cl -c -DANSI -Ic:\jdk1.3.1\include -Ic:\jdk1.3.1\include\win32 Example3a.c
link /dll /out:jicmath.dll Example3a.obj %LINK_CNL_STATIC%

Note: LINK_CNL_STATIC is an environment variable set by the cnlenv.bat batch file, which is supplied with the CNL product.

Execute Example3a:

java Example3a

solution for Example3a
x[0]= (1.000000000000001,-1.000000000000004)
x[1]=(1.9999999999999993,4.000000000000002)
x[2]=(3.0000000000000004,1.7177035475332614E-16)

Example3b.java :

```
import java.lang.Runtime;
import com.imsl.math.*;      // use this with JMSL
// import VisualNumerics.math.*;    use this with JNL

class Example3b
{
      public void calllsg ()
      {
            int n=3;

            Complex a[] = {
                  new Complex (1.0, 1.0),
                  new Complex (2.0, 3.0),
                  new Complex (3.0, -3.0),
                  new Complex (2.0, 1.0),
```

```
                    new Complex (5.0, 3.0),
                    new Complex (7.0, -5.0),
                    new Complex (-2.0, 1.0),
                    new Complex (-4.0, 4.0),
                    new Complex (5.0, 3.0)
            };
            Complex b[] = {
                    new Complex (3.0, 5.0),
                    new Complex (22.0, 10.0),
                    new Complex (-10.0, 4.0)
            };
            Complex x[]  = new Complex[3];

             // Solve Ax = b
            // call native interface code
            imsl_ji_c_lin_sol_gen (n, a, b, x);
            System.out.println ("solution for Example3b ");
             new PrintMatrix("x").print(x);


        //  this statement can be used to to print results if using JNL
        //System.out.println ("x[0]= ("+x[0].re + "," + x[0].im + ")   x[1]=("
        //                              +x[1].re + "," + x[1].im + ")   x[2]=("
        //                              +x[2].re + "," + x[2].im + ")" );


    }

    public static void main(String args[])
    {
          Example3b lsrun = new Example3b();
          lsrun.calllsg();
    }

// declare native interface
      public native void imsl_ji_c_lin_sol_gen (int n, Complex[] a, Complex[] b,
                                                Complex[] x);

//  load shared library containing C interface code
      static
      {
         System.loadLibrary("jicmath");
      }
}
```

Example3b.c :

```
#include <stdlib.h>
#include <imsl.h>
#include <jni.h>
#include "Example3b.h"

/* java interface to c_lin_sol_gen  */

JNIEXPORT void JNICALL Java_Example3b_imsl_1ji_1c_1lin_1sol_1gen
  (JNIEnv *env, jobject obj, jint n, jobjectArray a, jobjectArray b, jobjectArray x)
{
    d_complex     *ac, *bc, *xc;
    int i;
    jobject  cmplxobj;
    jclass   cls;
    jfieldID  fidre, fidim;
    jmethodID  mid;

/* allocate d_complex arrays for z_lin_sol_gen  */

    ac = malloc(n*n *sizeof(d_complex));
    bc = malloc(n * sizeof(d_complex));
    xc = malloc(n * sizeof(d_complex));

/* get class and field ids for Class Complex  in JMSL */
  cls = (*env)->FindClass (env, "com/imsl/math/Complex");

/*  use this statement to find Class Complex  if using JNL  :
  cls = (*env)->FindClass (env, "VisualNumerics/math/Complex");    */

  fidre =  (*env)->GetFieldID (env, cls, "re", "D");    /* real  */
  fidim =  (*env)->GetFieldID (env, cls, "im", "D");    /* imaginary  */

/* convert Java object array a to d_complex array ac   */
  for (i=0; i<n*n; i++) {
     cmplxobj = (*env)->GetObjectArrayElement (env, a, i);
     ac[i].re = (*env)->GetDoubleField (env, cmplxobj, fidre);
     ac[i].im = (*env)->GetDoubleField (env, cmplxobj, fidim);
  }


/* convert Java object array b to d_complex array bc   */
  for (i=0; i<n; i++) {
```

```
    cmplxobj = (*env)->GetObjectArrayElement (env, b, i);
    bc[i].re = (*env)->GetDoubleField (env, cmplxobj, fidre);
    bc[i].im = (*env)->GetDoubleField (env, cmplxobj, fidim);
  }

    imsl_z_lin_sol_gen (n, ac, bc, IMSL_RETURN_USER, xc, 0);

/* get method id of constructor for class Complex    */
    mid = (*env)->GetMethodID (env, cls, "<init>",  "(DD)V");

/*  copy solution back to Java Complex object array x */
      for (i=0; i<n; i++) {
          cmplxobj = (*env)->NewObject (env, cls, mid,  xc[i].re, xc[i].im);
           (*env)->SetObjectArrayElement (env, x, i, cmplxobj);
    }

      free (ac);
      free (bc);
      free (xc);
}
```

Execute Example3b:

```
java Example3b

solution for Example3b
              x
              0
0  1.000-1.000i
1          2+4.000i
2 3.000+0i
```