Template-Driven Interfaces
for Numerical Subroutines

Jon L. Bentley
Mary F. Fernandez
Brian W. Kernighan
Norman L. Schryer

AT&T Bell Laboratories
Murray Hill, NJ 07974

ABSTRACT

This paper describes a set of interfaces for numerical subroutines. Typing a short (often one-line) description allows one to solve problems in application domains including least-squares data fitting, differential equations, minimization, root finding, and integration. Our approach of ‘‘template-driven programming’’ makes it easy to build such an interface: a simple one takes a few hours to construct, while a few days suffice to build the most complex program we describe.

1. Introduction

Numerical subroutine packages are one of the oldest and most effective approaches to software reuse. Experts familiar with a good library can rapidly combine tested and robust components into useful software.

As effective as they are for experts, however, most large libraries are difficult for the casual user. Suppose, for instance, that a programmer desires to find a root of a nonlinear equation. This usually entails looking up the appropriate routine, modifying a sample program, compiling the program with links to the appropriate libraries, testing (and possibly debugging) the code, and interpreting the output to find the answer. This process can easily take an hour or two. Rice [1989, p. 2] summarizes the problem:

> The user must write code in the target programming language which creates the input and output data structures and which invokes the procedures. This code is usually lengthy compared to invocation of the library procedures.

Many solutions have been proposed for this problem. Gill, Murray, Picken, Wright [1979] describe principles for constructing subroutine libraries that are easy to use. Large environments such as MATLAB and S provide elegant interfaces to sophisticated software libraries. Such environments are excellent for both expert and novice users, but require many programmer-years to build.

We wish to make it easy for individual programmers to quickly build effective interfaces for their software libraries. In this paper we describe a set of programs that provide an interface to the Port subroutine library described by Fox [1984]. Each program solves a popular form of a common numerical problem. Here, for instance, is a dialog in which we first find a root of the function $\sin(x)$ in the interval $[3,4]$ and then find a root of $x - e^{1-x}(1 + \ln\ x)$ in the interval $[0.1, 0.9]$:

Maria Fernandez is currently with the Department of Computer Science, Princeton University, Princeton, NJ 08544. Electronic addresses: `jlb@research.att.com`, `mff@princeton.edu`, `bwk@research.att.com`, `nls@research.att.com`.

```
$ root 'sin(x)' 3 4
3.14159265
$ root 'x - exp(1-x)*(1+log(x))' .1 .9
0.49673636
$
```

(The dollar sign is the system prompt.) Behind the scenes, the Root program uses a *template* to write, compile and execute a Fortran program that calls the appropriate subroutine. We refer to this approach as ''template-driven programming''.

We undertook to construct similar interfaces for other parts of the Port library, with the following design goals:

- The interfaces should be accessible to novices, yet powerful enough to be convenient for experienced users.
- The problem specifications should be succinct.
- The interfaces should be easy to build, from a few hours and a few dozen lines of code, to a day or two and a few hundred lines.

This paper illustrates the technique on several major parts of the Port library. Section 2 introduces our template-driven approach to building interfaces, and Section 3 describes least-squares data fitting in detail. Section 4 surveys several problem domains, and Section 5 describes the more complex problem domain of differential equations. Other applications of template-driven programming are described in Section 6, and conclusions are offered in Section 7.

2. Template-Driven Programming

In this section we will introduce our template-driven approach to building interfaces by studying the implementation of the Root program that we saw earlier. That program calls the Port function `dzero` to find the zero of a function over a specified interval. The documentation of most function libraries illustrates the use of a function with a small example. We generalize that example to a *template* program that calls the function:

```
double precision dzero
external f
write (*,*) 0d0 + dzero(f, 0d0 + @LO@, 0d0 + @HI@, 0d0)
stop
end

double precision function f(x)
double precision f, x
f = @EXP@
return
end
```

This template has three parameters, `EXP`, `LO`, and `HI`, which are surrounded by at-signs (''`@`''). The Root program reads the three arguments from its command line, substitutes them into this template to produce a Fortran program, then compiles and executes the Fortran program, which writes the answer.

It is straightforward to implement this approach on many systems. We have, of course, tailored our implementation to our computing environment: the numerical routines are from the Port library, we call the routines from Fortran programs, and our interfaces are implemented in Awk. Here is the body of an Awk implementation of a minimal Root program:

```
BEGIN { subarr["EXP"] = ARGV[1]
        subarr["START"] = ARGV[2]
        subarr["END"] = ARGV[3]
        dotemplate("root.tplt", "junk.f", subarr)
        system("f77 junk.f -lport 2>junk.err; a.out")
        exit
}
```

(Aho, Kernighan and Weinberger [1988] describe the Awk programming language.) The first three lines read the command-line arguments, the fourth line calls the `dotemplate` function to instantiate the template `root.tplt` into the Fortran file `junk.f`, and the fifth line compiles the program and executes the object code, which writes the answer.

The `dotemplate` function is passed the name of a template file, the name of an output file, and a substitution array of name-value pairs; here is a trivial implementation:

```
function dotemplate(templatefile, outfile, subarr,  i) {
        while (getline <templatefile > 0) {
                for (i in subarr)
                        gsub("@" i "@", subarr[i])
                print $0 >outfile
        }
}
```

Appendix B describes a more complete function for instantiating templates.

This minimal program correctly handles correct programs; it is implemented with a 9-line template and 14 lines of Awk. The complete Root program is more careful with error checking: Does the command line have precisely three arguments? Did the Fortran compile successfully? If not, then report the error messages. The complete Root program, including the template itself and code for error checking and template interpretation, requires about 40 lines (or 30 lines with a library of template functions).

The Root program is easy to implement as a template-driven Awk program, but hard to implement directly in Fortran. One possible Fortran implementation parses an expression and interprets it at run time; another choice is to write and compile a Fortran program from within Fortran. Neither choice is particularly appealing. Our implementation parcels out the jobs to the right tools: an Awk program interprets the template, the UNIX® shell calls the Fortran compiler to compile the expression, the Port library supplies an effective solver. Although we've used the tools most convenient on our home system, similar tools could be used in other environments to implement the essential ideas: simple, concise interfaces that use template-driven programs.

3. Least-Squares Regression

In the data-fitting problem we are given a set of x, y pairs that are assumed to be modeled by the equation $y = f(x,p)$, where p is a vector of reals. The goal is to find the vector p that minimizes the sum of the squares of the residuals

$$\min_{p} \sum_{i=1}^{n} [y_i - f(x_{i,}p)]^2$$

The Port library offers several routines for various forms of this task, all based on the excellent algorithm of Dennis, Gay and Welsch [1981] (widely known as `NL2SOL`).

We provide access to that algorithm through a program called L2fit (for Least Squares Fit). We will illustrate our program on a file that contains data on traveling salesman tours through point sets randomly distributed on the unit square. The x value is the number of points and the y value is the length of a certain tour through the points. We have reason to believe that the tour length should grow as $a + bx^c$, where a, b and c are real numbers (synonyms for p_1, p_2 and p_3, in the model $y = f(x,p)$). We therefore type the following command:

```
$ l2fit -lxy 'a+b*x**c' data/tourlen.d
0.630579 0.733034 0.4998811
$
```

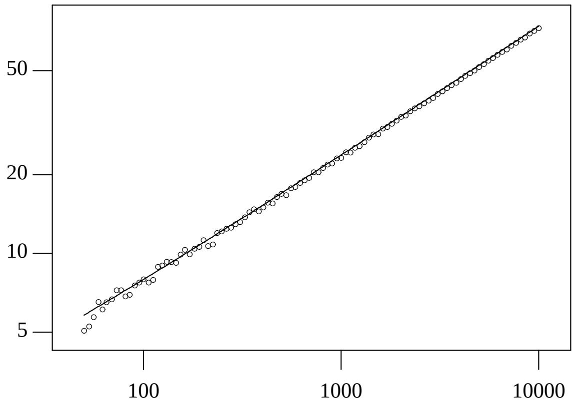
About ten VAX-8550 CPU seconds later, the program writes the computed values of *a*, *b* and *c*, in alphabetical order on the second line. As a side effect of the computation, L2fit produces the Troff output file `l2fit.out`, which is displayed in Figure 1.

L2FIT SUMMARY

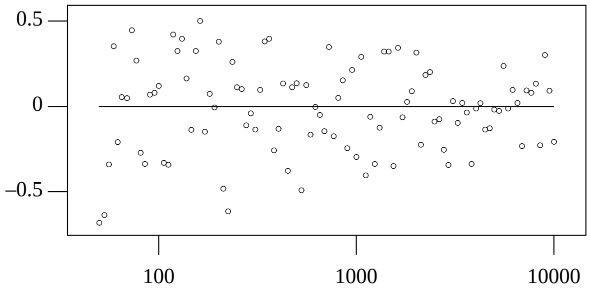
Command line: `l2fit -troff -s0.6 -lxy a+b*x**c data/tourlen.d`
Least squares regression type: Unweighted
Input data file: data/tourlen.d
Input expression: a+b*x**c $a + bx^c$
Canonical form: p(1)+p(2)*x**p(3) $p_1 + p_2 x^{p_3}$

Parameters	$a = p_1$	$b = p_2$	$c = p_3$
Initial:	0	0	0
Final:	0.630579	0.733034	0.499881
Standard errors:	0.12083	0.0166733	0.00240624

Algorithm termination: x- and relative function convergence



Circles: input (x,y) pairs. Line: least squares fit $y = f(x,p)$.



Circles: residuals $y - f(x,p)$.

Figure 1. L2FIT output on TSP data.

The text at the top of Figure 1 summarizes the computation. The command line typed by the user is repeated; this is indispensable when searching through a pile of L2fit runs. In this case, the `-lxy` flag requests that graphs be plotted with logarithmic *x* and *y* scales. The data is in the file `data/tourlen.d`. Two flags were added to the command line shown earlier to prepare the `l2fit.out` file for inclusion in this Troff document: `-troff` applies the appropriate document production programs and the `-s0.6` shrinks the graphs down to 60% of their default size. The final values of the parameters are displayed, along with their standard errors (multiplying by 1.96 gives 95% confidence intervals).

The top graph displays both the input data and the fitted function $y = 0.63 + 0.733x^{0.49988}$. The residuals in the bottom graph are quite well-behaved: they appear to decrease in value as x grows, but they are roughly normally distributed. These two graphs together give us confidence that the tour length grows as $y = 0.63 + 0.733\sqrt{x}$. The next command was therefore

```
$ l2fit -lxy 'a + b*sqrt(x)' data/tourlen.d
0.636145 0.732208
$
```

The new `l2fit.out` file (which is not included in this paper) shows that the computed values of a and b are quite close to their old values, but the standard error of a decreases from 0.121 to 0.044 and that of b decreases from 0.0167 to 0.0010. The graphs are essentially unchanged.

This example is typical of the user's view of a session with L2fit. L2fit provides the following services behind the scenes:

- It uses a template to write a Fortran program that calls the appropriate Port routine. It compiles the program and checks the Fortran compilation status to find errors in the input expression.
- It executes the program and reads and interprets the output of the least-squares routine. Some values are extracted directly, while others require more processing (for instance, the standard errors are the square roots of the diagonals of the covariance matrix).
- It performs certain straightforward data processing tasks, such as checking that each line in the input file contains two numeric fields, counting the number of parameters in the expression to determine how large to make the vector p, and sorting the x values to draw the function in the output graph.
- It uses another template to prepare a summary page in `l2fit.out`. That template uses the Troff language for typesetting and the Grap language for graphical display of data described by Bentley and Kernighan [1986].
- It performs minor translations among the various languages used. The variable a in the input expression is replaced by $p(1)$ in the Fortran program and by the computed value in the Grap output. The Fortran exponentiation operator ‘‘`**`’’, the Grap operator ‘‘`^`’’ and the Eqn operator ‘‘`sup`’’ are properly exchanged. The Grap `log` function returns the base-ten logarithm, so it is replaced with a natural logarithm function.

Mechanizing the job makes it much easier and faster to use the least-squares fitting routines in Port. But more importantly, L2fit supports *better* data fitting by automatically generating graphical displays along with numerical answers.

Figure 2 contains a second example of L2fit output. In this case each x value is the number of objects in a search tree and the y value is the corresponding average number of nodes visited during a search. Theory suggests that a logarithmic growth is one possibility, so we try the functional form $y = a + b \log x$. Because there are ten y observations at each x value, the `-w` flag calls for a weighted regression: each x value is assigned the mean y value, together with a weight inversely proportional to the sample standard deviation at that x.

The primary graph shows that the best fit equation $y = 15.62 + 0.282 \log x$ is not an accurate description of the data. Note that the small variances at larger x values give them greater weight; the fit is better in that region. The weighted residuals have reasonable dispersal, but their means show a great deal of structure. For this data, the computed parameters have very little meaning; the pictures show that the model is poor. A second experiment, fitting the model $y = a + bx^c$ to this data, shows that the data is accurately described by $19.2 - 26.0x^{-0.39}$.

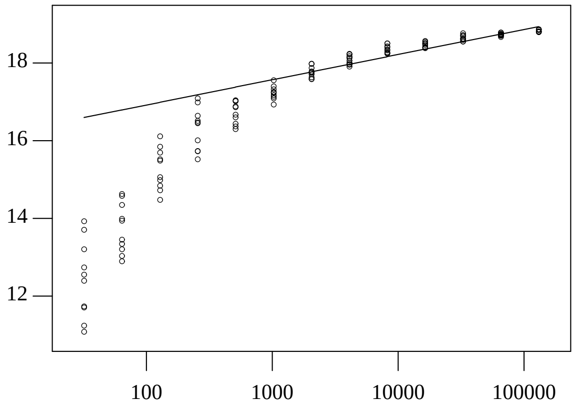
Appendix A describes L2fit in more detail. Because L2fit is a UNIX system filter, it may be combined with other tools on that system. For instance, one may desire to fit data to an exponential model, which is accomplished by this command:

L2FIT SUMMARY

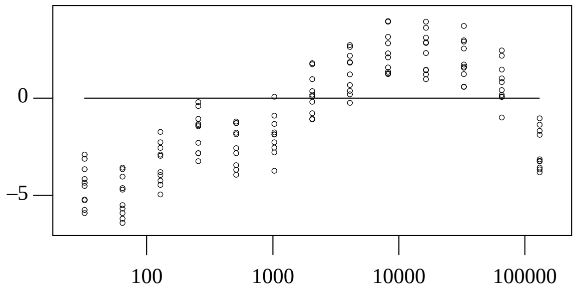
Command line: `l2fit -troff -s0.6 -w -lx a+b*log(x) 1 1 data/kdnodes.d`
Least squares regression type: Weighted
Input data file: data/kdnodes.d
Input expression: a+b*log(x) $a + b \log(x)$
Canonical form: p(1)+p(2)*log(x) $p_1 + p_2 \log(x)$

Parameters	$a = p_1$	$b = p_2$
Initial:	1	1
Final:	15.621	0.281858
Standard errors:	0.484768	0.0440454

Algorithm termination: x- and relative function convergence



Circles: input (x,y) pairs. Line: least squares fit $y = f(x,p)$.



Circles: weighted residuals $[y - f(x,p)]/stddev(x)$.

Figure 2. L2FIT output on search tree data.

```
$ l2fit -lxy 'a * x**b' datafile
```

The mathematical model underlying least-square regression assumes that the residual errors are independently chosen from the same normal distribution. If the errors are normal on a logarithmic scale, however, one should take the logarithm of both variables before performing the regression, as in this command:

```
$ awk '{ print log($1), log($2) }' datafile | l2fit -lxy 'a + b*x'
```

The Awk program takes the logarithm of both fields in the input file. Awk programs are useful for many common tasks in data analysis, such as filtering out subsets of data and re-expressing data. Some people use L2fit to perform a single fit on a single data set. At the other extreme, we have built scripts that perform a dozen L2fit regressions and combine them into a single Troff output file.

No single program can cover all possible cases of data fitting, and L2fit is missing some desirable features. For instance, one might want to specify a Fortran routine as the function to be fit. The underlying `NL2SOL` program can easily handle functions of many variables, but the L2fit interface is restricted to the single independent variable *x*. Interested users can use L2fit on similar problems with the `-t` flag to leave temporary files that may be used as templates for solving the more complex problems.

Appendix B describes the implementation of L2fit in detail. A simple version of L2fit performs only the least-squares regression to calculate the parameters; it does not prepare the graphical summary. It is implemented as a 50-line Awk program and a 40-line Fortran template. The complete L2fit is a 330-line Awk program that uses a 45-line Fortran template; it also uses a 60-line Troff and Grap template to produce the output.

4. A Survey of Problem Domains

In this section we survey template-driven interfaces for Port routines in several problem domains. We start with programs that use the format

<operation> *<expression>* *<lower bound>* *<upper bound>*

The Int program performs numerical integration; here is a numerical approximation to $\int_1^e 1/x \; dx$:

```
$ int '1/x' 1 2.718281828
1.00002317
$
```

The default error tolerance is 10^{-3}; it can be adjusted by an input flag.

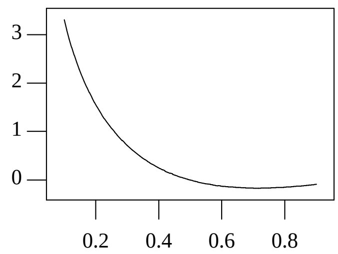
We saw the Root program earlier; here is an invocation to find a root of $x - e^{1-x}(1 + \ln x)$ in the interval $[0.1, 0.9]$:

```
$ root 'x - exp(1-x)*(1+log(x))' .1 .9
0.49673636
$
```

If we want more insight about the function, we can plot it over the interval with the command

```
$ show 'x - exp(1-x)*(1+log(x))' .1 .9
$
```

This produces a file `show.out` that contains a picture like this:



Show has many of the same options as L2fit (logarithmic scales, output suitable for Troff input, saving the temporary files, text labels, etc.), as well as an option for changing the number of points at which the function is plotted. A more sophisticated version of Show can plot multiple equations and *x*, *y* data files. The related Smooth program computes a smooth function through its input *x*, *y* data set; its output is a set of *x*, $Smooth(x)$ pairs, and it produces a picture of the smoothed function as a side-effect.

For some purposes, numeric function values are more useful than plots; the Feval program (for function evaluation) computes such values:

```
$ feval 'x - exp(1-x)*(1+log(x))'
.1
   3.30384
.9
   -0.0887295
.49
   0.0126441
.5
   -0.00591477
$
```

Feval is given a single Fortran expression; it reads a sequence of x values from its input and prints the corresponding output values. As with most programs on the UNIX system, the input is terminated by typing ‘‘Control-d’’. Feval is the only program in this paper designed to be used interactively.

The Min program performs multivariate minimization over a region specified by upper and lower bounds for each variable. We first apply it to our favorite univariate function and interval:

```
$ min 'a - exp(1-a)*(1+log(a))' '.1<=a<=.9'
  0.710633            -0.168715
$
```

The variables to be minimized are named `a`, `b`, `c`, etc., just as in L2fit, and ranges are given as inequalities (both `b>5` and `c<=10` also work). The first output value is the real value of a where the function is minimized, and the second is the value of the function at that real. Here is a more substantial input file, using a long format in which each field is given on a separate line:

```
# Problem 5 from Hock and Schittkowski [1981]
#  Min at a*  =   1/2 - pi/3  ~  -0.547198
#         b*  =  -1/2 - pi/3  ~  -1.547198
#         f(a*,b*) = -sqrt(3)/2 - pi/3 ~ -1.91322
sin(a+b) + (a-b)**2 - 1.5*a + 2.5*b + 1
-1.5 <= a <= 4.0
-3.0 <= b <= 3.0
0
0
```

The first four lines are comments, the fifth line contains the expression to be minimized, the next two lines give the bounds, and the last two lines specify the starting values of the two variables. The file is named `hock5`; it is invoked by this command:

```
$ min hock5
  -0.547196 -1.5472          -1.91322
$
```

The five programs sketched in this section were all straightforward to implement as template-driven Awk programs. They share a library of Awk functions that contains about 150 lines of code (a shorter implementation of L2fit also uses that library). Beyond the library code, Feval takes just 6 lines of Awk, Root and Int are both implemented in 30 lines (including 10-line Fortran templates), Show takes 110 lines (with a 30-line Troff/Grap template), and Min takes 180 lines (with a 50-line Fortran template).

5. Differential Equations

Simple problems in the domains we have seen so far have simple descriptions. We turn now to differential equations, where elementary problems are more difficult to describe. Because of this, problem descriptions are more complicated, even though they often rely heavily on conventions and abbreviations appropriate to the domain. Our goal in this section is not to present the details of the programs, but rather to give the flavor of our succinct descriptions of these complicated mathematical objects.

We start by considering the solution of *ordinary differential equations* or *ODE*s in the explicit form $x'(t) = f(t,x)$ for t in the interval (t_{start}, t_{stop}), where f is a vector-valued function of time t and the solution vector $x(t)$. Given initial conditions $x(t_{start})$ this problem typically has a unique solution. For example, the *ODE* system $S' = C$, $C' = -S$ with initial conditions $S(0)=0$ and $C(0)=1$ has the solution

$S(t) = sin(t)$ and $C(t) = cos(t)$. This *ODE* for S and C is solved numerically by the command:

```
$ odes vars "S,C" ode "S'=C; C'=-S" x 0 1 on 0 1
```

This interface uses the `ODES` solver from the Port library. The fields in this command can be grouped into name-value phrases, where a phrase consists of one or more fields and is usually identified by the first field in the phrase.

`odes`: The name of the underlying numerical software, also the name of this interface.

`vars`: Assigns the names ‘‘`S`’’ and ‘‘`C`’’ to the dependent variables, so $x = (S, C)$.

`ode`: Determines the components f of the *ODE* system to be solved. In this case, $f(t, S, C) = (C, -S)$.

`x`: Initial conditions for the dependent variables, $x(t_{start}) = (0, 1)$.

`on`: The time interval, $t_{start} = 0$ and $t_{stop} = 1$.

The above command is converted into 60 lines of Ratfor, a Fortran preprocessor dialect, by a 140-line Awk program. The Awk script for the *ODE* interface was initially written in a couple of hours. A few additional hours added control of the initial time step, error recovery, etc.

We can solve more complicated problems by considering a broad class of *partial differential equations* (or *PDE*s) in one space variable in the semi-linear divergence form

$$\frac{\partial}{\partial x}\ \mathbf{a}(t,\ x,\ \mathbf{u},\ \mathbf{u}_x,\ \mathbf{u}_t,\ \mathbf{u}_{xt}) = \mathbf{f}(t,\ x,\ \mathbf{u},\ \mathbf{u}_x,\ \mathbf{u}_t,\ \mathbf{u}_{xt})$$

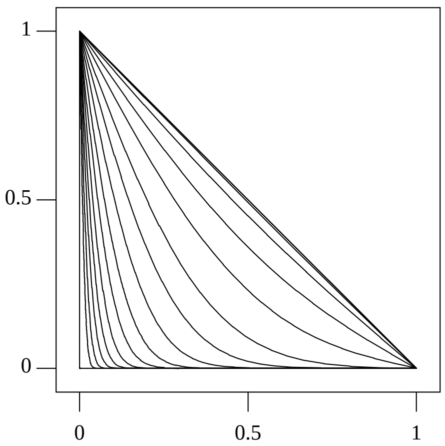
where $\mathbf{u}(t,x)$ is a vector of *PDE* variables, $\mathbf{a}$ and $\mathbf{f}$ are vector-valued functions of their arguments, for $L \le x \le R$ and $u_x \equiv \partial u / \partial x$, etc. The boundary conditions have the form

$$\mathbf{b}_L(t,\ \mathbf{u}(t,L),\ \mathbf{u}_x(t,L),\ \mathbf{u}_t(t,L),\ \mathbf{u}_{xt}(t,L)) = 0$$

$$\mathbf{b}_R(t,\ \mathbf{u}(t,R),\ \mathbf{u}_x(t,R),\ \mathbf{u}_t(t,R),\ \mathbf{u}_{xt}(t,R)) = 0$$

where $\mathbf{b}_L$ and $\mathbf{b}_R$ are vector-valued functions.

We start by considering the heat equation for the temperature variable T. The equation is $T_t = T_{xx}$, over the time range $0 \le t \le 1$ and the space range $0 \le x \le 1$. The initial condition is that $T(0,x) = 0$ and the boundary conditions are $T(t,0) = 1$ and $T(t,1) = 0$. These initial and boundary conditions conflict at $t = x = 0$ and, for small time t, the solution of the *PDE* is approximately $erfc(x/2\sqrt{t})$. Thus, for the default initial time-step of $\sqrt{\varepsilon}$, where ε is the machine rounding error (roughly 5×10^{-9} on the VAX architecture) the first plot is a unit spike of width roughly 10^{-4}. The evolution of this system and its final solution are shown in this graph:



The code has automatically gone from 2 mesh points to 33, enough to model this nasty function on the first time step to the default 0.1% accuracy. The solution finally relaxes to its equilibrium value of $1 - x$

at time $t = 1$ where only 5 mesh points are needed.

The graph above was produced by this command:

```
$ poss vars T af 'd Tx=Tt' bc 'T=1|T=0' t 0 1 x 0 1 u 0
```

We will, once again, sketch the fields in this command by describing the phrases:

`poss`: The program name, for Partial and Ordinary differential equations in Space with Smooth splines (this name was derived from POST, for Partial and Ordinary differential equations in Space and Time).

`vars`: Assigns the single *PDE* variable the name `T` (names must be upper case).

`af`: The equations are given in the form $\partial a/\partial x = f$. The letter `d` specifies the differential $\partial/\partial x$; `d Tx=Tt` states that $T_{xx} = T_t$, where `Tx` denotes T_x and `Tt` denotes T_t, the partials of T with respect to x and t.

`bc`: This specifies the boundary conditions, with left and right separated by the ‘‘`|`’’ character; the left condition is $T(t,0) = 1$, and the right is $T(t,1) = 0$.

`t`: The phrase `t 0 1` gives the range for t, and the next phrase gives the range for x.

`u`: Specifies the initial condition, $u(0,x)=0$.

The underlying Port software generates the necessary Jacobian information, the partial derivatives of a, f and bc with respect to the dependent variables u, u_t, etc., by finite-difference methods. Thus, the user does not need to provide such information. This was crucial for the success of these interfaces; specifying the partial derivatives is too cumbersome for most users. When the user needs exact, instead of approximate, derivatives, the interface uses Maple to compute them symbolically. (Char *et al* [1988] describe Maple.)

Although the description of the heat equation is less than transparent for a naive user, it is much more succinct than traditional approaches to the problem. The above specification is translated into 170 lines of Ratfor. The Awk program that performs the translation is 220 lines long and was originally written in an afternoon; several additional half-hour sessions have since added bells and whistles, such as being able to set storage limits and change integration methods.

We have built interfaces for several kinds of differential equation solvers. The software underlying Poss is an adaptive *PDE* solver in one space variable based on an adaptive mesh spline fitter and the fixed mesh *PDE* solver `POST`; it dynamically determines what mesh to use to obtain the solution to the desired accuracy. Another program solves *PDE*s using a fixed mesh, based on an extension of the `POST` software, while another interface uses continuation code for solving problems that become difficult as a parameter changes. Other interfaces solve stiff ordinary differential equations in implicit form with the `IODE` solver of the Port library. Benvenuti *et al* [1991] use these interfaces to solve a number of problems, including the drift-diffusion, energy balance and fluid dynamics models of semiconductor device behavior.

As an experiment, we built a significantly more sophisticated interface to the Port *PDE* solvers. Figure 3 describes a *PDE* in four variables used for semiconductor device modeling (graciously supplied by Bill Coughran). The specification uses the notation `d[con,x]` to denote $\partial con/\partial x$; this allows it to pass complete information about the partial derivatives to the underlying solver, whereas the earlier code always used finite-difference derivatives. For completeness, Figure 4 shows the same *PDE* described in the original interface, together with some help from the UNIX shell for naming parameters.

The program that processes the description in Figure 3 was constructed using the language development tools described by Kernighan and Pike [1984, Chapter 8]. It serves as an interface to three different but related Port routines, for handling *ODE*s, implicit *ODE*s, and *PDE*s. The program is implemented with 160 lines of Lex (for lexical analysis), 1070 lines of Yacc (for parsing), 2100 lines of C, and a 250-line Fortran template. (All of these files could probably be shortened by half by the coding style that we used in the Awk programs.) In addition to handling the syntactic structure shown above, the program performs extensive error checking; for instance, it identifies the line number in the source file of Fortran syntax errors. This interface also ensures that the variables and equations are appropriate for the type of equation. It also gives more useful error messages; earlier interfaces may pass faulty equations through to the Fortran template, so that errors are reported by the compiler.

```
problem pde
  pdevar pot con temp vel  # potential, concentration, temperature, velocity
  precision double         # Double precision solution
  mesh x 2 [0,2d-2] 101
  error 1d-2 1d-2
# CONSTANT EXPRESSIONS
  K    = 5.9380d-6     # scaled Boltzmann’s constant
  M    = 9.0352d-13    # scaled effective mass
  MU   = 36.193        # scaled mobility
  Q    = 1.7814d-3     # scaled elementary charge
  T0   = 1             # scaled ambient temperature
  VS   = 3.3579d4      # scaled saturation velocity
  V    = 10            # voltage applied at right
  N    = 3.6d7-0.25*(1+tanh(1d3*(x-5d-3)))*(1+tanh(1d3*(1.5d-2-x)))*3.557d7
                       # scaled background doping
  TAUP = M* MU * T0/(Q*temp)
  TAUW = 0.5* M * MU *T0/(Q*temp) + 1.5*MU*temp/((temp+T0)*VS**2)
  KP   = (1.5*MU*con)
# INITIAL VALUES
  init pot  log(abs( N ))+V*x/2d-2
  init con  N
  init temp T0
  init vel  0
# PDES
  a1 = d[pot,x]
    f1 = con-N
  a2 = con*vel
    f2 = 0
  a3 = -con*K*temp
    f3 = M * con * ( vel*d[vel,x]+vel/TAUP ) - Q*d[pot,x]*con
  a4 = KP * d[temp,x]
    f4 = 1.5*con*d[temp,x]*vel+con*temp*d[vel,x]+ \
         con*( M * vel**2/(2 * K) + 1.5*(temp-T0)) / \
         TAUW - con* M * vel**2 / (K * TAUP)
# BOUNDARY CONDITIONS
  bcl_pot = pot-log(abs(N))
    bcr_pot = pot-log(abs(N))-V
  bcl_con = con-N
    bcr_con = con-N
  bcl_temp = temp-T0
    bcr_temp =  temp-T0
  bcl_vel = d[vel,x]
    bcr_vel = d[vel,x]
```

Figure 3. A description of a *PDE*.

There are tradeoffs between Awk and Yacc as implementation languages. Awk is best suited for fast implementation, particularly if the execution speed of the resulting product is not of primary concern. Awk is convenient for string processing and symbol-table management. It is also good at handling keyword-value pairs; it is weak at parsing anything more complicated. By contrast, Yacc is intended for languages with richer grammatical structure, especially nested constructions. A Yacc-based program (with semantic actions implemented in C) will also execute much faster; error detection and recovery are also better. The price of using Yacc is that implementation time will be significantly longer.

6. Other Template-Driven Programs

Template-driven programs apply far beyond interfaces for numerical functions. An obvious extension is to nonnumeric functions: given an efficient sort function, we can make a sort pipe that uses a template C program. Templates could also provide pipe-like interfaces to functions in domains such as regular-expression pattern matching, random number generation, cryptology, and data compression. In many of these applications, however, it is more effective to call the function from a single C program

```
# P   potential
# C   concentration
# T   temperature
# V   velocity
K="(5.9380e-6)"     # scaled Boltzmann"s constant
M="(9.0352e-13)"    # scaled effective mass
MU="(36.193)"       # scaled mobility
Q="(1.7814e-3)"     # scaled elementary charge
T0="(1)"            # scaled ambient temperature
VS="(3.3579e4)"     # scaled saturation velocity
N="(3.6e7-0.25*(1+tanh(1e3*(x-5e-3)))*(1+tanh(1e3*(1.5e-2-x)))*3.557e7)"
                    # scaled background doping
VB="(10)"           # voltage applied at right
TAUP="($M * $MU * $T0/($Q*T))"
TAUW="(0.5* $M * $MU *$T0/($Q*T) + 1.5*T/((T+$T0)*$VS**2))"
KP="(1.5*$MU*C)"
poss                                                                        \
  vars "P,C,T,V;"                                                           \
  af "d Px = C-$N;                                                          \
      d C*V = 0;                                                            \
      d -C*$K*T = $M * C * ( V*Vx+V/$TAUP ) - $Q*Px*C;                      \
      d $KP * Tx = 1.5*C*Tx*V+C*T*Vx+ C*( $M * V**2/(2 * $K) +              \
                   1.5*(T-$T0))/ $TAUW - C* $M * V**2 / ($K * $TAUP)"       \
  bc "P-log(abs($N)); C-$N; T-$T0; Vx |                                     \
      P-log(abs($N))-$VB; C-$N; T-$T0; Vx"                                  \
  k 2 x on 0 2e-2 ndx 101                                                   \
  u "(log(abs( $N ))+$VB*x/2e-2)" "$N" "$T0" "0"                            \
  errpar 1e-2 1e-2
```

Figure 4. A Poss Description of the *PDE* in Figure 3.

without using a template.

Templates are most effective when an object must be processed by a compiler. Most of the examples we have seen so far in this paper deal with Fortran expressions; it is possible to build and interpret an expression tree at run time, but it is easier to pass the expression to the compiler via a template. We will now study a program that uses a template to study the run-time cost of various C expressions. Here is most of the template, which repeatedly evaluates the expression in a tight loop:

```
#include <math.h>
main()
{       double  d1, d2, d3, d4, d5;
        int     n;

        d1 = d2 = d3 = d4 = d5 = 0.0;
        @INIT@;
        for (n = @N@; n > 0; n--) { @EXP@; }
}
```

The complete schema also includes declarations and initializations for variables of type `int` and `float`.

Here is an interactive session with the resulting `ctime` program:

```
$ ctime
set d2 = 31.4159; d3 = 27.1828
d1 = d2 + d3
  1.0 mics/loop      0.1u 0.0s 0r
d1 = d2 / d3
  4.0 mics/loop      0.4u 0.0s 0r
d1 = floor(d2)
  10.0 mics/loop     1.0u 0.0s 1r
d1 = sin(d2)
  31.0 mics/loop     3.1u 0.0s 4r
d1 = log(d2)
  30.0 mics/loop     3.0u 0.0s 4r
d1 = sqrt(d2)
  41.0 mics/loop     4.1u 0.0s 5r
$
```

The first line uses the `ctime` keyword `set` to assign the variables `d2` and `d3` nonzero values. Subsequent pairs of lines contain the expression typed by the user followed by the averaged cost and the output of the UNIX `time` command. This timing program is by no means perfect: an optimizer could move some computation out of the timing loop, and it ignores caching. Nevertheless, we have found this program to be useful in its limited domain.

Apart from its applications, though, the `ctime` program illustrates the power of template-driven programming. The complete programs is about 60 lines: 20 lines of template, 20 lines of primary Awk functions, and 20 lines of supporting Awk functions that we copied from other programs (such as template instantiation and error message reporting). We wrote the `ctime` program in a couple of hours, and converted it to the Fortran `ftime` in about ten minutes. In a couple of hours we built a version that times C constructs across a half dozen types of machines that are easily reached by network from our home machine. The template-driven style made it easy to construct all of these variants.

7. Conclusions

The interfaces that we have described provide easy access to most of the major areas in the Port library. We have aimed at handling the common cases, which cover a high percentage of users, if not a high percentage of the total CPU time used in Port routines. Although many of our users are novices, we have been pleasantly surprised that numerical experts also seem to make extensive use of the interfaces.

The template-driven interfaces are easy to build in hours or days; they don't require weeks. The investment of time in building one is usually returned, with interest, in the first few applications. We believe that building an interface can easily pay for itself as a numerical routine is being tested and debugged. The quality of our software improved as a result of having these interfaces available: we can now quickly answer questions about the functional behavior and time complexity of our routines, and generate broad but short test cases.

The code for several of these interfaces is available from `netlib`. To receive an index to the collection and instructions on how to get what you want, type

```
echo "send index from templates" | mail netlib@research.att.com
```

The semi-symbolic nature of these Awk interfaces has changed the way we build numerical routines in Fortran. For example, in the solution of two-dimensional *PDE*s, the coupling of variables is important. If the variables are coupled in a nonlinear lower-triangular manner, then various iterative methods may be cost-effective compared to the default direct factorization schemes. An interface could ask the user to state whether the problem is triangular, but that adds complexity for the user and may lead to errors. It is easy for an Awk program to determine whether a given expression contains a specified variable, and thereby compute the triangular structure of the equations to be solved.

The programs that we describe all operate independently. Most are batch programs, not interactive. They are not cooperative: there is no elegant way to take the output of one into another. Integrating these interfaces into the philosophy of scientific computing tools proposed by Coughran and Grosse [1989] remains an important open problem.

Acknowledgments

We are grateful for the helpful comments of Rick Becker, Bill Coughran, Eric Grosse, Colin Mallows, Daryl Pregibon, Margaret Wright, and several anonymous referees.

References

Aho, A. V., B. W. Kernighan and P. J. Weinberger [1988]. *The AWK Programming Language*, Addison-Wesley, Reading, MA.

Bentley, J. L. and B. W. Kernighan [1986]. ‘‘GRAP — A Language for Typesetting Graphs’’, *Communications of the ACM 29*, 8, August 1986, pp. 782-792.

Benvenuti, A., M. R. Pinto, W. M. Coughran, Jr., N. L. Schryer, C. U. Naldi and G. Ghione [1991]. ‘‘Evaluation of the influence of convective energy in HBTs using a fully hydrodynamic model’’, *IEDM Technical Digest 91*, pp. 499-502.

Chambers, J. M., W. S. Cleveland, B. Kleiner and P. A. Tukey [1983]. *Graphical Methods for Data Analysis*, Wadsworth, Boston, MA.

Char, B. W., K. O. Geddes, G. H. Gonnet, M. B. Monagan, S. M. Watt [1988]. *MAPLE Reference Manual, Fifth Edition*, WATCOM Publications Limited, Waterloo, Ontario, Canada.

Coughran, W. M. and E. H. Grosse [1989]. ‘‘A philosophy for scientific computing tools’’, *SIGNUM Newsletter 24*, pp. 2-7.

Dennis, J. E., Jr., D. M. Gay and R. E. Welsch [1981]. ‘‘An adaptive nonlinear least-squares algorithm’’, *ACM Transactions on Mathematical Software 7*, 3, pp. 348-368.

Fox, P. A. [1984]. *The PORT Mathematical Subroutine Library*, AT&T Bell Laboratories, May 8, 1984.

Gill, P. E., W. Murray, S. M. Picken and M. H. Wright [1979]. ‘‘The design and structure of a Fortran program library for optimization’’, *ACM Transactions on Mathematical Software 5*, 3, pp. 259-283.

Hock, W. and K. Schittkowski [1981]. *Test Examples for Nonlinear Programming Codes*, Springer-Verlag, Berlin.

Kernighan, B. W. and R. Pike [1984]. *The UNIX Programming Environment*, Prentice-Hall, Englewood Cliffs, NJ.

Rice, J. R. [1989]. ‘‘Composition of libraries, software parts and problem solving environments,’’ Purdue University Computer Sciences Department CSD-TR-852, January 1989, 13 pp.

Appendix A. L2FIT Man Page

NAME

l2fit — Least-Squares Fit

SYNOPSIS

l2fit [option ...] expression [startvals ...] [filename]

DESCRIPTION

`l2fit` fits the function given in the expression to *x, y* pairs in a data file or on the standard input. Suppose, for instance, that `datafile` contains pairs in which *y* is a quadratic function of *x* plus perhaps some other ‘‘error’’ or ‘‘noise’’. This command finds the three quadratic coefficients $p(1)$, $p(2)$ and $p(3)$:

```
l2fit 'p(1)*x**2 + p(2)*x + p(3)' datafile
```

The values of the three parameters appear in order on the standard output, and as a side effect `l2fit` writes the Troff output file `l2fit.out` in the current directory. That file is a one-page summary of the fit that includes numbers (such as standard errors) and graphs of the function and the residuals. One should almost always study the `l2fit.out` file before relying on the data summarized on standard output.

The program assumes that the *x, y* input pairs are modeled by the equation $y = f(x,p)$; it attempts to find a vector *p* that minimizes the sum of the squares of the residuals $y[i] - f(x[i],p)$. The function *f* is given as the Fortran expression `expression`; it may contain only the variables *x* and $p(1)$, $p(2)$, ... `l2fit` recognizes ‘‘ˆ’’ as an abbreviation for ‘‘**’’, it recognizes ‘‘*a*’’ and ‘‘$p1$’’ as abbreviations for ‘‘$p(1)$’’, it recognizes ‘‘*b*’’ and ‘‘$p2$’’ as abbreviations for ‘‘$p(2)$’’, etc.

The algorithm starts with an initial guess of the vector *p*, and refines the guess by Gauss-Newton iteration until it arrives at a local minimum (which may not be a global minimum). `l2fit` uses the default starting vector of zero. If the algorithm is unable to find an optimal value for *p* starting from zero, the user may provide a starting vector by giving its floating-point components following `expression`. Thus if the program `gen.data` creates quadratic data that we expect to be well modeled by $10x^2 - x + 3$, we can fit the data with the following command:

```
gen.data | l2fit 'a*xˆ2 + b*x + c' 10 -1 3
```

The main work of `l2fit` is performed by the `dn2f` routine from the Port library. By default, `l2fit` uses unweighted regression. If there are at least two distinct *y* values for each *x* value, then a weighted least-squares regression invoked by the `-w` flag uses the sample variance at each x value to increase the accuracy of the regression. `l2fit` recognizes several options presented as flags:

`-w`	weighted regression
`-t`	save all temporary files; see FILES below
`-i`	invisible: do not produce the file `l2fit.out`
`-lx`	log x scale in Grap output
`-ly`	log y scale in Grap output
`-lxy`	log both scales in Grap output
`-x`Text	description of x variable
`-y`Text	description of y variable
`-s`Number	shrink the graphs in `l2fit.out` by this factor
`-c`Text	comment: text ignored
`-troff`	prepare `l2fit.out` for inclusion in a Troff document with the `.so` command

Complicated data fitting can lead to irritatingly long command lines. `l2fit` may therefore be invoked with a single argument that names a file. That file contains the arguments in the order described above, one argument per line. Beware that shell characters in the file (such as quote marks) are passed through untouched and are not interpreted by the shell. Characters on a line following the pound sign # are discarded as comments.

EXAMPLES

```
l2fit -lx -w -t 'a + b*x^c' 20 -1 -.5 searchdata.d
```

Fit the data in `searchdata.d` to the function $a + b*x**c$, using the starting values $a = 20$, $b = -1$, $c = -0.5$. Because each x value has many y values, we use a weighted regression (`-w`). The graphs in the output file `l2fit.out` have logarithmic x scales (`-lx`), and the temporary files `l2temp.*` are not deleted (`-t`).

FILES

`l2fit` uses several files in the current directory of the form `l2temp.*`; the `-t` flag causes these temporary files not to be deleted. The most useful of these files are:

`l2temp.f`

This is the Fortran program that calls the optimization routine. If your function will not fit in a single expression, you might consider modifying this file to solve your problem.

`l2temp.g`

This is the Grap file (including some Troff) that eventually makes `l2fit.out`. It may be processed with the pipeline `grap l2temp.g | pic | eqn | troff > l2fit.out`.

`l2temp.p`

This is the file produced by the Port routine `dn2f`. It can be helpful for studying details of the least-squares computation; it is especially handy when the program fails to converge. See the Port documentation for details.

SEE ALSO

P. A. Fox, The PORT Mathematical Subroutine Library, AT&T Bell Laboratories, May 8, 1984.

BUGS

There are many contexts in which `l2fit` fails to find the ‘‘right’’ answer. Sometimes the Port `dn2f` routine detects convergence problems; in that case an error message is produced on `stderr` and in `l2fit.out`. In other cases the routine converges to a local minimum that is a poor fit to the function; the graphs of the function fit and residuals usually provide evidence of this. Problems such as these are remedied by supplying either a more appropriate function or better starting values.

The program requires a lot of CPU cycles. On a VAX-8550, a 100-line data file requires about 3 seconds to compile and load the Fortran program, 2 seconds to execute the program to find optimal values, 3 seconds to prepare `l2fit.out` (the `-i` flag avoids the output and the corresponding CPU time), and about 2 seconds in the Awk program itself. Good starting values can slightly decrease the optimization time, while poor values can greatly increase it.

Because the program uses temporary files with fixed names, only one invocation of the program may be run at a time in a given directory.

Appendix B. L2FIT Implementation

This appendix describes the implementation of L2fit. We will first describe a subset of L2fit in detail, and then sketch the complete program.

The first version of L2fit does not allow the user to supply starting parameters; it always starts at the zero vector. To write the Fortran program that calls the Port library routine `dn2f`, it uses a *template* in which variables surrounded by at-signs (‘‘`@`’’) are replaced as the template is instantiated as a Fortran program. In this particular program, `@N@` and `@NP@` are replaced by integers, and `@EXP@` is a Fortran expression. (For historical reasons, the program refers to the template as a schema.)

```
c Schema for Fortran l2fit program, with following subs made:
c  @N@:   Number of x, y observations
c  @NP@:  Number of p parameters
c  @EXP@: Expression to be fit
         integer i, iv(2000), liv, lty, lv, ui(1), lp
         double precision v(1000)
         external dummy, resid
         double precision ty(@N@,2), p(@NP@)
         data liv /2000/, lv /1000/
         data lty /@N@/, lp /@NP@/
         do 10 i = 1, lp
                  p(i) = 0.0
   10    continue
         do 20 i = 1, @N@
                  read (*, *) ty(i,1), ty(i,2)
   20    continue
         ui(1) = lty
         iv(1) = 0
         call dn2f(lty, lp, p, resid, iv, liv, lv, v, ui, ty, dummy)
         stop
         end
c
         subroutine dummy
         return
         end
c
         subroutine resid(n, lp, p, nf, r, lty, ty, uf)
         integer n, lp, nf, lty
         double precision p(lp), r(n), ty(lty, 2)
         external uf
         integer i
         double precision x
         do 10 i = 1, n
                  x = ty(i, 1)
                  r(i) = ty(i, 2) - (@EXP@)
   10    continue
         return
         end
```

This program is similar to the example in the Port manual. The main program reads the data and calls the DN2F optimization routine. The `resid` routine computes the residuals given the current vector. The `dummy` routine performs no action, but is required by DN2F.

We turn now to the minimal L2fit program, which is a subset of the complete program. The command line contains a Fortran expression (in the independent variable x and the vector components $p(i)$) and an optional file name. The output is the computed parameters on the standard output.

```
BEGIN {
  # Get arguments from command line
        expr = ARGV[1]
        ARGV[1] = ""
  # Make datafile and compute n = # of x, y pairs
        while (getline > 0) {
                print $1, $2 >"junk.d"
                n++
        }
        close("junk.d")
  # Compute m = number of p(i) elements in exp
        s = expr
        gsub(/[ \t]/, "", s)
        while (match(s, /p\([0-9]+\)/)) {
                i = 0 + substr(s, RSTART+2, RLENGTH-3)
                if (i > m) m = i
                s = substr(s, RSTART+RLENGTH)
        }
  # Write, compile and execute Fortran program
        subarr["N"] = n
        subarr["EXP"] = expr
        subarr["NP"] = m
        doschema("l2fit.min", "FORTRAN", "junk.f", subarr)
        system("f77 junk.f -lport 2>/dev/null; a.out <junk.d >junk.c")
  # Extract and print answers
        while (getline <"junk.c" > 0) {
                if      (state == 0 && $2 == "FINAL") state = 1
                else if (state == 1 && NF == 0) state = 2
                else if (state == 2 && NF == 0) state = 3
                else if (state == 2 && NF == 4) printf "%g ", 0+$2
        }
        printf "\n"
}

function doschema(schemafile, marker, outfile, subarr,  temp, i) {
        while (getline <schemafile > 0)
                if ($1 == "@@@" && $2 == marker) break
        while (getline <schemafile > 0) {
                if ($1 == "@@@") break
                temp = $0
                for (i in subarr)
                        gsub("@" i "@", subarr[i], temp)
                print temp >outfile
        }
        close(schemafile)
        close(outfile)
}
```

The main tasks of the Awk program are described in the comments. Computing the number of $p(i)$ elements in the Fortran expression uses standard Awk tricks with regular expressions, and extracting the answers from the Port output uses a four-state machine.

The most interesting part of the program is the function `doschema`, which is presented in full generality of template processing (even though this minimal L2fit makes only restricted use of it). It is passed the name of `schemafile`, which (potentially) contains several schemas, each of which starts with a line of the form

```
@@@ <markername>
```

The next argument is the name of the marker of the desired schema. The third argument is the name of the output file, which contains the instantiated schema. The fourth argument is the substitution array, so `subarr["EXP"]` contains the text string that is substituted for `@EXP@`. This routine is similar to the ‘‘form letter’’ program discussed in Section 4.2 of Aho, Kernighan and Weinberger [1988].

Because it can be a bother to keep track of several distinct but related files, we package the template and the program together in a single file. The following packaging can be copied by all and understood by the UNIX adept:

```
# l2fit.min: min version of least-squares fit of a data file to an expression
cat >/dev/null <<'ENDSCHEMAS'
@@@ FORTRAN
   [Fortran schema goes here]
@@@
ENDSCHEMAS
awk '
   [Awk program goes here]
' "$@"
```

The complete version of L2fit is a natural extension of this miniature version. The Fortran template is about 45 lines long (an increase of 5 lines); it contains code to initialize the vector $p(i)$ from an input file. The program also contains a 60-line template of Troff and Grap commands that are used to make the file `l2fit.out`.

The most substantial change is to the Awk program; the complete version is about 330 lines long. It performs a great deal more error checking: Does each line in the data file contain two numeric fields? Is the command line in the proper format? It allows the arguments to appear either on the command line or in a specified file, one argument per line, with comments. It performs a number of data processing tasks, such as sorting and computing means and standard deviations (for weighted regressions). It uses Awk's regular expressions to convert among the Fortran, Grap and Eqn languages.