A FORTRAN PRIMER

A brief introduction to the principal scientific computing language

John D. Landstreet
Department of Physics and Astronomy
University of Western Ontario

Introduction

FORTRAN is one of the oldest high-level (i.e. intelligible to humans) computing languages. It developed essentially in response to the need for a language for doing numerical mathematics, and it has always specialized in this field. It is not as flexible as some other languages, particularly in that it communicates with the operating system of the computer to only a limited extent, but in its specialized field of numerical computation it is an extremely powerful language. It is the computing language used by many (perhaps most) physicists and astronomers today. A vast literature of well-tested programmes is available in the language, which is certainly one of the reasons for its continuing popularity.

FORTRAN has gone through much evolution, developing from FORTRAN 66 to FORTRAN 77 to FORTRAN 90. FORTRAN 77 is the version of the language used by most people, with the richest literature of available programmes, but FORTRAN 90 (which contains new constructs useful for matrix mathematics and for parallel computing, for example) is gradually coming into more use.

FORTRAN is a language with a large number of constructs and rules, but it is possible to master the essential ideas and some of the most common constructs, to the point of being able to write useful programmes, in a few hours. This introduction to FORTRAN tries to teach you elementary FORTRAN by use of an accompanying example programme, fitline.f. I assume that you have been exposed to some programming (e.g. BASIC, TURING, PASCAL, etc), so that the fundamental ideas of programming are somewhat familiar. This introduction focuses on teaching you the specifics of FORTRAN.

The programme fitline.f

Look at the accompanying programme, fitline.f, which is listed at the end of this document and available in the lab. This programme does a very simple task. It reads in data from a short file with \( n_{ap} \) measurements of an independent variable \( x_n \) and a dependent variable \( y_n \). Each measurement of \( y_n \) has an accompanying measurement error \( \sigma_n \). For example, the \( x_n \) could be a series of times, and the \( y_n \) could be the corresponding position of an unaccelerated ball rolling along a horizontal track; \( \sigma_n \) would then be the uncertainty or error in the value of \( y_n \) at time \( x_n \). Now suppose these data define an approximately linear series. If you guess (from inspection, for example) the coefficients \( A \) and \( B \) for a simple straight line fit, \( u_n = Ax_n + B \), to the set of measurements, what fitline.f does is to evaluate the quality of your chosen fit by calculating the value

\[
\chi^2 = \sum_{n=1}^{n_{ap}} \frac{(u_n - y_n)^2}{\sigma^2}.
\]

For \( n_{ap} \) data points, the value of \( \chi^2/(n_{ap} - 2) \), called the reduced chi-squared of the fit, should be a number of order 1; and the smaller the number, the better the fit. Using fitline.f, you can guess successive values of \( A \) and \( B \) and discover which ones give the best fits to the data set.
This is of course a very simple task, and probably much less than you would want such a
programme to do for you. It would not be hard to extend \texttt{fitline.f} to actually determine the
coefficients $A$ and $B$ of the best fit, decide whether a straight line is an appropriate fitting
function, etc, etc. I have limited the programme to this one simple task of simply computing
the reduced chi-squared of a chosen straight-line fit in order to keep the programme short
and easy to follow. It is designed to teach you FORTRAN, not do very much useful work.

\texttt{fitline.f}, line by line

Now examine the programme listing. It starts with a line

\begin{verbatim}
program fitline
\end{verbatim}

that simply announces the name of the programme. This line (and all other lines intended
for the computer to execute) is indented by 6 spaces. The first six spaces on each line of
instructions is reserved for numbering for statements (you will see how numbered statements
are useful below), and for a continuation symbol ($) in space 6, which indicates that the
line is a continuation of the previous line. In addition, FORTRAN ignores all characters
that appear after space 72 on each line. Note also that FORTRAN is insensitive to case –
and it ignores spaces (except for the initial six), although they are very useful for making
the programme more readable!

The next line starts with c. Any line which has a c in space 1 is considered a comment
and is ignored by the computer. \textit{It is essential to document your programme with comments
as you write so that you can recall its logic later, or someone else can read it.} Roughly
speaking, about half the lines of a programme should be explanatory comments. It is
particularly valuable to set out the purpose of the programme at the beginning – and also
to give it a usefully mnemonic name if possible, so that you can find it again later.

Then we have

\begin{verbatim}
real x(20),ydata(20),yfit(20),sigma(20),A,B,rdata
integer n,ndata
\end{verbatim}

FORTRAN has a number of data types, such as integers, real number, double precision real
number (which are just real numbers with more digits of precision than usual), complex
numbers, (ascii) characters, etc. Each of these variable types requires a specific amount of
storage to be set aside in memory for every variable needed in a programme, and so the
type of each variable must be specified. Most data types must be identified at the start of
the programme (before any executable commands are issued), but real and integer variable
may be introduced as you go; their types are (usually) identified by a naming convention.
FORTRAN variable names are made up of combinations of up to six letters and numbers
(starting with a letter). FORTRAN assumes that any variable name that starts with one
of the letters i – j – k – l – m – n is an integer, while any other initial letter denotes
a real variable. However, it is good programming practice to spell out the types of all
the variables you use in your programme, as I have here. (You can override the naming
convention identifying integers and reals if you wish.)

Some of the variables are actually \textit{arrays}. Thus the variable \texttt{x} has 20 memory spaces
set aside for it, for \texttt{x} values \texttt{x(1)}, \texttt{x(2)}, ... \texttt{x(20)}. The statement

\begin{verbatim}
real x(20)
\end{verbatim}

specifies that the real variable \texttt{x} is an array for which the programme reserves 20 real number
spaces (elements) in memory. Arrays of higher dimension are also possible.

Following the prologue in which data types are announced, etc, we start on the com-
mands which are actually executed by the programme as it runs. First,

\begin{verbatim}
open(unit=4,file=’fitpar.dat’,status=’old’)
\end{verbatim}
read(4,*) A, B
open(unit=5, file='data.dat', status='old')
read(5,*)ndata
The first and third lines "open" disk files, which means that the programme can access the contents. Each file is given a particular code number (the unit number) used later when the file is actually read. Unit 4 is the code number given to the file fitpar.dat which contains your guess of the fit parameters A and B; unit 5 is associated with the file data.dat containing the data that you want to fit. The status='old' setting tells the programme that it should expect to find fitpar.dat and data.dat already present on the disk, and should report an error if it does not; it is also possible to open new files (status='new') which should not previously exist, or reuse (write over) files already in existence (status='unknown').

After opening an input file, data is read from it with read commands. FORTRAN normally reads a file line by line; one read command usually reads one line (although it can be instructed to read several lines one after another explicitly). You can read all the data on one line or only part of the data, but if you tell FORTRAN to read eight numbers from a line where there are only four, the programme will report an error and quit.

A read command like
read(4,*) A, B
tells the programme to read the values of two numbers on the first line of the disk file assigned to unit code 4, and store them in the memory locations labeled A and B. The asterisk after the unit number tells the programme to assume that the two variables are on one line, separated by one or more spaces. (The format of the data on the line can also be specified more precisely, as we will discuss below.)

The line
rdata = real(ndata)
tells the programme to convert the integer ndata into a real number and store the real number as the variable rdata (a new – real – variable is created; the existing integer variable ndata is not changed in any way by this translation operation).

The next line
print*, 'Fit tested: y = ', A, '*x + ', B
provides some instant feedback to the programme user. This command prints a line on the screen announcing the fit to be tried by the programme; it intermixes pre-assigned text (such as Fit tested: y = ) which is identified by single quotes, with commands to print the values of A and then later B. Successive print operations are separated by commas. The asterisk again tells the programme to use its own judgment as to how to format A and B. Frequent print commands like this are very useful for debugging programmes, as they tell you how far you have gotten correctly.

We then come to a very useful construct, the block-if structure. This construct allow the programme to follow different paths depending on the value of some variable or on whether some condition is satisfied. The basic structure is
if (some condition is true) then
    some commands
elsif (some other condition is true)
    some commands
else
    still other commands
endif
Only the first and last lines of this structure are required, or several different elseif
commands may be included, as needed. The conditions are often of the form \( \text{ndata} \geq 20 \), a logical statement which is true if the variable \( \text{ndata} \geq 20 \) and false otherwise. The logical relations \( \geq \) (greater than or equal to), \( \lt \) (strictly less than), \( \text{eq} \) (equals), etc. have obvious abbreviations.

Here I have used the block-if statement to test if the data set under consideration is small enough to fit into the arrays I have set up. If \( \text{ndata} > 20 \) then there are too many data points for the amount of storage created initially, and the programme stops with a warning message. (Such test sprinkled through the programme are a valuable aid to debugging when the time comes.) If the data set is small enough, the programme is allowed to proceed.

In this case, we enter another of FORTRAN’s central programming constructs, the do-loop. This is a series of instructions that is repeated until some condition is satisfied; usually the instructions are repeated for a specified number of times. The loop has the form

\[
do \ 10 \ n = 1, \\text{n tot}
\text{some commands}
10 \ continue
\]

which causes the commands between the \( \text{do } 10 \ n = 1, \text{n tot} \) and the \( 10 \ continue \) lines to be executed \( \text{n tot} \) times. The loop ends at whichever statement carries the number 10; almost always this is a “dummy” \( continue \) statement, as shown here.

In this loop, two things are done: the first command reads one line from the file associated with unit 5 (repeated commands read successive lines; FORTRAN keeps a pointer going on each open file to remind it what it has read so far), and the second command computes and stores the \( y \) value of the straight line fit. The computation of the fitted value would often be done right here, with explicit commands, but I have chosen to do this by calling a function \( rline(x,A,B) \) which is defined in a separate (tiny) programme below. I have computed \( yfit \) in this roundabout way to demonstrate the syntax of a function call, because calling externally defined functions and subroutines is pretty common programming practice.

An externally defined function is set up like the function \( rline(x,A,B) \) defined at the end of the main programme. The main programme usually passes one or more numbers to the function (in this case a value for \( x, A, \) and \( B \)), and the function computes and returns to the main programme a single number which is associated with its name. This is of course very similar to normal usage of function names; \( \text{exp}(x) \) is a shorthand for the \( value \) of the exponential function \( \text{exp} \) for some value of \( x \). Our function, \( rline(x,A,B) \), simply calculates the value of \( rline = A * x + B \). Note that an external function is a watertight box that can have line numbering, variable names, etc., that are completely independent of those in the main programme; \( x(n) \) in the main programme becomes \( u \) in our function. This allows functions and other sub-programmes to be written once but used over and over in various programmes. The sub-programme finishes its work with \( \text{return} \) which tells the computer to return control to the calling programme.

FORTRAN also has a lot of built-in (library) functions as well that you can use as needed. The value of a function is evaluated by simply naming it with an argument. For example, to set a variable \( y \) equal to the value of the exponential function \( e^{1.5} \) you would write

\[
y = \text{exp}(1.5)
\]

and to evaluate the exponential \( e^x \) for whatever value the variable \( x \) happens to have right now, write

\[
y = \text{exp}(x)
\]

Other useful functions include \( \text{sin} \), \( \text{cos} \), \( \text{tan} \), \( \text{alog} \), \( \text{alog10} \), and \( \text{abs} \) (respectively the
sine, cosine, tangent, natural log, base 10 log, and absolute value), as well as more exotic functions such as the complementary error function \texttt{erfc}.

At this point, we have read in the parameters of the straight line fit, read in the data set \(x(n), y(n)\) and \(\sigma(n)\), and computed for each data point \(x(n)\) the value of \(yfit(n)\) predicted by the fit. It still remains to compute the reduced chi-squared of the fit. This is done by means of another separate programme unit, a \texttt{subroutine} called \texttt{chisq}. \texttt{Chisq} is called like an external function, but now no value is assigned to the name of the subroutine, but instead all values to be returned by the subroutine are just listed among the variables passed by the call. Thus

\[
call \texttt{chisq(ydata, yfit, sigma,ndata, chi2)}
\]
sends the values of \texttt{ydata, yfit, sigma,} and \texttt{ndata} to the subroutine, and receives in return the value of \texttt{chi2}. This allows a sub-programme to compute more than one number on each call. In this case, we could have done the job with a function (or even within the programme), but this illustrates the syntax for a subroutine call.

The subroutine again looks very much like a normal programme, with variable types, block-if statements, a do loop, and so on. You should be able to figure out what it does by examining the statements in it. However, it illustrates one feature of most programming languages that may be confusing. It contains the statement

\[
\texttt{chi2 = chi2 + ((y1(n) - y2(n))/s(n))**2}
\]
which is not to be read like an normal mathematical statement. Instead what this statement means is that in this step of the do loop, the programme reads the current value of \texttt{chi2}, adds to it the value of the second term, and stores the result in the variable storage location \texttt{chi2} in place of what was there previously. Thus the value of the variable \texttt{chi2} is changed on each step of the do loop as terms are added to it, one by one. (Incidentally, the FORTRAN notation for raising a number \texttt{C} to the power \texttt{D} is \texttt{C ** D}.)

The main programme finishes by printing a summary of the results to the screen for instant gratification, and by opening and writing to a file \texttt{out.dat} which contains a fuller report of the programme’s work. The statements that write to \texttt{out.dat} contain instructions for formatting the output through the association of a \texttt{format} statement with each \texttt{write} statement. Thus the statements

\[
\texttt{write(6,600) A, B}
\]

\[
\texttt{600 format(’Fit tested: Y = ’,f8.4,’*X + ’,f8.4)}
\]
tells the programme to write the values of the variable \texttt{A} and \texttt{B} to unit number 6 (the file \texttt{out.dat}) interleaved with pre-assigned text in quotations. The format of the numbers is specified with \texttt{f8.4} which says to write a real number in eight spaces, with four following the decimal place (woe to you if the number is too big or too small for this format). The number could be written in scientific notation as \texttt{e10.3} which tells the programme to write 23.02 as 0.230e2. Integers are specified as \texttt{14}, for example, which tells the programme to write the number in four spaces. Write statements may be given expressions as well as simple variable names to write.

Similar format statements may be used on input if desired, but are commoner on output. On input they are useful if the data are organized into neat columns, especially if you want to skip some of the data.

The programme finishes with a \texttt{stop} command, followed by \texttt{end}: that’s all, folks!

\textbf{Compiling and running a programme}

FORTRAN is a \texttt{compiled} language, which means that the list of commands is converted from commands that are (more or less) intelligible to the \texttt{programmer} to commands that
are intelligible to the *computer* by a programme called a **compiler**. The FORTRAN source programme is written in a way which is (almost) independent of the machine on which it is intended to run (some compilers support useful "extensions" of the language, but these may not be portable from one machine to another, and so are best avoided). But the machine on which the programme runs has a very particular syntax for executable programmes.

Thus the compiler depends on the machine architecture *and* on the operating system that runs everything else. The compiler for a Sun UNIX work station is different from the compiler for a Linux PC, which is different from the compiler for the same PC running Windows. These compilers even have different names. For UNIX the compiler is usually called **f77** (for FORTRAN 77, it's different for FORTRAN 90); for Linux it is either **f77** or **g77**. For Windows the old Microsoft compiler we have is called **fl**, but you will use a newer, more powerful compiler from Salford Software called **ftn77**. Furthermore, each specific compiler has (many, many) options, most of which are found on most compilers but specified in different ways. Consequently, it is difficult to give general instructions for compiling a programme on many different machines. I will tell you how to do this on UNIX, Linux, and on Windows.

For UNIX and Linux systems, compile the file **fitline.f**, which contains your programme, with the following command, given in the directory where the programme is located (the > symbol is not typed by you; it is the prompt!):

> f77 -o fitline fitline.f

This should only take a couple of seconds to execute. It will result in the creation of an executable (object) file named **fitline** which is what you will need to actually execute the programme (the -o **fitline** part of the compilation command specified this name for the executable file). If you have made some typing (or other) errors, the compiler may find them and report them to you. In this case, fix them and try again. Unfortunately, many types of errors sneak by this step.

Once your programme is compiled successfully, you can try running it. You will need the data files **fitpar.dat** and **data.dat** in the same directory as **fitline**. Execute the programme by giving as a command the name of the executable file you have generated:

> fitline

Nothing more should be needed; the programme should find its input files, execute, print its few lines to the screen and to the file **out.dat**, and exit. If something goes wrong, you may have error messages from the operating system or simply clues (or rubbish) in the screen printout and/or in **out.dat**. If everything goes correctly, you can use a graphics tool (e.g. **xmgr** on UNIX or Linux) to plot the input data and the resulting fit as reported in **out.dat**. If you don't like your fit, change the parameters and try again. As **fitline** reports smaller and smaller values of the reduced chi-squared, you should like the fit more and more.

The same basic steps are needed on a Windows machine. First, make sure that the FORTRAN compiler **ftn77** is available on your machine, by using the **find** command under Windows to look for a programme named **ftn77.exe**. Then you need to open an MS-DOS (command line) window. Create your own working directory (folder) on the machine, naming it after yourself so we can tell it is yours

> mkdir homer

(I assume your name is Homer), and below this folder, create another called (say) **fortran**. Change to this directory with the command

> cd fortran

From this directory, copy the files in my directory C:\jdl\fortran to your own **fortran**
directory. At the command line prompt (C:\your folder>) type the command
C:\your folder> ftn77 fitline.for
This should result in several lines of output, compilation of the programme, and creation of
the files FITLINE.OBJ and FITLINE.EXE. To execute the programme, simply type
C:\your folder> fitline
The screen output should appear almost instantly, and the file OUT.DAT will be found
in the folder. As above, debug your programme if necessary. To use the programme to
improve the fit of your line to the data, try inserting different values of the parameters
into the file fitpar.dat until you get a satisfactorily small value of the reduced chi-squared,
which should also lead to a fit that looks good by eye. Be sure to plot up your data and
the fit as you go, using Matlab’s plot function or Microsoft Excel’s graphical capabilities.

Useful references

- http://netlib.org This web site is the source of a remarkable number of public domain
programmes for mathematical computation. You can find programmes here for cal-
culation of special functions, for solving systems of differential equations, for matrix
inversion, etc, etc. Check it out.

- Theory and Problems of Programming with FORTRAN 77, by W. E. Mayo and M.
good basic introduction to programming with FORTRAN 77, except for the fact that
it does not use the WRITE statement at all! However, it is otherwise clearly written.

- Advanced Scientific FORTRAN, by D. R. Willé (Chichester: John Wiley & Sons),
1995. Willé’s book is a good introduction both to some of the issues involved in
numerically demanding scientific computing, and to FORTRAN 90. It is not an
introduction to FORTRAN.
Appendix: a listing of fitline.f

    program fitline
    c
    c For the Fortran class exercises, I need an example programme. This
    c is a program to fit a straight line to a small data set,
    c using chi-squared as the goodness of fit parameter, adjusting the
    c fitting parameters by hand. The calculation of \chi^2 is done by
    c a subroutine, and the computation of the fitted y values by a
    c function, in order to show how these constructs are used. The
    c programme is documented internally, as ALL programmes should be.
    c
    c Define the types of various variables, and the dimensions of arrays
    real x(20),ydata(20),yfit(20),sigma(20),A,B,rdata
    integer n, ndata
    c Open data files and read in fit parameters (unit 4) and data (unit 5)
    open(unit=4,file='fitpar.dat',status='old')
    read(4,*),A, B
    open(unit=5,file='data.dat',status='old')
    read(5,*), ndata
    rdata = real(ndata)
    c Print summary of problem to screen
    print*, 'Fit tested: y = ',A,'x + ',B
    c Test size of data file to see if it fits into arrays. If no, issue
    c warning and stop.
    if (ndata > 20) then
      print*, 'Too many data points, dummy!'
      stop
    c If size is ok, read in the data
    else
      do 10 n=1,ndata
         read(5,*), x(n), ydata(n), sigma(n)
      10     yfit(n) = rline(x(n),A,B)
    endif
    c Use subroutine chi2 to calculate the chi-squared value chi2 for the fit
    call chi2(ydata,yfit,sigma,ndata,chi2)
    c Print summary of results to screen
    print*
    print*, 'Degrees of freedom = ', ndata - 2
    print*, 'Reduced chi-squared of fit = ', chi2/(rdata-2.0)
c Write results to a disk file
   open(unit=6,file='out.dat',status='unknown')
   write(6,600) A, B
   600 format(2'Fit tested: Y = ',f8.4,'*X + ',f8.4)
   write(6,610)
   610 format(2'/ N   X   YDATA   YFIT   SIGMA'/)
   do 20 n = 1,ndata
      write(6,620) n, x(n), ydata(n), yfit(n), sigma(n)
   620 format(i4,4f10.2)
   20 continue
   write(6,630) chi2/(rdata-2.0)
   630 format(2'/Reduced chi squared of fit = ',f10.2)
   stop
end

*******************************************************************************
function rline(u,a,b)
   c The function rline evaluates the expression rline = a*u + b and returns
   c the value of rline to the calling programme
   real rline,u,a,b
   rline = a*u + b
   return
end
*******************************************************************************
subroutine chisq(y1,y2,s,nup,chi2)
   c The subroutine chisq evaluates the value of chi2 for an array of
   c measured values y1, an array of fitted values y2, and an array
   c of sigmas s by summing up the needed terms one at a time.
   real y1(20),y2(20),s(20),chi2
   integer n, nup
   c Initialize the value of chi2 to zero. (Don't assume that this has
   c been done by the operating system!)
   chi2 = 0.0
   do 10 n = 1, nup
   c Check for division by zero
      if (s(n) .eq. 0.0) then
         print*, 'HEY! One of the sigmas is zero!' stop
      c If ok, add the terms of chi2 one by one
      else
         chi2 = chi2 + ((y1(n) - y2(n))/s(n))**2
      endif
   10 continue
   return
end
*******************************************************************************