Wrapping Fortran libraries

Arjen Markus
arjen.markus@deltares.nl

May 2010

Abstract

There exist numerous libraries in C or Fortran that can be used to solve all manner of mathematical-numerical problems, such as LAPACK for linear algebra problems. These libraries comprise the experience of many mathematicians and software engineers. One of the goals of the Ftcl project is to make Tcl extensions for such libraries. The Wrapfort tool, akin to Critcl, especially, is designed to generate the required code.

1 Introduction

Using existing (numerical) libraries is often a recommendable practice: libraries such as LAPACK [**?**] for solving linear algebra problems comprise the experience of many mathematicians and software engineers in the form of easy-to-use routines. The main problem in using such libraries is that they are often programmed in a different language than the one you want to use, typically C or Fortran.

To use them you need to create a convenient interface, if one does not already exist, between the target language, Tcl in this case, and the implementation language (we will focus on Fortran in this paper). This interfacing is done by writing a set of so-called *wrapper* functions/routines that can be called from Tcl and invoke the appropriate Fortran routines in turn.

We need not restrict our attention to such general-purpose libraries. It is quite often desirable to have whole applications available as a set of separate commands. Here is an example:

Suppose you have a simulation program of a system of rivers, canals and reservoirs. This simulation program is used to investigate how the system is best managed - in terms of availability of drinking water, navigability of the rivers or the production of energy from the reservoirs. The various goals can be programmed into the computational core of the program, but that is a very rigid way of working: changing such rules requires changing the program, whereas the physics simulated in the program will not change.

Separating the program into individual routines, some concerned with the physics, others with the management rules makes it possible to run the program within a more flexible Tcl program. The Tcl program consists of high-level commands that translate into the low-level tasks in the computational core.

Using such high-level commands makes it easier to focus on the management rules, rather than their implementation.

Creating Tcl commands that invoke Fortran routines with a minimum of human effort is what this paper is about. We will mainly discuss the Wrapfort tool that is part of the Ftcl project (Markus, 2010a).

2 Related work

Several well-known extensions and packages have similar goals as Wrapfort:

- SWIG and Critcl both generate the required C code from a minimum of information:
  - SWIG (Beazley, 1998) takes the C prototypes an automatically produces wrapper functions for a variety of (dynamic) programming languages. The drawback of this approach is that it is tied to C as the language that is to be wrapped and, more importantly, the interfaces can not be geared to the target language – *cf* the next section.
  - Critcl (Wippler, 2002) is able to generate the code on the fly and build the extension automatically without any separate user action. As it relies on the user to define the interface, the drawback described above for SWIG does not hold. It does require a bit more work from the user, of course.

- TclODE (Kenny, 2008) is an extension that exports the functionality of a well-known library for solving systems of ordinary differential equations to Tcl. The original Fortran code treanslated to equivalent C code using the f2c utility. The main interfacing problems were:
  - Dealing with the COMMON blocks that are used in the original code
  - Automatically computing the derivatives of functions that are implemented in Tcl procedures.

3 Some examples

Let us consider a simple example to illustrate what needs to be done. The library "specfunc" by Shanjie Zhang and Jianming Jin (1996) contains more than 100 routines, all written in FORTRAN 77, for the evaluation of special mathematical functions, such Bessel functions of an arbitrary order and their zeros and spherical wave functions. A typical routine is AIRYB with the interface:

```
      SUBROUTINE AIRYB(X,AI,BI,AD,BD)
C
C       =====================================================
C       Purpose: Compute Airy functions and their derivatives
C       Input:   x  --- Argument of Airy function
C       Output:  AI --- Ai(x)
C                BI --- Bi(x)
C                AD --- Ai'(x)
C                BD --- Bi'(x)
C       =====================================================
C
        IMPLICIT DOUBLE PRECISION (A-H,O-Z)
```

(The IMPLICIT statement means that all variables whose name starts with a letter like A, O or W, are double precision floating point numbers, unless they are explicitly declared.)

Given an argument x it will evaluate the Airy functions of the first and second kind, Ai(x) and Bi(x), and their first derivatives Ai'(x) and Bi'(x).

And here we run into one issue that we need to address: should a Tcl command that wraps this Fortran routine also provide the values of these four functions or should we make four different commands, one for each mathematical function? If the first, do we change the original arguments, so that the interface behaves like this:

```
proc airy {x ai_ bi_ ad_ bd_} {
    upvar 1 $ai_ ai
    upvar 1 $bi_ bi
    upvar 1 $ad_ ad
    upvar 1 $bd_ bd

    ...
}
```

– the last four arguments passed to the command are regarded as the names of the variables that will hold the values.

Alternatively, we can create a command that returns the four values in a list:

```
proc airy {x} {
    ...

    return [list $ai $bi $ad $bd]
}
```

There is another, more technical issue as well: looking only at the declaration of the arguments in the Fortran code we can not deduce the role of the arguments: are they input or output arguments or both? In C this might be clear for scalar arguments, but not for arrays. It may even be unclear from the mere declaration whether something is a pointer to a scalar or a pointer to an array and in that case what the size of that array is:

```
void sumarray( int n, int *array, int *sum ) {
    int i;

    *sum = 0 ;
    for ( i = 0 ; i < n; i ++ ) {
        *sum = *sum + array[i];
    }
}
```

There can be no doubt about the first argument, but only by examining the body of the function can it be made clear if "array" is an array or a pointer to a scalar, like "sum".

This aspect of various programming languages makes it impossible to (completely) automate the generation of wrappers that present a convenient interface.

The routines in the LAPACK library present some other problems:

- Some arguments are simply work arrays - the user must provide memory for the operation, but on return the arrays do not contain useful information.[1] Quite often the optimal size depends on the problem data at hand.
- Input arrays (vectors and matrices) are specified by two or more arguments: the array itself, the number of rows, the number of columns and for additional flexibility the leading dimension (this is primarily meant for internal use).

Translating the interface to these routines verbatim to Tcl feels very clumsy:

```
      SUBROUTINE DBDSDC( UPLO, COMPQ, N, D, E, U, LDU, VT, LDVT, Q, IQ,
     $                   WORK, IWORK, INFO )
*
```

[1] FORTRAN 77 provided no standard way for dynamic allocation - most computers 30 years ago when that standard was developed did not offer that possibility.

```
*  -- LAPACK routine (version 3.2) --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     November 2006
*
*     .. Scalar Arguments ..
      CHARACTER          COMPQ, UPLO
      INTEGER            INFO, LDU, LDVT, N
*     ..
*     .. Array Arguments ..
      INTEGER            IQ( * ), IWORK( * )
      DOUBLE PRECISION   D( * ), E( * ), Q( * ), U( LDU, * ),
     $                   VT( LDVT, * ), WORK( * )
*     ..
*
```

would become (using a dummy procedure again to illustrate the role of the arguments):

```
proc dbdsdc { uplo compq n d_ e_ u_ ldu vt_ ldvt q_ iq_ work iwork info_ } {
    #
    # d, e, u, vt, q, and info are input/output or output arguments
    #
    # No actual need to output e, though, on exit it contains no useful
    # information
    #
    upvar 1 $d_    d
    upvar 1 $e_    e
    upvar 1 $u_    u
    upvar 1 $vt_   vt
    upvar 1 $q_    q
    upvar 1 $info_ info

    ...
}
```

But the arguments n, ldu, and ldvt are all array dimensions. In this case n is the order of the bidiagonal matrix formed from the diagonal d and the off-diagonal e. The arguments ldu and ldvt are the leading dimensions for the matrices u and vt, respectively. They would be equal to n in most circumstances and certainly when you use this routine directly (it is not a "driver" routine, so you would not normally use it).

The arguments work and iwork provide workspace. It is much more efficient to allocate arrays of the correct size in the C code and pass those on to the Fortran routine than to create them on the Tcl side.

These considerations lead to a much reduced interface - all the details above must be handled by the wrapper code:

```
proc dbdsdc { uplo compq d_ e_ u_ vt_ q_ iq_ info_ } {
    upvar 1 $d_    d
    upvar 1 $e_    e
```

```
    upvar 1 $u_    u
    upvar 1 $vt_   vt
    upvar 1 $q_    q
    upvar 1 $info_ info

    ...
}
```

Or, if all the output arguments are returned in a list:

```
proc dbdsdc { uplo compq d e } {

    ...

    return [list $dnew $u $vt $q $iq $info]  ;# Leaving out "e"
}
```

Conclusion: to create convenient (natural) interfaces we need to look at each routine we want to wrap with considerable care.

4 Automating what can be automated

Luckily, not all is lost: we can automate most of the tasks involved in creating a wrapper for such routines. First of all, we can distinguish a number of roles for the arguments:

- input data (scalars or arrays)
- sizes of arrays
- workspace (always arrays)
- output (scalars or arrays)
- options
- function name (this kind of argument is discussed in a subesequent section)

For each role we can provide boilerplate code. An input array "v" would be implemented as:

Declaration:

```
long *v; int size__v;
```

Initialisation:

```
if ( WrapCopyIntListToArray( interp, objv[1], &v, &size__v ) != TCL_OK ) {
    Tcl_SetResult( interp, "Argument 1 must be a list of integers", NULL );
    return TCL_ERROR;
}
```

Clean-up:

```
ckfree((char *)v);
```

The variable "size__v" is used to record the length of the array. From the user's point of view it can be accessed via the macro "size(v)". (Rows and columns of matrices are similarly treated).

The code is always the same, only the name of the variable and its position in the argument list change.

Within the Wrapfort extension such an input array is specified as:

```
double-array v input
```

The other roles also have associated C code and they can be specified in a very similar way:

```
integer k {assign 1}
double-array x {allocate n}
double y result
```

Currently Wrapfort deals with only one result variable at a time and that must be a scalar quantity. It is possible to define more than one output argument, but that requires some hand-crafted C code. Here is an example:

The routine LAGZO in the specfunc library fills two arrays that are passed a sarguments with the zeros of the Laguerre polynomial of order n and the corresponding weights for a quadrature rule. In other words:

```
subroutine lagzo( n, x, w )
    integer n
    double precision x(n), w(n)
    ...
end
```

The wrapper for this routine simply returns a list of two lists:

```
lassign [::Specfunc::laguerreZeros $n] x w
```

It is generated from the following code:

```
Wrapfort::fproc ::Specfunc::laguerreZeros lagzo {
    integer       n  input
    double-array  x  {allocate n}
    double-array  w  {allocate n}
    code {} {
        lagzo( &n, x, w );
        {
            Tcl_Obj *result[2];
            if ( WrapCopyDoubleArrayToList( interp, x, n, &result[0] ) != TCL_OK ) {
                Tcl_SetResult( interp, "Can not copy array to Tcl list", NULL );
                return TCL_ERROR;
            }
            if ( WrapCopyDoubleArrayToList( interp, w, n, &result[1] ) != TCL_OK ) {
                Tcl_SetResult( interp, "Can not copy array to Tcl list", NULL );
                return TCL_ERROR;
```

```
            }
            Tcl_SetObjResult( interp, Tcl_NewListObj( 2, result ) );
        }
    }
}
```

This complete example brings us to a second task, this one of rather technical nature: we can put any code in the body for the `fproc` command, but names of Fortran routines are mangled by the compiler to avoid conflicts and to ensure their names are not case-sensitive (where, furthermore, each compiler uses a different method). The second argument to `fproc` is therefore the name of the Fortran routine that is being wrapped. From that name we can then generate code like:

```
#ifdef FTN_UNDERSCORE
#   define lagzo lagzo_
#endif
#ifdef FTN_ALL_CAPS
#   define lagzo LAGZO
#endif
void __stdcall lagzo(); /* Important! */
```

to ensure the C compiler is given the right name and *calling convention*.

All in al, Wrapfort generates all the boilerplate code we need:

- An initialisation function for the package
- The package index script
- The invocations to Tcl_CreateObjCommand() to register the commands
- The wrapper functions themselves

(It does not create a corresponding makefile though - that still needs to be done)

The body of C code that is passed to Wrapfort is responsible for calling the Fortran routine in the correct way. Unfortunately this is very difficult to automate in general and then there are one or two nasty aspects that you must understand (cf. Appendix A):

- Where C distinguishes passing by value and passing by reference, Fortran only uses passing by reference.[2] Hence the line:

  ```
  lagzo( &n, x, w );
  ```

  where all arguments are in fact (C) pointers.

- Fortran expects the (declared) length of a string to be passed as a hidden argument. Some compilers put it right after the string argument, others put it at the end. Wrapfort provides two macros to take care of this:

[2]Strictly speaking C passes only by value and the Fortran standard does not enforce any argument passing method. But for practical purposes the above is an accurate enough description.

– STRING(a) to pass the name `a` and, if needed, the length of string `a`

– STRINGLEN(a) to pass, if needed, the length of string `a`

The use is a trifle awkward - the second macro includes a comma, if it is not empty, as illustrated by this LAPACK routine:

```
dposvx( STRING(fact), STRING(uplo), &n, &nrhs, a, &lda, af,
    &ldaf, STRING(equed), s, b, &ldb, x, &ldx, &rcond, ferr, berr,
    work, iwork, &info  STRINGLEN(fact) STRINGLEN(uplo) STRINGLEN(equed) );
```

Sometimes it is possible to automate the generation of even the Wrapfort commands. The LAPACK source code is documented in a very regular way: each argument is documented with its type and its role. In the case of arrays (one- or two-dimensional) the size is also documented:

```
      SUBROUTINE DGELS( TRANS, M, N, NRHS, A, LDA, B, LDB, WORK, LWORK,
     $                  INFO )
*
*  -- LAPACK driver routine (version 3.2) --
*  -- LAPACK is a software package provided by Univ. of Tennessee,    --
*  -- Univ. of California Berkeley, Univ. of Colorado Denver and NAG Ltd..--
*     November 2006
*
...
*
*  Purpose
*  =======
*
...
*
*  Arguments
*  =========
*
*  TRANS   (input) CHARACTER*1
*          = 'N': the linear system involves A;
*          = 'T': the linear system involves A**T.
*
*  M       (input) INTEGER
*          The number of rows of the matrix A.  M >= 0.
*
*  N       (input) INTEGER
*          The number of columns of the matrix A.  N >= 0.
*
*  NRHS    (input) INTEGER
*          The number of right hand sides, i.e., the number of
*          columns of the matrices B and X. NRHS >=0.
*
*  A       (input/output) DOUBLE PRECISION array, dimension (LDA,N)
*          On entry, the M-by-N matrix A.
*          On exit,
```

```
*            if M >= N, A is overwritten by details of its QR
*                       factorization as returned by DGEQRF;
*            if M <  N, A is overwritten by details of its LQ
*                       factorization as returned by DGELQF.
*
```

This feature made it possible to wrap the hundreds of routines in the LAPACK library almost automatically. Fine-tuning these interfaces is still a manual task, but at least it is manageable.

5 External functions

Quite often a general-purpose library relies on the ability to pass user-defined functions. Typical examples are:

- Integration routines have a function argument, so that the user can pass the name of the function to be integrated
- The MINPACK library (anonymous, 2010) minimizes functions using various algorithms. The functions are supplied as an argument.

Wrapfort can deal with this type of interfacing. Consider the routine `HYBRD1` from the MINPACK library that can be used to find the zeros of a system of nonlinear functions:

```
subroutine hybrd1(fcn,n,x,fvec,tol,info,wa,lwa)
integer n,info,lwa
double precision tol
double precision x(n),fvec(n),wa(lwa)
external fcn
```

The argument `fcn` is supposed to be a subroutine of four arguments with the following signature:

```
subroutine fcn(n,x,fvec,iflag)
integer n,iflag
double precision x(n),fvec(n)
----------
calculate the functions at x and
return this vector in fvec.
---------
return
end
```

The argument `iflag` should be set only to terminate the search procedure. This can be translated in an error condition: if the Tcl procedure returns an error, the wrapper code sets the argument `iflag`.

The `fexternal` command arranges for the generation of a C function that can be called from Fortran and that in turn invokes the Tcl command or procedure whose name was passed. Thus:

```
Wrapfort::fexternal fcn {
    fortran {
        integer      n      input
        double-array x      {input n}
        double-array fvec   {output n}
        integer      iflag  output
    }
    toproc {
        x      input
        fvec   result
    }
    onerror {
        *iflag = -1;
    }
}
```

(note the code fragment associated with the `onerror` keyword) and the accompanying interface to the `HYBRD1` routine:

```
Wrapfort::fproc findRoot hybrd1 {
    external     fcn    {}
    double-array x      input
    double       tol    input

    double-array fvec   {allocate  size(x)}
    integer      n      {assign    size(x)}
    double-array wa     {allocate  (n*(3*n+13))/2}
    integer      info   local
    integer      lwa    {assign    size(wa)}
    double-array xfinal {result    size(x)}
    integer      i      local

    code {Actually call the routine in this way} {
        hybrd1( fcn, &n, x, fvec, &tol, &info, wa, &lwa );
        for (i = 0; i < n; i ++) {
            xfinal[i] = x[i];
        }
        /* Check the result - set a message if there was an error */
        if ( info != 1 && info != 3 ) {
            WrapErrorMessage( "No satisfactory result achieved" );
        }
    }
}
```

Some care must be taken - the implementation uses a global variable to pass the name of the Tcl command/procedure. This is not thread-safe nor is it safe to use indirect recursion, but it is a straightforward solution.

6 Avoiding C programming altogether

In the description up to now the wrapping has involved *some* C code, hence the requirement for the user to have at least elementary knowledge of C and of the way C and Fortran interface, as well the presence of a suitable C compiler. This has the additional drawback that we can not simply supply a precompiled library.

Within the Ftcl project there is a small library of routines, callable from Fortran, that allows the user to create Tcl extensions directly in Fortran [2]. The subroutines that implement a Tcl command have this interface:

```
subroutine cmd_routine( cmdname, noargs, ierror )
   character(len=*)    :: cmdname
   integer             :: noargs
   integer             :: ierror

   ...
end subroutine cmd_routine
```

The Ftcl library then provides routines to access the actual arguments (the array of Tcl_Objs as found in C API) and it provides routines to register the commands in a very similar way as the C API:

```
call ftcl_make_command( cmd_routine, "routine" )
```

Via the routine verb+ftcl_provide_package+ you can register the package:

```
call tcl_provide_package( "package", "version", error )
```

When creating an extension in C, you would also create an initialisation function with a name like Package_Init() that will be called when the shared library or DLL is loaded. The following trick avoids the need to create such a function for Fortran-based extensions:

```
package ifneeded A 1.0 [list load \
    [file $dir $LIBNAME[info sharedlibextension]] Ftclpkg]
```

The function Ftclpkg_Init() is predefined and each Fortran extension using Ftcl is supposed to define a routine package_init (with that exact name) that does the actual initialisation.

7 Further developments

7.1 Performance

The current wrapping strategy used with the Wrapfort tool is to use Tcl lists for both arrays and matrices of data. That is: an ordinary Tcl list of double precision floating-point numbers is first transformed into a C/Fortran array of numbers and then passed on to the Fortran routine or function. On output the reverse process is used.

This is convenient on the Tcl side, but it is also somewhat costly. If we use a byte array instead to store the data on the Tcl side, the transformation is

no longer required. A simple experiment has a shown a gain of 402010b). The experimental evidence thus is very limited. Further experiments will have to show if it is really worth the extra effort.

Constructing and updating a Tcl byte array via the [binary] command, however, is rather expensive and would annihilate any performance gain on the wrapping side. Probably a better way is to develop a set of dedicated commands that take care of the storage and retrieval.

There are roughly two methods to implement such commands:

- Define a new data type via Tcl's `Tcl_RegisterObjType()` function
- Use a Tcl list to store the data and the associated structure. Such a list could contain the following elements:
  - A keyword to identify the type of data.
  - One or more elements to describe the data size (number of rows and columns for a matrix for instance).
  - The data themselves, stored as a byte array, but treated as an opaque data structure.

7.2 Wrapping C libraries

While the main goal of Wrapfort is to make wrapping Fortran libraries as easy as possible, the same approach can be taken to wrap C libraries like FFTW for Fourier transformations (FFTW, 2010).

The difference with Critcl would be the emphasis:

- Critcl is capable of compiling on the fly, whereas with Wrapfort this is always a separate step.
- Wrapfort focuses on numerical data.

A first glance at the API of FFTW has revealed that it presents more or less similar challenges with respect to wrapping as the Fortran libraries discussed here, but it also provides new ones:

- It has its own memory allocation functions to ensure the best possible alignment.
- It works with an opaque data structure, the *transformation plan*, that needs to be passed from one function to the next.

Wrapping FFTW will lead to further expanding the data types that Wrapfort supports, because of such opaque data structures and because complex numbers are very important with Fourier transforms.

8 References

David Beazley (1998)
Tcl Extension Building with SWIG
http://www.swig.org/papers/TclTutorial98/TclTutorial98.pdf

Jack Dongarra et al. (2010)
LAPACK - Linear Algebra PACKage
http://www.netlib.org/lapack/

Matteo Frigo and Steven G. Johnson (2010)
FFTW http://www.fftw.org

Kevin Kenny (2008)
An ODE solver for Tcl: old Fortran in a new interface
http://tclode.sourceforge.net/tclode.pdf

Arjen Markus (2010a)
Ftcl - Combining Fortran and Tcl
http://ftcl.sf.net

Arjen Markus (2010b)
wrapper extension for LAPACK
http://wiki.tcl.tk/25451

Jorge More', Burt Garbow abd Ken Hillstrom (1999)
The MINPACK package
http://www.netlib.org

Jean-Claude Wippler
Critcl – Compiled Run-time in Tcl
http://www.equi4.com/pub/docs/vancouver/pres2.htm

Shanjie Zhang and Jianming Jin (1996)
Computation of Special Functions
John Wiley and Sons, Inc., New York, 1996, ISBN 0-471-11963-6

Appendix: Interfacing C and Fortran

Calling Fortran routines from C or vice versa has several aspects that need to be dealt with, some depend on the combination of compilers used, others are due to the intrinsic difference in the programming language: not all Fortran data types are supported by C and vice versa. This means that sometimes data conversions are required (notably logicals in Fortran must be converted into integers in C and back, C structs can not be comfortably passed to Fortran and so on).[3]

Things to be considered are:

- Fortran does not have call by value, but only call by reference. So from the C point of view you always pass addresses.
- Strings in Fortran are actually made of two elements: the string itself and the length. The length is passed as a hidden argument, which is however

[3]The current Fortran 2003 defines a standard method of interfacing to C, but this requires changes to the Fortran code.

visible on the C side, and the position of that argument depends on the chosen Fortran compiler (and possibly the compiler options).

- The names of Fortran routines as seen by the linker may be mangled in different ways to avoid name clashes with system libraries and to ensure the name is case-insensitive. A popular scheme is to translate the name to lower-case and append an underscore.
- On Windows there used to be a problem with the calling convention in both C and Fortran libraries. The choice of calling convention was important for the handling of the stack on returning from a function or subroutine. As the `stdcall` convention has been deprecated, the issue is less important than it used to be, but it still exists.

In the *Ftcl* project a dedicated program attempts to find out which name mangling method and what calling convention is used and how string lengths are passed. The results are stored as a set of C macros.