FIG. 1. The header file `runFortObject.h`.

FIG. 2. The implementation file `runFortObject.m`.

FIG. 3. The Fortran file `hellosub.f`.

FIG. 4. The implementation file `runFortObject.m`.

FIG. 5. The files `make.preamble` and `make.postamble`.

FIG. 6. The `rhoSky` Main Window.

FIG. 7. The `rhoSky` Plot Window.

FIG. 8. The `updateOutputData`, `sendPlotDataToWindow`, and `provideDataStream` methods.

[1]W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Veterling, *Numerical Recipes: The Art of Scientific Computing* (Cambridge U. P., New York, 1986), esp. Chap. 15; see also W. H. Press and S. A. Teukolsky, Computers in Physics **6**, 188-191 (1992).

[2]From AT&T, available via anonymous FTP in a version specifically compiled for the NeXT from `sonata.cc.purdue.edu` in `/pub/next/2.0-release/binaries`.

[3]NeXT, Inc., *NeXTstep Reference Manual*, Addison Wesley, ISBN 0-201-58136-1, 1992; Useful supplementary documents available by anonymous FTP from the `sonata` archive, Ref. 2, are: NeXT, Inc., `NextStep_Concepts`, in directory `/pub-/next/docs`; M. Mahoney, `IB_tutorial`, in `/pub/next/Newsletters/SCaNeWS`; and J. Glover, *Short-Course on Object-Oriented Programming*, `UHOOPclass`, in `/pub/next/docs`.

[4]Y. Igarashi et al., Nucl. Phys. **B259**, 721-729, 1985. This model has a defect: the skyrmion solution is not really stable. See Z. F. Ezawa and T. Yanagida, Phys. Rev. D **33**, 237, 1986 and J. Kunz and D. Masak, Phys. Lett. **B179**, 146-152, 1986.

[5]C. Fletcher, `nxyPalette1.2`, available from `sonata`, Ref. 2, in directory `/pub-/next/submissions`.

[6]D. Jesperson and T. Pulliam, `nxyPlot1.8`, available from `sonata`, Ref. 2, in directory `/pub/next/submissions`.

The interface described above is specific to one physics problem, but it is clear that the principles involved can be applied to many different problems involving the solution of coupled differential equations. In fact, we already have adapted (re-used) the code, with *very* little work, to solve some coupled equations for a quantum hadrodynamic problem involving pions and σ-mesons.

This conversion only took about eight hours and most of that time was spent in writing the prologue to the code which describes the problem being solved. Essentially, only the Fortran subroutines `LOAD1`, `LOAD2`, and `DERIVS`, which specify the equations being solved, had to be changed. There were also some problem-specific changes that had to be made to the input and output Forms of the Main Window (different coupling constants, different numbers of functions to be solved for, etc.), but these were minor.

In my experience as a programmer, the quickness of this turnaround, from original conception to the production of useful results, was quite remarkable.

ACKNOWLEDGEMENTS

This work evolved from research done in collaboration with Michael Mattis and James L. Hughes. I want to thank Bryan Travis, Jeremy Brackbill, James Gubernatus, and Klaus Lackner for useful discussions regarding the interface described here and what other projects could be approached with it. An important component of the `rhoSky` interface has been the `nxyPalette` for the Plot Window; its author, Charles Fletcher of Techno-Sciences, Inc., was very helpful to me in getting it to work. Finally, Kim Maltman made useful comments for improving the manuscript.

- `provideDataStream`, which tells `nxyView` where to find the data to plot. (`RunFortObjectInstance` is `nxyView`'s "delegate", which means that for some messages received by `nxyView`, it asks its delegate to respond.)

Finally, there is one additional method (not mentioned in the header file, i.e., not "public"), `displayIntmdteFs`, for handling the display of intermediate results.

Likewise, the Fortran file is now considerably expanded. The communication with the Objective-C code continues to be, as above, through a set of subroutine calls. These subroutines were "written" quickly by converting the original Fortran main program (which itself largely consisted of subroutine calls) into several (top-level) subroutines, already mentioned above with regard to the methods connected with buttons: `startsub`, `continuesub`, and `finishsub`. There are two additional (top-level) subroutines, `writeplotdata`, called by `updateOutputData`, and `observables`, called by `finishsub`. The latter subroutine originally was a *separate* Fortran post-processor program for calculating, from the final F, F', G and G', the quantities that eventually appear in the `m_rho`, `M_sky`, and `GoldRat` FormCells in the Main Window. Finally, there are several lower-level routines needed for the Runge-Kutta integration, such as `DERIVS`, which calculates the right-hand-sides of the four first-order equations.

What can *not* be shown in the figures of this article is the real-time ease of use of the `rhoSky` application. Calculation proceeds quickly, with results and graphs appearing within a second or so after clicking a button. The interface in fact *encourages* a more exploratory approach to the equations than one might undertake when searching for solutions in a "batch mode". Most importantly, it is much more fun to "drive" than the old command-line program ever was.

IV. PORTABILITY OF CONCEPT

The application also has a Plot Window for displaying the present (or final) calculated $F(r)$ and $G(r)$ (Fig. 7). The code for plotting in this window was taken from `nxyPalette`,[5] a custom-built object in the public domain. `nxyPalette` in turn is based upon a publicly available NeXTstep plotting application, `nxyPlot`.[6] The expanded `RunFortObject.m` file, discussed below, includes three methods needed for plotting (besides those methods already in the `nxyView` object provided by `nxyPalette`). The graph in the Plot Window is updated after every pass (Run, Continue, or Finish Up). As shown in Fig. 7, the user sees the curves, initially discontinuous, smooth out as a solution is achieved.

As mentioned, the `RunFortObject.m` file is more complex and now contains eight methods. Some of these are connected to buttons:

- `startFortMethod:sender`, the action of the Run button, which calls the Fortran subroutine `startsub`.
- `continueFortMethod:sender`, the action of the Continue button, which calls subroutine `continuesub`.
- `finishFortMethod:sender`, the action of the Finish Up button, which calls subroutine `finishsub`.
- `clearForNewRun:sender`, the action of the Clear button, which simply calls the `clearNXYView` method provided by `nxyView` and clears the output Form displays.

The three methods involved in plotting graphs are (Fig. 8):

- `updateOutputData`, which calls writeplotdata_().
- `sendPlotDataToWindow`, which also checks that the data is plottable.

- An input Form object for various input parameters: coupling constants of the Lagrangian, particle masses, the matching radius $r_F \equiv$ `xf`, the asymptotic radius $r_2 \equiv$ `x2`, the Runge-Kutta precision parameter, and the value of $F(0)$ (which is usually taken to be π, as indicated in Eq.(3)).

- An input Form for the starting values of the four scale parameters (A, B, C, D) which are adjusted to make F, G continuous and smooth at r_F.

- A Form for showing the present values of the adjusted scale parameters.

- A Form for displaying the discontinuities at the match point and the calculated corrections to the scale parameters.

- An output Form for displaying post-processed computations, such as the Skyrmion mass, `M_sky`.

- Default values in the Form Cells for input parameters; these can be changed by user by clicking on the desired cells and editing them, before clicking on Run.

- Various control buttons:

  Run—does a Clear and performs first pass of a new calculation with present input parameters.

  Continue—goes on to next pass, if desirable.

  Finish Up—writes out final solution to output.data and perform and display post-processing computations.

  Clear—clears the Plot Window, discussed below, and the output form cells.

The evolution of the Main Window as one goes through the steps to find a solution is shown in Fig. 5.

The `rhoSky` application calculates the classical pion and ρ-meson field functions for a ρ-stablized skyrmion.[4] This is done using the so-called "Hedgehog Ansatz", which assumes the field function solutions are particular tensor covariants times spherically symmetric functions. In this case, since there are two meson fields, there are two functions, $F(r)$ and $G(r)$.

Mathematically, we want to solve two nonlinear differential equations for F and G knowing the boundary conditions and indicial behavior at $r = 0$ and appropriate asymptotic behavior (exponential damping). The equations are

$$\begin{aligned} & r^2F'' + 2rF' + (a-1)\sin 2F \\ & \qquad +2a(G-1)\sin F - m_\pi^2 r^2 \sin F \ = \ 0 \ , \qquad (1) \\ & r^2G'' - (m_\rho^2 r^2 + 2)G + 3G^2 \\ & \qquad -G^3 + m_\rho^2 r^2(1-\cos F) \ = \ 0 \ . \qquad (2) \end{aligned}$$

Near $r = 0$ the solutions must behave as

$$F = \pi - Ar + O(r^3) \ , \quad G = 2 - Br^2 + O(r^4) \ , \qquad (3)$$

where the scale parameters A and B are constants to be determined by the nonlinearity of the equations themselves. Similarly, the desired asymptotic behavior at large r is

$$F(r) \rightarrow C\frac{\exp(-m_\pi r)}{r} \ , \qquad (4)$$

$$G(r) \rightarrow \frac{m_\rho^2}{(m_\rho^2 - 4m_\pi^2)}\frac{F^2(r)}{2} + D\exp(-m_\rho r) \ , \qquad (5)$$

where C and D are the other two scale parameters (constants) to be determined.

The Main Window for the `rhoSky` application is assembled using the Interface Builder as in the "Hello, Fortran!" example discussed above, but it involves more objects (Fig.1):

- The Fortran subroutine reads the input data from `input.data` and massages it. It then writes its output to the file `output.data`, closes the file, and returns.
- `runFortMethod` continues by opening `output.data`, reads it, and finally displays the number there in `outputForm`, i.e., in the Main Window.

Three details in this code should be noted. First, Fortran is case-insensitive, but Unix filenames do care about case. Second, the codes in Figs. 2 and 3 include `printf`'s and `print`'s, seen here as commented out, for debugging purposes. If the application is launched from a terminal window (a shell), these debug printouts would appear there. The ability to print out intermediate results turned out to be very useful for debugging. Finally, it is necessary to include the `#import` statements for the header files `appkit/Form.h` and `stdio.h`, so that the C compiler will not complain about things it cannot find.

The final step is to "make" the application, to test it, and to iterate as necessary. The IB generates a standard Unix Makefile, which is not to be touched. However, the code developer can customize that Makefile as needed by creating `make.preamble` and `make.postamble` files, which are read and executed by the standard Makefile. In this particular application one needs to declare `OTHER_OFILES` to force recompilation of `hellosub.f`, if changed (Fig. 4). Also, to be able to find the Fortran I/O routines, etc., one must declare the f2c library. (Alternatively, one could declare the library by adding it to the "other libs" category in the Project Inspector window in the IB.)

Such a NeXTstep application does work, and, as a learning tool, it provided the clues for how to frontend a more ambitious Fortran code, such as that discussed in the next section.

III. THE FRONTEND FOR THE COUPLED ODE SOLVER CODE

`RunFortObject` is made the IB also creates an *instance* of it, `RunFortObject-Instance`. In this class one declares the existence of two outlets, `inputForm` and `outputForm`, and one method, `runFortMethod:sender`. (The argument "sender" will refer to the Run button object.)

The `RunFortObjectInstance`'s outlets, `inputForm` and `outputForm`, now need to be specified, i.e., the `RunFortObjectInstance` object must know what graphical objects elsewhere on the screen they refer to. Likewise, the Run button must know what object to tell that it has been clicked and what message should be sent there. These connections are made by dragging a line, again using the mouse, from the objects to their corresponding target objects. Thus code in the `RunFortObject` which refers to, say, the `outputForm` will know which Form object in the Main Window it should send the Fortran answer to for display. Similarly, the Run button is connected to the `RunFortObjectInstance` as target and will trigger the action given by `runFortMethod`.

Finally, by clicking on the Parse menu item, the IB creates a class header file, `runFortObject.h` (see Fig. 1), and a shell for the class implementation file `runFort-Object.m`. Nothing has to be done with the header file but the `runFortObject.m` file needs fleshing out with respect to its code. The following material was added, by hand, to `runFortMethod` (see Fig. 2):

- First, read the `inputForm`, write its value to a file, `input.data`, and close the file.

- Call the Fortran code, which is actually the subroutine `hellosub` in the `hellosub.f` file (see Fig. 3). The compiler-translator, f2c, creates a C function corresponding to this subroutine, `hellosub_()`, which can then be called by that name in the C code.

vice versa)?

II. TO PROVE THE CONCEPT

To answer that question, consider a simple “Hello, Fortran!” application, which does the following:

- The user inputs (clicks, then types) a number into a small text window, a FormCell object labeled “Input”, in an on-screen window representing the “Hello, Fortran!” application.

- Clicking on a Run button in that window reads that number and sends it to a Fortran program.

- The Fortran code multiplies it by 7 and displays the “answer” in another FormCell object (labeled “Output”) in the window.

To build such a program as a NeXTstep application one first builds the window using the Interface Builder[3] (IB), a developer’s toolkit that comes with every NeXT workstation having the extended distribution. How the IB is used will only be described briefly; the reader is referred to Ref. 3 for details.

After launching the IB, one uses the mouse to drag in and drop various graphical objects from the IB palette in the application’s Main Window. For the “Hello, Fortran!” application it is only necessary to add two Forms (both one cell only) for input and output and a Run button.

Now one creates a custom class, call it `RunFortObject`, by subclassing the generic `Object` class. This object will contain all the specialized behavior we need to control the graphical interface window and access the Fortran module. At the time chosen to have the input read from an editable input file, but for present purposes it was preferable to query the user.) This mode of operating was quite tolerable, perhaps close to optimum, during the time I was running the code on a VAX mainframe.

Soon, however, because of a change of station, I had to transfer my Fortran computing from the VAX to a NeXT workstation. Consideration of cost led me to use the public domain "compiler" f2c, which translates Fortran code into C code, and then compiles that.[2] This, as will be seen, turned out to be a serendipitous choice.

After the move to the NeXT, however, some things about running the program in text mode became annoying. For example, performing another calculation meant starting a new job and re-entering all the input parameters. Another annoyance was the graphing of the resulting solutions. On the VAX, plots of solutions could be made automatically (and somewhat interactively) by writing out the graphics files to my terminal running in Tektronix 4014 emulator mode. On the NeXT, however, the plotting had to be done by post-processing an output file after each run. I began looking for a better way to work.

Perhaps more of a reason to make a change came from the difficulties of the nonlinear equations themselves. It is sometimes tricky getting the code to find reasonable Newton-Rapheson corrections if the initial scale parameters are poorly choosen. If one could immediately see graphs of the results of the first pass, that would help in making a good initial choice. (If it's very bad, one would not proceed with the run and try another choice.) Likewise, seeing graphs of intermediate pass results would allow user to opt out of what turned out to be a bad choice of starting scale parameters.

Thus, it soon became obvious that one should try putting a user-friendly interface, a frontend, on the Fortran ODE solver program. However, the NeXTstep operating system and graphical user interface is based on Objective-C, an object-oriented extension of C. So the basic question is how to get Objective-C to talk to Fortran (and

I. INTRODUCTION AND BACKGROUND

In the course of my research as a nuclear physicist I recently had occasion to write a Fortran code for solving coupled, nonlinear differential equations. The code, which originally ran on a VAX, used appropriate Numerical Recipes[1] to carry out a Runge-Kutta integration of the equations.

In particular, the code solves second-order coupled ODEs for two functions, $F(r)$ and $G(r)$, which is a system of four first-order equations when recast for the Runge-Kutta procedure. The boundary conditions at the origin are known and the equations themselves determine indicial behaviors near $r = 0$. There are also desired asymptotic behaviors at large distances (e.g., exponential falloffs). However, one doesn't know the sizes of derivatives at the origin or the asymptotic normalizations, since these are determined by the nonlinearities of the equations. So, what the user of the code does is to choose these four scale parameters arbitrarily for a first pass through the equations. These parameters are then subsequently refined by successive iterations until the equations are well-satisfied.

This procedure involves shooting out from the origin, using an adaptive Runge-Kutta routine, to some intermediate distance, r_F (e.g., 0.5 fm). Then one shoots back from a large value of r (e.g., 2 fm) to r_F. The discontinuities at r_F (in the functions F and G and their derivatives) then determine, using a generalized Newton-Rapheson technique, corrections to the initial scale parameters that should tend to drive the next discontinuities toward zero. The code then goes on to do another pass through the above shooting procedure. The program continues iterating until it achieves a good solution.

This Fortran program was invoked from and ran in a (text only) terminal window. The user could semi-interactively input parameters, i.e., the code would use defaults and query the user on each if he or she wanted to make a change. (One could have

An Interactive NeXTstep Interface to a Fortran Code for Solving Coupled Differential Equations

Richard R. Silbar

Theoretical Division, Los Alamos National Laboratory

University of California, Los Alamos, New Mexico 87545

(Received August 15, 1992)

This paper describes a user-friendly frontend to a Fortran program that integrates coupled nonlinear ordinary differential equations. The user interface is built using the NeXTstep Interface Builder, together with a public-domain graphical palette for displaying intermediate and final results. The main problem is how to communicate between the Objective-C environment of NeXTstep and the Fortran code. This is resolved by breaking up the Fortran into separate subroutines, corresponding to the various control buttons in the interface. These subroutines are then compiled in such a way that they can be called as ordinary C functions.

Los Alamos National Laboratory Preprint LA-UR-92-2541

Submitted to *Computers in Physics*