An ODE solver for Tcl: old Fortran in a new interface

Kevin B. Kenny
Computational Biology Laboratory, GE Global Research Center, Niskayuna, NY
kennykb@research.ge.com

Abstract

LSODAR is a Fortran subroutine for integration of ordinary differential equations that has been actively maintained for over twenty years and gives high-quality results for a variety of problems. This paper demonstrates how such a "dusty deck" can be adapted to interface cleanly to a modern high-level language like Tcl, be made safe to use in the presence of threads and recursion, and take advantage of the symbolic computation capabilities available to Tcl. The result is an ODE solver for Tcl that is production-ready, at the expenditure of considerably less effort than it would take to develop comparable capabilities targeted specifically at Tcl. It presents a few tricks that should be handy in connecting other legacy Fortran applications to Tcl.

1. Introduction

Fortran has for decades been the first choice of language for scientific programming. Because of this, a great many Fortran codes are available for solving numeric problems, such as statistical analysis, root finding, and the integration of differential equations. Many of the best are available free of charge, having been developed by US Government researchers at taxpayer expense. These "treasures of the national labs" deserve to be more widely known outside the Fortran community, but the tremendous difference in style between Fortran and more modern languages inhibits their broader acceptance. This paper describes how one such code, LSODAR (Livermore Solver for Ordinary Differential equations with Automatic method selection and Root-finding) (4) was adapted to use in Tcl.

Among the issues that have to be addressed for this code are adapting the long parameter lists of the Fortran calls to something more Tcl-friendly, managing dynamically allocated memory, dealing with Fortran callbacks (callbacks to EXTERNAL functions) at the Tcl level, and dealing with issues like recursion and multithreading.

The combination is arguably greater than the sum of the parts, because Tcl's ability to perform symbolic calculations can be exploited in the combined code. The combined Fortran-Tcl system has the ability to use Tcl's symbolic capabilities to perform symbolic differentiation to exploit backward-differencing formulas.

2. Background

The motivation for this work was the observation that a number of the author's colleagues were performing numerical simulations of ordinary differential equations (ODE's) in the context of cell biology(1), pharmacokinetics(5), and other biological systems. The systems tended to divide into two groups. The first group was accepted by the biologist users, but generally was specialized for a single narrow problem, and often had tremendous issues of numerical stability (because of the use of ad hoc solvers for the ODE's). The second took care with the numerical analysis, often by using well-tested third party codes, and often offered considerably more generality, but tended to be

Fortran programs usable only by their authors. Clearly, some sort of middle ground was needed.

At the same time, a number of others were working on developing the interface between Tcl and Fortran. Having already realized Tcl's utility for ad hoc scientific calculations(8), Arjen Markus had developed a rudimentary interface for Fortran programs to evaluate Tcl scripts(7). Working together with Jean-Claude Wippler and Steve Landers, he also explored a "proof of concept" of an automatic interface generator whereby Fortran subprograms could be integrated into Tcl(6). The idea of integrating Tcl and Fortran was certainly in the air.

These two developments combined to motivate the development of a Tcl interface to a general-purpose Fortran ODE solver, LSODAR (Livermore Solver for Ordinary Differential equations [Automatic method selection with Rootfinding])(4). LSODAR was chosen because it is well qualified (it has been well maintained for twenty years!), and because it requires fairly little "care and feeding" in the area of solution method selection (Many other libraries require the user to recognize the realms in which their ODE's are stiff and non-stiff and choose methods accordingly, risking horrible numerical instabilities if the wrong methods are chosen.) Moreover, LSODAR offers a root-finding capability, which was essential to stop the simulation at bifurcation points in the mixed strategy (ODE's plus discrete event simulation) that some of the author's colleagues were pursuing.

3. The Tcl interface to LSODAR

The first approach to connecting LSODAR with Tcl was to do a fairly straightforward translation of the application programming interface. It became obvious fairly quickly, however, that this approach was not going to succeed – or at the very least, was not going to produce a calling sequence that was anywhere near to idiomatic Tcl. If we examine Figure 1, it becomes quickly obvious why this is.

```
      SUBROUTINE DLSODAR (F, NEQ, Y, T, TOUT, ITOL, RTOL, ATOL, ITASK,
     1             ISTATE, IOPT, RWORK, LRW, IWORK, LIW, JAC, JT,
     2             G, NG, JROOT)
      EXTERNAL F, JAC, G
      INTEGER NEQ, ITOL, ITASK, ISTATE, IOPT, LRW, IWORK, LIW, JT,
     1   NG, JROOT
      DOUBLE PRECISION Y, T, TOUT, RTOL, ATOL, RWORK
      DIMENSION NEQ(*), Y(*), RTOL(*), ATOL(*), RWORK(LRW), IWORK(LIW),
     1   JROOT(NG)
```

Figure 1. Calling sequence for LSODAR.

First, there is a long list of parameters – twenty or so. It is difficult to imagine any Tcl command accepting that many positional parameters. Moreover, a number of the parameters (NEQ, LRW, LIW, and NG) simply serve to give the lengths of arrays. In addition, only the arrays ATOL, RTOL,Y and JROOT are actually used by the caller,

either before or after the call[1]. RWORK, IWORK and G are simply working storage provided by the caller because Fortran 77 has no dynamic memory allocation.

A more awkward problem is that three of the parameters are EXTERNAL – which, in Fortran, means that they are pointers to Fortran subprograms that will be invoked by the called routine. These callbacks need to be adapted so that they can be delivered into Tcl. Clearly, at least a small development effort was needed to make a module that would present a Tcl-friendly interface and support calls back and forth between Tcl and Fortran.

A. Choice of compiler

Since Tcl's object level API's are all in C, and the interfaces between C and Fortran are not terribly portable among compilers, the decision was made to have all the code in C. For this reason, the first thing done was to take all the Fortran code needed for LSODAR and pass it through 'f2c' – a Fortran compiler that has C code as output(3). This C code, plus the C source code for four builtin functions from the associated runtime library, provided a code base that was independent of the Fortran compiler on any target system; it is simply included in the source code of the distributed module. It was tested on one Linux system by comparing the output of the LSODAR example programs compiled with 'f2c' and the output from the Fortran versions compiled with GNU 'g77'(2); output was identical.

B. Designing a Tcl interface

Rather than having a large number of positional parameters, the Tcl interface uses keyword-value syntax, with sensibly chosen defaults. Figure 2 shows a trivial example, where a simple harmonic oscillator is modeled for an eighth-cycle of its operation.

```
set solver [odesolv::odesolv -atol 1e-6 -rtol 1e-6 -indvar theta \
             -- s {$c} c {-$s}]
set s 0.0
set c 1.0
set theta 0.0
set troubles [$solver run [expr {atan(1.0)}]]
puts [list $theta $c $s]
rename $solver {}
```

Figure 2. Trivial example of use of the solver

The program in Figure 2 is read: "Create an ODE solver instance, and store it in variable 'solver'. The independent variable for the system of ODE's is ϑ, and the results are expected to be computed to either six decimal places (an absolute tolerance of 10^{-6}) or to six significant figures (a relative tolerance of 10^{-6}). The system to be solved is:

$$\begin{aligned} \frac{ds}{dt} &= c \\ \frac{dc}{dt} &= -s \end{aligned} .$$

Initial values are $\vartheta = 0, s = 0, c = 1$, and results are requested for $\vartheta = \pi / 4$."

[1] Not quite true – the IWORK and RWORK arrays also contain further optional parameters for the call. We are dealing with a truly Byzantine interface here!

Unsurprisingly, when run, the program prints that at the end, s and c are both within 10^{-6} of $\sqrt{2}$.

Note that the right-hand sides of the system are Tcl expressions: they are actually evaluated by making Tcl_ExprDoubleObj callbacks back into the interpreter where the solver instance was created.

C. Connecting Tcl and Fortran

The solver constructor command is fairly straightforward; it simply builds a C data structure that contains all the parameters (plus sensible defaults for any named parameters that have been omitted). It then constructs a Tcl command that has this data structure as its ClientData, and returns the name of the constructed command. The client data structure also includes pointers to the arrays that the Fortran code uses.

The instance command supports three subcommands, 'run', 'continue' and 'status.' The 'run' command starts an integration; the 'continue' command continues an integration already in progress. The 'status' command simply returns a dictionary containing a group of the internal variables that the integrator uses, reporting on things like the integration method in use, the number of steps taken so far, the size of the last successful step and the next step to be attempted.

The 'run' and 'continue' commands invoke the Fortran code in fairly straightforward fashion. The only thing that is at all unusual is that the callbacks F (which computes the vector of derivatives, JAC (which computes the Jacobian matrix of F) and G (which computes the algebraic equations whose roots are to be found) are sent back to C code within the interface module for further processing. The callback for F, for instance, is shown in Figure 3.

```
static void
SolverInstEvalDeriv(
    integer* neq,          /* Number of equations, but also
                            * instance pointer */
    doublereal* t,          /* Value of the independent variable */
    doublereal* y,         /* Values of the model variables */
    doublereal* ydot)             /* Values of the derivatives */
{
    Solver* solverPtr = (Solver*) neq;
    Tcl_Interp* interp = solverPtr->interp;
    int systemc;
    Tcl_Obj** systemv;
    int i;
    int status = TCL_OK;
    status = CopyOutModelVariables(solverPtr, t, y, &systemc, &systemv);

    /* Evaluate the derivatives */
    for (i = 0; (status == TCL_OK) && (i < *neq); ++i) {
       status = Tcl_ExprDoubleObj(interp, systemv[2 *i + 1], ydot + i);
    }
    if (status != TCL_OK) {
       longjmp(solverPtr->errorExit, 1);
    }
}
```

Figure 3. Callback that evaluates the derivatives in the solver

In the figure, we see first a horrible kludge. The parameter, ‘neq’, which gives the number of ODE’s in the system, is also the first member of the client data structure. Casting it back to the client data lets the evaluator find all the other information it needs. Believe it or not, this hack is actually documented in the manual for LSODAR.

The ‘CopyOutModelVariables’ call copies the Fortran state vector to Tcl variables in the current scope. Following this, each of the ‘neq’ derivatives is evaluated in Tcl. If an error occurs, the solver is aborted (by the expedient of longjmp’ing around the Fortran code, since LSODAR does not provide for an error exit); otherwise, we return to the Fortran to continue solving the ODE’s.

4. Troubles that needed fixing

This level of interface served for quite some time, until its user base expanded a little. The first real trouble came from unexpected recursion. A user had created a Tk GUI that allowed running multiple systems at once. For some reason, one of the derivatives was evaluated by an expression that contained an invocation of a Tcl procedure, which in turn contained an [update]. The event loop was invoked, and dispatched the solver for another system of equations, and the code crashed.

The problem was that the code generated by ‘f2c’ contained static data in several labeled COMMON blocks. This static storage had to be made local to a single instance of the solver. This proved to be surprisingly easy.

First, every file that designated a given labeled COMMON block has a ‘struct’ declaration simultaneously laying it out and defining it, as shown in Figure 4. A #define giving the name to be used to refer to the block follows immediately.

```
struct {
    doublereal rowns[209], ccmax, el0, h__, hmin, hmxi, hu, rc, tn, uround;
    integer init, mxstep, mxhnil, nhnil, nslast, nyh, iowns[6], icf, ierpj,
           iersl, jcur, jstart, kflag, l, lyh, lewt, lacor, lsavf, lwm, liwm,
            meth, miter, maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje,
           nqu;
} dls001_;

#define dls001_1 dls001_
```

Figure 4. Machinery of ‘f2c’ COMMON blocks.

Each subprogram that appears in the file begins with a left brace ‘{‘ on a line by itself. This is the only place where ‘f2c’ generates such a line.

A simple Tcl script is able to rewrite the declaration to the code shown in Figure 5.

```
typedef struct {
    doublereal rowns[209], ccmax, el0, h__, hmin, hmxi, hu, rc, tn, uround;
    integer init, mxstep, mxhnil, nhnil, nslast, nyh, iowns[6], icf, ierpj,
           iersl, jcur, jstart, kflag, l, lyh, lewt, lacor, lsavf, lwm, liwm,
            meth, miter, maxord, maxcor, msbp, mxncf, n, nq, nst, nfe, nje,
           nqu;
} dls001_;
extern Tcl_ThreadDataKey Odesolvdls001_Key;
```

Figure 5. Rewritten COMMON block interface.

It can then add a codeburst to the head of every procedure to get the current locations of the blocks, as shown in Figure 6.

```
    dls001_ *dls001_Ptr = (dls001_ *)
        OdesolvGetCommon("dls001_", &Odesolvdls001_Key, sizeof(dls001_));
    dlsa01_ *dlsa01_Ptr = (dlsa01_ *)
        OdesolvGetCommon("dlsa01_", &Odesolvdlsa01_Key, sizeof(dlsa01_));
    dlsr01_ *dlsr01_Ptr = (dlsr01_ *)
        OdesolvGetCommon("dlsr01_", &Odesolvdlsr01_Key, sizeof(dlsr01_));
```

Figure 6. Retrieving COMMON block addresses at the head of procedures

(OdesolvGetCommon is a thin layer around Tcl_GetThreadData that calls Tcl_Panic if the thread data have not been initialized.)

When a solver instance is started via the ‘run’ or ‘continue’ subcommands, it saves any existing thread data, sets the thread data to point to instance-specific copies of the COMMON blocks, and enters the Fortran code. In this way, the usage of COMMON blocks is made safe for multiple instances of the solver, multiple interpreters, and even multiple threads. Since this change was made, no further mystery crashes have been observed.

5. Further convenience

One optional parameter to LSODAR is a subroutine that evaluates the Jacobian of the derivatives (that is, the matrix of second derivatives). Supplying this subroutine can accelerate convergence, and in the case of very stiff systems of equations, may be necessary for numerical stability.

Even in Tcl, writing code to evaluate the Jacobian may be onerous. Since most of the uses of the package so far have involved fairly simple expressions for the derivatives (in the sense that they do not require command substitutions or arrays, but make do with Tcl’s built-in functions and scalar variables), it looked feasible to address the problem with symbolic differentiation.

The ‘odesolv’ package looks for a ‘math::symdiff’ package. If it finds it, and the user has not supplied a Jacobian matrix, it invokes ‘math::symdiff::jacobian’ to try to develop one symbolically. If it succeeds, it invokes the Fortran code with the symbolic Jacobian; if it fails, it invokes the Fortran code with no Jacobian supplied and hopes for the best.

Figure 7 shows an example of computing a symbolic Jacobian for a system of equations.

```
% foreach row [math::symdiff::jacobian {
    y1 {-0.04 * $y1 + 10000. * $y2 * $y3}
    y2 {0.04 * $y1 - 10000. * $y2 * $y3 - 3.0e-7 * $y2 * $y2}
    y3 {3.0e-7 * $y2 * $y2}
}] {puts $row}
-0.04 {(10000.0 * $y3)} {(10000. * $y2)}
0.04 {-(((10000.0 * $y3) + ((3.0e-7 * $y2) + (3e-007 * $y2))))} {-((10000. *
$y2))}
0.0 {((3.0e-7 * $y2) + (3e-007 * $y2))} 0.0
```

Figure 7. Calculating derivatives symbolically

The details of the 'symdiff' package are described in a companion paper.

6. Results

It is customary in an experience study like this one to present comparison results with other techniques for achieving the same goals. In this case, it is difficult, because it is hard to find other attempts at a production-quality ODE solver in Tcl.[2] Certainly, any attempt to compare the 'odesolv' package with the ODE solver in Tcllib is comparing apples with oranges: 'odesolv' offers automatic switching between Adams-Moulton and Gear backward differencing, while Tcllib does only fourth-order Runge-Kutta; 'odesolv' has adaptive stepsize control and error estimation, while the Tcllib method lacks these features; and so on. Nevertheless, it is possible at least to set up benchmark problems with the two systems.

The first problem, a nonstiff system, was the simple harmonic oscillator of Figure 2. Integration was carried out starting at $\vartheta = 0$ and continuing to $\vartheta = 13\pi/6$, a little more than one cycle. The second problem was the extremely stiff set of equations $\frac{du}{dt} = 998u + 1998v \quad \frac{dv}{dt} = -999u - 1999v$, carried out for $0 \le t \le 16$.
For the stiff solution, LSODAR chose the Adams-Moulton solver for the initial nonstiff settling time until t = 0.017, and then switched to Gear's backward difference formula. Table 1 shows the performance comparison.

Table 1 . Performance comparisons of Tcllib's integrator and 'odesolv'.

Problem	Nonstiff		Stiff	
Metric	Tcllib	Odesolv	Tcllib	Odesolv
Number of evaluations of derivatives	256	194	32768	472
Number of evaluations of Jacobian	N/A	5	N/A	21
Run time (ms)	4.13	0.55	284.00	2.03

The results show that to achieve the same accuracy of results 'odesolv' took roughly the same number of iterations in the nonstiff case, but arrived at the answer nearly an order of magnitude faster than tcllib's integrator. Tcllib's integrator proved unsuitable for the

[2] It may be possible to achieve this functionality using Tcl with SCILAB; I have not yet succeeded at this, but admittedly haven't tried very hard.

stiff problem. For fewer than 16384 iterations of the Runge-Kutta step, its numeric stability was so catastrophic that it crashed with a floating-point overflow rather than returning results. By contrast, 'odesolv' was, as expected, quite well behaved, switching to a backward difference formula and needing two orders of magnitude less time (and comparably fewer evaluations of the derivatives).

7. Conclusions

This work demonstrates a comprehensive reworking of a legacy Fortran interface into idiomatic Tcl. It shows how integrating high-quality scientific codes into Tcl can enhance the capabilities of both. It presents a few interesting techniques for integrating Fortran and Tcl; in particular, anchoring client data to the address of a Fortran parameter, replacing COMMON blocks with thread-specific data are both new techniques (as far as the author is aware). Finally, it demonstrates that combining Tcl's ability for symbolic computation and for reconfiguration yields a whole that is greater than the sum of the parts.

References

1) Beer, A.J., Haubner, R., Goebel, M., Luderschmidt, S., Spilker, M.E., Wester, H.J., Weber, W.A., Schwaiger, M. "Biodistribution and pharmacokinetics of the αVβ3-selective tracer ^{18}F-galacto-RGD in cancer patients." *J Nucl Med.* 46:8 (August, 2005), pp.1333-41.
2) Burley, James Craig. *Using and porting GNU Fortran.* Version 3.4.6. Cambridge, Mass: Free Software Foundation, 2006.
3) Feldman, S.I., Gay, David M., Maimone, Mark W., Schryer, N. L. "A Fortran-to-C converter." Computing Science Technical Report 149, Murray Hill, N.J.: AT&T Bell Laboratories, March, 1995. Reprint available at http://www.netlib.org/f2c/f2c.pdf.
4) Hindmarsh, Alan C. "ODEPACK: A systematized collection of ODE solvers." In *Scientific Computing* (R.S. Stepelman, *et al., eds.*) (Volume 1 of IMACS Transactions on Scientific Computation) Amsterdam: North-Holland, 1983, pp. 55-64. Reprint available at http://www.llnl.gov/CASC/nsde/pubs/u88007.pdf.
5) Kiehl, Thomas R., Mattheyses, Robert M., Simmons, Melvin K. "Hybrid simulation of cellular behavior." *Bioinformatics* 20:3 (March 2004), pp. 316-322.
6) Landers, Steve, and Wippler, Jean-Claude. "CriTcl - beyond Stubs and compilers." Proc. Ninth Intl. Tcl/Tk Conference, Vancouver, B.C. (September 2002). Reprint available at http://aspn.activestate.com/ASPN/Tcl/TclConferencePapers2002/Tcl2002papers/wippler-critcl/critcl.pdf.
7) Markus, Arjen. "Combining Fortran and scripting languages." ACM SIGPLAN Fortran Forum 21:3 (2002), pp. 10-18.
8) Markus, Arjen. "Doing mathematics with Tcl." Proc. Third European Tcl/Tk User Meeting. Munich (June, 2002). Reprint available at http://www.t-ide.com/tcl2002e/mathematics.pdf.