Exercise Solutions

not included in the Solutions Manual **for**

Classical Fortran second edition

Michael Kupferschmid

Copyright © 2023 Michael Kupferschmid. ב"ה

All rights reserved. Except as permitted by the fair-use provisions in Sections 107 and 108 of the 1976 United States Copyright Act, no part of this book may be stored in a computer, reproduced, translated, or transmitted, in any form or by any means, without prior written permission from the author.

This book, "Exercise Solutions" by Michael Kupferschmid, is licensed under CC-BY 4.0. Anyone who complies with the terms specified in
`https://creativecommons.org/licenses/by/4.0/legalcode.txt`
may use the work in the ways therein permitted. Inquiries and requests for permission to use material from the book in other ways should be emailed to the appropriate address at the **contact** tab of the website from which it was downloaded.

The computer programs presented in this book are included only for their instructional value. They have been carefully checked and tested, but they are not guaranteed for any particular purpose, and they should not be used in any application where their failure to work as expected might result in injury to persons, damage to property, or economic loss. Michael Kupferschmid offers no warranty or indemnification and assumes no liabilities with respect to the use of any information contained in this book. For more general disclaimers see the **disclaimers and permissions** tab of the web site from which this book was downloaded.

About This Supplement

When the Second Edition of **Classical FORTRAN** was published in 2009, I included in its Solutions Manual answers to only about half of the Exercises. Instructors who wanted to avoid the effort of solving the problems could assign those, while instructors who worried that their students might have gained access to the Solutions Manual could assign the others. Readers who were not teaching a graded course from the text did not have legitimate access to solutions for any of the problems.

Students cannot be prevented from sharing their work, so over time solutions to the Exercises in any popular book become common knowledge, even if they are not in a Solutions Manual and even if the Solutions Manual has not leaked out. It has by now probably become less important to keep some of the answers secret, so I am here making public many of the solutions that are not included in the Solutions Manual; the ones that remain unpublished are listed below.

text §	Exercises having solutions that remain unpublished
0.10	2-4 6 10-23
5.8	6 19(c-g,i-l,n,o)
9.10	23
10.9	18 24 27-29 32 34 37-39
11.9	29
13.13	1 3 4 6 14-19 22 23 25 26 28 30 31 33-38 40-45
14.8	5 8 10 11 15 16
15.4	8 9 17
16.4	1-3 5 8 10 11 14-16
17.4	4 20 30
18.8	1-23

A course instructor can continue to assign those problems on the assumption that their solutions are not widely known, along with problems from the Solutions Manual on the understanding that students earning credit may not consult it even if they do gain access. Meanwhile, graduate students who need FORTRAN for their research, professionals who need it for their work, and others who are learning the language on their own can benefit from studying the solutions published here after trying the problems as they work through the text.

The Solutions

Each solution begins with the Exercise number and its difficulty rating. As explained in the Solutions Manual, problems marked [E] test the student's recall from the text; problems marked [H] require some thought and perhaps some programming; problems marked [P] require the student to provide a program as part of the solution.

0.10.1 [H] (a) The people in the machine room on the other side of the glass were clerks and technicians, not scientists. Most of the programs they ran were written by professional coders to do business data processing. During the 1960's it was commonly estimated that out of every 100 hours of computer time the world consumed, 90 were devoted to running COBOL programs for general ledger accounting, inventory management, and printing paychecks and insurance policies. Of the remaining 10 hours it was often said that 9 were devoted to running FORTRAN programs for numerical calculations, and that of those 9 hours 8 were spent solving $\mathbf{A}\mathbf{x} = \mathbf{b}$ for problems in engineering design. The remaining hour was divided between scientific research, almost all of it unclassified, and instructional computing for college students. Many of the FORTRAN programs were written by engineers or scientists, but some of them were also written by programmers who were neither. Computer users then, like computer users now, were about average in both arrogance and sanity.

(b) Most computer use during the mainframe era was for mundane activities necessary to commerce and industry, and was thus harmful *and* beneficial to the same extent that commerce and industry are both.

(c) Early computers were easily damaged by heat, humidity, and tiny particles of air-borne dirt. Mainframes were kept in antiseptic rooms mainly to protect them from those environmental hazards, and to a lesser degree because they were used to process sensitive information such as payroll data. Because the machines were an expensive and limited resource, access was allocated by the universities and companies that owned them to users with a legitimate need. Mainframe users did recognize one another as members of a professional fraternity, but anyone could join if they learned how to program. Computing today is in contrast fractured into numerous subcultures that are often mutually dismissive and hostile to outsiders.

(d) The introduction of the personal computer trivialized the technology and ultimately lead to our social media culture, but whether that constitutes progress is a question that can be answered only by historians (if there are any) in the distant future.

(e) Our embrace of the personal computer as an information appliance has, especially with the recent introduction of artificial intelligence to process vast aggregations of data, dramatically *increased* society's demand for technical computing.

0.10.5 `[H]` It is easy (though tedious) to approximate the value of x for which

$$f(x) = \sin(x) - \tfrac{1}{2}x = 0 \quad \text{on} \quad [a,b] = [\tfrac{1}{2}, 2\tfrac{1}{2}]$$

with a hand-calculator, by organizing the numerical calculations in a table like the one below. The function values are shown to three significant figures, but it is only their signs that matter to the algorithm.

iteration	a	$f(a)$	b	$f(b)$	$x = (a+b)/2$	$f(x)$
0	0.5	0.229	2.5	−0.652	1.5	0.247
1	1.5	0.247	2.5	−0.652	2.0	−0.0907
2	1.5	0.247	2.0	−0.0907	1.75	0.109
3	1.75	0.109	2.0	−0.0907	1.875	0.0166
4	1.875	0.0166	2.0	−0.0907	1.9375	−0.0352
5	1.875	0.0166	1.9375	−0.0352	1.90625	−0.00886
∞					1.89549426703398	0

After five iterations we have $x = 1.90625 \pm 0.03125 \approx 1.90625 \pm 1.6\%$. Each iteration of the bisection algorithm halves the uncertainty in the estimate of the root.

0.10.7 `[E]` Some calculations are more gracefully described in FORTRAN than in MATLAB, and many calculations take much longer in MATLAB than they do in FORTRAN. It is possible to use MATLAB (or its free work-alike Octave) only on a computer where the program is installed. Engineers and scientists need to know how to *write* programs for doing numerical calculations because they frequently encounter problems that no existing software can solve. They need to know how to *read* programs written in FORTRAN because it is the language in which numerous extant programs are written. FORTRAN is often preferable to other procedural languages because it is easier to learn and use, has free compilers providing strong optimization, and lends itself better to vector and parallel processing.

0.10.8 `[E]` (a) As explained in §0.3, FORTRAN resembles a peasant language because it is "plain and direct" yet "richly expressive" and because it "sustains a magnificent literature" of numerical software. (b) The bourgeoisie might naturally be expected to have a low opinion of peasants.

0.10.9 `[E]` FORTRAN is a special-purpose language for floating-point calculations, which are seldom needed in most programming jobs. Employers assume when they advertise engineering and scientific jobs requiring a knowledge of FORTRAN that applicants will know the language. There are some ads for engineers and scientists in the New York Times, but those positions are often advertised in specialty publications instead. Lisp, a special-purpose language for artificial intelligence, is seldom mentioned either, for the same reasons.

1.5.11 [H] FORTRAN is best suited to numerical computing tasks while a compiler does mostly symbol manipulation, so a language such as C is a far better choice than FORTRAN for writing a compiler. However, FORTRAN compilers can be, and have been, written in FORTRAN. To compile such a compiler, or to compile a C compiler that is written in C, it is necessary to initially use a compiler that was written in some other language (typically assembler). Of course, once the new compiler has been compiled it can be used to recompile itself or subsequent compilers written in the same language.

2.9.12 [H] This program performs the calculation, and prints zero for the "bit more."

```
QTY=(12.+144.+20.+3.*SQRT(4.))/7. + 5.*11.
BIT=QTY-9.**2
PRINT *,'bit more=',BIT
STOP
END
```

3.8.6 [H] First consider the case that n is even (legend attributes this argument to K. F. Gauss at age seven).

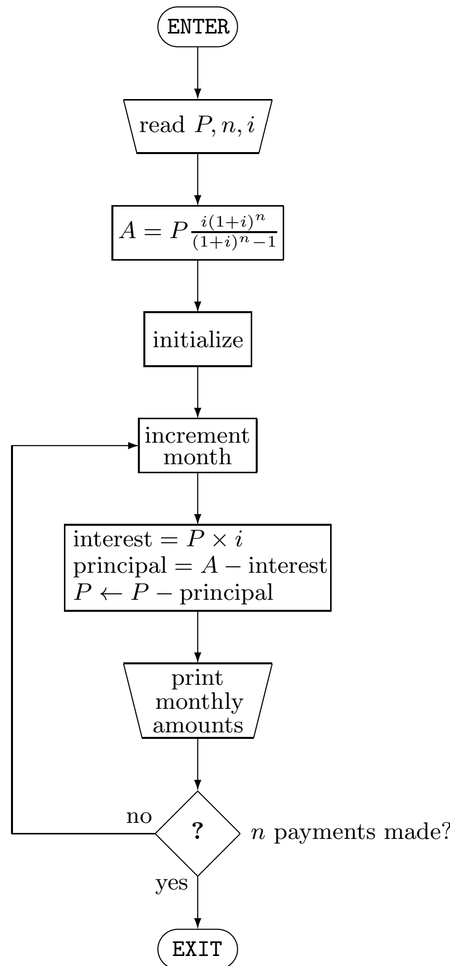
$$\begin{aligned}
\sum_{i=1}^{n} i &= 1+2+3+\cdots+(n/2)+[(n/2)+1]+\cdots+(n-2)+(n-1)+n \\
&= (1+n)+[2+(n-1)]+[3+(n-2)]+\cdots+\{(n/2)+[(n/2)+1]\} \\
&= \sum_{j=1}^{n/2}(n+1) \\
&= (n+1)\sum_{j=1}^{n/2} 1 \\
&= (n+1)(n/2) \\
&= \tfrac{1}{2}n(n+1).
\end{aligned}$$

If n is odd, then $n-1$ is even and we can use the previous result as follows.

$$\begin{aligned}
\sum_{i=1}^{n} i &= \sum_{i=1}^{n-1} i + n \\
&= \tfrac{1}{2}(n-1)[(n-1)+1]+n \\
&= \tfrac{1}{2}(n-1)(n)+\tfrac{1}{2}(2n) \\
&= \tfrac{1}{2}(n)(n-1)+\tfrac{1}{2}n(2) \\
&= \tfrac{1}{2}(n)(n-1+2) \\
&= \tfrac{1}{2}(n)(n+1).
\end{aligned}$$

Thus, the sum is equal to the given formula whether n is even or odd, as was to be shown.

3.8.18 `[P]` (a) The flowchart below gives an algorithm for performing the calculation described in the problem statement.



The flowchart precisely specifies the flow of control required for the calculation but it omits many details that are needed in a program, such as variable names and the FORTRAN statements to be used in implementing the operations described in the boxes.

(b) The program on the next page is one implementation of the algorithm flowcharted above. First the program reads the loan principal `P`, the number `N` of monthly payments, and the real interest rate *per month* `RATE` from the user. `RATE` is assumed to be provided as a decimal, rather than as a percentage. Then the formula given in the problem statement is used to compute the monthly payment `A`.

The interest part of each payment has a fractional part, so it can't be represented in this program by a variable having a name such as `I` or `INT` (such names *can* be used for real variables, but only by using a type declaration as explained in textbook §4). Since the interest is money charged for use of the money loaned, the program instead adopts the real variable name `USE`.

```
      READ *,P,N,RATE
      A=P*(RATE*(1.+RATE)**N)/((1.+RATE)**N-1.)
      AGGUSE=0.
      AGGPRN=0.
      BAL=P
      MONTH=0
    1 MONTH=MONTH+1
        USE=RATE*BAL
        AGGUSE=AGGUSE+USE
        PRN=A-USE
        AGGPRN=AGGPRN+PRN
        BAL=BAL-PRN
        PRINT *,MONTH,USE,PRN,BAL,AGGUSE,AGGPRN
      IF(MONTH.LT.N) GO TO 1
      STOP
      END
```

We need to accumulate the aggregate interest and principal paid, so AGGUSE and AGGPRN, the variables representing these quantities, are initialized to zero. Then the loan balance BAL is initialized to the principal value P, the loop counter MONTH is initialized to zero, and control flow enters the loop beginning at statement 1. The loop gets executed repeatedly for successive values of MONTH until the test at the bottom no longer finds MONTH less then the number of months in the loan.

In each pass through the loop, the program computes the interest USE due on the remaining balance, finds the principal amount PRN of the month's payment as the total payment A less the interest charge, and reduces the balance owed BAL by the principal amount paid. AGGUSE and AGGPRN are incremented to accumulate the interest and principal paid. The PRINT statement reports, for each payment, the quantities required in the problem statement. Compiling and running the program above produces the dialog shown below.

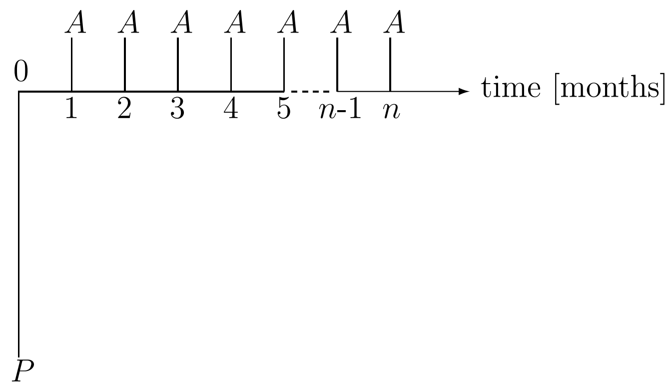
```
unix[1] f77 mortgage.f
unix[2] a.out
10000 12 .005
 1  50.  810.66272  9189.33691  50.  810.66272
 2  45.946682  814.716064  8374.62109  95.9466858  1625.37878
 3  41.8731041  818.789612  7555.83154  137.819794  2444.16846
 4  37.7791557  822.883545  6732.94824  175.598953  3267.052
 5  33.6647415  826.997986  5905.9502  209.263702  4094.05005
 6  29.5297508  831.132996  5074.81738  238.793457  4925.18311
 7  25.3740864  835.288635  4239.52881  264.167542  5760.47168
 8  21.1976433  839.465088  3400.06372  285.365173  6599.93652
 9  17.0003185  843.662415  2556.40137  302.365479  7443.59912
 10  12.7820063  847.880737  1708.52063  315.147491  8291.47949
 11  8.54260254  852.120117  856.400513  323.690094  9143.59961
 12  4.28200245  856.380737  0.0197753906  327.972107  9999.98047
```

The user entered the f77 command to compile the program and the a.out command to run it, and then entered the values of P=10000, N=12 months, and RATE=.005 per month. The program wrote the last 12 lines. From this output we see that the final payment consists

of about `4.28` in interest and about `856.38` in principal, leaving a balance of about zero (actually a little less than `2` cents) and a principal paid of about `10000` (actually a little more than `9999.98`). The total interest paid on the loan is about `327.97` (which is of course quite a bit less than $.005 \times 12 \times 10000 = 600$ because the amount owed declines as the loan is paid off).

Roundoff errors in the floating-point calculations account for the tiny balance left unpaid in this simulation. FORTRAN is seldom used for calculations involving money, but when it is two-part values can be used instead of real variables, as explained in textbook §18.4, to make the results accurate to the penny. In §9 we shall see how formatted I/O can be used to align tables more neatly than the one printed by this program.

(c) We can derive the formula for A by considering the time series of money transfers pictured below.



The loan P is taken at time zero. Payments of A commence at the end of month 1 and repeat through the end of month n. The first payment earns interest for $n-1$ months, so at the end of the series its value is $A(1+i)^{n-1}$. The second payment earns interest for $n-2$ months, and so on, so the total future value F of the payments at the moment after the final one is given by the following sum.

$$F = A(1+i)^{n-1} + A(1+i)^{n-2} + \cdots + A(1+i) + A = A\sum_{k=0}^{n-1}(1+i)^k$$

Recall that the sum of a geometric series is

$$\sum_{k=0}^{n-1} r^k = \frac{1-r^n}{1-r}.$$

Thus the future value of the payments is

$$F = A\frac{1-(1+i)^n}{1-(1+i)} = A\frac{1-(1+i)^n}{-i}$$

But the future value of the loan P is $F = P(1+i)^n$, so for the payments to retire the loan it must be that

$$
\begin{aligned}
A\frac{(1+i)^n-1}{i} &= P(1+i)^n \\
A &= P(1+i)^n\frac{i}{(i+1)^n-1} \\
A &= P\frac{i(1+i)^n}{(1+i)^n-1}
\end{aligned}
$$

as was to be proved. □

3.8.21 [P] (a) Here is a FORTRAN program that implements the algorithm specified in the flowchart.

```
      EPS=.001
      XL=1.
      XR=3.
C
C     find midpoint of current interval
    3 X=(XL+XR)/2.
      IF(ABS(XR-XL).LT.EPS) GO TO 1
      F=SIN(X)-0.5*X
      IF(ABS(F).LT.EPS) GO TO 1
C
C     shorten interval by excluding half that does not contain root
      FL=SIN(XL)-0.5*XL
      IF(F*FL .LT. 0.) GO TO 2
      XL=X
      GO TO 3
    2 XR=X
      GO TO 3
C
C     convergence tolerance satisfied
    1 PRINT *,X
      STOP
      END
```

(b) When the program is run it prints out

```
 1.894531250
```

which yields $\sin(x) - \frac{1}{2}x \approx 0.8 \times 10^{-3}$.

(c) The first test uses the fact that F*FL will be negative only if F and FL differ in sign. It is simple and easy to type correctly, but it uses a multiplication and it requires some analysis to understand what it is really testing. The second test explicitly compares the signs of F and FL, so it does not require a multiplication and what it is doing is more obvious from the text of the statement, but it is longer and harder to type without making a mistake. Both of these tests fall through if FL is zero, even if F is far from zero.

4.10.17 [P] Here is a REAL*8 version of the program given in the solution to Exercise 3.8.17(b). In addition to typing all of the real variables REAL*8, I have appended D0 to cast each real constant as REAL*8 and changed the elementary function references from ALOG to DLOG.

```
C     make all of the real variables double precision
      REAL*8 A,B,H,SUM,X,ANS
C
C     set limits, number of strips, strip width
      A=0.5D0
      B=3.0D0
      N=1000
      H=(B-A)/1000.D0
C
C     start with the first two terms
      SUM=DLOG(A)/A+4.D0*DLOG(A+H)/(A+H)
C
C     so far we have added up two terms
      I=2
      X=A+H
C
C     others have coefficients in the pattern 2,4,2,4...
    1 X=X+H
      SUM=SUM+2.D0*DLOG(X)/X
      I=I+1
      X=X+H
      SUM=SUM+4.D0*DLOG(X)/X
      I=I+1
      IF(I.LT.N) GO TO 1
C
C     finally add in the last (N+1st) term
      SUM=SUM+DLOG(B)/B
C
C     compute the integral and print it out
      ANS=(H/3.D0)*SUM
      PRINT *,'the integral is about ',ANS
      STOP
      END
```

When run this program prints the following answer, which is correct to 10 digits.

```
 the integral is about   0.36324797339455928
```

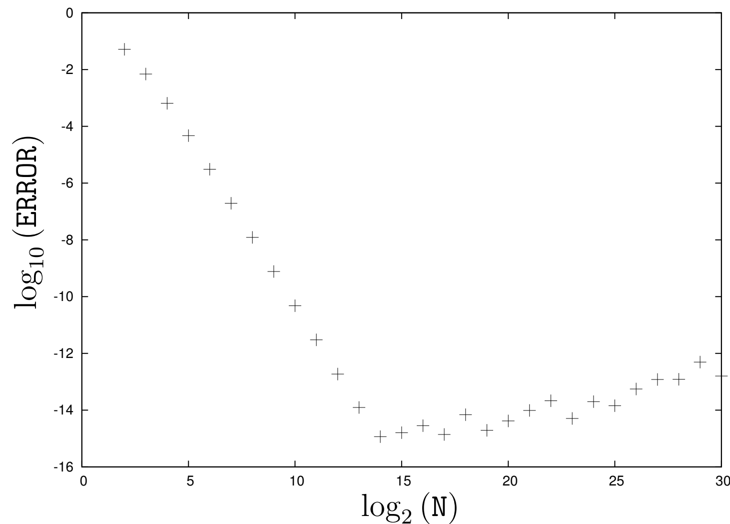
As in the first program of the earlier solution, I arbitrarily chose N=1000 strips. Increasing the number of strips makes the approximation better.

Revising the above program as in Exercise 3.8.17(c) yields the code shown on the next page. Its calculations are much more precise than those of the REAL*4 program, so the optimal strip width will be much smaller. This program therefore considers values of $\log_2$ (N) up to 30, which is the largest value that can be used since N is an INTEGER*4 variable. As explained in textbook §4.1, an INTEGER*4 can represent positive values up to only $2^{31} - 1$, so 2^{31} would be too big. Even with this limitation, the program runs for a long time.

```
C     make all of the real variables double precision
      REAL*8 A,B,H,SUM,X,ANS,ERROR,EXACT
C
C     set the limits of integration
      A=0.5D0
      B=3.0D0
C
C     compute the actual integral of ln(x)/x from A to B
      EXACT=0.5D0*(DLOG(B)**2-DLOG(A)**2)
C
C     compute the integral using 30 values of N
      N=2
      L=0
    2 L=L+1
        N=2*N
        H=(B-A)/DFLOAT(N)
C
C       start with the first two terms
        SUM=DLOG(A)/A+4.D0*DLOG(A+H)/(A+H)
C
C       so far we have added up two terms
        I=2
        X=A+H
C
C       others have coefficients in the pattern 2,4,2,4...
    1   X=X+H
        SUM=SUM+2.D0*DLOG(X)/X
        I=I+1
        X=X+H
        SUM=SUM+4.D0*DLOG(X)/X
        I=I+1
        IF(I.LT.N) GO TO 1
C
C       finally add in the last (N+1st) term
        SUM=SUM+DLOG(B)/B
C
C       compute the error and print it out
        ANS=(H/3.D0)*SUM
        ERROR=DABS(ANS-EXACT)
        IF(ERROR .NE. 0.D0) PRINT *,L+1,DLOG10(ERROR)
      IF(L.LT.30) GO TO 2
      STOP
      END
```

The output of this program is shown in the table on the left at the top of the next page, and plotted in the graph on the right. Notice that here the error in the approximation is never zero (in the solution to Exercise 3.8.17 one of the errors came out zero by coincidence). Once again the error initially goes down as `N` increases and then goes back up as roundoff error becomes more important than discretization error.

$\log_2$ (N)	$\log_{10}$ (ERROR)
2	-1.2859188277456728
3	-2.1567534899105518
4	-3.1894528576213759
5	-4.3270319399194044
6	-5.5108293469857106
7	-6.7095121506465700
8	-7.9122468033865809
9	-9.1160186539221097
10	-10.320049526857987
11	-11.524143819082155
12	-12.728763778729110
13	-13.907314902806824
14	-14.933400471121065
15	-14.793221767956029
16	-14.548049589757047
17	-14.857679757182947
18	-14.158709752846928
19	-14.711551721504708
20	-14.380558502463284
21	-14.010107098040836
22	-13.670159036346483
23	-14.291831938509429
24	-13.699317265087696
25	-13.844000059891755
26	-13.254752044323757
27	-12.919760854535166
28	-12.913592077767513
29	-12.305498418343610
30	-12.798040618859222



Now the smallest error of about 10^{-15} is achieved when $\log_2(\mathtt{N}) = 14$, corresponding to an optimal strip width of

$$h^* = \frac{b-a}{n^*} = \frac{3-\frac{1}{2}}{2^{14}} \approx 0.00015$$

Using REAL*4 calculations the lowest error of about $10^{-7.5}$ was achieved when $\log_2(\mathtt{N}) \approx 8.5$, corresponding to $h^* \approx 0.007$ (and to a much faster running time). Using REAL*8 calculations instead of REAL*4 doubled the number of correct digits in the best answer obtainable with this algorithm (as well as the number of correct digits obtained using N=1000).

The error curve for REAL*8 rises much less sharply after its minimum than the curve for REAL*4, so the accuracy of this calculation is less sensitive to the choice of h so long as h is small enough.

4.10.19 `[E]` The denominator of the i'th term in the sine series is $(2i-1)!$, so the previous denominator was $(2[i-1]-1)!$. Thus we only need to show that $(2[i-1]-1)!(2i-1)(2i-2) = (2i-1)!$. A little algebra

$$
\begin{aligned}
(2[i-1]-1)!(2i-1)(2i-2) &= (2i-2-1)!(2i-1)(2i-2) \\
&= (2i-3)!(2i-1)(2i-2) \\
&= (2i-1)(2i-2)(2i-3)! \\
&= (2i-1)(2i-2)(2i-3)(2i-4)\cdots 1 \\
&= (2i-1)!
\end{aligned}
$$

yields the desired result, so the claim in textbook §4.5 must be true. □

4.10.21 `[P]` (a) Here is a program that solves the first part of the problem.

```
      REAL*8 TWOPI,A,P
      TWOPI=8.D0*DATAN(1.D0)
      I=0
    1 I=I+1
        A=0.02D0*DFLOAT(I-51)
        P=(1.D0-A+(1.D0/TWOPI)*DSIN(TWOPI*A))**(1.5D0)
        PRINT *,A,P
      IF(I.LT.101) GO TO 1
      STOP
      END
```

(b) To avoid the possibility of a `NAN` on the last value of α, it is only necessary to stop the loop one iteration early and print out the final coordinates separately.

```
      REAL*8 TWOPI,A,P
      TWOPI=8.D0*DATAN(1.D0)
      I=0
    1 I=I+1
        A=0.02D0*DFLOAT(I-51)
        P=(1.D0-A+(1.D0/TWOPI)*DSIN(TWOPI*A))**(1.5D0)
        PRINT *,A,P
      IF(I.LT.100) GO TO 1
      A=1.D0
      P=0.D0
      PRINT *,A,P
      STOP
      END
```

4.10.22 [P] (a) The following program prints a table of the digit frequencies predicted by Benford's law, and their sum.

```
C     This program computes Benford's law first-digit frequencies.
C
      INTEGER*4 D
      REAL*8 F,SUM
C
C -----------------------------------------------------------------
C
C     table and add up the digit frequencies
      SUM=0.D0
      D=1
    1 F=DLOG10(1.D0+1.D0/DFLOAT(D))
      PRINT *,D,F
      SUM=SUM+F
      D=D+1
      IF(D.LE.9) GO TO 1
C
C     report the sum of the digit frequencies
      PRINT *,'SUM=',SUM
      STOP
      END
```

The d in Benford's formula is an integer, so the program begins by declaring `D` to be an `INTEGER*4` variable. The digit frequencies and their sum are real, so following the advice of §4.3 it declares `F` and `SUM` to be `REAL*8`. Type declarations come before the first executable statement, so they are placed in a **preamble** and separated from the executable code. The first **stanza** of executable code loops `D` through the values `1...9`, evaluating Benford's formula for each value and printing out the results. It also accumulates the sum of the frequencies. When the loop is done, the second stanza reports the sum and stops the program. The program produces this output.

```
 1  0.301029996
 2  0.176091259
 3  0.124938737
 4  0.096910013
 5  0.079181246
 6  0.0669467896
 7  0.057991947
 8  0.0511525224
 9  0.0457574906
 SUM=  1.
```

Even though the frequency calculations are carried out with the limited precision of floating-point numbers, the sum of the frequencies is found to be exactly 1.

(b) To evaluate the sum analytically, notice that

$$\begin{aligned}\log_{10}\left(1+\frac{1}{i}\right) &= \log_{10}\left(\frac{i+1}{i}\right) \\ &= \log_{10}\ (i+1) - \log_{10}\ (i).\end{aligned}$$

Then

$$\begin{aligned}\sum_{d=1}^{9}\log_{10}\left(1+\frac{1}{d}\right) &= (\log_{10}\ 2 - \log_{10} 1) \\ &+ (\log_{10}\ 3 - \log_{10} 2) \\ &+ (\log_{10}\ 4 - \log_{10} 3) \\ &+ (\log_{10}\ 5 - \log_{10} 4) \\ &+ (\log_{10}\ 6 - \log_{10} 5) \\ &+ (\log_{10}\ 7 - \log_{10} 6) \\ &+ (\log_{10}\ 8 - \log_{10} 7) \\ &+ (\log_{10}\ 9 - \log_{10} 8) \\ &+ (\log_{10} 10 - \log_{10} 9) \\ &= \log_{10} 10 - \log_{10} 1 \\ &= 1.\end{aligned}$$

4.10.24 [P] The first harmonic number greater than 10 is the first partial sum of the series

$$\sum_{i=1}^{n} 1/i$$

that exceeds 10. Here is a program that accumulates the sum, starting with $i = 1$, until it exceeds 10.

```
      REAL*8 H
      I=0
      H=0.D0
    1 I=I+1
      H=H+1.D0/DFLOAT(I)
      IF(H.LE.10.D0) GO TO 1
      PRINT *,'the ',I,'th harmonic number is ',H
      STOP
      END
```

When compiled and run this program produces the output shown below.

```
unix[1] a.out
 the  12367th harmonic number is   10.000043
unix[2]
```

The next program adds up the first 12367 terms in the opposite order.

```
      REAL*8 H
      I=12368
      H=0.D0
    1 I=I-1
      H=H+1.D0/DFLOAT(I)
      IF(I.GT.1) GO TO 1
      PRINT *,'the  12367th harmonic number is ',H
      STOP
      END
```

Its output is identical to that of the first program, so changing the order of the additions makes no difference to the digits printed. The true sum is `10.000043008275808`, correct to 17 digits.

4.10.27 [H] (a) If A and B differ in sign and each is a representable floating-point value, then A+B will be smaller in absolute value than either so no overflow will occur. If A and B are of the same sign then it suffices to check for an overflow in finding the sum of their absolute values, as in this code segment.

```
      REAL*8 A,B,C,HUGE/Z'7FEFFFFFFFFFFFFF'/
      :
      IF((A.GT.0.D0 .AND. B.GT.0.D0)  .OR.
     ;   (A.LT.0.D0 .AND. B.LT.0.D0)) THEN
         IF(DABS(A).GT.HUGE-DABS(B)) THEN
            PRINT *,'floating-point overflow on addition'
            STOP
         ENDIF
      ENDIF
      C=A+B
      :
```

(b) The signs of the numbers matter only for determining the sign of the quotient, so assume for a moment that they are both positive. If both numbers are zero their quotient is NaN, and our fixup should not apply. If E is 1 or more and D is a representable floating-point value, then D/E will be no bigger than D so no overflow will occur. If E is less than 1, then no overflow results from multiplying it by the largest representable value. Hence we can use a code segment like this one to anticipate an overflow in the division.

```
      REAL*8 F,D,E,HUGE/Z'7FEFFFFFFFFFFFFF'/
      :
      IF(D.EQ.0.D0 .AND. E.EQ.0.D0) THEN
C        let the division produce a NaN
         F=D/E
      ELSE
         IF(DABS(E).GE.1.D0) THEN
            F=D/E
         ELSE
            IF(HUGE*DABS(E) .GT. DABS(D)) THEN
              F=D/E
            ELSE
              F=HUGE
              IF((D.GT.0.D0 .AND. E.LE.0.D0)  .OR.
     ;           (D.LT.0.D0 .AND. E.GE.0.D0)) F=-F
            ENDIF
         ENDIF
      ENDIF
      :
```

4.10.28 [H] The given statements are equivalent *except* when the product FL*F yields the −0. byte code. In that case the result of the test depends on how the compiler translates the FORTRAN source text into machine instructions. Consider the programs listed below, in both of which the product underflows to minus zero.

```
      REAL*4 FL,F
      FL=1.E-30
      F=-1.E-30
C
      IF(FL*F .LT. 0.) THEN
         PRINT *,'true'
      ELSE
         PRINT *,'false'
      ENDIF
      STOP
      END
```

```
      REAL*4 FL,F,P
      FL=1.E-30
      F=-1.E-30
      P=FL*F
      IF(P .LT. 0.) THEN
         PRINT *,'true'
      ELSE
         PRINT *,'false'
      ENDIF
      STOP
      END
```

When I compiled these programs (using `gfortran` version `Ubuntu 4.3.2-1ubuntu12`) and ran them, the one on the left printed `true` but the one on the right printed `false`. Minus zero tests equal to zero, not less, so in the IF statement in the program on the left the sign of the product, and hence the outcome of the test, might be determined without performing the multiplication.

4.10.30 [P] (a) Here is a program that implements Knuth's recursion.

```
      XBAR=0.
      I=0
    1 PRINT *,'enter next value'
      READ *,X
      IF(X.EQ.0.) THEN
         PRINT *,'average=',XBAR
         STOP
      ELSE
         I=I+1
         XBAR=XBAR+(X-XBAR)/FLOAT(I)
         GO TO 1
      ENDIF
      END
```

This code uses an input value of zero to flag the end of the data, so it can't be used to average a set of data that includes zero values. In Chapter 9 of the text you will learn a less clumsy way to signal the end of input to an interactive program. When the program is run it produces output like this.

```
unix[1] f77 knuth.f
unix[2] a.out
 next value
-10
 next value
14
 next value
3
 next value
20.5
 next value
8
 next value
0
 average=  7.0999999
unix[3]
```

The average of the given (nonzero) numbers is 7.1, so the program produces the correct result within roundoff error.

(b) The advantage of Knuth's recursion is that it does not require the allocation of storage for the vector of input values and can therefore be used without knowing in advance how many there will be (or how to store vectors in FORTRAN). A disadvantage of the recursion is that it involves N subtractions and N divisions besides the usual N additions, which makes it both slower and more subject to roundoff error.

(c) Knuth's recursion is not obvious but is easily derived using the following simple argument. Suppose that $\bar{x}$ is the average of the first $(i-1)$ values. Then

$$\begin{aligned}
\bar{x} &= \frac{x_1 + x_2 + \cdots + x_{i-1}}{i-1} \\
(i-1)\bar{x} &= x_1 + x_2 + \cdots + x_{i-1} \\
x_1 + x_2 + \cdots + x_{i-1} + x_i &= (i-1)\bar{x} + x_i \quad = \quad i\bar{x} - \bar{x} + x_i \\
\bar{x} &\leftarrow \frac{i\bar{x} - \bar{x} + x_i}{i} \quad = \quad \bar{x} + \frac{x_i - \bar{x}}{i}.
\end{aligned}$$

4.10.40 [P] (a) Here is the program of textbook §3.4 rewritten to make the variables D, E, and F complex.

```
COMPLEX*8 D,E,F
READ *,A,B,C
D=CMPLX(B**2-4.*A*C,0.)
E=CMPLX(-B/(2.*A),0.)
F=CSQRT(D)/CMPLX(2.*A)
PRINT *,'X1=',E+F,' X2=',E-F
STOP
END
```

Now the roots (possibly repeated) are always E+F and E-F, so no tests are required. When the program is compiled and run it produces the following exchange with a user.

```
unix[1] a.out
1 2 3
 X1= (-1.00000000    ,  1.4142135    )  X2= (-1.00000000    , -1.4142135    )
unix[2] a.out
1 -4 4
 X1= (  2.0000000    ,  0.0000000    )  X2= (  2.0000000    ,  0.0000000    )
```

(b) Making all of the variables COMPLEX results in an even simpler program.

```
COMPLEX*8 A,B,C,D,E,F
READ *,A,B,C
D=B**2-4.*A*C
E=-B/(2.*A)
F=CSQRT(D)/(2.*A)
PRINT *,'X1=',E+F,' X2=',E-F
STOP
END
```

When this program is compiled and run it produces the following exchange.

```
unix[1] a.out
(1,0) (-4,0) (4,0)
 X1= (  2.0000000    , -0.0000000    )  X2= (  2.0000000    ,  0.0000000    )
unix[2] a.out
(1,1) (1,1) (1,1)
 X1= (-0.50000000    ,-0.86602539    )  X2= (-0.50000000    , 0.86602539    )
```

The final problem is worked out analytically below.

$$
\begin{aligned}
0 &= (1+i)x^2 + (1+i)x + (1+i) \\
x &= \frac{-(1+i) \pm \sqrt{(1+i)^2 - 4(1+i)(1+i)}}{2(1+i)} \\
&= -\frac{1}{2} \pm \frac{\sqrt{-6i}}{2(1+i)} = -\frac{1}{2} \pm \frac{\sqrt{3}(i-1)}{2(i+1)} = -\frac{1}{2} \pm i\frac{\sqrt{3}}{2}.
\end{aligned}
$$

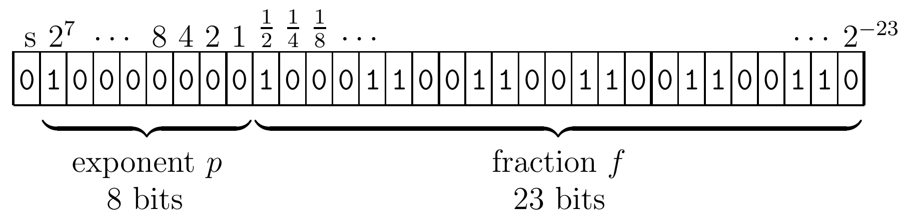
4.10.41 [H] When the logarithm example of textbook §4.6.2 is compiled and run it produces the approximation 0.68817180 to $\ln(2) =$ .69314718055994531, and stops when I=100. Here is the program rewritten to eliminate the logical variable EVEN and to use REAL*8 floating-point values. Notice the REAL*8 constants 0.D0, 1.D0, and 0.01D0, and the use of DFLOAT and DABS instead of FLOAT and ABS.

```
      LOGICAL*4 DONE/.FALSE./
      REAL*8 X,XOLD
      I=0
      X=0.D0
    1 IF(DONE) THEN
         PRINT *,X
         STOP
      ELSE
         I=I+1
         X=X+1.D0/DFLOAT(I)
         XOLD=X
         I=I+1
         X=X-1.D0/DFLOAT(I)
         DONE=(DABS(X-XOLD).LT.0.01D0) .OR. (I.GE.100)
         GO TO 1
      ENDIF
      END
```

Because the termination test is performed only in the THEN part of the IF-THEN-ELSE-ENDIF, EVEN or some other variable that alternates between two values is needed to alternate the sign of the terms in the ELSE part if only one term is to be added to the sum at each iteration. To avoid the need for such a variable this version of the program adds *two* terms (and increments I twice) in each iteration. This is called **unrolling** the loop, and is discussed in textbook §15.2.6. The convergence test in the example measures the improvement that is made by adding *one* term of the series, so for comparison XOLD is set here to the value of X *after* the first of the two terms is added in each iteration. When the program is compiled and run it produces the approximation 0.68817217931019570, and also stops when I=100. Making the real variables double precision did not have much effect on the accuracy of the estimate, and neither approximation is very good.

If we substitute DONE=(DABS(X-XOLD).LT.1.D-9) in the program listed above, it produces the approximation 0.69314718006071308 after accumulating 999999974 terms of the series. This result is not nearly accurate or fast enough to be of any practical use, but it does illustrate the notoriously slow convergence of the power series for logorithm and show why the elementary function library uses a different algorithm!

4.10.45 `[H]` According to textbook §4.2, a `REAL*4` number has the following bit pattern.



Its value is normalized if $p > 0$, unnormalized if $p = 0$ and $f \neq 0$, or one of the special bit patterns for zero, $\pm\infty$, or NaN. This program distinguishes those three cases.

```
C     specify the number to be checked
      INTEGER*4 I/Z'40466666'/
      EQUIVALENCE(I,X)
C
      INTEGER*4 PINF/Z'7F800000'/,MINF/Z'FF800000'/,NAN/Z'7FFFFFFF'/
      INTEGER*4 MASK/Z'FF000000'/
      LOGICAL*4 NML
C
C ------------------------------------------------------------------
C
C     check whether the number is zero, infinity, or NAN
      NML=.TRUE.
      IF(I.EQ.0   ) NML=.FALSE.
      IF(I.EQ.PINF) NML=.FALSE.
      IF(I.EQ.MINF) NML=.FALSE.
      IF(I.EQ.NAN ) NML=.FALSE.
      IF(.NOT.NML) THEN
        PRINT *,'X=',X,' is a special value'
        PRINT *,'so it is neither normalized nor unnormalized'
        STOP
      ENDIF
C
C     not a special value; shift out the sign bit
      J=ISHFT(I,1)
C
C     extract the biased exponent p from the first byte
      J=IAND(J,MASK)
C
C     if p is nonzero, the number is normalized
      NML=J.NE.0
      IF(     NML) PRINT *,'X=',X,' is normalized'
      IF(.NOT.NML) PRINT *,'X=',X,' is unnormalized'
      STOP
      END
```

As listed, the program checks the bit string `01000000010001100110011001100110` used as the example in §4.2. That bit string is set in `I` using a compile-time initialization to the corresponding hexadecimal constant. The `EQUIVALENCE` statement overlays the `REAL*4 X` on `I`, giving it the same bit pattern. It is necessary to manipulate the bit pattern in integers because the logical functions `ISHFT` and `IAND` require arguments that are `INTEGER*4`.

The code below the line, which is written to work no matter what value is specified for `X`, begins by checking for the special values. Testing for equality between a real number and NaN yields unequal even if the real number *is* NaN, but comparing the bit patterns as integers yields equal if both are `7FFFFFFF` (that is just the bit pattern of the largest possible

INTEGER*4). For our example all of these tests fail, because the given bit pattern is not one of the special values.

ISHFT is invoked to shift the whole bitstring left by one bit. This throws away the sign bit, which is irrelevant to whether the number is normalized, and moves the 8 bits of the biased exponent p to the front of the number. The IAND performs a bitwise logical AND between the shifted bitstring and MASK, which is initialized using a hexadecimal constant to the bitstring 11111111000000000000000000000000. This leaves J containing only the bits of p. For the example, one of those bits is nonzero so the test J.NE.0 sets NML to .TRUE.; if the exponent were zero, denoting an unnormalized number, NML would be set to .FALSE.. For the example the program reports that X is normalized.

Changing the initial value of I to Z'00466666' gives X a zero exponent, so the IAND sets J to 0, NML comes out .FALSE., and the program reports the number is unnormalized.

Replacing the compile-time initialization of I with the byte code for zero, $\pm\infty$, or NaN results in NML being set to .FALSE. in the first stanza of executable code, and then the program reports that X is neither normalized nor unnormalized.

4.10.46 [H] On a big-endian processor the 52 fraction bits of a `REAL*8` quantity are its rightmost bits, so to set the least-significant 15 of them to zero we can do a bitwise AND between the rightmost word of the value and the mask `11111111111111111000000000000000`. The most obvious approach to this problem uses arrays, which will be introduced in Chapter 5, but it is possible to manipulate just the rightmost word of a `REAL*8` using only the language features we have learned so far. The program below illustrates one approach.

```
      REAL*8 X/1.23456789012345678D0/
      COMPLEX*8 Y
      EQUIVALENCE(X,Y)
      REAL*4 Z,W
      EQUIVALENCE(Z,I)
      EQUIVALENCE(W,J)
      INTEGER*4 MASK/Z'FFFF8000'/
C
C     this code is for big-endian processors
      PRINT *,X
      Z=AIMAG(Y)
      J=IAND(I,MASK)
      Y=CMPLX(REAL(Y),W)
      PRINT *,X
      STOP
      END
```

Overlaying the `COMPLEX*8` variable `Y` on `X` makes the second word of `X` the imaginary part of `Y`. The code copies that bit pattern into the `REAL*4` variable `Z`, which is `EQUIVALENCE`d to the `INTEGER*4` variable `I`. Then `J` can be found as the logical OR of `I` with `MASK`, which is initialized to the byte code for the bit pattern `11111111111111111000000000000000`. But `J` is `EQUIVALENCE`d to `W`, so we can use `W` to put the resulting bit pattern back into the imaginary part of `Y`, and that replaces the second word of `X`.

On a little-endian processor the reversal of the order of the bits in `X` also reverses the first and second words in `Y`, so the least-significant word of `X` becomes the *first* word, or real part, of `Y` and the most-significant word of `X` becomes the second word or imaginary part of `Y`. This requires that the program be changed as follows. The declarations stanza is unchanged, because the compiler will do the initializations consistently for whichever endian-ness the processor has.

```
      REAL*8 X/1.23456789012345678D0/
      COMPLEX*8 Y
      EQUIVALENCE(X,Y)
      REAL*4 Z,W
      EQUIVALENCE(Z,I)
      EQUIVALENCE(W,J)
      INTEGER*4 MASK/Z'FFFF8000'/
C
C     this code is for little-endian processors
      PRINT *,X
      Z=REAL(Y)
      J=IAND(I,MASK)
      Y=CMPLX(W,AIMAG(Y))
      PRINT *,X
      STOP
      END
```

When I compiled and ran this program on my little-endian machine, it produced the following output.

```
unix[1] a.out
   1.2345678901234567
   1.2345678901183419
unix[2]
```

The effect of zeroing out the 15 least-significant bits of `X` is to make the number slightly wrong. Changing `MASK` to `FFFFFFFF`, so that none of the bits get zeroed out, leaves `X` unchanged at its initial value. Changing `MASK` to `00000000`, so that the least-significant 32 bits of `X` get zeroed out (about 60% of the fraction bits), yields the result shown below, in which 10 of the 17 significant digits are wrong (again about 60% of them).

```
unix[3] a.out
   1.2345678901234567
   1.2345676422119141
unix[4]
```

Further degradations of precision could be obtained by zeroing out bits in the most-significant word of `X`. In modeling the loss of precision it is sometimes more realistic to flip a certain number of less-significant bits at random, rather than setting them to zero.[1]

The contortions required by this Exercise are meant to test your understanding of number storage formats and some built-in functions, not to provide an ideal way of simulating degraded precision in floating-point calculations. Bit manipulations are far easier and less obscure in C, so if you need to make lots of numbers imprecise you should write a C function to do that and invoke it from your FORTRAN program.

[1]See **Kupferschmid, Michael** and **Ecker, J. G.**, A note on solution of nonlinear programming problems with imprecise function and gradient values, *Mathematical Programming Study* 31 129-138, 1987.

5.8.19 [P] (a) The program given as the solution to Exercise 3.8.21 is reproduced below.

```
      EPS=.001
      XL=1.
      XR=3.
C
C     find midpoint of current interval
    3 X=(XL+XR)/2.
      IF(ABS(XR-XL).LT.EPS) GO TO 1
      F=SIN(X)-0.5*X
      IF(ABS(F).LT.EPS) GO TO 1
C
C     shorten interval by excluding half that does not contain root
      FL=SIN(XL)-0.5*XL
      IF(F*FL .LT. 0.) GO TO 2
      XL=X
      GO TO 3
    2 XR=X
      GO TO 3
C
C     convergence tolerance satisfied
    1 PRINT *,X
      STOP
      END
```

The free loop in this code is produced by branching back to statement `3` if both convergence tests fail. To bound the iterations we can replace this free loop by a `DO` loop as shown below.

```
      KMAX=100
      EPS=.001
      XL=1.
      XR=3.
C
C     perform up to KMAX iterations
      DO 3 K=1,KMAX
C          find midpoint of current interval
           X=(XL+XR)/2.
           IF(ABS(XR-XL).LT.EPS) GO TO 1
           F=SIN(X)-0.5*X
           IF(ABS(F).LT.EPS) GO TO 1
           KUSED=K
C
C          exclude interval half that does not contain root
           FL=SIN(XL)-0.5*XL
           IF(F*FL .LT. 0.) GO TO 2
           XL=X
           GO TO 3
    2      XR=X
    3 CONTINUE
C
C     convergence tolerance satisfied or KMAX iterations used
    1 PRINT *,X,KUSED
      STOP
      END
```

The original solution followed as literally as possible the flowchart given in Exercise 3.8.21, but the revised program does not, and other stylistic improvements should occur to you as you look at this code.

Now, if convergence is not attained in 100 iterations the program stops anyway. In 100 iterations the initial interval width $\texttt{XR} - \texttt{XL} = 2$ would be reduced to

$$2 \times \left(\tfrac{1}{2}\right)^{100} \approx 1.6 \times 10^{-30} \ll \texttt{EPS} = 0.001$$

Thus $\texttt{KMAX} = 100$ is sure to be enough so that at least the convergence test on interval width is satisfied. Running the program produces the following output.

```
unix[1] a.out
   1.8945313              8
```

The root is the same as reported in the solution to Exercise 3.8.21, and was found in 8 iterations.

(b) Here is the Simpson's Rule program of the solution to Exercise 3.8.17, rewritten to use DO loops instead of free loops.

```
C     set limits, number of strips, strip width
      A=0.5
      B=3.0
      N=1000
      H=(B-A)/1000.
C
C     start with the first two terms
      SUM=ALOG(A)/A+4.0*ALOG(A+H)/(A+H)
      X=A+H
C
C     others have coefficients in the pattern 2,4,2,4...
      DO 1 I=3,N,2
           X=X+H
           SUM=SUM+2.0*ALOG(X)/X
           X=X+H
           SUM=SUM+4.0*ALOG(X)/X
    1 CONTINUE
C
C     finally add in the last (N+1st) term
      SUM=SUM+ALOG(B)/B
C
C     compute the integral and print it out
      ANS=(H/3.0)*SUM
      PRINT *,'the integral is about ',ANS
      STOP
      END
```

As in the code using a free loop, the first two terms and the last term are computed separately from the others, so that the repetitive pattern of alternating coefficients can be isolated in the loop (this works because $\texttt{N} = 1000 \geq 4$). The DO loop uses an increment of 2 because the loop body includes the calculation of two terms. On the first pass the loop adds to SUM terms 3 and 4 of the series, on the second pass terms 5 and 6, and so on until its last pass adds in terms N-1 and N.

When this program is compiled and run it yields the same result as the one given in the solution to Exercise 3.8.17.

(d) The program listed in the solution to Exercise 3.8.20(a) has a free loop over values of the variable M. Rewriting it to use a DO loop requires that we specify an upper limit on M. We know from that solution that M=25 and M=106 solve the problem, so an upper limit of 200 will be adequate. The DO loop is bounded, so we need not count the solutions as a way of exiting the loop, and the code below removes those statements.

```
      DO 1 M=1,200
C          put M coconuts in the pile
           N=M
C
C          first person wakes up
           IF(N-3*(N/3).NE.1) GO TO 1
           N=2*(N/3)
C
C          second person wakes up
           IF(N-3*(N/3).NE.1) GO TO 1
           N=2*(N/3)
C
C          third person wakes up
           IF(N-3*(N/3).NE.1) GO TO 1
           N=2*(N/3)
C
C          division of the rest at dawn
           IF(N-3*(N/3).NE.0) GO TO 1
C
           PRINT *,M
    1 CONTINUE
      STOP
      END
```

Running the program produces this output.

```
unix[1] a.out
          25
         106
         187
unix[2]
```

Now the calculation reveals a third answer, which is given by the closed-form solution formula for $c = 3$:

$$(1 + pc)p^p - (p - 1) = (1 + 3 \times 2)3^3 - (3 - 1) = 187.$$

The program listed in the solution to Exercise 3.8.20(b) also has a free loop over M. This time rewriting it to use a DO loop is not so easy, because the user can input any number of stranded people and that makes it hard to decide on an upper limit for M. In such a case all we can do is use the largest possible DO index, $2^{31} - 1$, and restore the test on how many solutions have been found as a way to exit the loop before the limit on M is reached. This might seem like just another way to form a loop that is essentially free, but it prevents the loop index from ever being incremented past the largest positive integer and thereby becoming negative.

```
C     find out how many people there are
      READ *,NP
C
      K=0
      DO 1 M=1,2147483647
           IP=0
           N=M
C
C          the people wake up one after another
    2      IP=IP+1
           IF(N-NP*(N/NP).NE.1) GO TO 1
           N=(NP-1)*(N/NP)
           IF(IP.LT.NP) GO TO 2
C
C          division of the remaining pile at dawn
           IF(N-NP*(N/NP).NE.0) GO TO 1
C
           PRINT *,M
           K=K+1
           IF(K.EQ.2) STOP
    1 CONTINUE
      STOP
      END
```

When this program is compiled and run it produces the output shown below.

```
unix[1] a.out
3
           25
          106
unix[2] a.out
7
       823537
      6588338
unix[3]
```

These answers for $p = 3$ are the same as we got before; the two solutions for $p = 7$ show that `M` must be allowed to get big even for modest values of p. You can confirm by experiment that for values of p that require the smallest solution to be more than the largest permissible `DO` index, the program produces no output.

(h) The next page shows a program that uses a `DO` loop to evaluate the formula

$$\phi = \left(1 - \alpha + \frac{1}{2\pi}\sin(2\pi\alpha)\right)^{3/2}$$

at 101 values of α evenly spaced from -1 to 1.

```
      REAL*8 ALPHA,PHI,TWOPI/6.2831853071795865D0/
      DO 1 I=1,101
           ALPHA=-1.D0+0.02D0*DFLOAT(I-1)
           PHI=(1.D0-ALPHA+(1.D0/TWOPI)*DSIN(TWOPI*ALPHA))**1.5D0
           PRINT *,ALPHA,PHI
    1 CONTINUE
      STOP
      END
```

When it is compiled with `gfortran` and run it produces output some of which is shown below.

```
unix[1] a.out
 -1.0000000000000000        2.8284271247461903
 -0.97999999999999998       2.8283155517840486
   :                          :
  0.98000000000000009       3.81446243721525688E-007
  1.0000000000000000                            NaN
unix[2]
```

For `ALPHA=1.D0` the program reports `NaN` for `PHI`, as predicted in the statement of Exercise 4.10.21, but from inspection of the formula it is clear that $\phi(1) = 0$. The program below reports the correct value and also the computed value of the quantity inside the outer parentheses.

```
      REAL*8 ALPHA,PHI,TWOPI/6.2831853071795865D0/,QTY
      DO 1 I=1,100
           ALPHA=-1.D0+0.02D0*DFLOAT(I-1)
           PHI=(1.D0-ALPHA+(1.D0/TWOPI)*DSIN(TWOPI*ALPHA))**1.5D0
           PRINT *,ALPHA,PHI
    1 CONTINUE
      ALPHA=1.D0
      PRINT *,ALPHA,0.D0
      PRINT *
      QTY=(1.D0-ALPHA+(1.D0/TWOPI)*DSIN(TWOPI*ALPHA))
      PRINT *,'when ALPHA=',ALPHA,' quantity=',QTY
      STOP
      END
```

When it is compiled and run the end of its output is as shown below. Logarithms are used in raising to a real power, so $(-3.9 \times 10^{-17})^{3/2}$ is what came out `NaN` before.

```
unix[3] a.out
   :                          :
  0.98000000000000009       3.81446243721525688E-007
  1.0000000000000000        0.0000000000000000

 when ALPHA=  1.0000000000000000       quantity= -3.89804309105147787E-017
unix[4]
```

(m) Here is the program presented in the solution to Exercise 4.10.39.

```
      REAL*8 Z/0.05D0/,R,HMAG
      COMPLEX*16 H
      K=0
      R=0.D0
    1 K=K+1
        R=R+0.1D0
        H=DCMPLX(1.D0,0.D0)/DCMPLX(1.D0-R**2,2.D0*Z*R)
        HMAG=DSQRT(DREAL(H)**2+DIMAG(H)**2)
        PRINT *,R,HMAG
      IF(K.LT.100) GO TO 1
      STOP
      END
```

The free loop can be replaced by a DO loop like this.

```
      REAL*8 Z/0.05D0/,R,HMAG
      COMPLEX*16 H
      R=0.D0
      DO 1 K=1,100
           R=R+0.1D0
           H=DCMPLX(1.D0,0.D0)/DCMPLX(1.D0-R**2,2.D0*Z*R)
           HMAG=DSQRT(DREAL(H)**2+DIMAG(H)**2)
           PRINT *,R,HMAG
    1 CONTINUE
      STOP
      END
```

These programs produce identical output, but the second version is one line shorter and less work to understand.

5.8.28 [P] (a) Here is a program that tests the conjecture for each starting value between 1 and 100. (b) It prints out how many values are generated until a 1 appears, counting the starting value and the 1.

```
C     try each starting value
      DO 1 N=1,100
           I=N
           K=1
    2      K=K+1
           IF(MOD(I,2).EQ.0) THEN
C             even; halve it
              I=I/2
           ELSE
C             odd; multiply by 3 and add 1
              I=3*I+1
           ENDIF
           IF(I.NE.1) GO TO 2
           PRINT *,N,' leads to a 1 for term',K
    1 CONTINUE
      STOP
      END
```

When it is compiled and run the program produces output some of which is shown below.

```
unix[1] a.out
 1 leads to a 1 for term 4
 2 leads to a 1 for term 2
 3 leads to a 1 for term 8
 4 leads to a 1 for term 3
 5 leads to a 1 for term 6
 6 leads to a 1 for term 9
 :
 100 leads to a 1 for term 26
```

5.8.29 `[P]` The next page lists a program that finds both answers by exhaustively testing all pairs of five-digit integers manufactured as described in the problem statement. When the program is compiled and run (for about 15 minutes on a 1GHz processor) it produces the following output.

```
max difference is  97531 when A= 1234 B= 98765
min difference is  247 when A= 49876 B= 50123
```

The program is very inefficient, because it generates all $10^{10} = 10000000000$ combinations of the digits and rejects all but the $10! = 3628800$ that are permutations as required. It is possible to write a FORTRAN program that generates *only* the permutations, but that requires programming techniques more sophisticated than those discussed so far (e.g., see §11.7 and think of arranging the permutations in a tree). It also tests twice as many number pairs that do contain permutations of the digits as needed, because it is the absolute difference that matters. Thus, for example, `A=98765`, `B=1234` is generated and tested even though it yields the same maximum absolute difference as `B=98765`, `A=1234`.

A little thought leads humans to the first answer, that the largest difference occurs when `A=01234` and `B=98765`, so no actual calculations are needed in that case. The integers that yield the smallest difference, `A=49876` and `B=50123`, also have digit patterns that suggest they might be explained by some simple logical argument. Unfortunately, it is seldom obvious how to automate this sort of reasoning in a low-level programming language.

```
      INTEGER*4 I(10),A,B,DIF
      INTEGER*4 AMAX,AMIN,BMAX,BMIN,DIFMAX/00000/,DIFMIN/99999/
C
C     generate all possible combinations of the digits 0-9
      DO 1 I1=0,9
      DO 1 I2=0,9
      DO 1 I3=0,9
      DO 1 I4=0,9
      DO 1 I5=0,9
      DO 1 I6=0,9
      DO 1 I7=0,9
      DO 1 I8=0,9
      DO 1 I9=0,9
      DO 1 I10=0,9
C
C          if two digits are the same it's not a permutation
           I(1)=I1
           I(2)=I2
           I(3)=I3
           I(4)=I4
           I(5)=I5
           I(6)=I6
           I(7)=I7
           I(8)=I8
           I(9)=I9
           I(10)=I10
           DO 2 J=1,9
           DO 2 K=J+1,10
                IF(I(K).EQ.I(J)) GO TO 1
    2      CONTINUE
C
C          the digits are all different; find the 5-digit integers
           A=10000*I1+1000*I2+100*I3+10*I4+I5
           B=10000*I6+1000*I7+100*I8+10*I9+I10
C
C          remember which combinations yield max and min differences
           DIF=IABS(A-B)
           IF(DIF.GT.DIFMAX) THEN
              DIFMAX=DIF
              AMAX=A
              BMAX=B
           ENDIF
           IF(DIF.LT.DIFMIN) THEN
              DIFMIN=DIF
              AMIN=A
              BMIN=B
           ENDIF
    1 CONTINUE
C
C     report the results
      PRINT *,'max difference is ',DIFMAX,' when A=',AMAX,' B=',BMAX
      PRINT *,'min difference is ',DIFMIN,' when A=',AMIN,' B=',BMIN
      STOP
      END
```

5.8.30 [P] Here is a program that performs the required calculation.

```
      REAL*8 HH(30),H,DD,D
      H=3.6D0
      D=42.D0
      DD=10.D0*12.D0
      HH(1)=0.D0
      DO 1 N=2,30
           HH(N)=(HH(N-1)+H)*(DD+D*DFLOAT(N-1))/(DD+D*DFLOAT(N-2))
    1 CONTINUE
      DO 2 N=1,30,2
           PRINT *, N,HH(N),N+1,HH(N+1)
    2 CONTINUE
      STOP
      END
```

Notice that DD, which represents D, the distance from the stage to the first row of seats, must be in inches. The factors $(n-1)$ and $(n-2)$ are converted from INTEGER*4 to REAL*8 in order to avoid mixing modes in the calculation. When compiled and run, the code produces the output listed below. Two results are printed on each line to save space.

```
 1   0. 2   4.86
 3   10.6533333 4   17.1878431
 5   24.3369871 6   32.011131
 7   40.1434568 8   48.6822342
 9   57.586229 10   66.8218027
 11   76.3609909 12   86.180179
 13   96.259161 14   106.580451
 15   117.128767 16   127.890643
 17   138.854119 18   150.008505
 19   161.344185 20   172.852468
 21   184.525456 22   196.355945
 23   208.337332 24   220.463546
 25   232.728987 26   245.128471
 27   257.657185 28   270.310652
 29   283.084693 30   295.975401
```

The slope or **rake** of the auditorium floor increases with increasing distance from the stage (or movie screen). Building a floor that curves in this way is more expensive than using a constant rake, so the patrons of most theatres find themselves trying to look between the people sitting in front of them, or unable to see.

5.8.31 `[P]` The formula[2] for the deflection y makes sense only if the quantities used to evaluate it have compatible units, but some lengths are given in feet and others in inches. If we convert all of the lengths to inches and w to lb/in, then the units in the formula will be

$$\frac{[\text{lb/in}]}{[\text{lb/in}^2][\text{in}^4]}\Big([\text{in}^2][\text{in}^2] - [\text{in}][\text{in}^3] + [\text{in}^4]\Big) = [\text{in}]$$

so y will be computed in inches. The data for the problem are then as summarized below.

quantity	given value		units-corrected value		variable
l	6	[ft]	`6*12`	[in]	`L`
w	$320/l$	[lb/ft]	`320/(6*12)`	[lb/in]	`W`
E	1600000	[lb/in^2]	`1600000`	[lb/in^2]	`E`
a	1.625	[in]	`1.625`	[in]	`A`
b	8.25	[in]	`8.25`	[in]	`B`

The problem statement calls for 20 values of x between 0 and l, including both end values, so there are 19 equal intervals of width `DX` between consecutive values of x. Here is a program that performs the calculation.

```
      REAL*8 L,W,E,A,B,I,DX,X,Y
      L=6.D0*12.D0
      W=320.D0/L
      E=1600000.D0
      A=1.625D0
      B=8.25D0
      I=A*(B**3)/12.D0
      DX=L/19.D0
      DO 1 K=1,20
           X=DX*DFLOAT(K-1)
           Y=(W/(2.D0*E*I))*( (L**2)*(X**2)/2.D0
     ;                       - L*(X**3)/3.D0
     ;                       + (X**4)/12.D0 )
           PRINT *,'x=',X,' y=',Y
    1 CONTINUE
      STOP
      END
```

When it is compiled and run the program generates the output shown at the top of the next page. The deflection is zero at the fixed end and increases monotonically to about an eighth of an inch at the free end. The given modulus of elasticity and dimensions suggest the beam might be a wooden 2×9 standing on edge, so this amount of deflection under a distributed load of only $53\frac{1}{3}$ pounds per foot seems plausible.

[2]See **Den Hartog, J. P.**, *Strength of Materials,* Dover, New York, 1961, §18.

```
unix[1] a.out
 x=  0. y=  0.
 x=  3.78947368 y=  0.000656329473
 x=  7.57894737 y=  0.00253366404
 x=  11.3684211 y=  0.00550017282
 x=  15.1578947 y=  0.00943155811
 x=  18.9473684 y=  0.0142110554
 x=  22.7368421 y=  0.0197294335
 x=  26.5263158 y=  0.0258849941
 x=  30.3157895 y=  0.0325835724
 x=  34.1052632 y=  0.0397385367
 x=  37.8947368 y=  0.0472707885
 x=  41.6842105 y=  0.0551087623
 x=  45.4736842 y=  0.0631884261
 x=  49.2631579 y=  0.0714532809
 x=  53.0526316 y=  0.0798543609
 x=  56.8421053 y=  0.0883502336
 x=  60.6315789 y=  0.0969069997
 x=  64.4210526 y=  0.105498293
 x=  68.2105263 y=  0.11410528
 x=  72. y=  0.122716662
unix[2]
```

5.8.32 [P] (a) Here is a program that performs the calculation and prints the required table of values.

```
      REAL*8 Q,FN
      PRINT *,'N  Q'
      DO 1 N=2,10
           FN=DFLOAT(N)
           Q=DSQRT((FN-1.D0)/(FN+1.D0))*(FN/DSQRT(FN**2-1.D0))**N
           PRINT *,N,Q
    1 CONTINUE
      STOP
      END
```

(b) For $n = 1$ we find

$$q(1) = \sqrt{\frac{n-1}{n+1}}\left(\frac{1}{\sqrt{n^2-1}}\right)^1 = \frac{\sqrt{n-1}}{\sqrt{n+1}(\sqrt{n-1}\sqrt{n+1})} = \frac{1}{n+1} = \frac{1}{2},$$

so $\lim_{n\to 1} q(n) = \frac{1}{2}$. The formula given in the problem statement for $q(n)$ and coded in the solution to part (a) cannot simply be evaluated for $n = 1$ because that results in a division by zero. Modifying the program to handle the case of $n = 1$ separately results in the following code.

```
      REAL*8 Q,FN
      PRINT *,'N  Q'
      DO 1 N=1,10
           FN=DFLOAT(N)
           IF(N.EQ.1) THEN
             Q=0.5D0
           ELSE
             Q=DSQRT((FN-1.D0)/(FN+1.D0))*(FN/DSQRT(FN**2-1.D0))**N
           ENDIF
           PRINT *,N,Q
    1 CONTINUE
      STOP
      END
```

The value of `N` appears in the formula several times as a real number, so to avoid having to use `DFLOAT` repeatedly the code assigns `FN` once and uses that in the formula. In raising to a whole-number power `N` is used in preference to `FN` (see §15.2.4). When this program is compiled and run, it produces this output.

```
N  Q
1  0.5
2  0.769800359
3  0.84375
4  0.881318877
5  0.904224537
6  0.919685526
7  0.930834735
8  0.939259212
9  0.945850803
10  0.95114984
```

(c) To experiment with some very large values of n we can use this program.

```
      REAL*8 Q,FN
      PRINT *,'N  Q'
      DO 1 I=1,10
           FN=10.D0**I
           Q=DSQRT((FN-1.D0)/(FN+1.D0))*(FN/DSQRT(FN**2-1.D0))**FN
           PRINT *,FN,Q
    1 CONTINUE
      STOP
      END
```

Now the `DO` indexes not `N` but the power of ten that `N` is. The largest value that will fit in an `INTEGER*4` is only about 2×10^9 so we must avoid computing `N` as an integer and use `FN` for the power. When this program is compiled and run it produces the output shown below.

```
 N  Q
  10.  0.95114984
  100.  0.995012396
  1000.  0.999500125
  10000.  0.999950001
  100000.  0.999995
  1000000.  0.9999995
  10000000.  0.999999951
  100000000.  0.99999999
  1.E+09  0.999999999
  1.E+10  1.
```

From these experimental results it appears that

$$\lim_{n\to\infty} q(n) = 1.$$

(d) As n becomes large, the 1s in the formula for $q(n)$ become negligible by comparison and $q(n) \to 1$. A more rigorous demonstration can be made by recalling that

$$\lim_{x\to\infty} \left(1 + \frac{1}{x}\right)^x = e,$$

the base of natural logarithms, and then reasoning as shown on the next page.

$$
\begin{aligned}
[q(n)]^2 &= \left(\frac{n-1}{n+1}\right)\left(\frac{n}{\sqrt{n^2-1}}\right)^{2n} = \left(\frac{n-1}{n+1}\right)\frac{n^{2n}}{(n^2-1)^n} \\
[q(n)]^2 &= \frac{(n-1)(n^{n-1})(n^{n+1})}{(n+1)(n-1)^n(n+1)^n} = \frac{(n^{n-1})(n^{n+1})}{(n-1)^{n-1}(n+1)^{n+1}} \\
[q(n)]^2 &= \frac{\dfrac{n^{n-1}}{(n-1)^{n-1}}}{\dfrac{(n+1)^{n+1}}{n^{n+1}}} = \frac{\left(\dfrac{n}{n-1}\right)^{n-1}}{\left(\dfrac{n+1}{n}\right)^n\left(\dfrac{n+1}{n}\right)} \\
[q(n)]^2 &= \frac{\left(1+\dfrac{1}{n-1}\right)^{n-1}}{\left(1+\dfrac{1}{n}\right)^n\left(\dfrac{n+1}{n}\right)}. \\
\left(\frac{n+1}{n}\right)[q(n)]^2 &= \frac{\left(1+\dfrac{1}{n-1}\right)^{n-1}}{\left(1+\dfrac{1}{n}\right)^n} \\
\left(\frac{n+1}{n}\right)\lim_{n\to\infty}[q(n)]^2 &= \frac{e}{e} = 1 \\
\lim_{n\to\infty}[q(n)]^2 &= \lim_{n\to\infty}\frac{n}{n+1} = \lim_{n\to\infty}\frac{d(n)/dn}{d(n+1)/dn} = \lim_{n\to\infty}\frac{1}{1} = 1.
\end{aligned}
$$

The last line makes use of L'Hospital's rule to find the limit of $n/(n+1)$.

This determination of $\lim_{n\to\infty}[q(n)]^2$, which is due to C. E. Lemke, also yields the approximation that when n is large, $q(n) \approx \sqrt{n/(n+1)}$, and a little extra work shows that this is in turn approximately $1 - 1/(2n)$. A program to calculate $q(n)$ for arbitrary values of n might save some processor time by using one or the other approximation, rather than the given formula, for sufficiently large values of n.

5.8.35 `[H]` (a) On a little-endian processor the given initial value of `I(1)` is actually stored as `0000F03F`, so the byte pattern for `DW` is `0000F03F00000000` in memory. The `REAL*8` value is therefore the one corresponding to the big-endian byte pattern `000000003FF00000`, which represents an unnormalized `REAL*8` value. The code below initializes a doubleword to that byte pattern and prints the value.

```
      REAL*8 DW/Z'000000003FF00000'/
      PRINT *,DW
      STOP
      END
```

When compiled with `gfortran` and run, the program produces the output shown below.

```
unix[1] a.out
  5.29980882362664448E-315
unix[2]
```

This agrees with the result given in the problem statement, to the digits printed there. The reason that initializing the elements of `I` separately does not work as intended on a little-endian machine is that *the bytes of each word in* `I` *are stored in reversed order, while the words in* `DW` *are stored in the order shown.*

(b) Anticipating the byte reversal, we could change the compile-time initialization of `I` like this to give `DW` the value `1.D0` on a little-endian machine.

```
      REAL*8 DW
      INTEGER*4 I(2)/Z'00000000',Z'3FF00000'/
      EQUIVALENCE(DW,I)
      PRINT *,DW
      STOP
      END
```

Now the words are reversed too, so the byte string stored for `DW` is `000000000000F03F`, which is the little-endian version of `3FF0000000000000` and therefore represents `1.D0` as shown in the table of textbook §4.7.

(c) Initializing `DW` to the desired byte string directly, as in the second program given with the problem statement, results in the bytes being stored as `000000000000F03F` on a little-endian machine. That is the little-endian version of `3FF0000000000000` and thus represents `1.D0` as desired. *When* `DW` *is initialized directly, there are no array elements to end up in the wrong order so the whole byte string gets reversed in memory on a little-endian machine,* and represents the intended value. Of course, if the program is compiled for a big-endian machine the bytes get stored in the order shown and also represent the intended value.

5.8.36 `[H]` The Index entry for "bugs, disappearing" refers us to textbook §14.3.4, but the discussion in §5.1 provides enough information to answer this question. Here is a program (which I will assume is in a file named `bug.f`) illustrating the phenomenon.

```
      INTEGER*4 ARRAY(5)
      DO 1 I=1,6
C          PRINT *,I
           ARRAY(I)=0
    1 CONTINUE
      STOP
      END
```

When the program is compiled using `gfortran` (version `4.3.2-1ubuntu12`) the executable that is produced runs in an endless loop until it is interrupted.

```
unix[1] gfortran bug.f
unix[2] a.out
^C
unix[3]
```

The assignment `ARRAY(6)=0` changes the fullword next after the memory the compiler assigned to hold the array, and that somehow interferes with the `DO` loop indexing mechanism so that its test for completion is never satisfied. If the `C` is removed from the `PRINT` statement and the program is compiled the same way, the endless loop disappears.

```
unix[4] gfortran bug.f
unix[5] a.out
1
2
3
4
5
6
unix[6]
```

Adding the `PRINT` statement has displaced the quantity involved in loop indexing to some other memory location, so when the word following the array gets overwritten with zero that mechanism is no longer interfered with and the loop terminates normally. The `PRINT` evidently works as intended too, because the output we requested appears on the screen. Perhaps the word that gets zeroed out incorrectly now holds some quantity that is important only before the loop begins. In a more complicated program the zeroed-out word might play some crucial role later in the processing, so it is never "good enough" to leave the actual bug undiscovered and just keep the `PRINT` statement, hoping that since it somehow makes the visible evidence go away now everything is all right. Whenever you notice this sort of behavior it is essential to investigate further, by setting the compiler option to check for subscript overflows and perhaps by using a debugger, to find the actual mistake. In this case the error is either that the `DO` limit of `6` is too big or that `ARRAY` is dimensioned too small.

A runaway subscript can lead to bafflement and disaster in any language, including C and its many derivatives. Always use the compiler option for detecting subsubscript overflows, if one is available, unless execution speed is very important *and* you have already tested thoroughly enough with subscript checking on that you are very sure the code is right. For UNIX™ FORTRAN compilers, use `-C` or `-fbounds-check`, depending on the version.

6.8.8 [H] Here is a subroutine that implements the algorithm of §5.6 to find the product of two 10×10 matrices.

```
      SUBROUTINE MATMPY(A,B, C)
      REAL*4 A(10,10),B(10,10),C(10,10)
      DO 1 I=1,10
      DO 1 J=1,10
           C(I,J)=0.D0
           DO 2 K=1,10
                C(I,J)=C(I,J)+A(I,K)*B(K,J)
    2      CONTINUE
    1 CONTINUE
      RETURN
      END
```

The core of this routine is the nest of loops shown in the complete program of §5.6, with the limits set to 10. Here, however, the initialization `C(K,J)=0.D0` is needed before the loop over `K`. In the complete program the whole matrix `C` was initialized to zero at compile time, but `MATMPY` might be called repeatedly and each element of `C` needs to start at zero each time.

This routine is of limited usefulness because of the fixed sizes of the arrays, a defect that we will remove in §7.

6.8.20 [P] The code listed on the right below satisfies the requirements of the exercise. The statements that make up the body of subroutine MYRAND are similar to those appearing in the solution to Exercise 4.10.48.

```
      INTEGER*4 ISEED/123457/
      DO 1 K=1,100
           CALL MYRAND(ISEED)
           PRINT *,ISEED
    1 CONTINUE
      STOP
      END
```

```
      SUBROUTINE MYRAND(ISEED)
      INTEGER*4 A/843314861/,B/453816693/
      INTEGER*4 PLUS/Z'7FFFFFFF'/
      ISEED=A*ISEED+B
      ISEED=IAND(ISEED,PLUS)
      RETURN
      END
```

When these routines are compiled together and run, the program produces output that begins like this.

```
 1421872482
 307751087
 232397240
 927916749
 787739646
 970500123
 688995764
 991583257
 1558705242
 1344884295
 1388564848
 1693637157
 41462390
 1665615667
 1224462572
 :
```

6.8.23 [H] (a) A compile-time initialization for B would start like this

```
      REAL*8 B(100)/1.D0,1.D0,2.D0,3.D0,3.5D0,3.8333333333333335D0,
     ;              4.1190476190476195D0,...
```

and go on for many continuation lines. To avoid the tedium of calculating or typing in the 100 values one could write a program that prints the necessary source statements, and then paste them into the code for PM, but we won't learn how to do that until Chapter 10. Having the function begin with a page of data would also make it hard to read, so this approach has enough drawbacks that the solution involving detection of first entry is probably better.

(b) Moving the initialization of B to a separate subroutine yields code like this.

```
      REAL*8 X(100),Y(100),PM
      REAL*8 B(100)
      CALL SETB(B)
      READ *,X
      PRINT *,PM(X,B)
      READ *,Y
      PRINT *,PM(Y,B)
      STOP
      END

      FUNCTION PM(A,B)
      REAL*8 PM,A(100),B(100)
      PM=0.D0
      DO 2 I=1,100
           PM=PM+A(I)*B(I)
    2 CONTINUE
      RETURN
      END

      SUBROUTINE SETB(B)
      REAL*8 B(100)
      B(1)=1.D0
      B(2)=1.D0
      DO 1 I=3,100
           B(I)=B(I-1)+1.D0/B(I-2)
    1 CONTINUE
      RETURN
      END
```

This is a good solution if SETB might also be called in other circumstances; if not, it requires someone reading the code to understand two subprograms instead of one, and to notice that SETB is invoked first in the main.

On the other hand, since PM now just computes a dot product, it could be replaced by a library routine, such as the version of DDOT we will encounter in textbook §7.1, and that would make the program simpler rather than more complicated. The cleanest solution to this problem is thus to compute B first using SETB and then to find the results to print by invoking a Chapter 7 version of DDOT. If the calculation performed by PM had been more complicated, so that this simplification were not possible, then once again detecting first entry might still have been simpler.

6.8.28 [H] (a) Here is a statement that computes I mod K without using any built-in functions.

```
      J=I-K*(I/K)
```

Integer division truncates, so I/K evaluates to the quotient without the remainder. Multiplying by K yields the multiple of K that is in I. Subtracting that from I yields the remainder from the integer division of I by K. For example, if I=17 and K=5 we compute (I/K)=3, K*(I/K)=5*3=15, and J=I-K*(I/K)=17-15=2, which is the correct remainder.

(b) The MOD function is defined for negative values of I or K, but not for K=0. Here is a program that tries some combinations of values using both the formula given above and the built-in MOD function.

```
      DO 1 K=-3,3
           IF(K.EQ.0) GO TO 1
           DO 2 I=-3,3
                J=I-K*(I/K)
                PRINT *,I,K,J,MOD(I,K)
    2      CONTINUE
    1 CONTINUE
      K=0
      I=10
      PRINT *,MOD(I,K)
      STOP
      END
```

When the program is compiled with gfortran and run it produces the output shown on the next page. The remainder from division by +1 or -1 is always zero, so I have replaced 12 lines showing that result by vertical ellipses : to shorten the output. The nest of loops executes successfully because the test skips iterations in which K=0. Then the final invocation of MOD with K=0 causes a run-time error. This is reported as a Floating point exception, which suggests that the MOD function used by gfortran might make use of floating-point instructions internally.

(c) The reasoning given above to explain how the calculation J=I-K*(I/K) works does not involve any assumptions about the sign of either I or K, so the formula finds the remainder, just as MOD(I,K) does, whether I is positive, negative, or zero and whether K is positive or negative.

```
unix[1] a.out
          -3          -3           0           0
          -2          -3          -2          -2
          -1          -3          -1          -1
           0          -3           0           0
           1          -3           1           1
           2          -3           2           2
           3          -3           0           0
          -3          -2          -1          -1
          -2          -2           0           0
          -1          -2          -1          -1
           0          -2           0           0
           1          -2           1           1
           2          -2           0           0
           3          -2           1           1
          -3          -1           0           0
           :           :           :           :
           3           1           0           0
          -3           2          -1          -1
          -2           2           0           0
          -1           2          -1          -1
           0           2           0           0
           1           2           1           1
           2           2           0           0
           3           2           1           1
          -3           3           0           0
          -2           3          -2          -2
          -1           3          -1          -1
           0           3           0           0
           1           3           1           1
           2           3           2           2
           3           3           0           0
Floating point exception
unix[2]
```

6.8.30 `[P]` (a) The functions below determine whether an integer is prime.

```
      FUNCTION PRIME(N)
      LOGICAL*4 PRIME
      M=IABS(N)
      PRIME=.FALSE.
      IF(M.EQ.0) RETURN
      K=M-1
      DO 1 I=2,K
           IF(MOD(M,I).EQ.0) RETURN
    1 CONTINUE
      PRIME=.TRUE.
      RETURN
      END
```

```
      FUNCTION PRIME(N
      LOGICAL*4 PRIME
      M=IABS(N)
      PRIME=.FALSE.
      IF(M.EQ.0) RETURN
      K=IFIX(SQRT(FLOAT(N)))
      DO 1 I=2,K
           IF(MOD(M,I).EQ.0) RETURN
    1 CONTINUE
      PRIME=.TRUE.
      RETURN
      END
```

A negative number can be prime (such as -3) or composite (such as $-4 = -2 \times +2$), so both routines begin by finding the absolute value of the argument. If the number is zero each routine returns `.FALSE.` because zero is not divisible by itself and is therefore not prime. If the number is not zero each routine checks for factors.

The routine on the left examines all possible factors from `2` through `M-1`, because if the number is composite it will be divisible by at least one of those values. In that case `MOD(M,I)`, the remainder from dividing `M` by `I`, will be zero. If a divisor is found the routine returns with `PRIME` set to `.FALSE.` but if no divisor is found it sets `PRIME=.TRUE.` before returning.

The routine on the left actually checks more possible divisors than it needs to, because if a number has factors at least one of them will be no greater than the square root of the number. For example, the composite number $51983 = 227 \times 229$ has one factor less than $\sqrt{51983} \approx 227.998$, so if we check for factors through 227 that is sure to be enough. On the right above is a version of `PRIME` that uses this idea to reduce the number of trial divisors. Now the search runs from `2` only up to the largest integer not exceeding $\sqrt{\texttt{N}}$.

(b,c) This program determines that 123457 is prime, and prints the first 100 primes.

```
      LOGICAL*4 PRIME
C
      IF(PRIME(123457)) THEN
         PRINT *,'123457 is prime'
      ELSE
         PRINT *,'123457 is composite'
      ENDIF
C
      NP=0
      N=0
    1 N=N+1
      IF(PRIME(N)) THEN
         PRINT *,N
         NP=NP+1
      ENDIF
      IF(NP.LT.100) GO TO 1
      STOP
      END
```

6.8.35 [H] The smallest representable positive floating-point number is (according to textbook Chapter 4) $4.9406564584124654 \times 10^{-324}$. For the exponential function to yield that number,

$$\begin{aligned} e^x &\approx 4.9 \times 10^{-324} \\ x &\approx \ln(4.9 \times 10^{-324}) \\ x &\approx -744 \end{aligned}$$

To compute this value exactly we can use the following program.

```
      REAL*8 X,TINY/Z'0000000000000001'/
      X=DLOG(TINY)
      PRINT *,'DEXP(',X,')=',DEXP(X)
      STOP
      END
```

When compiled (with `gfortran`) and run this program produces the following output.

```
unix[1] a.out
 DEXP( -744.44007192138122      )= 4.94065645841246544E-324
unix[2]
```

Experiments reveal that the most negative `X` for which `DEXP(X)` returns a nonzero value is actually $\texttt{C0874910D52D2053}_{16}$ = $\texttt{-7.451332191014757D+02}_{10}$, but the value returned is still `4.94065645841246544D-324`. The argument value -745.1332191014757 is more negative than $-\ln(R_{max}) \approx -710$ (see textbook Section 6.6.1) because the smallest representable value is **subnormal**. The smallest subnormal has only 1 bit of precision, so the digits of its decimal equivalent are not signficant. Practical numerical calculations should be scaled in such a way that the argument of `DEXP` is never more negative than -710, even though the function returns values for more negative arguments, unless we are content to know that the result is "negligibly small" and don't need to know its actual value.

6.8.42 [H] On every machine I have tried, the compiler links in the FORTRAN system routine for `DSQRT` rather than the local alternative provided in the given source code, and the program prints an approximation to the square root of 2 rather than printing 4. This illustrates that if you unwittingly give one of your routines a name that FORTRAN recognizes as a built-in function, it is the built-in function that will be invoked. If you suspect this might be happening, you can put a `PRINT` statement in your code to confirm that it is actually being entered.

7.4.1 `[E]` It is occasionally desirable simply to give an array the same fixed dimensions in all of the routines where it appears. For example, a program having to do with the game of chess might use a two-dimensional array that is 8×8 to represent the board, and these dimensions are not likely to change over the life of the program. In such a case it is unnecessary to use adjustable dimensions, so the code will be simpler if they are fixed instead. It might also be desirable in such an application for the programmer to be constantly reminded of the actual fixed dimensions of an array by seeing the dimensions given as numbers in each routine where the array is used. The `changeall` script of text §18.6 can of course be used to change all occurrences of a fixed dimension if that turns out to be necessary after all.

Much more often it is inconvenient simply to give an array the same fixed dimensions in all of the routines where it appears, because in most applications array sizes vary from one problem instance to the next. Then the dimensions must be adjusted everywhere and the code recompiled for each new problem size. The difficulty of doing this is increased if some of the routines are maintained in a subprogram library rather than with the main programs that use them, because it then might be necessary to change the array sizes for the different programs as well as for different problem sizes solved by each program.

If a program invokes some subprogram more than once and different invocations pass arrays of different sizes, then it is impossible to use fixed array dimensions in the subprogram.

7.4.2 `[E]` In Classical FORTRAN, adjustable dimensions *cannot* be used in declaring an array where it is originally allocated, such as in a main program. Doing that would amount to dynamic memory allocation, which is not allowed (but see text §17.1.2 about dynamic memory allocation in modern FORTRAN).

7.4.5 `[E]` [B] The last, because only that dimension is not used by the machine code of the subprogram in calculating the memory addresses of array elements.

7.4.6 `[E]` (a) The program prints out `5`. In `TRYDIM` the dimension is stated as `L(1:6)` so `L(5)` refers to the fifth fullword in `L`. In the main program, the fifth fullword in `L` is set by the compile-time initialization to the value `5`. The dimensioning information given for `L` in the main program is irrelevant in this case.

(b) Changing the dimensioning in `TRYDIM` to `L(*)` makes no difference at all; the program still prints `5`. Separate compilation makes it impossible for `TRYDIM` to know how `L` is dimensioned in the main program, so when the compiler finds the dimension `L(*)` in the subroutine it can only assume that `L(5)` refers to the fifth fullword of `L`, starting from the address that is passed for that parameter. In the main program that is just the starting address of `L`. Thus the dimensioning information given for `L` in the main program is also irrelevant in this case.

The point of this exercise is that care must be exercised in passing arrays that are dimensioned to start at some element other than 1.

7.4.8 `[H]` Memory for the vector `Z` is allocated in `SUB`, where space is provided for only 5 elements. Thus the index `J` in `ISMSQ` must not exceed `5` no matter how `Z` is declared in `ISMSQ`. This means that `N+1` can be no larger than `5`, so `N` can be no larger than `4`. Because `Z` is declared assumed-size within `ISMSQ` and compilation is separate, the compiler has no way of knowing how big `Z` really is, so using the `-C` option does *not* permit detection of a value of `J` (hence `N`) that is too big. For the code itself to detect when `N > 4`, a test must be included in `SUB`, where the actual dimension of `Z` is set. Here is a version of `SUB` that writes an error message and stops if `N` is too large.

```
      SUBROUTINE SUB(N)
      INTEGER*4 Z(5)/1,2,3,5,8/
      IF(N.GT.4) THEN
         PRINT *,'N=',N,' > 4; error'
         STOP
      ENDIF
      PRINT *,ISMSQ(Z,N)
      RETURN
      END
```

Now when `SUB` is called by the following main program,

```
      CALL SUB(7)
      STOP
      END
```

this output is produced.

```
 N= 7 > 4; error
```

7.4.9 `[P]` (a) This subprogram meets the requirements. Passing the matrix size in `N` allows the code to be used for checking matrices of arbitrary size, and passing the leading dimension of `A` in `LDA` allows `A` to be dimensioned larger than needed in the invoking routine.

```
      FUNCTION SYMMAT(A,LDA,N)
      LOGICAL*4 SYMMAT
      REAL*8 A(LDA,*)
      SYMMAT=.FALSE.
      DO 1 I=1,N
      DO 1 J=1,N
           IF(A(I,J) .NE. A(J,I)) RETURN
    1 CONTINUE
      SYMMAT=.TRUE.
      RETURN
      END
```

(b) If the elements of A result from different floating-point calculations, then roundoff makes it unlikely that $a_{i,j}$ will be *exactly* equal to $a_{j,i}$ for all i and j, even if the matrix would be symmetric in the absence of roundoff. Thus it *does* make sense to introduce an equality tolerance.

The code given in answer to part (a) repeats many comparisons (e.g., testing both `A(1,2).NE.A(2,1)` and `A(2,1).NE.A(1,2)`) and performs others that are unnecessary (e.g., `A(1,1).NE.A(1,1)`). If the loops ran `I=1,N-1` and `J=I+1,N` this superfluous work could be avoided. For example, if `N=4` the comparisons performed would be

```
A(1,2) .NE. A(2,1)
A(1,3) .NE. A(3,1)
A(1,4) .NE. A(4,1)
A(2,3) .NE. A(3,2)
A(2,4) .NE. A(4,2)
A(3,4) .NE. A(4,3)
```

and these are exactly the ones that are necessary to check for symmetry in a 4×4 matrix. Thus, it might also be a good idea to rewrite the loops in this way. When `N=1` the new limits on the `DO` loops will be out of order, and a careless user might accidentally invoke `SYMMAT` with a non-positive value of `N` or with a value of `LDA` that is less than `N`, so the code shown on the next page handles those cases separately.

The main program exercises the function using a matrix that is slightly unsymmetric, but $|a_{1,2} - a_{2,1}| = 10^{-15} < \texttt{TOL} = 10^{-14}$ so the program reports that the matrix is symmetric. If $a_{1,2}$ is changed to $2 + 10^{-13}$, the program no longer reports that A is symmetric.

```
      REAL*8 A(10,10)/100*0.D0/,TOL/1.D-14/
      LOGICAL*4 S,SYMMAT
      LDA=10
      N=2
      A(1,1)=1.D0
      A(1,2)=2.000000000000001D0
      A(2,1)=2.D0
      A(2,2)=4.D0
      S=SYMMAT(A,LDA,N,TOL)
      IF(S) PRINT *,'the matrix is symmetric'
      STOP
      END
C
      FUNCTION SYMMAT(A,LDA,N,TOL)
      LOGICAL*4 SYMMAT
      REAL*8 A(LDA,*),TOL
      IF(LDA.LT.N .OR. N.LT.1) THEN
         SYMMAT=.FALSE.
         RETURN
      ENDIF
      IF(N.EQ.1) THEN
         SYMMAT=.TRUE.
         RETURN
      ENDIF
C     here it must be that N is greater than 1
      SYMMAT=.FALSE.
      DO 1 I=1,N-1
      DO 1 J=I+1,N
           IF(DABS(A(I,J)-A(J,I)).GT.TOL) RETURN
    1 CONTINUE
      SYMMAT=.TRUE.
      RETURN
      END
```

7.4.10 [H] The storage for an array is allocated in the highest-level routine where it is declared, so in this example the compiler obeys the dimensioning of A in the main program and reserves 10 doublewords for its elements. The dimension of 20 elements given in subroutine XYZ plays no role in the calculations used by XYZ to figure out the addresses (back in the main program) at which to store the elements A(I) that are assigned in the loop. As explained in §7.1.1 of the text, a vector that is passed as a subprogram parameter can be dimensioned (*) in the subprogram, because its actual length is not determined there. The largest value of N that can safely be read in by XYZ is 10, the size of A that is specified where it is first dimensioned.

If the length of a vector parameter is given as a number in the subprogram, rather than as *, it *is* used to check for subscript overflows if the -C compiler option is given (see §14.1.2 of the text). In this example, the erroneous dimension of 20 stated in XYZ would mislead the subscript-checking code generated by -C into not reporting an error until N $>$ 20, but by that time the 10 doublewords of storage *following* A would also have been overwritten by the loop.

Array subscript overflows are not always fatal to a program, because the memory locations that are overwritten by exceeding the dimensioned size might be unused or contain parts of the machine code that are no longer needed. It is also possible for the overwriting to change the values of other variables in ways that cause the program to generate incorrect results, rather than to crash. Thus, an array subscript overflow might cause serious trouble that is not noticed by the program's user.

Here is XYZ with the typographical error (in the dimensioning of A) corrected.

```
      SUBROUTINE XYZ(A)
      REAL*8 A(10)
      READ *,N
      IF(N.GT.10) STOP
      DO 1 I=1,N
           A(I)=DFLOAT(I)
    1 CONTINUE
      RETURN
      END
```

This version also adds a test to stop the program if N gets too big.

7.4.11 [H] Here is a version that can be used with matrices of arbitrary size.

```
      SUBROUTINE EVEN(A,LDA, B,LDB,M,N)
      REAL*8 A(LDA,*),B(LDB,*)
      DO 1 I=1,M
      DO 1 J=1,N
           B(I,J)=A(2*I,J)
    1 CONTINUE
      RETURN
      END
```

Most compilers allow expressions for adjustable dimensions, so you could say `A(2*LDB,*)` instead of passing `LDA`. This has the virtue of making it clearer to a human reader that the routine is intended to be called with `A` having twice as many rows as `B`. However, building the assumption into `EVEN` means that the subroutine *won't work* if the arrays don't happen to be dimensioned that way where they are allocated. The safest solution is to pass the leading dimensions separately, as shown.

7.4.15 [H] (a) What gets printed is `A(2,2)=2+3*(2-1)=5`. (b) Here is a version of the program that uses adjustable dimensions in `PRT22`.

```
      INTEGER*4 A(10,10)/100*0/
      N=3
      DO 1 I=1,N
      DO 1 J=1,N
           A(I,J)=J+N*(I-1)
    1 CONTINUE
      CALL PRT22(A,10)
      STOP
      END
```

```
      SUBROUTINE PRT22(A,LDA)
      INTEGER*4 A(LDA,*)
      PRINT *,A(2,2)
      RETURN
      END
```

Running this program also produces the output `5`.

7.4.18 [E] A magic subprogram name is one that is incorporated in the source text of another routine, so that it cannot be changed without recompiling the routine in which it appears. For example, the function name `FCN` is hard-coded into the version of subroutine `BISECT` listed in textbook §6.4, so to use that version of `BISECT` the function whose zero is sought *must* be computed by a function subprogram named `FCN`. This makes it difficult to use that version of `BISECT` for solving two different equations in the same program, or to use that version of `BISECT` along with some other routine that needs to invoke a different function named `FCN`.

The use of a magic subprogram name can be avoided by declaring the subprogram name that might change to be `EXTERNAL` and passing it as a formal parameter to the routine in which it will be invoked, as shown in textbook §7.2.2. When the name of one subprogram is passed to another, it must be declared `EXTERNAL` in both the calling routine and the called routine.

A library routine that invokes a user-supplied subprogram should always make the name of that subprogram `EXTERNAL` and receive the name as a parameter, rather than requiring the user-supplied subprogram to have a particular name.

7.4.19 [E] Recall from textbook §6.2 that FORTRAN passes subprogram parameters **by reference**, so what gets passed for each parameter is its address rather than its value. The address that gets passed for a scalar variable is the location in memory that the compiler has chosen to hold the variable's value. In contrast, the address that must be passed for a parameter that is a subprogram name is the address of the first machine-language instruction in the object code for that subprogram (technically that will be the beginning of the subprogram's entry sequence, but you can think of it as corresponding to the subprogram's first executable statement). Declaring a subprogram name that is a parameter to be `EXTERNAL` is the only thing that allows the compiler to distinguish it from an ordinary scalar variable. Then, instead of allocating space in the calling routine to store a scalar value, the compiler generates a request for the loader, when it links together the executable program, to fill in the address of the external routine as the subprogram parameter value.

7.4.21 [P] (a) Listed on the next page is a routine that meets the stated requirements. If there is no inverse in the interval, or if convergence to the root takes too long, the subprogram writes an error message and stops.

The program listed below uses `FINV` to approximate $x = \sqrt{y}$ for $y = 2$, and prints the result `1.41421348`.

```
      REAL*8 FINV
      EXTERNAL F
      PRINT *,FINV(F,0.D0,10.D0,2.D0)
      STOP
      END
C
      FUNCTION F(X)
      REAL*8 F,X
      F=X*X
      RETURN
      END
```

(b) Using the features of FORTRAN that have been introduced so far, there is no way to accomplish this calculation by calling `BISECT` from within `FINV`. To use `BISECT` it would be necessary to pass it a function, say `G`, that returns `F(X)-Y`. Unfortunately, there is no way to send the `EXTERNAL` name of the function to be inverted from `FINV` into `G`, because it is `BISECT` rather than `FINV` that calls `G`. If we were willing to accept the restriction that the subprogram for the function to be inverted *must* be named `F`, so that a function name would not have to be passed to `G`, there would still be a problem in passing `Y` into `G`. This could be solved by using `COMMON`, which is introduced in textbook §8.

```
C
Code by Michael Kupferschmid
C
      FUNCTION FINV(F,A,B,Y)
C     This routine returns the value of X such that Y=F(X).
C
C     variable  meaning
C     --------  -------
C     A         lower limit on possible X values
C     B         upper limit on possible X values
C     DABS      Fortran function returns |REAL*8|
C     F         function value at trial point
C     F         the function to be inverted
C     FL        function value error at left end
C     FR        function value error at right end
C     I         index on bisections
C     TOL       convergence tolerance on X
C     X         trial point
C     XL        left end of interval on X
C     XR        right end of interval on X
C     Y         value at which to find corresponding X
C
C     formal parameters
      REAL*8 FINV,F,A,B,Y
      EXTERNAL F
C
C     local variables
      REAL*8 X,XL,XR,FX,FL,FR,TOL/1.D-06/
C
C ------------------------------------------------------------------
C
C     use bisection to solve F(X)-Y=0 for X
      XL=A
      XR=B
      FL=F(XL)-Y
      FR=F(XR)-Y
      DO 1 I=1,100
           X=0.5D0*(XL+XR)
           FX=F(X)-Y
           FINV=X
           IF(DABS(XR-XL).LE.TOL) RETURN
           IF(FX*FL .LT. 0.D0) GO TO 2
           IF(FX*FR .LT. 0.D0) GO TO 3
C
C          there is no root in the interval
           WRITE(0,901)
  901      FORMAT('No inverse on interval in FINV')
           STOP
C
C          the root is in the left half
    2      XR=X
           FR=FX
           GO TO 1
C
C          the root is in the right half
    3      XL=X
           FL=FX
    1 CONTINUE
C
C     convergence was not attained in 100 iterations
      WRITE(0,902)
  902 FORMAT('No convergence in 100 iterations in FINV')
      STOP
      END
```

7.4.22 `[H]` (a) What gets printed is `3`. The flow of control is as follows:

```
I=2              in main
CALL SUB(K,I,J)  in main
J=L(I)           in SUB   J=K(2) because L is really K
K=I+1            in K     K=2+1=3
RETURN           to SUB
RETURN           to main
PRINT *,J        in main  J=3
```

(b) If the `EXTERNAL` in the main program is omitted, then what is passed to `SUB` for `K` is the address of a variable rather than of a routine, so the function invocation in `SUB` fails with an `illegal instruction` error upon execution of `J=L(I)`.

(c) If the `EXTERNAL` in `SUB` is omitted, what happens depends on whether your compiler is smart enough to deduce that `K` is a function. If so, the program works just as it did when the `EXTERNAL` statement was included in `SUB`; if not, construction of the executable fails with a loader error about function `L` being missing.

8.8.2 `[E]` As discussed in textbook §8, `COMMON` storage is for transmitting data from a user-written routine that invokes a library routine to a user-written routine that is invoked by the library routine, without having to send the data through the library routine. As discussed in textbook §11, `COMMON` storage can also be used for sharing temporary workspace or fixed data.

Data in `COMMON` get from one routine to another by both routines having access to the memory locations where the data are stored.

8.8.3 `[E]` A `COMMON` block name resembles a `SUBROUTINE` name in that it does *not* have a type, it does *not* carry a value, and it can *not* be dimensioned as an array. A `COMMON` block name must be different from the names of subprograms used in a program because, like a subprogram name, it surives the compilation process and is used by the loader in the same way that a subprogram name is: the virtual memory address that a `COMMON` block has in the executable is assigned by the loader during linkage editing. A `BLOCK DATA` subprogram cannot be named.

8.8.10 [E] As explained in textbook §8.3.1 and §8.3.2, it is part of the definition of the COMMON statement that it causes the variables mentioned in it to be placed adjacent in memory, in the order they are listed, starting on a doubleword boundary associated with the COMMON block name. Those semantics don't permit the compiler to choose the alignment of the variables in memory.

If the meaning of COMMON were *changed* to be more permissive, a compiler might automatically rearrange the variables to minimize the number of inappropriate alignments, or introduce gaps to make all of the alignments correct. Then the placement of the variables in memory would have nothing to do with the order they were listed in the COMMON statement. As long as the compiler did this the same way in translating each COMMON statement and all of the COMMON statements describing a given block were identical, all of the memory references to those variables would be the same and the programmer would never know that they had been stored out of order or non-contiguously in memory.

But what if the COMMON statements referring to a given block in different routines are *not* the same? As illustrated in Exercise 8.8.8, it is sometimes desirable to think of the memory in a COMMON block in different ways in different routines, and because of separate compilation the compiler might rearrange things differently each time. Then the memory references to a given variable might *not* be the same in each routine using the COMMON block. In §11.2 and §11.3 we shall see that some important uses of COMMON *require* that the memory in a COMMON block be thought of and referred to differently in different routines. Thus it would not be possible to change the semantics of COMMON in the way we have conjectured without fundamentally altering the way it is used in classical FORTRAN, and that makes it logically impossible for the compiler to automatically align COMMON variables on appropriate memory boundaries.

As mentioned in §8.3.2, it is desirable for reasons of efficiency that variables in COMMON be aligned on memory boundaries corresponding to their lengths. The fact that this must be done by hand is one of the many reasons that COMMON should be used only in a few special circumstances and then with great care.

8.8.11 [H] The program prints 1, because the value of N gets changed everywhere as soon as the assignment statement in SUBX is executed. FORTRAN subprogram parameters are passed by reference, so the assignment statement in SUBX operates directly on the storage location where N is stored in the main program, namely the first fullword of the COMMON block /NCOM/. Then when SUBX calls SUBY, SUBY finds the value of N in the COMMON block already changed, and prints out the new value. Of course that happens before SUBY has returned to SUBX and before SUBX has returned to the main program.

8.8.12 [H] Making X a dummy parameter means that the actual address of X will be the location in memory of whatever variable is passed as the actual parameter, and might change from one call to the next. For example, in this program NUTSO is invoked twice.

```
      REAL*8 Z/1.D0/,W/2.D0/
      CALL NUTSO(Z)
      CALL NUTSO(W)
      STOP
      END
```

On the first call, NUTSO sees X as Z, the memory address containing 1.D0, but on the second call X is really W, the address containing 2.D0. This consequence of X being a formal parameter is contradicted by placing X in COMMON, because that gives X a *fixed* address in memory (which is, incidentally, neither the address of Z nor the address of W). The code in the problem statement, which is reproduced on the left below, therefore makes no sense.

```
      SUBROUTINE NUTSO(X)                     SUBROUTINE NUTSO(X)
      REAL*8 X                                REAL*8 X,XX
C     this is illegal                   C     this is legal
      COMMON /COMX/ X                         COMMON /COMX/ XX
      :                                       XX=X
                                              :
```

The revised code on the right receives X as a formal parameter and copies its value into the COMMON block. Now executing the main program shown above will result in NUTSO giving the variable XX in COMMON the value 1.D0 at the beginning of the first call and changing its value to 2.D0 at the beginning of the second call.

8.8.13 [E] The addresses in memory of A, B, C, and D are determined by their positions in COMMON and are therefore all *different*. Naming any two of them in the same EQUIVALENCE would tell the compiler they occupy the *same* location in memory. Thus only the third alternative suggested,

```
      COMMON /AC/ A,C
      COMMON /BD/ B,D
      EQUIVALENCE(A,X)
      :
```

makes sense. This causes the name X to be treated by the compiler as a synonym for A, both referring to the fullword that comes first in the COMMON block /AC/.

8.8.14 [E] In the example shown, COMMON /ONE/ I asserts that the variable I is stored in the first word of the COMMON block /ONE/, which has a particular fixed address in memory. The statement COMMON /TWO/ I then asserts that I is stored in the first word of the COMMON block /TWO/, which has a *different* fixed address in memory. This makes no sense, because the word named by I cannot be in two different memory locations at the same time. A FORTRAN variable name refers to a *single* location in memory, the unique place where a scalar or the first element of an array is stored.

8.8.16 `[E]` (a) A `PARAMETER` constant is a symbol that the compiler replaces by the literal for its value everywhere it appears in the source code, before the source code is translated into machine language. Thus the given code segment is equivalent to this statement.

```
COMMON /CONST/ 746.0
```

If this were legal some other routine in the same program might similarly give an initial value to the first fullword of `/CONST/`, and the two values might differ; in that case it would be ambiguous which should be used. Further, some other routine might at run time *change* the value of a variable it had placed in `/CONST/`, which would make the statement above no longer true! Even if it presented no logical difficulties this construction would be impossible on a practical level, because separate compilation requires that the assignment of `COMMON` storage be performed by the loader, when the executable is linked. Thus only the loader can initialize `COMMON` storage, and `PARAMETER` constants can't be in `COMMON`.

(b) Variables in `COMMON` can be initialized at compile time only by using a `BLOCK DATA` subprogram, like this one.

```
BLOCK DATA
COMMON /CONST/ WPHP
REAL*4 WPHP/746.0
END
```

8.8.17 [H] When the given program is compiled with `gfortran` no error message results. However, when the executable is run the output written is different from what was apparently intended:

```
unix[1] gfortran junk.f
unix[2] a.out
          0
unix[3]
```

From the given source code it appears that `I` and `J` will occupy the same location in memory, so assigning `I=1` should set `J=1` as well, but that is not the effect produced. The program listed below elicits a warning (which is not very helpful) from the compiler. An executable is produced, but running it produces the same output as before.

```
      COMMON /EQIV/ K
      K=2
      CALL SUB
      STOP
      END
C
      SUBROUTINE SUB
      COMMON /EQIV/ I
      COMMON /EQIV/ J
      I=1
      PRINT *,J
      RETURN
      END
```

```
unix[4] gfortran junk.f
junk.f:8.19:

      COMMON /EQIV/ I
                  1
Warning: Named COMMON block 'eqiv' at (1) shall be of the same size
unix[5] a.out
          0
unix[6]
```

When the `COMMON` declaration involving `I` is removed from the program listed above, the remaining code works as expected.

```
unix[11] gfortran junk.f
unix[u] a.out
          2
unix[4]
```

Another compiler might behave differently. Trying to `EQUIVALENCE` variables in the way suggested by this Exercise is obviously not a good idea.

8.8.20 [P] The solution to Exercise 6.8.20 is reproduced here for convenient reference.

```
      INTEGER*4 ISEED/123457/
      DO 1 K=1,100
           CALL MYRAND(ISEED)
           PRINT *,ISEED
    1 CONTINUE
      STOP
      END
```

```
      SUBROUTINE MYRAND(ISEED)
      INTEGER*4 A/843314861/,B/453816693/
      INTEGER*4 PLUS/Z'7FFFFFFF'/
      ISEED=A*ISEED+B
      ISEED=IAND(ISEED,PLUS)
      RETURN
      END
```

In this code MYRAND applies the mixed congruential algorithm to the input value ISEED, generating an output value that is then returned in ISEED. It has no internal memory of the number it returned on its previous invocation, so that information must be carried in the *caller's* value of ISEED. This makes MYRAND simple, but as explained in the statement of this exercise it might be inconvenient to pass ISEED between various routines that invoke MYRAND.

(a) This exercise suggests changing MYRAND to remember its previous result and use that in the algorithm, so that the input value of ISEED is irrelevant. In the code provided, N is initialized at compile time inside MYRAND and used in computing the new value of ISEED. This new value is then stored back into N for use on the next invocation of MYRAND. This allows MYRAND to be called from various routines of a program without the need for those routines to share the value of ISEED with each other. This is advantageous because it reduces the complexity of the program by eliminating that requirement.

(b) To set the initial value of N from within a main program, so that we can start from any number (rather than always starting at 123457) we can introduce a COMMON block as shown in the modified code below.

```
      COMMON /SETRNS/ N
      N=123459
      CALL SUBR1
      :

      SUBROUTINE SUBR1
      :
      CALL MYRAND(IRN)
      :
      use IRN
      :
```

```
      SUBROUTINE MYRAND(ISEED)
      INTEGER*4 A/843314861/,B/453816693/
      INTEGER*4 PLUS/Z'7FFFFFFF'/
      COMMON /SETRNS/ N
      ISEED=A*N+B
      ISEED=IAND(ISEED,PLUS)
      N=ISEED
      RETURN
      END
```

The COMMON block name /SETRNS/ is meant to suggest SETting the Random Number Seed. Now the main program sets the starting seed to N=123459 and then invokes SUBR1 (and possibly other routines, which in turn invoke other routines, and so on). In SUBR1, MYRAND is invoked to get a pseudorandom value in IRN, which is then used somehow. Other routines in the call tree rooted at the main program might also invoke MYRAND, without knowing about each other. Each call to MYRAND results in N getting a new value (the result most recently returned by MYRAND). The values returned by MYRAND are pseudorandom numbers in the sequence starting with 123459.

8.8.22 [P] The function FINV listed below meets the requirements of the problem. It calls the version of BISECT given last in textbook §7.2, which is reproduced on the right except that I have here increased the iteration limit from 10 to 100.

```
      REAL*8 A,B,Y,X,FINV
      A=0.D0
      B=5.D0
      Y=2.D0
      X=FINV(A,B,Y)
      PRINT *,'X=',X
      STOP
      END

      FUNCTION F(X)
      REAL*8 F,X
      F=X**2
      RETURN
      END

      FUNCTION FINV(A,B,Y)
      REAL*8 FINV,A,B,Y,YY
      COMMON /WYE/ YY
      EXTERNAL G
      REAL*8 XL,XR,X,F
      INTEGER*4 RC
      YY=Y
      XL=A
      XR=B
      CALL BISECT(G,XL,XR, X,F)
      FINV=X
      RETURN
      END
```

```
      SUBROUTINE BISECT(FCN,XL,XR, X,F)
      EXTERNAL FCN
      REAL*8 XL,XR,X,FCN,FL,FR,F
      FL=FCN(XL)
      FR=FCN(XR)
      DO 1 I=1,100
           X=0.5D0*(XL+XR)
           F=FCN(X)
           IF(F*FL .LT.0.D0) THEN
              XR=X
              FR=F
           ELSE
              XL=X
              FL=F
           ENDIF
    1 CONTINUE
      RETURN
      END

      FUNCTION G(X)
      COMMON /WYE/ Y
      REAL*8 G,X,Y,F
      G=F(X)-Y
      RETURN
      END
```

The main program sets the search interval $x \in [0, 5]$ and invokes FINV to find $x = f^{-1}(2)$. The subprogram F defines the function whose inverse is to be found, $f(x) = x^2$. Thus the calculation should produce a result approximating $x = \sqrt{2}$.

FINV begins by copying the value of Y into YY, which is shared via COMMON block /WYE/ with subprogram G (where it is again called Y. Now G can invoke the function whose inverse is to be found, which we have agreed to always call F, and compute $g(x) = f(x) - y$.

Next FINV copies the search interval ends into XL and XR, and invokes BISECT to solve $g(x) = 0$. BISECT repeatedly invokes G (which BISECT names FCN) and returns the X where G(X) is close to zero. At that value of X, F(X) will be close to Y, so X is returned as FINV. This process can fail if $f(x)$ does not have a unique inverse on the interval specified, but for $f(x) = x^2 = 2$ and $x \in [0, 5]$ the algorithm is successful as shown below.

```
unix[1] a.out
 X=   1.4142135623730949
unix[2]
```

8.8.23 [P] Here is a version of the program that works as required.

```
      REAL*8 X(3)
      EXTERNAL Z
C
C     share the observations with the error function
      COMMON /DATA/ Y
      REAL*8 Y(100)
C
C     read the observations once
      READ *, Y
C
C     find parameter values that fit the model to the data
      CALL OPT(Z,X)
C
C     report the optimal parameter values
      PRINT *,X
      STOP
      END
```

Now the main program reads 100 observations from the keyboard once and stores them in the `COMMON` block.

```
      FUNCTION Z(X)
      REAL*8 Z,X(3),YHAT(100)
C
C     receive the observations from the main program
      COMMON /DATA/ Y
      REAL*8 Y(100)
C
C     use the model to predict what the observations should be
      CALL MODEL(X,YHAT)
C
C     compute the error between the predictions and measurements
      Z=0.D0
      DO 1 I=1,100
           Z=Z+(YHAT(I)-Y(I))**2
    1 CONTINUE
      RETURN
      END
```

Then the `FUNCTION` subprogram uses the observations in the `COMMON` block to compute `Z`.

This is the typical problem described in textbook §8.2, where a user-written main program calls a library subroutine (here `OPT`) that invokes a user-written function (here `Z`) that needs some data (here `Y`) that is known in the main program. The use of `COMMON` illustrated here is the typical solution adopted.

This problem could instead be solved by modifying only the function `Z`, to make it read `Y` only on its first call (see textbook §6.5). That avoids the need to introduce a `COMMON` block, but it complicates `Z` with code to detect its first call and it requires that the compiler treat `Y` as static, rather than as an automatic array (see textbook §17.1.2).

The parameter estimation problem that is the motivation for this Exercise is discussed in more detail in §8.5 of my book *Introduction to Mathematical Programming,* which can be downloaded from the same website where you found this document.

9.10.1 `[E]` *Free-format input* is performed by the `READ *` and `READ(unit,*)` statements. They invoke library routines that can figure out numerical values expressed in a wide variety of ways. The precise behavior of FORTRAN I/O library routines is implementation-dependent, but in almost all implementations leading and trailing blanks are ignored, values can be separated by blanks or commas or both, whole-number real values need not be given with a decimal point, and a real value can expressed in fraction-exponent form in any way that makes mathematical sense. For example, if a `READ *` or `READ(5,*)` statement reads the real values 12.0 and 7.5 from the keyboard, the user could enter any of the following strings.

```
12   7.5000
  +12.,7.5
 12.0,   0.75e+1
1.2e1  75e-1
1.20e+01  7.5e+00
     0.12E+2            7.5
```

It is usually a good idea to use free format for inputs, so that they don't have to be entered in a certain way.

Free-format output is performed by the `PRINT *` and `WRITE(unit,*)` statements. They invoke library routines that decide for you how to format the output, based on the types and values of the variables. The decisions they make are highly implementation-dependent, so you can't count on free-format outputs being the same from one system to another. It is often preferable to format your own outputs so that you can predict what they will look like.

Unformatted I/O is performed by the `WRITE(unit)` and `READ(unit)` statements. They invoke library routines that simply copy bytes to or from the storage locations in memory named by the variables, from or to a device such as a disk file. Unformatted I/O is always much faster than formatted I/O, and often the data occupies less space on the storage device. However, the data are bytes representing the numerical values, as discussed in §4, rather than numerals, so they are unreadable to the human eye.

Formatted I/O is performed by the `WRITE(unit,fmt)` and `READ(unit,fmt)` statements, where `fmt` is the number of a `FORMAT` statement specifying the typographical appearance of the data.

9.10.2 `[E]` (a) Most FORTRAN I/O libraries give `READ(*,*)` and `WRITE(*,*)` the same behavior as `READ *` and `PRINT *`, respectively. Unless your compiler does not, the only difference between these statements is that the ones using `(*,*)` are slightly more complicated, and correspond to the forms of `READ` and `WRITE` that include unit or `FORMAT` statement numbers. (b) I introduced the simpler forms for two reasons: first, the `PRINT *` and `READ *` statements are widely used, so if you are going to be comfortable reading old code you should learn about them; and second, the use of the complicated forms might have distracted you from the other topics that were the focus of the first 8 chapters by begging the question why there are *two* asterisks.

9.10.4 [E] No, a FORMAT statement cannot be used as a branch target. A FORMAT statement is not executable, so control cannot be transferred to it. In this book FORMAT statements usually appear immediately after their first use (as a hint to what the I/O statement is doing) but they can actually be located anywhere in the source text of the routine where they are used. In translating your FORTRAN source program into machine language, the compiler arranges for the system routine that actually performs each I/O operation to receive the text of the corresponding FORMAT statement so that it can know how to interpret input records or format output records. Thus the FORMAT statement itself is never executed.

9.10.7 [E] The pattern .␣D±␣␣ corresponds to the D format field specification D6.1 There are 6 print positions altogether, and 1 digit after the decimal point. No print position has been provided for the sign of the number, so this format is suitable only for values in the closed interval [0.D0, 1.7976931348623157D+308], that is between zero and the largest representable REAL*8 value (see text §4.7).

9.10.10 [E] Here is a code segment that reads the four values from the file described.

```
      :
      OPEN(UNIT=3,FILE='data')
      READ(3,*)
      READ(3,*)
      READ(3,901) I,J,K,L
  901 FORMAT(I5,I6,I7,I8)
      :
```

In this code segment, a unit number other than 3 could have been chosen. If the file is assumed to be redirected to standard-in and the READ statements use unit 5 rather than unit 3 as shown, then the OPEN can be omitted. The first two READ statements in the code segment just skip over the first two lines in the file, so that I, J, K, and L are read from the third line as required by the problem statement. Notice that L will be read as 8219765 even though the 8-character field containing that value in the given data includes a leading zero.

9.10.11 `[H]` (a) The `FORMAT` statement prints the character string

```
0.3141592653589793238462643D+01
```

skips to the next line, and then prints the value of `PI` using fraction-exponent form in 31 characters with 25 digits following the decimal point. This is the same pattern of characters as printed on the first line, which makes it easy to compare the value that was actually stored to the one on the right side of the assignment statement.

(b) Running the program produces this output.

```
unix[1] a.out
0.3141592653589793238462643D+01
0.3141592653589793115997963E+01
unix[2]
```

(c) Only the first 16 significant digits were stored correctly and printed back out. This happens because of the limited precision with which real numbers can be represented in a computer. As explained in §4.2, `REAL*8` values are stored using enough fraction bits to preserve 15, 16, or 17 decimal digits of precision, depending on the value represented; in this case the machine representation is precise to 16 digits. It never makes sense to print a `REAL*8` quantity using a format that provides for more than 17 significant digits, because although more digits will print they are meaningless.

9.10.15 `[H]` The left-hand expression can be expanded into separate sums, like this.

$$\sum_{i=1}^{N}(x_i^2 - 2\mu x_i + \mu^2) = \sum_{i=1}^{N} x_i^2 - 2\mu\sum_{i=1}^{N} x_i + \mu^2\sum_{i=1}^{N} 1$$

But

$$\mu = \frac{1}{N}\sum_{i=1}^{N} x_i$$

so

$$N\mu = \sum_{i=1}^{N} x_i$$

Then we have

$$\begin{aligned}\sum_{i=1}^{N}(x_i^2 - 2\mu x_i + \mu^2) &= \sum_{i=1}^{N} x_i^2 - 2\mu N\mu + N\mu^2 \\ &= \sum_{i=1}^{N} x_i^2 - N\mu^2\end{aligned}$$

as was to be proved. □

9.10.16 [H] Standard-out and standard-error are normally connected to the display, but both can be redirected. If error messages must be written to the display even when standard-out and standard-error are redirected, the program must attach some other unit to the display. Textbook §9.4.1 illustrates the use of the UNIX™ device name /dev/tty for input from the keyboard, and a little experimentation reveals that this device can also be used for output to the screen.

The program of the case study in textbook §9.5 is revised below to attach unit 1 to /dev/tty and write its error messages there.

```
C     improved descriptive statistics program
      REAL*8 X,SUM,SSQ,ANS(2)
C
C     make sure error messages go to the screen
      OPEN(UNIT=1,FILE='/dev/tty')
C
C     collect the data
      SUM=0.D0
      SSQ=0.D0
      N=0
      LNUM=0
    3 WRITE(0,901)
  901 FORMAT('? ',$)
        LNUM=LNUM+1
        READ(5,*,END=1,ERR=2) X
        N=N+1
        SUM=SUM+X
        SSQ=SSQ+X**2
      GO TO 3
    2 WRITE(1,902) LNUM
  902 FORMAT('bad data at record',I11)
      GO TO 3
C
C     compute and report the statistics
    1 IF(N.EQ.0) THEN
        WRITE(1,903)
  903   FORMAT('no input data')
        STOP
      ENDIF
      IF(N.EQ.1) THEN
        ANS(1)=SUM
        ANS(2)=0.D0
      ELSE
        ANS(1)=SUM/DFLOAT(N)
        ANS(2)=DSQRT((SSQ-DFLOAT(N)*(ANS(1))**2)/DFLOAT(N-1))
      ENDIF
      WRITE(6,904) N,(ANS(J),J=1,2)
  904 FORMAT(/'samples=',I11
     ;       /'mean=',D11.4
     ;       /'standard deviation=',D10.3)
      STOP
      END
```

When the program is compiled and run it yields the conversations shown on the next page. The first time I ran the program I entered 5 numbers and then a ˆD, and the program printed the output shown. The second time I ran the program I redirected the output (unit 6) to the file junk and the prompts (unit 0) to /dev/null. I entered the same 5 numbers

and then a ^D as before, but now the prompts got sent down the drain and the output went to the file. To prove that the output went to the file I used the UNIX™ `more` command to copy the contents of `junk` to the screen. Finally, I ran the program and presented it with some bad data to get the first error message on the screen. When I entered a ^D in response, ending the input process without having successfully entered any values, the second error message appeared.

```
unix[1] a.out
? 1.3
? 2.7
? -14
? 6.95
? 12
?
samples=          5
mean= 0.1790D+01
standard deviation= 0.976D+01
unix[2] a.out > junk 2> /dev/null
1.3
2.7
-14
6.95
12
unix[3] more junk

samples=          5
mean= 0.1790D+01
standard deviation= 0.976D+01
unix[4] a.out > junk 2> /dev/null
xyz
bad data at record          1
no input data
unix[5]
```

It is hard to anticipate every circumstance in which a program might be run, so it is seldom a good idea to insist on using either the screen or the keyboard rather than allowing them to be redirected. Can you think of a situation in which this version of the statistics program would be hard to use because it insists on writing its error messages to the screen?

9.10.17 [H] Here is the program of textbook §9.5 revised to simulate an adding machine.

```
C     adding machine simulator
      REAL*8 X,SUM,SSQ,ANS(2)
C
C     collect the data
    4 SUM=0.D0
      N=0
    3 WRITE(0,901)
  901 FORMAT('+ ',$)
        READ(5,*,END=1,ERR=2) X
        N=N+1
        SUM=SUM+X
      GO TO 3
    2 WRITE(6,902)
  902 FORMAT('bad data ignored')
      GO TO 3
C
C     report the sum
    1 IF(N.EQ.0) THEN
        WRITE(6,903)
  903   FORMAT('no input data')
        STOP
      ELSE
        WRITE(6,904) SUM
  904   FORMAT('= ',F9.2)
        GO TO 4
      ENDIF
      END
```

The first stanza of executable code prompts for input like the statistics program did, but now it prints a `+` to indicate that the numbers entered will be added. It handles input errors similarly to the statistics program but accumulates only the sum of the entries. On EOF the first `READ` branches to `1`, and if there are no input values the program stops; otherwise it writes the sum and goes back to `4` so the user can enter another set of values. When the program is compiled and run it produces output like this.

```
unix[1] a.out
+ 1.23
+ 4.56
+ -7.89
+ 10.11
+ =      8.01
+ 2.75
+ 1.13
+ =      3.88
+ no input data
unix[2]
```

In response to the prompt following the `10.11` I entered ˆD, which caused the program to finish the line with "`=      8.01`" and write the next prompt. After the `1.13` I entered ˆD, which caused the program to finish that line with "`=      3.88`" and write the next prompt. Then I entered ˆD again, causing the program to finish the line with `no input data` and stop.

9.10.19 [P] Here is one possible SUBROUTINE that prints an array as described, and a main program to invoke it with a test array. (Many other solutions are possible.)

```
      INTEGER*4 A(3,5)
      DO 1 J=1,5
      DO 1 I=1,3
           A(I,J)=J+5*(I-1)
    1 CONTINUE
      CALL APRT(A,3,3,5)
      STOP
      END
```

```
      SUBROUTINE APRT(A,LDA,M,N)
      INTEGER*4 A(LDA,*)
      DO 1 I=1,M
         IF(MOD(N,3).EQ.0) THEN
            WRITE(6,903) (I,J,A(I,J),J=1,N)
903         FORMAT(3(’ A(’,I1,’,’,I1,’)=’,I2))
         ENDIF
         IF(MOD(N,3).EQ.1) THEN
            IF(N.GT.3) WRITE(6,903) (I,J,A(I,J),J=1,N-1)
            WRITE(6,901) I,N,A(I,N)
901         FORMAT(1(’ A(’,I1,’,’,I1,’)=’,I2))
         ENDIF
         IF(MOD(N,3).EQ.2) THEN
            IF(N.GT.3) WRITE(6,903) (I,J,A(I,J),J=1,N-2)
            WRITE(6,902) (I,J,A(I,J),J=N-1,N)
902         FORMAT(2(’ A(’,I1,’,’,I1,’)=’,I2))
         ENDIF
    1 CONTINUE
      RETURN
      END
```

This main program declares A to be 3 $\times$ 5, but any size could be used because the leading dimension of A and the numbers of rows M and columns N to be printed are passed as parameters. If each printed line is to contain no more than three values, then all the lines might be full, or the last line might contain only two values, or the last line might contain only one value. These three cases are distinguished by testing whether the remainder from dividing N by 3 is 0, 1, or 2, so that appropriate implied-DO and FORMAT statements can be used to write the line. The code above produces the following output.

```
 A(1,1)= 1 A(1,2)= 2 A(1,3)= 3
 A(1,4)= 4 A(1,5)= 5
 A(2,1)= 6 A(2,2)= 7 A(2,3)= 8
 A(2,4)= 9 A(2,5)=10
 A(3,1)=11 A(3,2)=12 A(3,3)=13
 A(3,4)=14 A(3,5)=15
```

In this solution to the problem, each output item is labeled with its array element; other labeling schemes might avoid the need to distinguish the three types of last line or result in other simplifications, at the expense of making it slightly more difficult for a reader to tell which elements are printed where. The formats used for the element indices and values were chosen to suit the test array used here, but a general-purpose routine for printing integer arrays would need to use I11 fields to accommodate positive and negative integers up to the maximum possible value, or object-time formats as described in §10. APRT labels each output item as an element of an array named A, but a general-purpose routine should receive the name as a parameter, as also described in §10.

9.10.20 `[P]` The program listed below meets the requirements of the problem statement.

```
C     This program reads a time in hh:mm format and an increment
C     in minutes, and prints the resulting time in hh:mm format.
C
C     variable  meaning
C     --------  -------
C     H         hours part of a time
C     INC       increment in minutes
C     M         minutes part of a time
C     T         time in minutes
C
      INTEGER*4 H,M,T
C
C -----------------------------------------------------------------
C
C     convert the starting time to minutes
      WRITE(6,901)
  901 FORMAT('starting time: ',$)
      READ(5,902) H,M
  902 FORMAT(I2,1X,I2)
      T=60*H+M
C
C     add the increment to get the ending time
      WRITE(6,903)
  903 FORMAT('increment: ',$)
      READ(5,*) INC
      T=T+INC
C
C     convert the ending time back to hours and minutes
      H=T/60
      M=T-60*H
    1 IF(H.GE.24) THEN
         H=H-24
         GO TO 1
      ENDIF
    2 IF(H.LT. 0) THEN
         H=H+24
         GO TO 2
      ENDIF
      IF(M.LT. 0) THEN
         H=H-1
         M=M+60
         GO TO 2
      ENDIF
C
C     print the resulting time
      IF(H.GE.10 .AND. M.GE.10) WRITE(6,904) H,M
  904 FORMAT('ending time: ',I2,':',I2)
      IF(H.GE.10 .AND. M.LT.10) WRITE(6,905) H,M
  905 FORMAT('ending time: ',I2,':0',I1)
      IF(H.LT.10 .AND. M.GE.10) WRITE(6,906) H,M
  906 FORMAT('ending time: 0',I1,':',I2)
      IF(H.LT.10 .AND. M.LT.10) WRITE(6,907) H,M
  907 FORMAT('ending time: 0',I1,':0',I1)
      STOP
      END
```

The prompts for `starting time` and `increment` use the `FORMAT` code `$` to make the cursor hang. The starting time is read with a fixed format because it is expected, based on the

problem statement, to consist of a 2 digit hours value, a colon, and a 2 digit minutes value. The increment is read with free format because it might be any number of minutes. The resulting time is written as a 2-digit hour value, a colon, and a 2-digit minutes value. Four different `FORMAT` statements are required to get leading zeros (such as in the example given with the problem statement). In §13.9.4 we shall see that an `I2.2` field specifier can be used to achieve this effect much more easily.

The conversion of the final minutes value in `T` back to hours and minutes is tricky because `H` might be greater than 24 or negative, or `M` might be negative. If any of those things happen the out-of-range values are adjusted by the free loops until they are in-range.

When the program is compiled and run it produces output like that shown below.

```
unix[1] a.out
starting time: 17:45
increment: 20
ending time: 18:05
unix[2] a.out
starting time: 21:30
increment: 480
ending time: 05:30
unix[3] a.out
starting time: 05:30
increment: -480
ending time: 21:30
unix[4]
```

9.10.24 [P] Here is a program that does what is asked by the exercise. So that the input files can be named at run time, it uses the GETFIL routine described in §9.4.2 of the text. If one file is longer than the other, the program compares the files down to the end of the shorter file and reports that the files are of unequal length.

```
C
C     This program compares two files of real numbers.
C
C     variable  meaning
C     --------  -------
C     A         a number read from file A
C     B         a number read from file B
C     DABS      function returns |REAL*8|
C     ERROR     absolute difference between A and B
C     GETFIL    routine attaches a file for a given purpose
C     LNUM      number of the line(s) just read in
C     TOL       maximum allowed difference between the numbers
C
      REAL*8 A,B,ERROR,TOL
C
C -----------------------------------------------------------------
C
C     attach the files
      CALL GETFIL('file A',6,' ',0,1,1)
      CALL GETFIL('file B',6,' ',0,2,1)
      LNUM=0
      OPEN(UNIT=3,FILE='errors')
C
C     read the tolerance from the user
      WRITE(6,900)
  900 FORMAT('tolerance: ',$)
      READ(5,*,END=1) TOL
C
C     read the two numbers that correspond
    4 READ(1,*,END=2) A
      READ(2,*,END=3) B
      LNUM=LNUM+1
C
C     write out their difference if it exceeds the tolerance
      ERROR=DABS(A-B)
      IF(ERROR.GT.TOL) THEN
         WRITE(3,901) LNUM,ERROR
  901    FORMAT(I10,1X,1PE13.6)
      ENDIF
      GO TO 4
C
C     end of file A was found first; now also at the end of file B?
    2 READ(2,*,END=1)
C
C     end of B first (=>A longer) or end of A but not B (=>B longer)
    3 WRITE(0,902)
  902 FORMAT('the files are of unequal lengths')
C
    1 STOP
      END
```

9.10.25 [H] The output produced by the program is shown below.

	r	e	s	u	l	t	:							
K	(		2	)	=					7				
K	(		3	)	=					9				

9.10.26 [H] The EXTERNAL in the main program causes the entry address of SUB to be passed to IADROF, which copies it into its return value. Thus the main program prints the entry address of the subroutine SUB. The address might have a 1 in its most significant (sign) bit, which would make it appear to be a negative integer, so a hexadecimal output format is used rather than an integer format. The entry address of SUB might be of some use in a program written in a lower-level language, but in FORTRAN it is just a curiosity. The exercise is meant only to reinforce the idea that EXTERNAL causes the address of one subprogram to be passed as a parameter to another subprogram. The given code uses the function IADROF, rather than just printing SUB from the main, because many compilers will not allow a symbol that has been declared EXTERNAL to be printed out or used in an assignment statement. Separate compilation prevents the compiler from knowing that what appears to be a variable in IADROF is actually an external symbol.

10.9.1 [E] (a) The scalar or CHARACTER*n form is required for the following uses.

an internal file that is read from or written to
the filename in an INQUIRE statement (see textbook §14.2.2) or in an OPEN statement
the name of a FUNCTION subprogram returning a character value
string parameters passed to or received from UNIX™ system subprograms.

(b) In Classical FORTRAN it is necessary to use a CHARACTER*1 vector when the individual characters of a string must be manipulated separately.

(c) The same string can be used in both ways in a single routine by using EQUIVALENCE to overlay a CHARACTER*n scalar with a CHARACTER*1 vector of n elements.

10.9.2 [E] There are 4 bits in a nybble, and 8 characters can be stored in a doubleword.

According to the ASCII character code, the hex constant Z'0C204F6E650A54776F07' represents a page eject followed by a space, then the word One, then a carriage return, then the word Two, and finally the bell character. I wrote the following program to confirm this interpretation, and while it is far from obvious it does illustrate some complications that sometimes arise.

```
      INTEGER*4 CONST(3)/Z'6E4F200C',Z'77540A65',Z'0000076F'/
      CHARACTER*1 CHARS(10)
      EQUIVALENCE(CONST,CHARS)
      WRITE(6,901) CHARS
  901 FORMAT(10A1)
      STOP
      END
```

Here the given hex constant is used to initialize the first 10 of the 12 bytes in the INTEGER*4 vector CONST, the final two bytes of which I set to the null character Z'00'. Unlike the now-obsolete g77, the gfortran compiler I used does not allow CHARACTER variables to be initialized using hex constants (it is hard to think of this as an improvement). Because my processor is little-endian (see textbook §4.8) the bytes within each word of CONST must be given in reverse order. Then I used EQUIVALENCE to overlay the first 10 bytes of CONST with the CHARACTER*1 vector CHARS, and printed that in the usual way. The program produced the following output, along with an audible *beep* from by the trailing bell character.

```
unix[1]

 One
Two
unix[2]
```

From this output it appears the gfortran I/O library prints a blank line for the page eject escape character.

10.9.5 [H] (a) One way to confirm that 09 is the byte code for a tab is by writing and running a FORTRAN program that prints that byte to a file, and then examining the file with an editor to verify that a tab is present. Another way is by using a text editor to insert tabs in a file, then reading the text into a program and confirming that the corresponding bytes have the hexadecimal value 09.

(b) The next page lists a program that reads lines from standard-in, translates each tab character into 8 blanks, and writes the transformed lines on standard-out. The heart of the program is the stanza titled `translate tabs to spaces`. That stanza consists of a free loop having counter `I`, which steps through the characters of `LINE` in search of the next tab. If a tab is found and its expansion into 8 blanks will fit in `LINE`, the `DO 3` loop copies characters forward to make room for the blanks and then the `DO 4` loop fills in the blanks. If the resulting line would be too long, control is transferred to statement 2 where an error message is written.

```
C
C     This program translates tabs to blanks for Exercise 10.9.5.
C
C     variable  meaning
C     --------  -------
C     I         first index on the characters of LINE
C     K         second index on the characters of LINE
C     L         length of LINE
C     LENGTH    function returns index of last nonblank in a string
C     LINE      text line read and written
C     TAB       byte code for a tab
C
      CHARACTER*1 LINE(100),TAB/Z'09'/
C
C ------------------------------------------------------------------
C
C     read a line
    6 READ(5,900,END=1) LINE
  900 FORMAT(100A1)
        L=LENGTH(LINE,100)
C
C       translate tabs to spaces
        I=1
    5   IF(LINE(I).EQ.TAB) THEN
           IF(L.GT.100-7) GO TO 2
           DO 3 K=L,I+1,-1
                LINE(K+7)=LINE(K)
    3      CONTINUE
           DO 4 K=I,I+7
                LINE(K)=' '
    4      CONTINUE
           I=I+7
           L=L+7
        ELSE
           I=I+1
        ENDIF
        IF(I.LE.L) GO TO 5
C
C       write the line
        WRITE(6,900) (LINE(K),K=1,L)
      GO TO 6
C
C     reached the end of the input file
    1 STOP
C
C     report an error
    2 WRITE(0,901)
  901 FORMAT('result line too long')
      STOP
      END
```

10.9.10 `[E]` When the given program is compiled and run, it produces the following exchange with the interactive user.

```
unix[1] a.out
test
test
unix[2]
```

The input from the keyboard consists of only 4 characters (I did not type any blanks after the word `test`) but when the value of `LINE` is read the characters after those I input got blanked out. It is easier to see what is happening if we change the program like this.

```
      CHARACTER*1 LINE(80)/80*'x'/
      WRITE(6,901) LINE
  901 FORMAT(80A1,'|')
      READ(5,902) LINE
  902 FORMAT(80A1)
      WRITE(6,901) LINE
      STOP
      END
```

When this program is compiled and run it produces the following exchange.

```
unix[4] a.out
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx|
test
test                                                                            |
unix[5]
```

Notice that the final write of `LINE` outputs 76 blanks after the text I input. Often it is undesirable to write lines with trailing blanks, and that can be avoided like this.

```
      CHARACTER*1 LINE(80)/80*'x'/
      READ(5,901) LINE
  901 FORMAT(81A1)
      L=LENGTH(LINE,80)
      WRITE(6,901) (LINE(K),K=1,L),'|'
      STOP
      END
```

Now we get this exchange

```
unix[7] a.out
test
test|
unix[8]
```

10.9.12 `[H]` (a) The `LENGTH` function of textbook §10.2 is reproduced on the left below.

```
      FUNCTION LENGTH(LINE,L)
      CHARACTER*1 LINE(L)
      DO 1 K=L,1,-1
           LENGTH=K
           IF(LINE(K).NE.' ') RETURN
    1 CONTINUE
      LENGTH=0
      RETURN
      END
```

```
      FUNCTION LENGTH(LINE,L)
      CHARACTER*1 LINE(*)
      DO 1 K=L,1,-1
           LENGTH=K
           IF(LINE(K).NE.' ') RETURN
    1 CONTINUE
      LENGTH=0
      RETURN
      END
```

In that version of the routine the fact that `LINE` is given one dimension tells the compiler that it is a vector, but the *value* of `L` does not matter unless subscript checking is on or some attempt is made to address an element of `LINE`. If `L` is negative or zero and the code is compiled using a FORTRAN-77 compiler, the loop will be skipped so no elements of `LINE` will be addressed. However, it is misleading to claim that `LINE` contains `L` elements if there is a possibility that `L` will be nonpositive. Also, a compiler that is sufficiently aggressive in checking array subscripts (`gfortran` is not) might complain if the adjustable dimension is nonpositive, even if no attempt is made to address the array.

The version of `LENGTH` shown on the right above avoids this quandary by using an assumed-size dimension. Now it is clear, to a human reader and to the compiler, that the dimensioned size of `LINE` in the calling routine might *not* be `L`. Of course this also prevents the compiler from inserting subscript-checking code into the executable, even if we ask it to do so. A better approach if `L` might be nonpositive would be to add a leading-dimension parameter separate from `L`, at the cost of a more complicated calling sequence.

(b) If the code is compiled following the FORTRAN-66 standard, one iteration of the `DO` loop will be performed even if `L` is nonpositive. An attempt will then be made to address element `L` of `LENGTH` even though that is outside the dimensioned bounds of the array, possibly causing a segmentation fault or a normal return with the wrong value of `LENGTH`. To prevent that we can modify the code as shown below.

```
      FUNCTION LENGTH(LINE,L)
      CHARACTER*1 LINE(*)
      IF(L.GT.0) THEN
         DO 1 K=L,1,-1
              LENGTH=K
              IF(LINE(K).NE.' ') RETURN
    1    CONTINUE
      ENDIF
      LENGTH=0
      RETURN
      END
```

Now if `L` is nonpositive the loop is skipped no matter how the code was compiled, and `LENGTH=0` is returned. The price we pay for this safety is that the code is larger and tests `L` on each invocation even if the routine will never be compiled following the FORTRAN-66 standard and `L` will always be positive. Whether this is worthwhile is a judgment call that must be made by the programmer (I decided that it is definitely not).

10.9.14 `[P]` (a) According to the table in textbook §10.1, the alphabetical order of the given "words" is `<@>`, `>&<`, `Smith`, `XY(1+N)`, `apple`, `eat!`, `eats`, `smith`, `wombat`, `xylophone`.

(b) Here is a program that sorts a list of 100 words each having up to 24 characters. When compiled and run it produces the conversation shown to the right of the listing.

```
      CHARACTER*24 WORDS(100),WTEMP
      WRITE(6,901)
  901 FORMAT('unsorted:')
      DO 1 I=1,100
           READ(5,902,END=2) WORDS(I)
  902      FORMAT(A24)
           N=I
    1 CONTINUE
    2 DO 3 I=1,N-1
      DO 3 J=I+1,N
           IF(WORDS(I).GT.WORDS(J)) THEN
              WTEMP=WORDS(I)
              WORDS(I)=WORDS(J)
              WORDS(J)=WTEMP
           ENDIF
    3 CONTINUE
      WRITE(6,903)
  903 FORMAT(' '/'sorted:')
      DO 4 I=1,N
           WRITE(6,902) WORDS(I)
    4 CONTINUE
      STOP
      END
```

```
unix[1] a.out
unsorted:
smith
Smith
eat!
eats
XY(1+N)
xylophone
<@>
>&<
wombat
apple

sorted:
<@>
>&<
Smith
XY(1+N)
apple
eat!
eats
smith
wombat
xylophone
unix[2]
```

10.9.15 [P] Here is a Classical FORTRAN program for copying a file from standard-in to standard-out.

```
      CHARACTER*1 TEXT(257)
      LINE=0
    2 READ(5,901,END=1) TEXT
  901 FORMAT(257A1)
      LINE=LINE+1
      L=LENGTH(TEXT,257)
      IF(L.EQ.257) THEN
         WRITE(0,902) LINE
  902    FORMAT('input line ',I10,' is too long')
         STOP
      ENDIF
      WRITE(6,901) (TEXT(K),K=1,L)
      GO TO 2
    1 STOP
      END
```

I chose a maximum line length of 256 characters because I know from experience that most files I want to copy have lines shorter than that; someone else might pick a different number. To warn of lines longer than that the program reads up to 257 characters and stops with an error message if it finds a line of exactly that length. Of course in an input file with long lines it might happen that every one of them has a blank in column 257, in which case none of them will be detected.

In a language capable of reading and writing a line one *character* at a time that process could simply be repeated until the end of each input line was found, and there would be no limit to their length. There would also be no need to store more than 1 character at a time, so the program could use less memory as well as being faster and more generally useful. The C programming language provides a way to read or write one character at a time and returns the newline as a character on input. In Classical FORTRAN the `FORMAT` code `$` provides a way to read or write one character at a time, but there is no way to detect the end of a line. In FORTRAN-90 (as discussed in textbook Chapter 17) the `ADVANCE='NO'` option of `READ` and `WRITE` provides a way to read or write one character at a time, and there is an `EOR=` option (which *will* be mentioned in Chapter 17 if there is ever a Third Edition!) to detect the end of a line. These mechanisms are used in the file-copying program listed below.

```
      CHARACTER*1 CHAR
    3 READ(5,901,ADVANCE='NO',END=1,EOR=2) CHAR
  901 FORMAT(A1)
      WRITE(6,901,ADVANCE='NO') CHAR
      GO TO 3
    2 WRITE(6,*)
      GO TO 3
    1 STOP
      END
```

Single characters are read and written without advancing the input or output line. When the end of a record is reached the `EOR=2` causes a branch to statement 2, which writes a newline. The next `READ` then advances the line pointer to the next input line, and the process continues until the end of the file is reached. Then the `END=1` causes a branch to statement `1` and stops the program.

10.9.16 [P] Here is a program that copies records from standard-in to standard-out but omits adjacent repeated lines.

```
      CHARACTER*80 NEW,OLD
      CHARACTER*1 NEWC(80)
      EQUIVALENCE(NEW,NEWC)
C
C     read and write the first input line
      READ(5,901,END=1) NEW
  901 FORMAT(A80)
      L=LENGTH(NEWC,80)
      WRITE(6,902) (NEWC(K),K=1,L)
  902 FORMAT(80A1)
      OLD=NEW
C
C     read the next input line
    2 READ(5,901,END=1) NEW
C
C     skip if it is the same as the previous one
      IF(NEW.EQ.OLD) GO TO 2
      L=LENGTH(NEWC,80)
      WRITE(6,902) (NEWC(K),K=1,L)
      OLD=NEW
      GO TO 2
C
C     when the end of the input is reached we are done
    1 STOP
      END
```

Because Classical FORTRAN uses static memory allocation it is necessary to assume a fixed maximum line length, and here I have chosen 80 characters. Any input characters to the right of column 80 are simply ignored.

The program reads and writes the first line unconditionally. If `NEW` were simply written, the output would contain trailing blanks out through column 80 even if they were not in the input record. To avoid this, the `LENGTH` function of textbook §10.2 is used to find the index of the last nonblank in the input line and only those characters are written. This means that any trailing blanks that are present in the input text are missing in the output. To permit the printing of individual characters from the input line, the program uses `EQUIVALENCE` to overlay the `CHARACTER*80` scalar `NEW` on the `CHARACTER*1` vector `NEWC`.

Each subsequent line is read and compared to the line most recently written, and only if they differ is the new line written.

The UNIX™ `uniq` utility works like the program listed above but differs from it in several respects. First, `uniq` (probably written in C) can process lines of arbitrary length. Second, `uniq` preserves exactly any trailing blanks in the input, and it considers lines to be different if they are identical except for the numbers of trailing blanks they have. Third, `uniq` provides options that allow the user to change its behavior in various ways.

10.9.19 [P] Here is a library subprogram that satisfies the requirements of the problem.

```
C
Code by Michael Kupferschmid
C
      SUBROUTINE GETI4S(PR,LP,IDEF,LDEF, I,RC)
C     This routine prompts for and reads I, returning
C     the default value IDEF if the user sends a null response.
C
C     RC  meaning
C     --  -----------------
C      0  all went well
C      1  user sent EOF
C      2  an error occurred
C
C     variable  meaning
C     --------  -------
C     BTD       routine converts INTEGER*4 to numerals
C     DTB       routine converts numerals to INTEGER*4
C     I         integer returned
C     IDEF      default value to set for null response
C     K         index on the characters of a string
C     LDEF      digits for IDEF, or zero if no default value
C     LENGTH    returns index of last nonblank in a string
C     LI        number of digit positions allowed for default
C     LP        length of description
C     LPR       length of prompt
C     LT        index of last nonblank in user's response
C     MIN0      Fortran function for smaller of INTEGER*4s
C     PR        description of input value
C     PROMPT    routine prompts for input from the keyboard
C     PRSTR     prompt written
C     RC        return codes, then return code; see table above
C     SHIFTL    routine removes leading blanks from a string
C     TEXT      user response
C
C     formal parameters
      CHARACTER*1 PR(LP)
      INTEGER*4 RC
C
C     local variables
      CHARACTER*1 PRSTR(78)
      CHARACTER*78 TEXT
C
C ------------------------------------------------------------------
C
C     return the default in case of error
      IF(LDEF.GT.0) THEN
         I=IDEF
      ELSE
         I=0
      ENDIF
C
```

```
C     construct the prompt string
      LPR=0
      DO 1 K=1,78
           IF(K.LE.LP) THEN
              LPR=LPR+1
              IF(LPR.GT.78) GO TO 2
              PRSTR(LPR)=PR(K)
           ELSE
              PRSTR(K)=' '
           ENDIF
    1 CONTINUE
      IF(LDEF.GT.0) THEN
C        display the default
         LPR=LPR+1
         IF(LPR.GT.78) GO TO 2
         PRSTR(LPR)=' '
         LPR=LPR+1
         IF(LPR.GT.78) GO TO 2
         PRSTR(LPR)='['
         LPR=LPR+1
         LI=MIN0(78-LPR+1,LDEF)
         CALL BTD(IDEF, PRSTR(LPR),LI,RC)
         IF(RC.NE.0) GO TO 2
         CALL SHIFTL(PRSTR(LPR),78-LPR+1)
         LPR=LENGTH(PRSTR,78)
         LPR=LPR+1
         IF(LPR.GT.78) GO TO 2
         PRSTR(LPR)=']'
      ENDIF
      LPR=LPR+1
      IF(LPR.GT.78) GO TO 2
      PRSTR(LPR)=':'
C
C     write the prompt (and a space) and read the response
    4 CALL PROMPT(PRSTR,LPR)
      READ(5,900,END=3) TEXT
  900 FORMAT(A78)
C
C     interpret the response
      LT=LENGTH(TEXT,78)
      IF(LT.GT.0) THEN
C        the user entered a response; replace the default with it
         CALL DTB(TEXT,LT, I,RC)
         IF(RC.NE.0) GO TO 4
      ENDIF
      RC=0
      RETURN
C
C     handle EOF and errors
    3 RC=1
      RETURN
    2 RC=2
      RETURN
      END
```

10.9.20 [P] Here is a routine that changes the spacing of a line as described in the problem statement.

```
      SUBROUTINE SPACEL(TEXT,L)
      CHARACTER*1 TEXT(L)
C
C     copy the string to the left, ignoring multiple blanks
      J=1
      DO 1 I=2,L
C          is this character a blank?
           IF(TEXT(I).NE.' ') GO TO 2
C
C          this character is a blank; was the previous?
           IF(TEXT(I-1).NE.' ') GO TO 2
C
C          both this character and the previous character are blanks
           GO TO 1
C
    2      J=J+1
           TEXT(J)=TEXT(I)
    1 CONTINUE
C
C     fill the end of the string with blanks
      IF(J.EQ.L) RETURN
      JP=J+1
      DO 3 J=JP,L
           TEXT(J)=' '
    3 CONTINUE
      RETURN
      END
```

The main program below exercises `SPACEL` on the test string specified in the problem statement.

```
      CHARACTER*32 LINE/' More  and   more    blanks     '/
      WRITE(6,901) LINE
  901 FORMAT(A32,'|')
      CALL SPACEL(LINE,32)
      WRITE(6,901) LINE
      STOP
      END
```

When run this program prints the following output.

```
 More  and   more    blanks     |
 More and more blanks           |
```

10.9.21 [P] Here is a routine that satisfies the requirements of the problem statement.

```
      SUBROUTINE FINDST(LINE,LL,STRING,LS, KSTART)
C     This routine returns in KSTART the position in LINE where
C     STRING begins, or 0 if STRING is not found, or -1 if the
C     parameters don't make sense.
C
C     KSTART  meaning
C     ------  -------
C       -1    bad parameters
C        0    STRING was not found in LINE
C       >0    the position in LINE where STRING begins
C
C     variable  meaning
C     --------  -------
C     KL        index on the characters of LINE
C     KS        index on the characters of STRING
C     KSTART    see table above
C     LINE      line of text to be searched for STRING
C     LL        number of characters in LINE
C     LS        number of characters in STRING
C     STRING    string to be searched for
C
      CHARACTER*1 LINE(LL),STRING(LS)
C
C -----------------------------------------------------------------
C
C     eliminate impossible cases
      KSTART=-1
      IF(LL.LE.0 .OR. LS.LE.0 .OR. LL.LT.LS) RETURN
C
C     try to match STRING to LINE
      DO 1 KL=1,LL-LS+1
           DO 2 KS=1,LS
                IF(STRING(KS).NE.LINE(KL+KS-1)) GO TO 1
    2      CONTINUE
           KSTART=KL
           RETURN
    1 CONTINUE
C
C     there was no match
      KSTART=0
      RETURN
      END
```

If `LL` or `LS` is non-positive, or if the line is shorter than the string being sought, then the parameters don't make sense and the routine returns with `KSTART=-1`. Otherwise we need to compare `STRING` to successive substrings of `LINE` beginning at position `KL=1` in `LINE`. In order for `STRING` to fit in the characters of `LINE` beginning with `LINE(KL)`, `KL` can be no bigger than `LL-LS+1`, so that is the upper limit of the `DO 1` loop. For each starting position in `LINE`, the `DO 2` loop compares `STRING` to `LINE`. If a match is found, `KSTART` is set to `KL`; otherwise the routine returns with `KSTART=0`.

10.9.23 `[P]` (a) The program given in the Exercise writes only a negative value, but this one writes a negative value, a zero, and a positive value to see what happens when numbers get rounded to zero to fit into an `F4.1` field specification.

```
      WRITE(6,901) -.01,0.0,+.01
  901 FORMAT(F4.1)
      STOP
      END
```

When it is compiled using `gfortran 4.3.2-1ubuntu12` and run under Linux it produces the following output.

```
unix[1] a.out
-0.0
 0.0
 0.0
unix[2]
```

This implementation of the I/O library indicates by printing `-0.0` that a negative value has been rounded to zero (some implementations do not!) but it fails to distinguish between a positive value that has been rounded and a true zero.

(b) The subroutine listed on the next page meets the requirements of the problem. It assumes [16] that the field specification supplied will be exactly 4 characters long as in the example, so that it looks like `Fn.m`, where `m` is the number of digits to be printed after the decmial point and `n` is the total field width.

It begins [26-28] by using the `DTB` routine of textbook §18.1 to convert the numerals given for `n` and `m` to `INTEGER*4` values in `N` and `M`. If the conversion fails [29] or the field descriptor `FMT` is otherwise badly formed [30] the routine [31-33] writes an error message and stops. Otherwise [35-36] it copies those numerals into a template `F1` [19] which is equivalenced [21] to the `CHARACTER*6` object-time format `F6` [20].

Next [40] it compares $|\texttt{X}|$ to $\frac{1}{2} \times 10^{-\texttt{M}}$, which is the smallest value that will print as nonzero in a field with `M` digits after the decimal point. If the value is big enough [41-43] it uses the object-time format to write out the value, and returns.

If the value will print as zero, the routine [47-53] prints a zero with the appropriate prefix, and returns.

When the subroutine is compiled with the main program given in the problem statement and run, it produces the following output.

```
unix[4] a.out
-0.0
 0.0
+0.0
unix[5]
```

This output differs slightly from that shown in the exercise because my I/O library prints a zero before the decimal point when the field specification is `F4.1` and the value is zero.

```
 1       SUBROUTINE WRITEZ(FMT,X)
 2 C
 3 C     variable  meaning
 4 C     --------  -------
 5 C     DABS      Fortran function returns |REAL*8|
 6 C     DTB       routine converts numerals to an INTEGER*4
 7 C     F1        F6 as individual characters
 8 C     F6        the object-time format corresponding to FMT
 9 C     FMT       the given field specification
10 C     M         the number of digits to follow the decimal
11 C     N         the total field width
12 C     RCM       return code from DTB in converting M
13 C     RCN       return code from DTB in converting N
14 C     X         the value to be written
15 C
16       CHARACTER*1 FMT(4)
17       REAL*8 X
18 C
19       CHARACTER*1 F1(6)/'(','F','n','.','m',')'/
20       CHARACTER*6 F6
21       EQUIVALENCE(F1,F6)
22       INTEGER*4 RCM,RCN
23 C
24 C -----------------------------------------------------------------
25 C
26 C     find the n and m specified in FMT
27       CALL DTB(FMT(2),1,N,RCN)
28       CALL DTB(FMT(4),1,M,RCM)
29       IF(RCN.NE.0 .OR. RCM.NE.0 .OR. N.LT.M+2 .OR.
30      ;   FMT(1).NE.'F' .OR. FMT(3).NE.'.') THEN
31            WRITE(6,900) FMT
32   900      FORMAT('bad field specification ',4A1)
33            STOP
34       ELSE
35            F1(3)=FMT(2)
36            F1(5)=FMT(4)
37       ENDIF
38 C
39 C     will the value print as a zero?
40       IF(DABS(X) .GE. 0.5D0*10.D0**(-M)) THEN
41 C        no; write it using the given format
42          WRITE(6,F6) X
43          RETURN
44       ENDIF
45 C
46 C     yes; write the appropriate form of zero
47       IF(X .LT. 0.D0) WRITE(6,901)
48   901 FORMAT('-0.0')
49       IF(X .EQ. 0.D0) WRITE(6,F6) X
50       IF(X .GT. 0.D0) WRITE(6,902)
51   902 FORMAT('+0.0')
52       RETURN
53       END
```

10.9.25 [E] A numeral is the symbol we write to represent a number. For example, the numeral 2 is the character representing the number whose value is 2. In FORTRAN the numeral 2 is a single byte containing that numeral's ASCII character code, Z'32', which has the bit string 00110010. The number 2 is in contrast an INTEGER*4 constant (a fullword of 4 bytes) containing the bit string 00000000000000000000000000000010, which is the binary representation for the value +2.

The program below reads a number into N and writes the number that is in M.

```
      READ(5,901) N
  901 FORMAT(I2)
      M=N+3
      WRITE(6,901) M
      STOP
      END
```

When the program runs and you type in a value for N at the keyboard, the I/O library converts the numerals you type into a number that gets stored in N. It is that value that is used in the arithmetic for finding M. When the program writes out the resulting number M, the I/O library converts it into the numerals that get printed.

Of course, just as a numeral represents a number, a FORTRAN variable actually represents a storage location in memory. Thus it is not strictly correct to say, as I just did, that a numerical value is "in" N or M. The numerical value of a variable is in memory, at the address for which the variable is our symbolic name. If anything is to be usefully thought of as "in" N or M, it is the address in memory named by that variable.

It might seem an unfamiliar and difficult mental task to distinguish carefully between things and their names, but there are occasions in our everyday lives when we do that without stopping to think about it. For example, I know the difference between the names on my class roster and the students sitting before me, and I can sort the names without moving the people. In programming it is sometimes necessary to bear it consciously in mind whether we are manipulating values or their names. Mixing the two up fortunately accounts for only a small minority of the mistakes that people make in writing FORTRAN programs (but it accounts for many of the mistakes that people make when writing in assembler language).

10.9.26 [P] (a) Here is a version of `PROMPT` that copies `MSG` into an object-time `FORMAT`.

```
      SUBROUTINE PROMPT(MSG,LM)
C     This routine prompts on unit 0 for input from the terminal.
C
C     variable  meaning
C     --------  -------
C     FMT       object-time format
C     K         index on the characters of MSG
C     L         number of characters of MSG actually used
C     LM        length of MSG
C     MIN0      Fortran function for smaller of INTEGER*4s
C     MSG       prompt string
C
      CHARACTER*1 MSG(LM),FMT(106)
C
C -----------------------------------------------------------------
C
C     limit the prompt to 99 characters
      L=MIN0(LM,99)
      IF(L.LE.0) RETURN
C
C     construct the object-time format ('contentsofMSG ',$)
      FMT(1)='('
      FMT(2)=1H'
      DO 1 K=1,L
           FMT(2+K)=MSG(K)
    1 CONTINUE
      FMT(2+L+1)=' '
      FMT(2+L+2)=1H'
      FMT(2+L+3)=','
      FMT(2+L+4)='$'
      FMT(2+L+5)=')'
      DO 2 K=2+L+6,106
           FMT(K)=' '
    2 CONTINUE
C
C     write the prompt
      WRITE(0,FMT)
      RETURN
      END
```

The version of §10.4 limits `MSG` to 99 characters so that when converted to numerals `LM` will have no more than 2 digits. This version limits `MSG` to 99 characters because it is necessary to pick some limit so that `FMT` can be dimensioned. Now the object-time format is constructed character by character to include the contents of `MSG`. Any unused portion of `FMT` must be blanked out (by the `DO 2` loop) on each call in case a previous invocation put nonblanks there. Notice that the forward-quote character `'` used in `FMT` is specified using the Hollerith constant `1H'`, and that the `WRITE` statement no longer writes out the variable `MSG`.

(b) Here is a version of PROMPT that uses a fixed, rather than an object-time, format.

```
      SUBROUTINE PROMPT(MSG,LM)
C     This routine prompts on unit 0 for input from the terminal.
C
C     variable  meaning
C     --------  -------
C     L         number of characters of MSG actually used
C     LM        length of MSG
C     MIN0      Fortran function for smaller of INTEGER*4s
C     MSG       prompt string
C
      CHARACTER*1 MSG(LM)
C
C ------------------------------------------------------------------
C
C     limit the prompt to 99 characters
      L=MIN0(LM,99)
      IF(L.LE.0) RETURN
C
C     write the prompt string
      WRITE(0,901) MSG
  901 FORMAT(99(A1,$))
      RETURN
      END
```

As discussed in §9.1.1, if a FORMAT statement is used up before all the variables have been transmitted the I/O library returns to the nearest left parenthesis and reuses the field specifications that begin there. Unfortunately, we cannot omit the repetition factor from this FORMAT because when the I/O library returns to the nearest left parenthesis the $ flag is ignored. To prevent each letter of MSG from geting printed on a new line it is necessary to use a big enough repetition factor, so to be consistent with the other versions this PROMPT also imposes a limit of 99 characters in MSG. It is only because $ is included *inside* the repeated group that the prompt hangs when there are fewer than 99 characters in MSG.

The earlier versions of PROMPT insert a space after the MSG text, but in this version it is necessary for the blank to be included in MSG.

10.9.30 [H] This program reads a REAL*8 value into A, rounds it away from zero so that it has two digits after the decimal point when it is written in scientific notation, and outputs the result D.

```
      REAL*8 A,B,C,D,SGN
      INTEGER*4 PWR
C
C     get the number to be rounded off
    2 CALL PROMPT('A=',2)
      READ(5,*,END=1) A
C
C     find how many powers of 10 it contains
      PWR=IFIX(0.5+SNGL(DLOG10(DABS(A))))
C
C     find its sign
      SGN=0.D0
      IF(A.GT.0.D0) SGN=+1.D0
      IF(A.LT.0.D0) SGN=-1.D0
C
C     normalize the number to be nonnegative in [1,10)
      B=DABS(A)/(10.D0**PWR)
C
C     shift decimal 2 places right, chop, shift 2 places left
      C=0.01D0*DFLOAT(IFIX(0.5+SNGL(100.D0*DABS(B))))
C
C     restore the original sign and power of 10
      D=SGN*(10.D0**PWR)*C
C
      WRITE(6,901) D
  901 FORMAT('D=',1PE13.6)
      GO TO 2
    1 STOP
      END
```

When the program is compiled and run it produces the output shown below.

```
unix[1] ftn junk.f
unix[2] a.out
A= 1.23654
D= 1.240000E+00
A= -1.23654
D=-1.240000E+00
A= 123.654
D= 1.240000E+02
A= -123.654
D=-1.240000E+02
unix[3]
```

In two places this code finds the integer part of a positive real number by adding $\frac{1}{2}$ and truncating the result.

10.9.31 [P] Here is a function that meets the requirements of the problem statement.

```
C
      FUNCTION CHECK(FIELD)
C     This routine returns T if FIELD looks like -123.45
C
C     variable  meaning
C     --------  -------
C     DIGIT     the numerals
C     FIELD     string to be tested
C     K         index on the characters of SIGN or DIGIT
C     SIGN      +, -, or blank
C
C     formal parameters
      LOGICAL*4 CHECK
      CHARACTER*1 FIELD(7)
C
C     local variables
      CHARACTER*1 SIGN(3)/'+','-',' '/
      CHARACTER*1 DIGIT(10)/'0','1','2','3','4','5','6','7','8','9'/
C
C ------------------------------------------------------------------
C
      CHECK=.TRUE.
      DO 1 K=1,3
           IF(FIELD(1).EQ.SIGN(K)) GO TO 2
    1 CONTINUE
      CHECK=.FALSE.
    2 DO 3 K=1,10
           IF(FIELD(2).EQ.DIGIT(K)) GO TO 4
    3 CONTINUE
      CHECK=.FALSE.
    4 DO 5 K=1,10
           IF(FIELD(3).EQ.DIGIT(K)) GO TO 6
    5 CONTINUE
      CHECK=.FALSE.
    6 DO 7 K=1,10
           IF(FIELD(4).EQ.DIGIT(K)) GO TO 8
    7 CONTINUE
      CHECK=.FALSE.
    8 IF(FIELD(5).NE.'.') CHECK=.FALSE.
      DO 9 K=1,10
           IF(FIELD(6).EQ.DIGIT(K)) GO TO 10
    9 CONTINUE
      CHECK=.FALSE.
   10 DO 11 K=1,10
            IF(FIELD(7).EQ.DIGIT(K)) GO TO 12
   11 CONTINUE
      CHECK=.FALSE.
   12 RETURN
      END
```

Some work could be saved by returning as soon as the first format error is doscovered, but only by making the code longer and more complex. It is instructive to attempt a solution of this problem that avoids explicit branching.

10.9.36 `[P]` (a) The routine below simulates the C `assert()` macro.

```
      SUBROUTINE ASSERT(COND,STRING,LS)
      LOGICAL*4 COND
      CHARACTER*1 STRING(LS)
C
C     do nothing if the assertion is true
      IF(COND) RETURN
C
C     the assertion is false; report the error and stop
      WRITE(0,901) STRING
  901 FORMAT(80A1)
      STOP
      END
```

To understand how `ASSERT` works it is helpful to see how it is used. The program below uses `ASSERT` to avoid finding the square root of a negative number.

```
      READ *,X
      CALL ASSERT(X.GE.0.0,'negative argument to SQRT',25)
      Y=SQRT(X)
      PRINT *,Y
      STOP
      END
```

The computation of the logical value `COND` used by the subroutine is actually accomplished in the caller, where the expression `X.GE.0.0` gets evaluated so that its result can be passed as the first argument of `ASSERT`. The problem statement says "One argument should be the logical value to be tested and *the* other a character string describing the violation...", which suggests there should be only two arguments, but to facilitate using strings of arbitrary length in Classical FORTRAN `ASSERT` receives the string length as a third argument (in the manner of the character minipulation routines described in textbook §10). When the code above is compiled and run it produces the following output.

```
unix[1] a.out
2
   1.4142135
unix[2] a.out
-2
negative argument to SQRT
unix[3]
```

(b) The C `assert()` macro and the `ASSERT` subroutine given above implement a debugging strategy known as the **mousetrap**. A mousetrap is a code sequence that stops a program in such a way as to reveal the existence and location of an abnormal condition. `ASSERT` is useful only when the abnormal condition can be detected by evaluating some simple logical condition like the one in our example. Setting moustraps might be helpful for debugging while a program is still under development, but an application that is correct and in production use requires a more robust response to bad data. A finished program should try harder to recover when something goes wrong, or at least to preserve the results it has computed so far, rather than just reporting the error and then giving up.

10.9.42 `[H]` (a) On either kind of machine `BIG` is stored with the `13` byte at the high memory address. The `EQUIVALENCE` overlays `BIG` with `I`, an `INTEGER*4`, so `I` also has the `13` byte at its high memory address. (a) On a big-endian machine the high memory address is the least significant byte of an integer, so the value represented is $00000013_{16} = 1 \times 16^1 + 3 \times 16^0 = 19_{10}$ and the program prints `19`. On a little-endian machine the high memory address is the most significant byte of `I`, so the value represented is $13000000_{16} = 1 \times 16^7 + 3 \times 16^6 = 318767104_{10}$ and the program prints `318767104`. (b) The endianness of a processor can be determined by testing the value of an integer that has been initialized as in the program given in the exercise. Here is a program that prints `big-endian` or `little-endian` according to the byte ordering used by the processor on which it is run.

```
      CHARACTER*1 BIG(4)/Z'00',Z'00',Z'00',Z'13'/
      EQUIVALENCE(BIG,IBIG)
      IF(IBIG.EQ.19) THEN
         PRINT *,'big-endian'
      ELSE
         PRINT *,'little-endian'
      ENDIF
      STOP
      END
```

(c) To guard against the possibility that some entirely different scheme is used to store integers, we could improve the program to test both the big-endian and the little-endian byte ordering, like this.

```
      CHARACTER*1 BIG(4)/Z'00',Z'00',Z'00',Z'13'/
      EQUIVALENCE(BIG,IBIG)
      CHARACTER*1 LTL(4)/Z'13',Z'00',Z'00',Z'00'/
      EQUIVALENCE(LTL,ILTL)
      IF(IBIG.EQ.19) PRINT *,'big-endian'
      IF(ILTL.EQ.19) PRINT *,'little-endian'
      IF(IBIG.NE.19 .AND. ILTL.NE.19) PRINT *,'neither'
      STOP
      END
```

11.9.1 [E] Classical FORTRAN stores arrays in column-major order. For example, the array that is allocated by the following type statement

```
REAL*8 A(3,2)
```

is stored in a contiguous stretch of memory consisting of 3 $\times$ 2 = 6 doublewords, with A(1,1) at the lowest address followed by A(2,1), A(3,1), A(1,2), A(2,2), and A(3,2) in the successive doublewords at higher addresses. It is because the array elements are stored in that order, going down successive columns from left to right by varying the inner subscript of A most rapidly, that this order of storage is called column major.

Classical FORTRAN passes subprogram parameters by reference. In other words, what is passed to the subprogram is the address in memory where the value of the parameter (or in the case of an array the address of the *first* element of the array) is stored. The location in memory (relative to the beginning of the object program) where a variable will be stored is determined by the compiler when it processes the routine where the variable is first typed or dimensioned, so the storage assigned can be thought of as residing within that routine. Subprograms to which the variable is passed as a parameter can inspect or change the value of the variable by referring to that memory address, so no memory is needed or allocated for the parameter inside the subprogram.

11.9.2 [E] If A invokes B invokes C and C needs data from A but B does not need that data, COMMON can be used to pass the data around B. If it is necessary to conserve memory in a program by using the same storage for local variables in different routines whose values do not need to be preserved across calls, COMMON can be used to provide the shared storage. If constant data are needed in different routines and the flow of control through the routines would require the data to be present in routines not needing it if it were passed in formal parameters, COMMON can be used to communicate the constant data.

11.9.3 [P] The main program on the left below initializes A to the value on the right, calls ADDCOL to add up the columns, and reports the column sums.

```
      REAL*8 A(10,10),COLSUM(10)
      LDA=10
      N=3
      DO 1 I=1,N
      DO 1 J=1,N
           A(I,J)=DFLOAT(I+N*(J-1))
    1 CONTINUE
      CALL ADDCOL(A,LDA,N, COLSUM)
      DO 2 J=1,N
           PRINT *,'column',J,' sum=',COLSUM(J)
    2 CONTINUE
      STOP
      END
```

$$\mathtt{A} = \begin{bmatrix} 1 & 4 & 7 \\ 2 & 5 & 8 \\ 3 & 6 & 9 \end{bmatrix}$$

The ADDCOL routine listed on the left below has the calling sequence specified in the problem statement. It uses the value LDA passed in by the calling program for the leading dimension of A, but sets the second dimension of A and the length of COLSUM to the actual number N of columns and rows in use. These dimensions could have been set to * instead, but using N permits array subscript overflows to be detected if the -C compiler option is used. Notice that the §11.1 function VSUM, which is listed on the right below, must also be declared REAL*8 in ADDCOL.

```
      SUBROUTINE ADDCOL(A,LDA,N, COLSUM)
      REAL*8 A(LDA,N),COLSUM(N),VSUM
      DO 1 J=1,N
           COLSUM(J)=VSUM(A(1,J),N)
    1 CONTINUE
      RETURN
      END
```

```
      FUNCTION VSUM(V,N)
      REAL*8 VSUM,V(N)
      VSUM=0.D0
      DO 1 J=1,N
           VSUM=VSUM+V(J)
    1 CONTINUE
      RETURN
      END
```

The DO 1 in ADDCOL loops over the N columns of A and the corresponding elements of COLSUM. FORTRAN stores arrays in column-major order, so the elements of column J occupy the N doublewords in memory starting at the address of A(1,J). FORTRAN passes subprogram parameters by address, so invoking VSUM with A(1,J) passes it the address of the first element in the column. Then VSUM adds up the N elements in the column, which it sees as the vector V, and returns the sum for assignment to COLSUM(J) in ADDCOL. Compiling and running the program produces the following output.

```
unix[1] f77 main.f addcol.f vsum.f
unix[2] a.out
 column 1 sum=  6.
 column 2 sum=  15.
 column 3 sum=  24.
unix[3]
```

These are the correct column sums for matrix A.

11.9.5 [E] This question involves passing an array column, which is discussed in §11.1 of the text.

(a) FORTRAN passes subprogram parameters by reference, so what is passed for V is its starting address rather than its value. In the CALL statement, A(1,3) is the address of that array element, so what SUB sees for V is the elements of A beginning with A(1,3). The elements of A are stored in column-major order and the value passed for N, the length of V, is 10. Thus the vector V in SUB is really A(1,3), A(2,3) ... A(10,3), or the entire first column of A in the main program.

(b) Passing an array column to a subprogram is often desirable because it is convenient, saves machine time that would otherwise be wasted in copying the column into a vector, and saves the space that would otherwise be used in storing the vector.

(c) If the actual parameter is changed to A(3,1) then what SUB sees for V is a vector composed of A(3,1), A(4,1) ... A(10,1), A(1,2), and A(2,2). Those are the 10 elements of A in column-major order starting with A(3,1).

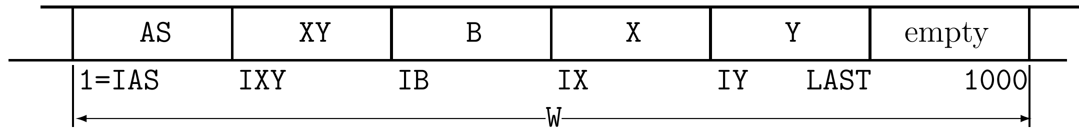
11.9.8 [E] The output produced is shown below.

```
unix[1] a.out
c
    d
e
    f
unix[2]
```

where the c and e are printed in column 1 and the d and f are printed in column 5.

The BLOCK DATA subprogram ensures that when the program is loaded the 9-character variable WORD has been initialized to the string abcdefghi. This is visible to the main program, through COMMON block /XXX/, as the 9-element vector of separate characters called STRING. The first executable statement in the program is the CALL to SUB, which passes for its first parameter the address of STRING(3) and for its second parameter the address of a memory location where the literal value 4 is stored. Control flows into the subroutine, where the first parameter V is a vector of L separate characters. Using the address provided for L, the entry code generated by the compiler for SUB retrieves the value 4 from the main program, so L=4 in SUB. Thus V contains the 4 characters starting with STRING(3), or c, d, e, and f. Next the WRITE statement prints V(1) through V(4) using the given FORMAT. The FORMAT specification prints one character, moves to the next line, tabs in 5 spaces, and prints one character. Because 4 characters must be printed altogether the I/O library returns to the nearest left parenthesis and repeats that pattern of field specifications, producing the output shown above. When the WRITE statement has been executed, control RETURNs to the main program, which then STOPs.

11.9.9 [E] The stretch of memory storing the workspace is pictured below to show the variables arranged within it and their starting indices in W. This view assumes that the sum of the lengths of the variables is less than 1000 doublewords so that some elements of W remain unused. The segments are not drawn to scale (e.g., XY is probably actually longer than X).



Here is a program that calculates the starting indices in W where its constituent arrays begin, and checks whether they will fit.

```
      REAL*8 W(1000)
    1 READ(5,*) M,N
      IF(M.LE.0 .OR. N.LE.0) THEN
         PRINT *,'M and N must be positive; try again'
         GO TO 1
      ENDIF
      IAS=1
      IXY=IAS+N*(N+1)/2
      IB=IXY+MAX0(N+M,2*N*M-N)
      IX=IB+N*M
      IY=IX+N
      LAST=IY+(M+1)-1
      IF(LAST.GT.1000) THEN
         PRINT *,LAST,' elements will not fit'
      ELSE
         PRINT *,'AS starts at ',IAS
         PRINT *,'XY starts at ',IXY
         PRINT *,' B starts at ',IB
         PRINT *,' X starts at ',IX
         PRINT *,' Y starts at ',IY
         PRINT *,'used ends at ',LAST
         PRINT *,'unused space ',1000-LAST,' elements'
      ENDIF
      STOP
      END
```

The output below shows results for several different M, N combinations.

```
unix[1] a.out
10 5
 AS starts at  1
 XY starts at  16
  B starts at  111
  X starts at  161
  Y starts at  166
 used ends at  176
 unused space  824 elements
unix[2] a.out
12 21
 AS starts at  1
 XY starts at  232
  B starts at  715
  X starts at  967
  Y starts at  988
 used ends at  1000
 unused space  0 elements
unix[3] a.out
100 50
 16376 elements will not fit
unix[4]
```

11.9.11 [H] (a) The arrangement used puts I first in W, so if NI integers are read they occupy the first NI fullwords of W. But then GETR is passed the address of W(NI+1), the NI+1'st *doubleword* in W, and fills in the REAL*8 values starting there. This leaves the NI+1'st through the 2*NI'th fullwords in W unused. If there are NI fullwords they take up NI/2 doublewords if NI is even, or NI/2+1 doublewords if NI is odd. So it would be better to pass GETR the starting address W(NI/2+MOD(NI,2)) and start storing the REAL*8s there. This would leave only one fullword unused, when NI is odd. It would be even better to store the REAL*8s first, so that there are no unused fullwords.

(b) The type mismatch is not a problem, but the warning message *is* a distraction.

(c) Here is a version of the program that eliminates the wasted space and the warning messages.

```
REAL*8 RW(1000)
INTEGER*4 IW(2000)
EQUIVALENCE(RW,IW)
KR=0
CALL GETR(RW(KR+1),NR)
KI=2*NR
CALL GETI(IW(KI+1),NI)
CALL SUBR(IW(KI+1))
CALL SUBR(13)
:
```

Now the J'th element of R is RW(KR+J) (here J is counting doublewords) and the J'th element of I is IW(KI+J) (here J is counting fullwords). Of course, as is always the case when using shared workspace, there will be a problem if we look for an element of R at RW(KR+J) and J is greater than NR, because then we're into I. It is much simpler, and easier to understand, if instead of putting real and integer values in the same shared workspace you can afford the potential inefficiency of having separate shared workspaces for integers and reals.

(d) To reject data sets that are too big we need to change the program something like this.

```
PARAMETER(MAXR=1000,MAXI=2*MAXR)
REAL*8 RW(MAXR)
INTEGER*4 IW(MAXI)
EQUIVALENCE(RW,IW)
KR=0
CALL GETR(MAXR,RW(KR+1),NR)
KI=2*NR
CALL GETI(MAXI-KI,IW(KI+1),NI)
CALL SUBR(IW(KI+1))
CALL SUBR(13)
:
```

```
SUBROUTINE GETR(LEFT,R,NR)
REAL*8 R(*)
READ(5,902) NR,(R(J),J=1,MIN0(NR,LEFT))
IF(NR.GT.LEFT) THEN
   :

SUBROUTINE GETI(LEFT,I,NI)
INTEGER*4 I(*)
READ(5,901) NI,(I(J),J=1,MIN0(NI,LEFT))
IF(NI.GT.LEFT) THEN
   :
```

Now GETR and GETI know how many values (REAL*8 or INTEGER*4 respectively) will fit in the space that is left, so they can avoid reading more than that and can take some action (such as reporting the problem and stopping the program) if too much data is found.

11.9.12 `[E]` Variables that are allocated locally in different routines occupy separate memory locations, and the total amount of memory they consume is the *sum* of their sizes s_i. Workspace that is allocated in `COMMON` can be used by several routines to hold different temporary variables in each, so that the same space is used repeatedly and the total memory required is the *largest* of the sizes s_i required by the individual routines. Because $s_i \geq 0$

$$\max_i (s_i) \leq \sum_i s_i$$

and memory might be saved by storing temporary local variables in shared `COMMON` workspace. Unfortunately, any saving in memory is obtained only by assuming two significant risks. First, some variable that is stored in the `COMMON` workspace might turn out not to be temporary after all. If its value is set on one call to a routine and referred to on a subsequent call, it might have been changed in between by one of the other routines sharing the workspace. Second, the practice makes the program harder to understand and therefore harder to write, debug, and maintain.

11.9.13 [E] (a) No, the construction proposed in the problem statement is not permissible. The declarations of A and B cause those arrays to be allocated by the compiler to fixed locations in memory within SUB, but the dummy parameter W might correspond to *different* actual parameters on different calls of SUB. If A and B were aligned to W as requested by the EQUIVALENCE statements, their locations in memory might have to *change* from one call of SUB to another, but Classical FORTRAN does only static memory allocation. The EQUIVALENCE statements do tell the compiler "think of A as the first half of W and B as the second half of W," but to act on them the compiler needs the actual address of W and that will not be known within SUB until run time. The program below elicits the error messages shown from gfortran.

```
      REAL*8 W(100)
      CALL SUB(W)
      STOP
      END
      SUBROUTINE SUB(W)
      REAL*8 W(100),A(50),B(50)
      EQUIVALENCE(W( 1),A(1))
      EQUIVALENCE(W(51),B(1))
      RETURN
      END
```

```
unix[1] gfortran wrong.f
wrong.f:7.23:

      EQUIVALENCE(W( 1),A(1))
                      1
Error: EQUIVALENCE attribute conflicts with DUMMY attribute in 'w' at (1)
wrong.f:8.23:

      EQUIVALENCE(W(51),B(1))
                      1
Error: EQUIVALENCE attribute conflicts with DUMMY attribute in 'w' at (1)
unix[2]
```

(b) Here, as described in textbook §11.2, is the standard (and legal) way of accomplishing the desired effect.

```
      REAL*8 W(100)
      CALL SUB(W(1),W(51))
      STOP
      END
      SUBROUTINE SUB(A,B)
      REAL*8 A(50),B(50)
      RETURN
      END
```

Now A and B, rather than being local to SUB, really are parts of W.

11.9.15 `[E]` (a) After `RTMP` and `ITMP` have been placed in `/WS/` like this

```
      SUBROUTINE ANOTHR(X,Y,N)
      REAL*8 X(N),Y(N)
C
C     share temporary workspace
      COMMON /WS/ RTMP,ITMP
      REAL*8 RTMP(5)
      INTEGER*4 ITMP(7)
```

the `COMMON` block will be $5 \times 8 + 7 \times 4 = 68$ bytes long, or longer if some other routine uses more space in it. For example, a routine that starts with the following statements could also be part of the program.

```
      FUNCTION ONEMOR(J)
C
C     share temporary workspace
      COMMON /WS/ TEXT
      CHARACTER*1 TEXT(1000)
```

In that case `/WS/` would contain 1000 bytes (maybe more if there are still other routines using it) and `RTMP` and `ITMP` would occupy the first 68 of them. The actual size of `/WS/` is determined during linkage editing as the maximum used by any routine in the program.

It is of course a basic requirement for putting these variables in shared temporary workspace that `TEXT` not be in use at the same time `RTMP` and `ITMP` are.

(b) The order in which `RTMP` and `ITMP` appear in `/WS/` *does* matter. Every `COMMON` block begins on a doubleword boundary in memory, and for reasons of efficiency it is desirable for variables in `COMMON` to be aligned on memory boundaries corresponding to their data types. Therefore, the `REAL*8` array `RTMP` should start on a doubleword boundary, and that would not happen if `ITMP` came first in `/WS/`.

If `ITMP` were an even number of words long, however, the order of the variables in `COMMON` would not matter because `RTMP` would be doubleword aligned either way.

11.9.16 `[E]` As explained in textbook §8.1, passing problem-specific data through a general-purpose subprogram makes the subprogram no longer general-purpose and thus not a candidate for inclusion in a library. That means its development costs will be amortized over only one use rather than many, so it is less likely to be correct. If the same functionality is required for a different configuration of problem-specific data in another part of the program or in a different project, the routine must be copied and re-customized, introducing replicate code (see textbook §12.2.2), which has many drawbacks.

If a variable that is only passed through has a type declaration or dimension (which it might acquire in the course of a subsequent revision even if it has neither now) then consistency demands (see textbook §12.4.5) that those attributes be given in the subprogram that passes it through and updated there whenever they are changed elsewhere. Letting a routine know about a variable means the programmer has to remember and worry about it forever after. This complicates future maintenance even more than it does the initial coding of the routine, as the programmer must repeatedly answer the question "Why is *that* here?"

The compiler needs to worry about the variable, too, which adds to compile time and might degrade optimization. The presence of a passed-through variable also lengthens the entry sequence and thus affects the run-time performance of the routine that passes it through, and the address of the passed-through variable must reside on two parameter stacks which wastes a little memory.

Finally, the presence of the passed-through variable exposes it to the risk of inadvertent alteration due to a coding error that incidentally affects its value.

Ideally all of the information sharing in any program is by means of formal parameters. However, if no program architecture can be found that avoids the need for variables passed in that way to be present in routines that do not actually need them, then `COMMON` should be used instead to pass data around as described in textbook §8.2.

11.9.17 [P] Here is an INPUT routine that meets the requirements of the Exercise.

```
      SUBROUTINE INPUT
      COMMON /LIST/ N,VALS
      INTEGER*4 VALS(100),VALUE,TEMP
C
C     read and count the integers
      OPEN(UNIT=1,FILE='values')
      N=0
    3 READ(1,*,END=1,ERR=2) VALUE
      IF(N.EQ.100) THEN
         WRITE(0,901)
  901    FORMAT('"values" contains too many lines')
         STOP
      ENDIF
      N=N+1
      VALS(N)=VALUE
      GO TO 3
C
C     sort them into increasing order
    1 IF(N.EQ.0) THEN
         WRITE(0,902)
  902    FORMAT('"values" is empty')
         STOP
      ENDIF
      DO 4 I=1,N-1
      DO 4 J=I+1,N
           IF(VALS(I).GT.VALS(J)) THEN
              TEMP=VALS(I)
              VALS(I)=VALS(J)
              VALS(J)=TEMP
           ENDIF
    4 CONTINUE
      RETURN
C
C     report bogus data
    2 WRITE(0,903) N+1
  903 FORMAT('"values" line ',I10,' is not an integer')
      STOP
      END
```

If values does not exist, the OPEN statement creates an empty file of that name. Thus the OPEN never fails, and if the file is empty the program can't tell whether it existed before the program ran. The bubble sort is slow, but it is better to be obvious than fast if there are really only 100 values as I have assumed. If a read error occurs the ERR=2 branch is taken before N gets incremented, so the number of the offending line is actually N+1.

Any routines that want to use the sorted list of numbers can include the COMMON statement and the declaration INTEGER*4 VALS(100). The number of values that were found in the file, N, is also available in the COMMON block. The problem statement says that the low-level routines sharing /LIST/ all expect the values to be in order, so it's important that none of them change VALS.

INPUT should be called exactly once.

11.9.20 [E] The given matrix is represented by storing the elements on and above its diagonal in a vector, in column-major order, like this

```
INTEGER*4 AS(15)/1,4,1,-1,7,2,0,8,9,3,6,-2,-3,-4,5/
```

The problem does not say whether the matrix is real or integer, but its entries are given as integers so that is what I have assumed. For full storage we need $5 \times 5 = 25$ elements, but using sparse storage we need only $\frac{1}{2} \times 5 \times (5+1) = 15$ elements.

11.9.21 [E] A symmetric matrix is its own transpose, so a matrix that is stored in symmetric storage mode is its own transpose.

11.9.22 [P] (a) The code on the left uses full storage mode for $\mathbf{Q}$, and (b) that on the right symmetric storage mode. The output written by each program is shown below the code.

```
      REAL*8 QF(3,3)/9*0.D0/
      REAL*8 X(3)/1.D0,2.D0,3.D0/
      DO 1 I=1,3
      DO 1 J=1,3
           QF(I,J)=QF(I,J)+X(I)*X(J)
    1 CONTINUE
      DO 2 I=1,3
           WRITE(6,901) (QF(I,J),J=1,3)
  901      FORMAT(3(1X,F5.1))
    2 CONTINUE
      STOP
      END
```

```
 1.0   2.0   3.0
 2.0   4.0   6.0
 3.0   6.0   9.0
```

```
      REAL*8 QS(6)/6*0.D0/
      REAL*8 X(3)/1.D0,2.D0,3.D0/
      L=1
      DO 1 J=1,3
      DO 1 I=1,J
           QS(L)=QS(L)+X(I)*X(J)
           L=L+1
    1 CONTINUE
      WRITE(6,903) QS(1),QS(2),QS(4)
  903 FORMAT(3(1X,F5.1))
      WRITE(6,902)       QS(3),QS(5)
  902 FORMAT(6X,2(1X,F5.1))
      WRITE(6,901)             QS(6)
  901 FORMAT(13X,F5.1)
      STOP
      END
```

```
 1.0   2.0   3.0
       4.0   6.0
             9.0
```

These programs start with $\mathbf{Q} = \mathbf{0}$ so that it is easy to see how the entries get changed by the rank-1 update, but in a real application the starting value of $\mathbf{Q}$ would of course in general be nonzero.

11.9.23 [P] If a matrix $M = [m_{i,j}]$ is stored in symmetric storage mode in the vector M then its elements can be extracted from M by using this formula.

$$m_{i,j} = \begin{cases} \mathtt{M}(i + (j-1)j/2) & i \le j \\ \mathtt{M}(j + (i-1)i/2) & i > j \end{cases}$$

The rule can be coded in a single expression as shown in the following subroutine, which prints out the full matrix.

```
      SUBROUTINE PRTSYM(M,N)
      REAL*8 M(*)
      DO 1 I=1,N
           WRITE(6,*) (M(MIN0(I,J)+(MAX0(I,J)-1)*MAX0(I,J)/2),J=1,N)
    1 CONTINUE
      RETURN
      END
```

To see how the code works consider the matrix

$$\mathtt{A} = \begin{bmatrix} 1. & 2. & 3. \\ 2. & 4. & 5. \\ 3. & 5. & 6. \end{bmatrix}$$

which is represented in symmetric storage mode by the vector A in the following program.

```
      REAL*8 AS(6)/1.D0,2.D0,4.D0,3.D0,5.D0,6.D0/
      CALL PRTSYM(AS,3)
      STOP
      END
```

When this main is compiled along with PRTSYM and run, it produces the output shown below.

```
unix[1] f77 main.f prtsym.f
unix[2] a.out
  1.  2.  3.
  2.  4.  5.
  3.  5.  6.
unix[3]
```

11.9.25 `[P]` Here is a routine that meets the requirements of the problem statement.

```
      SUBROUTINE FSTMPY(A,B,X, Y)
      REAL*8 A,B,X(4),Y(4)
      Y(1)=X(1)+B*X(3)
      Y(2)=A*X(2)
      Y(3)=B*X(1)+X(3)-X(4)
      Y(4)=-X(2)+A*X(4)
      RETURN
      END
```

The more general approach of §11.6.2 could also be used to avoid most of the unnecessary floating-point arithmetic that would be performed in a dense calculation. However, it would not avoid the 4 floating-point multiplications by 1 and -1, and it would entail a significant increase in the complexity of the `FSTMPY` routine and the additional integer arithmetic associated with the overhead of the `DO` loop and indirect array addressing. In a calculation where sparse data structures have a fixed pattern of nonzeros, special-purpose code such as that given above is always faster, and usually simpler, than code based on the approach needed when the pattern of nonzeros is varying.

11.9.30 `[E]` Indirect array addressing is the use of an array element as an array subscript. For example, `X(I(J))` uses indirect addressing to refer to the i_j'th element of `X`.

Indirect array addressing usually takes longer than direct array addressing because of the need for the machine code produced by the compiler to calculate the address of the required element in the address array (`J` in the example above) and retrieve the value of that element (`I(J)` in the example) before using that number to address the target array (`X` in the example).

Sometimes indirect array addressing can be avoided, as in the example below.

```
      INTEGER*4 I/2,4,6,8/
      REAL*8 X(8)                               REAL*8 X(8)
      :                                         :
      DO 1 J=1,4                                DO 1 J=2,8,2
           Y=Y+X(I(J))                               Y=Y+X(J)
    1 CONTINUE                                1 CONTINUE
```

However, in many situations indirect addressing is the best or only way of achieving an irregular memory reference pattern.

Indirect array addressing can involve using the *same* array for the address array and the target array, as illustrated in Exercises 5.8.3 and 5.8.4.

11.9.32 `[H]` One way for David to ensure that his data get read exactly once is for him to do that himself in the manner suggested by the following outlines for his `SUBA`, `SUBB`, and `INPUT` routines.

```
      SUBROUTINE SUBA
      COMMON /DATA/ VALUES
      REAL*4 VALUES(1000)
      CALL INPUT
      :
      code A to use VALUES
      :
      RETURN
      END
C
      SUBROUTINE SUBB
      COMMON /DATA/ VALUES
      REAL*4 VALUES(1000)
      CALL INPUT
      :
      code B to use VALUES
      :
      RETURN
      END
C
      SUBROUTINE INPUT
      COMMON /DATA/ VALUES
      REAL*4 VALUES(1000)
      LOGICAL*4 FIRST/.TRUE./
      IF(FIRST) THEN
         :
         code to read VALUES from the file
         :
         FIRST=.FALSE.
      ENDIF
      RETURN
      END
```

`INPUT` uses a compile-time initialization to set the first-call flag `FIRST` to `.TRUE.`, as described in textbook §6.5. When `LIB` invokes either `SUBA` or `SUBB` first and that routine calls `INPUT`, `INPUT` finds `FIRST` set to the initial value of `.TRUE.` and executes the code to read `VALUES` from the file. Then it sets `FIRST` to `.FALSE.`, so that all subsequent calls to `INPUT` return immediately without doing anything. `VALUES` is shared in `COMMON`, so after `INPUT` has been called the first time, whether that was by `SUBA` or `SUBB`, the data are available for use in both routines. This solution involves at least one empty call to `INPUT` but avoids the need to change either the main program or `LIB`.

11.9.33 [P] The routines below satisfy the requirements of the Exercise. To understand either it is necessary to recall that, as shown on textbook page 209, node I of a linked list is stored in row I of the array that holds the linked list.

```
      SUBROUTINE DELNOD(LIST,N,HEAD,I)
      INTEGER*4 LIST(100,2),HEAD
C
C     adjust the links
      IF(I.EQ.HEAD) THEN
C        no node links to node I; move head pointer to its successor
         HEAD=LIST(I,2)
      ELSE
C        some node J links to node I; find it
         DO 1 J=1,N
              IF(LIST(J,2).EQ.I) THEN
C                make it point to the node that node I points to
                 LIST(J,2)=LIST(I,2)
                 GO TO 2
           ENDIF
    1    CONTINUE
      ENDIF
C
C     make the deleted node point nowhere
    2 LIST(I,2)=-1
      RETURN
      END

      SUBROUTINE RELIST(LIST,N,HEAD)
      INTEGER*4 LIST(100,2),HEAD
C
C     seek a vacated node to collect
    5 DO 1 I=1,N
           IF(LIST(I,2).EQ.-1) THEN
              IC=I
C             move all the rows below this one up
              DO 2 J=I,N-1
                   LIST(J,1)=LIST(J+1,1)
                   LIST(J,2)=LIST(J+1,2)
    2         CONTINUE
C             and zero out the bottom row
              LIST(N,1)=0
              LIST(N,2)=0
              GO TO 3
           ENDIF
    1 CONTINUE
      RETURN
C
C     adjust links to rows below the collected one
    3 N=N-1
      DO 4 I=1,N
           IF(LIST(I,2).GT.IC) LIST(I,2)=LIST(I,2)-1
    4 CONTINUE
      IF(HEAD.GT.IC) HEAD=HEAD-1
      GO TO 5
      END
```

(a) To mark node I as deleted we need to set its link to -1. That will break the linked list, so before doing that we must find the node preceding node I in traversal order and make

that node link to the successor of node I. But If the deleted node I is the head node, so that there is no node preceding it in traversal order, all we need do is change HEAD to point to the successor of node I before marking node I as deleted. The DELNOD routine on the previous page does these things.

(b) Deleted nodes are not in the traversal sequence of a linked list, so to find them we must examine all of the rows of the array holding the linked list. The RELIST routine on the previous page examines the rows of LIST from top to bottom in search of a node that is deleted. If one is found, the row below it is copied up to replace it. Then the row below that is copied up, and so on until each row below has been copied over the one above it. Then the row that used to be at the bottom is set to zeros. That shortens the list by one row, and it requires any link to a row that was moved up to be reduced by one. This whole process is repeated until no deleted nodes remain.

To test the DELNOD and RELIST routines, I wrote the program on the next page, which uses the PRLIST subroutine given below.

```
      SUBROUTINE PRLIST(LIST,N,HEAD)
      INTEGER*4 LIST(100,2),HEAD
C
C     print the list in the order it is stored
      DO 1 I=1,N
           IF(I.EQ.HEAD) THEN
              WRITE(6,901) I,LIST(I,1),LIST(I,2)
  901         FORMAT('--> ',I2,1X,I2,1X,I2)
           ELSE
              WRITE(6,902) I,LIST(I,1),LIST(I,2)
  902         FORMAT('    ',I2,1X,I2,1X,I2)
           ENDIF
    1 CONTINUE
C
C     print the list in the order it is traversed
      WRITE(6,900)
  900 FORMAT(' ')
      IF(HEAD.EQ.0) STOP
      I=HEAD
    2 WRITE(6,903) LIST(I,1)
  903 FORMAT(I3,$)
      I=LIST(I,2)
      IF(I.GT.0) GO TO 2
      WRITE(6,904)
  904 FORMAT(/' ')
      RETURN
      END
```

PRLIST first prints the linked list in node order, marking the head node with an arrow. Then it traverses the list from head to tail, printing the values of the nodes in their linked order.

The program on the next page initializes a linked list having 8 nodes, and prints it out. Then it invites the user to delete nodes using DELNOD, and prints the resulting list after each deletion. When the user is finished deleting nodes (as signalled by entering ˆD) the program calls RELIST to remove the unused nodes, and prints the final list.

```
      INTEGER*4 LIST(100,2)/200*0/,HEAD
      INTEGER*4 VALUES(8)/1,-5,13,7,8,4,-2,0/
      INTEGER*4 LINKS(8)/6,7,0,5,3,4,8,1/
C
C     define a list suitable for testing
      N=8
      DO 1 I=1,8
           LIST(I,1)=VALUES(I)
           LIST(I,2)=LINKS(I)
    1 CONTINUE
      HEAD=2
      CALL PRLIST(LIST,N,HEAD)
C
C     delete a node
    2 CALL PROMPT('delete node',11)
      READ(5,*,END=3) I
      CALL DELNOD(LIST,N,HEAD,I)
      CALL PRLIST(LIST,N,HEAD)
      GO TO 2
C
C     node deletions are done; do garbage collection
    3 WRITE(6,900)
  900 FORMAT(' ')
      CALL RELIST(LIST,N,HEAD)
      CALL PRLIST(LIST,N,HEAD)
      STOP
      END
```

Output from a test run is shown below and on the next two pages. The first stanza shows the starting list, with all 8 nodes and the head at node 2, followed by the node values in traversal order.

```
unix[1] a.out
      1  1  6
-->   2 -5  7
      3 13  0
      4  7  5
      5  8  3
      6  4  4
      7 -2  8
      8  0  1

 -5 -2  0  1  4  7  8 13
```

The output stanzas on the next page show the results of deleting node 1, node 2 (which changes the head node), node 5, and node 7 (which changes the head node). Finally the user enters ˆD, which causes garbage collection to occur, and the resulting reduced list is printed out.

```
delete node 1
     1  1 -1
-->  2 -5  7
     3 13  0
     4  7  5
     5  8  3
     6  4  4
     7 -2  8
     8  0  6

 -5 -2  0  4  7  8 13

delete node 2
     1  1 -1
     2 -5 -1
     3 13  0
     4  7  5
     5  8  3
     6  4  4
-->  7 -2  8
     8  0  6

 -2  0  4  7  8 13

delete node 5
     1  1 -1
     2 -5 -1
     3 13  0
     4  7  3
     5  8 -1
     6  4  4
-->  7 -2  8
     8  0  6

 -2  0  4  7 13
```

```
delete node 7
     1  1 -1
     2 -5 -1
     3 13  0
     4  7  3
     5  8 -1
     6  4  4
     7 -2 -1
-->  8  0  6

  0  4  7 13

delete node ^D
     1 13  0
     2  7  1
     3  4  2
-->  4  0  3

  0  4  7 13

unix[2]
```

11.9.34 [E] Casablanca

11.9.36 `[H]` (a) If we are traversing the list *down,* from highest values to lowest, the head of the list is node 2, with the value `12`. It must point to the next lower value, which is the `5` at node 5, and so on. The completed list is shown below.

	node	value	up link	down link	
	1	4	5	6	
up tail ⇨	2	12	0	5	⇦ down head
up head ⇨	3	-1	4	0	⇦ down tail
	4	2	6	3	
	5	5	2	1	
	6	3	1	4	
	:				

Notice that the zero (tail) down link is in the same row as the head up link, and vice versa.

(b) Below and at the top of the next page, the code of §11.7 is revised to print the values ascending and then descending. Now `LIST` has `3` columns, and for each appended value it is necessary to update the down links as well as the up links.

```
      INTEGER*4 LIST(100,3)/300*0/,HEADUP/1/,HEADDN/1/,VALUE
      N=0
C
C     build the list
    2 READ(5,*,END=1) VALUE
      N=N+1
      CALL ADDNOD(LIST,N,HEADUP,HEADDN,VALUE)
      IF(N.LT.100) GO TO 2
C
C     print the list ascending
    1 IF(N.EQ.0) STOP
      I=HEADUP
    3 PRINT *,LIST(I,1)
      I=LIST(I,2)
      IF(I.NE.0) GO TO 3
C
C     print the list descending
      I=HEADDN
    4 PRINT *,LIST(I,1)
      I=LIST(I,3)
      IF(I.NE.0) GO TO 4
      STOP
      END
```

```
      SUBROUTINE ADDNOD(LIST,N,HEADUP,HEADDN,VALUE)
      INTEGER*4 LIST(100,3),HEADUP,HEADDN,VALUE
C
C     append the new value to the end of the list
      LIST(N,1)=VALUE
      IF(N.EQ.1) RETURN
C
C     does the new node belong before the up head?
      IF(VALUE.LE.LIST(HEADUP,1)) THEN
         LIST(N,2)=HEADUP
         HEADUP=N
      ELSE
         I=HEADUP
    2    NEXT=LIST(I,2)
         IF(VALUE.LE.LIST(NEXT,1) .OR. NEXT.EQ.0) THEN
           LIST(I,2)=N
           LIST(N,2)=NEXT
           GO TO 1
         ELSE
           I=NEXT
           GO TO 2
         ENDIF
      ENDIF
C
C     does the new node belong before the down head?
    1 IF(VALUE.GE.LIST(HEADDN,1)) THEN
         LIST(N,3)=HEADDN
         HEADDN=N
      ELSE
         I=HEADDN
    3    NEXT=LIST(I,3)
         IF(VALUE.GE.LIST(NEXT,1) .OR. NEXT.EQ.0) THEN
           LIST(I,3)=N
           LIST(N,3)=NEXT
           RETURN
         ELSE
           I=NEXT
           GO TO 3
         ENDIF
      ENDIF
      END
```

12.8.3 `[E]` (a) The Principle of Least Astonishment is a design rule requiring that a program "always do the least surprising thing." Thus, for example, user inputs that don't make sense should elicit an informative error message rather than crashing the program or causing it to perform unexpected or destructive actions.

(b) Egoless programming consists of making decisions in the design and construction of software so as to ensure that user needs are met, rather than so as to demonstrate the programmer's cleverness or express his contempt for lesser mortals. A mature and responsible programmer is instinctively egoless at work, so that only her professionalism, not her personality, is evident in the programs she writes.

(c) Data hiding is the design practice of ensuring that the data of a problem are visible only to those code segments that actually use or manipulate it. This is accomplished by dividing the calculation into subprograms and passing variables to them in such a way that each routine manipulates a logically-separate subset of the data.

(d) Successive refinement, also referred to as top-down design, is the process of decomposing the task to be performed by a program into successively smaller logically distinct subparts. Typically this results in a hierarchical program structure that can be realized with nested subprograms.

(e) Bottom-up design is a programming strategy in which top-down design is used to achieve a hierarchical program structure and then the code is written starting with the lowest-level subprograms and working up the tree to the main program.

(f) Replicate code is a sequence of operations or program statements that appears in more than one place, either within one program or in different programs. To avoid extra work in developing and maintaining software, it is preferable to replace each replication of a given non-trivial code sequence by the invocation of a subprogram in which the code sequence appears only once.

(g) Locality in source occurs when all of the code for a particular calculation is in one place and short enough to be comprehended at once. Locality in memory occurs when array storage is accessed in a pattern that allows the elements needed for a particular calculation all to be in fast memory at once.

(h) Pseudocode is a procedural description of a program written in a mixture of English (or some other natural language), mathematics, and FORTRAN (or some other programming language).

(i) Software maintenance is the process of revising a program after it has been written and is already in use, for the purpose of correcting errors, improving efficiency, or adding functionality.

(j) Often in the testing of a program, bugs are discovered after certain tests have already been passed. Regression testing is repeating all earlier tests after each bug is fixed, to make sure that fixing that bug did not introduce others.

12.8.4 [E] In addition to coding (writing comments and statements in a language such as FORTRAN) programming also entails problem formulation, user interface design, program design, writing external documentation, hand-checking finished code, testing, performance tuning, and maintenance.

12.8.5 [E] Open code is a sequence of FORTRAN statements that is not encapsulated in a subprogram. As explained in textbook §6, long stretches of repeated open code should be replaced by invocations of a subprogram that provides the same functionality for different sets of input data.

Open-source code is software whose source code is public and can be inspected by prospective users. Open-source programs are therefore available under the terms of a license agreement for free or in exchange for a voluntary contribution, and many are developed by a community of volunteer programmers rather than by any single individual or by a commercial enterprise.

12.8.6 [E] The program development steps should be performed in this order: user interface design, program design, external documentation, internal documentation, coding, hand checking, testing and revision, maintenance.

12.8.7 [E] (a) A tire suspended by a rope from a tree branch makes a dandy swing, even if it is rigged by a couple of kids who know nothing about engineering or mechanics. In contrast, the computer program used by the bank to compute your mortgage payment could be rendered completely wrong by changing a single `-` sign to a `+` sign (so that, say, the amount you owe *increased* rather than decreasing with each payment you made).

(b) This computer program contains an error that has no effect on its behavior.

```
 1  C     count from one to ten
 2        I=1
 3      1 PRINT *,I
 4        I=I+1
 5        IF(I.GT.10) STOP
 6        GO TO 1
 7        I=I+1
 8        IF(I.GT.10) STOP
 9        GO TO 1
10        END
```

Three executable statements [7-9] are unreachable because of the unconditional branch [6] before them, so their presence is an error (many production codes contain unreachable code, as illustrated by the example of textbook §13.11). Inadvertent statement repetitions like this one can result from a single errant keystroke when using an editor such as `vi`. But the program does count from 1 to 10 as advertised, and the mistake could never be found by testing. Compilers sometimes notice unreachable code, but this program elicited no complaint from `gfortran`. There are of course other sorts of error that can also have no effect on program behavior.

12.8.8 [E] The message could be changed to say `press enter when you are ready for the program to continue`.

12.8.9 [E] My (Ubuntu) version of `sort` has 24 options, a syntax rule prescribing the specification of field and character positions, and a list of acceptable suffixes for the specification of buffer size. The man page contains a warning about the `LC_ALL` environment variable (which I had never heard of) and gives alternative names for 20 of the options. Some of the options would be hard to use without knowing something about the inner workings of the program; for example, one can be used to specify another program for "compressing temporaries" but what a "temporary" is and why it might be good to compress it, the user is expected already to know. Thus, when its options are needed, `sort` becomes quite a bit more than the simple filter it first appears. Yet convincing arguments can be made for all of the functionality it provides, so it is not clear how it could be simplified. The program does have a `--help` option, which prints a useful summary of the man page.

12.8.13 [H] Here is a code segment that displays a menu of four choices and prompts for and reads the user's selection, guarding against bogus responses.

```
      :
    2 WRITE(0,901)
  901 FORMAT('Enter the number of the choice you like the best.'/
     ;       '  1 = vacation in Aruba'/
     ;       '  2 = checkout ride in a jet fighter'/
     ;       '  3 = romantic evening with your dream date'/
     ;       '  4 = three weeks of Marine basic training'/
     ;       'Choice: ',$)
      READ(5,*,END=1,ERR=2) LIKE
      IF(LIKE.LT.1 .OR. LIKE.GT.4) GO TO 2
      GO TO 3
    1 [what to do on end-of-file]
      :
    3 [what to do if a legal response is given]
```

If the user enters something other than an integer, or an integer that is not one of the allowed responses, the prompt is repeated. If the user sends an end-of-file, control transfers to the code at `1`. If the user enters one of the allowed responses, control transfers to the code at `3`.

The menu is written to unit `0` (standard-error) rather than to unit `6` (standard-out). It would not make sense for the menu to end up in a file if standard-out happens to be redirected (it is also possible that standard-error will be redirected, but far less likely). The `FORMAT` statement is laid out in such a way that someone reading the code can easily see how the menu will be displayed. The dollar-sign flag is used after the `Choice:` prompt to keep the typing cursor on the same line as the prompt. The `READ` statement uses free-format so that leading blanks in the user's response don't matter, and traps both end-of-file and errors (such as entering letters rather than numerals). The user's response is read into the variable `LIKE`, whose name was taken from the menu header. Other reasonable variable names such as `ANSWER` and `CHOICE` would need to be declared `INTEGER*4`.

A much more general solution to this problem would be to write a library subprogram that displays a menu specified in a subroutine parameter, prompts for and reads the user's response, handles error and end-of-file conditions, and returns the number of the user's menu choice. Such a routine could also present a default choice and return it if the user sends a null response, and mark alternatives as available or unavailable depending on context (similar to the way unavailable responses can be "grayed out" in a graphical user interface).

12.8.15 [E] A batch program must read all of its inputs from files and write all of its outputs to files, so the major change that must be made to an interactive program is to provide it with a way of doing that. An interactive user can observe and possibly correct error conditions, but for the program to run in the background all such eventualities must be anticipated and somehow responded to without user intervention. If this means stopping the program when an error occurs, sufficient diagnostic information must be written to a file so that the problem can be understood post-mortem. File output is frequently buffered, so it might be necessary for the program to flush its output buffers to guard against the loss of data in the event of a hardware or network problem or an unexpected program crash (e.g., an integer zero divide that was not protected against).

12.8.16 `[E]` A dense symmetric linear system with n variables has a coefficient matrix containing $\frac{1}{2}n(n+1)$ distinct values and a constant vector containing n values, so a lower bound on the memory required is $n + \frac{1}{2}n(n+1) = \frac{1}{2}n(n+3)$ storage locations. If $n = 2000$ and `REAL*8` values are used, that works out to $\frac{1}{2} \times 2000 \times 2003 \times 8 = 16024000$ bytes.

12.8.17 `[E]` Ideally a subprogram has source code about two pages (120 lines) long, including its preamble (which typically accounts for about half). Similarly, a subprogram's man page is ideally one or two pages long, including the example that shows how to invoke the routine. But both of these limits should be treated as somewhat flexible. Many routines can be much shorter than two pages, and one page is often enough for a man page. It can also happen that a natural unit of functionality is encompassed in a subprogram longer than two pages but too short (say, less than four) to warrant decomposition into smaller pieces, or that a routine's description requires more than two sides of man page (say, less than four) but not enough to justify a separate manual.

Here is a histogram of the lengths (in lines, including comments) of the general-purpose subprograms in my library. It includes 243 routines, so each dot stands for more than one.

```
   9.00 .............................................
  49.33 ..................................
  89.67 ................
 130.00 ......
 170.33 ...
 210.67 ..
 251.00 ..
 291.33 .
 331.67
 372.00
```

12.8.19 `[H]` (a) Here is the code with a preamble in the style of the textbook.

```
C
Code by Michael Kupferschmid
C
      FUNCTION STRAB(STRING,LS,TEMPLT,LT)
C     This routine returns T if STRING abbreviates TEMPLT.
C
C     variable  quantity
C     --------  --------
C     K         index on the elements of STRING and TEMPLT
C     LS        number of characters in STRING
C     LT        number of characters in TEMPLT
C     STRING    character string to be compared to TEMPLT
C     TEMPLT    template to which STRING is to be compared
C     UPCASE    function upper-cases a letter
C
      LOGICAL*4 STRAB
      CHARACTER*1 STRING(LS),TEMPLT(LT),UPCASE
C
C -----------------------------------------------------------------
C
C     a string that is empty or too long can't abbreviate template
      IF(0.LT.LS .AND. LS.LE.LT) GO TO 1
      STRAB=.FALSE.
      RETURN
C
C     compare the string to the template, ignoring case
    1 DO 2 K=1,LS
           IF(UPCASE(STRING(K)).EQ.UPCASE(TEMPLT(K))) GO TO 2
           STRAB=.FALSE.
           RETURN
    2 CONTINUE
      STRAB=.TRUE.
      RETURN
      END
```

(b) The program passes the columns of `STRNGS` to `STRAB`. The data initializations are in column-major order, so the columns of `STRAB` are `bana`, `naba`, and `banana`, so the program prints this output.

```
 T
 F
```

12.8.20 [E] Here is the `BISECT` routine of textbook §12.3.2 revised in the way described.

```
C
Code by David A. Scientist
C
      SUBROUTINE BISECT(FCN,XL,XR,TOL, X,F,RC)
C     This routine finds the value of X between XL and XR
C     where the function FCN is zero, and returns in F the
C     value of the function at that point.
C
C     RC value  meaning
C     --------  -------
C     -1        on input => write no messages
C      0        all went well
C      1        failure to converge in 100 bisections
C      2        no root in the interval
C
C     variable  meaning
C     --------  -------
C     DABS      Fortran function for |REAL*8|
C     DMAX1     Fortran function gives larger of REAL*8s
C     F         function value at current solution X
C     FCN       routine computes value of function
C     FL        function value at XL
C     FR        function value at XR
C     FTOL      F tolerance used
C     I         index on the iterations
C     MSG       T => write messages
C     RC        return code; see table above
C     TOL       tolerance vector (X,F)
C     X         midpoint of [XL,XR]
C     XL        left endpoint of interval containing root
C     XR        right endpoint of interval containing root
C     XTOL      X tolerance used
C
      REAL*8 FCN,XL,X,XR,FL,F,FR,TOL(2),XTOL,FTOL
      INTEGER*4 RC
      LOGICAL*4 MSG
C
C ------------------------------------------------------------------
C
C     set convergence tolerances
      XTOL=DMAX1(0.D0,TOL(1))
      FTOL=DMAX1(0.D0,TOL(2))
C
C     figure out whether messages are to be written
      MSG=(RC.NE.-1)
C
C     assume a root will be found
      RC=0
C
C     find the function values at the endpoints
      FL=FCN(XL)
      FR=FCN(XR)
```

```
C
C     permit only 100 bisections
      DO 1 I=1,100
C          bisect the interval
           X=.5D0*(XL+XR)
C
C          evaluate the function at the new point
           F=FCN(X)
C
C          check for convergence
           IF(DABS(XR-XL).LE.XTOL .AND. DABS(F).LE.FTOL) RETURN
C
C          decide which side the root is on and update the endpoints
           IF(F*FL .LT. 0.D0) GO TO 2
           IF(F*FR .LT. 0.D0) GO TO 3
C
C          there is no root in the interval
           IF(MSG) WRITE(0,901)
  901      FORMAT('No root on interval in BISECT')
           RC=2
           RETURN
C
C          the root is in the left half
    2      XR=X
           FR=F
           GO TO 1
C
C          the root is in the right half
    3      XL=X
           FL=F
    1 CONTINUE
C
C     convergence was not attained in 100 iterations
      IF(MSG) WRITE(0,902)
  902 FORMAT('No convergence in 100 iterations in BISECT')
      RC=1
      RETURN
      END
```

Now the `LOGICAL*4` variable `MSG` is set to `.FALSE.` if `RC` is `-1` on input or to `.TRUE.` otherwise. Then it is used to determine whether each `WRITE` statement gets executed. If an error is detected `RC` gets set to `1` in either case.

12.8.21 [E] Although the example of textbook §12.3.1 includes sections for DIAGNOSTICS, BUGS, and REFERENCES, it is possible that those would not make sense for some routines. For example, the TPVADD routine of textbook §18.4.1 has no return code and generates no diagnostics, the limitations of its applicability require no elaboration, and there is no work to cite (other than the textbook itself) for additional information about the calculation it performs. On the other hand, it might be handy to know that there is also a TPVSUB, so a SEE ALSO section would be appropriate. The table below summarizes my recommendations about man page sections.

always required	sometimes appropriate
NAME	WARNING
SYNOPSIS	FILES
DESCRIPTION	SEE ALSO
LINKAGE	DIAGNOSTICS
EXAMPLE	NOTES
	BUGS
	AUTHOR
	REFERENCES

12.8.22 [E] To begin writing the internal documentation for a routine, copy the key sentences out of the man page DESCRIPTION section and turn them into outline headings. Fill in the outline by referring to your design notes for the routine and to the references you used in devising the algorithm you will use, to complete a detailed list of the steps the routine must perform. Then turn each line of the list into a FORTRAN comment in the file where you will compose the source code. The actual coding process will consist of filling in FORTRAN statements between these stanza comments to carry out the steps that the each describes. You can put an END statement at the bottom of the file now.

At the beginning of the .f file, start the preamble of the routine. This includes an attribution of the routine's authorship, the SUBROUTINE or FUNCTION statement if it is a subprogram (based on the SYNOPSIS section of the man page), and a brief description of the routine's purpose (from the NAME section of the man page). Then come the column headings for the table of variables, which you will fill in as you write the code, followed by declarations for the variables appearing in the calling sequence or in COMMON if this is a subprogram, and a line across the page to mark the end of the preamble and the beginning of the executable code. At this stage you can fill in the variable definitions for the routine's parameters if there are any, and begin writing the code.

12.8.23 [E] A BLOCK DATA subprogram consists of a preamble followed by END, so its internal documentation should be similar to that of any other subprogram preamble: an attribution of authorship, a table of variables, and annotations on the COMMON statements, type declarations, and initializations to explain what they are for.

12.8.24 `[E]` If all variables are declared, `IMPLICIT NONE` can be used to detect misspelled variable names in the rest of the code. However, it is necessary for a reader of the code to search a long list of declarations to find the type of each variable.

If the only variables declared are those not correctly typed by default, a reader of the code needs to check only a short list of declarations to find the type of each variable or deduce that it has the default type for its name. However, `IMPLICIT NONE` cannot be used to detect misspelled variable names.

12.8.25 `[H]` The first character of a variable name may not be a numeral, so there are 26 1-character names `A-Z`. Names having more than one character can have letters or numerals in positions after the first, so the number of names having $k > 1$ characters is $26(26+10)^k$. Thus the total number of names that can be constructed using only the letters A-Z and the numerals 0-9 is $26 + 26 \times 36 + 26 \times 36^2 + 26 \times 36^3 + 26 \times 36^4 + 26 \times 36^5 = 1617038306$, or about 1.6 billion.

12.8.27 [P] Here is a program that does what the Exercise requires.

```
C
C     This program reads Fortran source text from standard-in,
C     makes comments mixed case and statements upper case,
C     and writes the result on standard-out.
C
C     variable   meaning
C     --------   -------
C     ALPHAL     lower case letters
C     ALPHAU     upper case letters
C     INSTR      T => we are inside a quoted string
C     K          index on the characters in TEST
C     L          index on the letters of the alphabet
C     LENGTH     function returns index of last nonblank in string
C     LINE       a line of text read in, changed, and written out
C     LT         the index of the last non-blank text
C     QUOTE      the character "'"
C     TEXT       the LINE as individual characters
C
      CHARACTER*72 LINE
      CHARACTER*1 TEXT(72)
      EQUIVALENCE(LINE,TEXT(1))
C
      CHARACTER*1 ALPHAL(26)/
     ;            'a','b','c','d','e','f','g','h','i','j',
     ;            'k','l','m','n','o','p','q','r','s','t',
     ;            'u','v','w','x','y','z'/
C
      CHARACTER*1 ALPHAU(26)/
     ;            'A','B','C','D','E','F','G','H','I','J',
     ;            'K','L','M','N','O','P','Q','R','S','T',
     ;            'U','V','W','X','Y','Z'/
C
C     prepare to keep track of whether we're in a quoted string
      LOGICAL*4 INSTR
      CHARACTER*1 QUOTE/Z'27'/
C
C -----------------------------------------------------------------
C
C     at the beginning we're not in a quoted string
      INSTR=.FALSE.
C
C     read a line from standard-in
    9 READ(5,900,END=1) LINE
  900 FORMAT(A72)
      LT=LENGTH(TEXT,72)
C
C     make all comments start the same way
      IF(LT.EQ.0) THEN
         TEXT(1)='C'
         LT=1
      ELSE
         IF(TEXT(1).EQ.'c') TEXT(1)='C'
         IF(TEXT(1).EQ.'*') TEXT(1)='C'
         IF(TEXT(1).EQ.'!') TEXT(1)='C'
      ENDIF
C
```

```
C     is this line a comment?
      IF(TEXT(1).EQ.'C') THEN
C        if it's a blank comment just write it out
         IF(LT.EQ.1) GO TO 2
C
C        leave mixed-case comments alone
         DO 3 K=2,LT
              DO 4 L=1,26
                   IF(TEXT(K).EQ.ALPHAL(K)) GO TO 2
    4         CONTINUE
    3    CONTINUE
C
C        upper case; translate all but first letter to lower case
         DO 5 K=2,LT
              DO 6 L=1,26
                   IF(TEXT(K).EQ.ALPHAU(L)) THEN
                      TEXT(K)=ALPHAL(L)
                      GO TO 5
                    ENDIF
    6         CONTINUE
    5    CONTINUE
         GO TO 2
      ENDIF
C
C     non-comment; if it is a non-statement leave it alone
      IF(LT.LT.7) GO TO 2
C
C     statement; translate non-string characters to upper case
      DO 7 K=7,LT
C          keep track of whether we are in or out of a string
           IF(INSTR) THEN
             IF(TEXT(K).EQ.QUOTE) INSTR=.FALSE.
           ELSE
             IF(TEXT(K).EQ.QUOTE) INSTR=.TRUE.
           ENDIF
C
C          don't translate the character if it's in a string
           IF(INSTR) GO TO 7
C
C          not in a string; translate the character
           DO 8 L=1,26
                IF(TEXT(K).EQ.ALPHAL(L)) THEN
                   TEXT(K)=ALPHAU(L)
                   GO TO 7
                ENDIF
    8      CONTINUE
    7 CONTINUE
C
C     write the altered line on standard-out
    2 WRITE(6,901) (TEXT(K),K=1,LT)
  901 FORMAT(72A1)
      GO TO 9
C
    1 STOP
      END
```

12.8.28 `[H]` (a) To understand a program containing n lines we must first consider each of the lines individually, then all potential pairwise interactions, then all potential interactions of 3 lines, and so on up to the potential interaction of all n lines. The total number of such interactions is thus

$$\binom{n}{1} + \binom{n}{2} + \cdots + \binom{n}{n} = \sum_{k=1}^{n} \binom{n}{k} = 2^n - 1$$

where the final result follows from the binomial theorem.

(b) Multiplying the number of statements in a program by x increases the number of potential interactions from $2^n - 1$ to $2^{xn} - 1$. For most programs $2^n \gg 1$ (e.g., if $n = 7$ then $2^n = 128 \gg 1$) so $(2^{xn} - 1)/(2^n - 1) \approx 2^{n(x-1)}$ and the longer program is potentially $2^{n(x-1)}$ times as hard to understand.

12.8.30 `[E]` Here is the given code segment rewritten to use names that are less confusing and that all conform to the naming rules of Classical FORTRAN. The name `SIX` must be declared `REAL*8` somewhere before the first executable statement.

```
PARAMETER(SIX=6.D0)
:
A=X(N)
I=FACT(J)
IP=I+1
K=IP*I
L=K*IP
M=K+IP
```

But the best names for these variables would be names based on the underlying application, which the given code segment does not hint at.

12.8.34 `[E]` The given replicate code could be greatly simplified by using a lowly subprogram that returns the sum of the first 3 elements of its vector argument. If the variables are `REAL*8` such a function might look like this.

```
FUNCTION SUM(V)
REAL*8 SUM,V(*)
SUM=V(1)+V(2)+V(3)
RETURN
END
```

Then the given code simplifies to this.

```
A=SUM(X)/SUM(Y)
B=SUM(Y)*SUM(W)
C=SUM(W)/SUM(Z)
D=SUM(Z)*SUM(X)
```

12.8.35 [E] The version number can be made a PARAMETER constant whose value is set in the preamble, and which is written out by name:

```
      :
      PARAMETER(VERS=3.1)
      :
C
C -----------------------------------------------------------------
C
      :
      WRITE(6,900) VERS
  900 FORMAT('This output is from program ABC, version ',F3.1)
      :
```

12.8.37 [E] An obvious flaw in the given code is that when X is zero there is a division by zero even though the value of the function is 1. Here is an improved version of the code.

```
      FUNCTION SINC(X)
      REAL*8 SINC,X
      IF(X.NE.0.D0) THEN
         SINC=DSIN(X)/X
      ELSE
         SINC=1.D0
      ENDIF
      RETURN
      END
```

12.8.38 [H] The given program, which is reproduced below, contains a typographical error.

```
      I=0
      DO 10 I=1.10
            PRINT *,I
   10 CONTINUE
      STOP
      END
```

Instead of DO 10 I=1.10, the author probably meant to type DO 10 I=1,10 With a comma the text is recognizable as a DO statement, but with a period it is not. After ignoring comment lines, the first thing FORTRAN compilers do is remove all blanks from the source text. The program above gets translated into machine code as though it had been typed in like this.

```
      I=0
      DO10I=1.10
      PRINT *,I
   10 CONTINUE
      STOP
      END
```

The name DO10I is strange but legal and it identifies, according to the default naming conventions, a REAL*4 variable. To this variable we assign the value 1.10 When I gets printed it is unchanged from its starting value of 0 Then there is a CONTINUE statement, which is also strange but legal. Even though we have never used CONTINUE except with DO, it can stand alone (see textbook §13.5.3) in which case it does nothing. When this program is compiled and run it prints out the zero value of I, rather than the list of values from 1 thru 10 that I would take on if the DO loop had been properly coded.

When the erroneous program is modified like this

```
      I=0
      DO10I=1.10
      PRINT *,I,DO10I
   10 CONTINUE
      STOP
      END
```

it prints the following output.

```
unix[1] a.out
           0   1.1000000
unix[2]
```

It is an essential feature of FORTRAN syntax that blanks *not* be significant outside of quoted strings (in parsing the source code even blanks appearing in columns 1-6 are ignored) so the mistake illustrated by this example cannot be detected by the compiler. The source code compiles without warning and runs without error, printing output different from what the programmer intended. If a mistake like this were made in a DO loop that does some calculation rather than printing things out, it might be overlooked even though it makes subsequent results incorrect. Only hand-checking, as described in §12.5, is likely to identify typos that do not render your code illegal.

12.8.39 [H] Suppose that 1.234567 is a floating-point value we calculate. If we print all of the digits, the number looks like it is correct to 7 significant figures. However, the data entering into the calculation might have been measured only approximately so the result is accurate to $\pm 10\%$. Then the true answer might be as much as $0.1 \times 1.234567 = 0.1234567$ higher or lower than the number we computed, falling in the interval $[1.1111103, 1.3580237]$. Thus we can be sure of only the first digit! We should use an F3.0 format and print ␣1. as the answer. It would be more useful to print the interval +[1.1,1.4], but that is more complicated than just specifying an output format.

If instead the value is 12.34567, then it might be wrong by as much as 1.234567 and the true answer falls in the interval $[11.111103, 13.580237]$ so again we can trust only the first significant figure. In this case we had better use a 1PE7.0 format and print ␣1.E+01 for the answer (that format will work no matter what the magnitude of the value is). Again it would be more informative, and equally truthful, to print the interval +[1.1,1.4]E+01. A routine for doing that is listed below.

```
      SUBROUTINE INTRVL(X,P)
C     This routine prints an interval containing X if the
C     value is accurate to plus or minus P percent.
C
C     variable  meaning
C     --------  -------
C     BTD       routine converts an INTEGER*4 to numerals
C     DABS      Fortran function returns |REAL*8|
C     DLOG10    Fortran function returns common log of a REAL*8
C     ES        exponent sign
C     FMT       (A1,'[',Fxx.xx,',',Fxx.xx,']E',A1,I2.2)
C     IABS      Fortran function returns |INTEGER*4|
C     IFIX      Fortran function returns INTEGER*4 for REAL*4
C     N         number of digits to print after the decimal
C     P         percent error in X
C     RC        return code; 0 => all went well
C     S         sign of the value
C     SNGL      Fortran function returns REAL*4 for REAL*8
C     T         power of ten in X
C     W         width of interval
C     X         value to be written
C     Y         X with sign and power of ten removed
C     YHI       upper end of interval
C     YLO       lower end of interval
C
      REAL*8 P,W,X,Y,YHI,YLO
      INTEGER*4 RC,T
      CHARACTER*1 ES,S
C
C     declare the object-time format template
      BYTE FMT(39)/Z'28',Z'41',Z'31',Z'2C',Z'27',Z'5B',Z'27',Z'2C',
     ;             Z'46',Z'78',Z'78',Z'2E',Z'78',Z'78',Z'2C',Z'27',
     ;             Z'2C',Z'27',Z'2C',Z'46',Z'78',Z'78',Z'2E',Z'78',
     ;             Z'78',Z'2C',Z'27',Z'5D',Z'45',Z'27',Z'2C',Z'41',
     ;             Z'31',Z'2C',Z'49',Z'32',Z'2E',Z'32',Z'29'/
C
C ----------------------------------------------------------------
C
```

```
C     remove the sign and power of ten from X
      Y=X
      IF(Y.GE.0.D0) S='+'
      IF(Y.LT.0.D0) S='-'
      Y=DABS(Y)
      T=IFIX(SNGL(DLOG10(Y)))
      Y=Y/(10.D0**T)
      IF(T.GE.0) ES='+'
      IF(T.LT.0) ES='-'
      T=IABS(T)
C
C     find the endpoints of the interval
      YLO=Y*(1.D0-0.01D0*P)
      YHI=Y*(1.D0+0.01D0*P)
C
C     figure out how many digits to print after the decimal
      W=YHI-YLO
      N=1-IFIX(SNGL(DLOG10(W)))
C
C     complete the object-time format
      CALL BTD(N+2,FMT(10),2,RC)
      IF(FMT(10).EQ.Z'20') FMT(10)=Z'30'
      CALL BTD(N,FMT(13),2,RC)
      IF(FMT(13).EQ.Z'20') FMT(13)=Z'30'
      CALL BTD(N+2,FMT(21),2,RC)
      IF(FMT(21).EQ.Z'20') FMT(21)=Z'30'
      CALL BTD(N,FMT(24),2,RC)
      IF(FMT(24).EQ.Z'20') FMT(24)=Z'30'
C
C     write the interval as +[range]E+xx
      WRITE(6,FMT) S,YLO,YHI,ES,T
      RETURN
      END
```

The compile-time initialization of FMT uses hex constants because I generated it with a program. The string could be given as a vector of CHARACTER*1s instead, but it includes quote marks and those are tricky to specify using quote marks (see textbook §10.1). The description of FMT in the variable list shows the template in readable form.

The power of ten in the difference between the endpoints (it will be negative for reasonable values of P) tells how many digits are the same in the two numbers. We want to print one more digit than that so N is 1 plus the negative of that power of ten.

One lesson of this Exercise is that a result known ±10% is not very accurate. Unfortunately, many engineering and scientific calculations use input data that are even less accurate than that! For example, the electrical resistance stated by the manufacturer for a standard resistor is only accurate to ±20%, and the most precise resistors commonly used are labeled with values accurate to only ±1%. Everyday laboratory measurements of almost any physical quantity are seldom more accurate than four significant digits, or ±0.01%. It's important to program with full precision (i.e., REAL*8) numbers to control the roundoff errors that beset all numerical methods, but in the end it is also prudent not to attach unwarranted significance to the trailing digits in the answers we find to real-life problems.

Given probability distributions for the data entering into a numerical calculation, simulation can be used to find the probability distributions of computed results. This is a difficult exercise but it can produce valuable insights into a calculation based on real-world data.

12.8.40 `[E]` According to textbook §12.5 the steps in hand-checking code are (1) make a listing of the code on paper; (2) review the logic; (3) make sure every symbol in the code appears in the variable list; (4) for each symbol in the list, answer The Three Questions:

> Does it have the right type?
> Does it have the right size?
> How does it get a value?

Hand-checking each routine is an essential step in the efficient construction of correct programs. **Are you hand-checking your code?**

13.13.2 `[H]` With 10 characters we can write programs of the form

```
I=0
STOP
END
```

Here there is room on the right of the assignment statement for only a single digit `0`...`9`, so the variable must be an integer `I`...`N`; hence there are $10 \times 6 = 60$ possible programs.

With 11 characters (4 characters on the line above `STOP`) there is still room only for an assignment statement, but now there are more possible variable names and values. If the variable has 2 characters there is only one left for the value, so the value must be a single digit as before and the variable must be an integer. Using only letters and numbers to make the name, this permits $6 \times (26 + 10) = 216$ names and 10 values, or 2160 variations. If the variable name has only one character, the value can have two. If the value is an integer, there are 90 possibilities of the form `10`...`99`, 10 of the form `+0`...`+9`, and 10 of the form `-0`...`-9`, for a total of 110, and 6 possible integer variable names, yielding a total of 660 combinations. If the value is real, it must contain a decimal point so there are 10 possibilities of the form `0.`, `1.`, etc and 10 of the form `.0`, `.1`, etc. There are 20 possible one-character non-integer variable names, yielding a total of 400 combinations. The total number of variations with 11 characters is therefore 2160+660+400=3220.

13.13.11 `[H]` Rewriting to eliminate the arithmetic `IF` and computed `GO TO` statements, we get the following equivalent code:

```
    5 IF(X.LT.0.D0) GO TO 18
      IF(X.EQ.0.D0) THEN
         X=-1.D0
         GO TO 18
      ENDIF
      IF(L.EQ.1) THEN
         IF(X.LE.1.D0) GO TO 18
         STOP
      ENDIF
      IF(L.EQ.2) THEN
         X=X-1.D0
         GO TO 5
      ENDIF
      IF(L.EQ.3) STOP
      PRINT *,L
   18 ...
```

If transformations beyond those specified in the problem statement are permitted, the first five lines of this code could be replaced by these two.

```
    5 IF(X.EQ.0.D0) X=-1.D0
      IF(X.LT.0.D0) GO TO 18
```

Now if `X` is less than zero at statement `5` control is still transferred to statement `18`; if `X` is equal to zero it still gets set to `-1.D0` and control still transfers to statement `18`.

These simplifications taken together do not result in a code sequence whose function comes close to being obvious. In such a situation you should feel complelled to question whether this complexity is an essential feature of the underlying problem or an artifact of its expression in a clumsy algorithm.

13.13.32 `[H]` Here is a version of `EXCH` that exchanges the values of its dummy parameters correctly even if the same actual parameter is passed for both `K` and `L`.

```
      SUBROUTINE EXCH(K,L)
      INTEGER*4 K(2),L(2),TEMP(2)
      TEMP(1)=K(2)
      TEMP(2)=K(1)
      L(1)=TEMP(1)
      L(2)=TEMP(2)
      RETURN
      END
```

```
unix[1] a.out
 2 1
 2 1
unix[2]
```

Copying `K` into `TEMP` saves those values (and switches their order) so that when `L` is replaced the correct values are used even if `K` and `L` were aliased in the `CALL`. When this routine is linked with the first main program in §6.2 of the text and run, it produces the output shown on the right, demonstrating that the new version of `EXCH` works.

14.8.17 [H] Using either g77 or a Sun compiler, the given program does *not* compile if the main and SUB are in the same file. Inside SUB, MOD is not dimensioned so it looks like a function reference, and the compiler objects to the function name being passed in but not declared EXTERNAL.

When the two routines are in different .f files the program compiles and links, but when the executable is run it elicits a run-time error (illegal instruction on the Sun and segmentation fault on the Linux machine). This is because SUB tries to invoke the function whose name is passed in. That address contains not machine instructions but data. MOD is the name of a built-in FORTRAN function (MOD(3,2) yields 1) but that does not seem to confuse either compiler.

14.8.18 [E] (a) Everyone should want to use make, to save time by recompiling only those parts of a program that have changed and to save trouble by coding the long-winded details once and for all in the Makefile rather than typing them interactively each time the program is to be built.

(b) It is based on the last change times of the dependencies that make decides whether to perform the rule associated with a target. If the files that are used in building the target have been modified since the target was last built, then the target needs to be rebuilt.

(c) To force make to update a target we can either remove the target or touch one of the files on which the target depends to make its last change time more recent than that of the target. Removing the target entails the risk of accidentally removing some precious other file instead because of a typing error, and changing the last change date of a dependency when the file didn't actually change makes it impossible to know when the file *did* actually change last, so some programmers are leery of both alternatives. Another approach is to make one of a target's dependencies (which might be its only dependency) an empty file, which we can touch before makeing the target.

14.8.19 [E] If draw.f invokes a subprogram in the library /usr/jones/lib/graphics then we can compile and link the program by issuing this UNIX™ command:

```
f77 draw.f -L/usr/jones/lib -lgraphics
```

14.8.20 [E] The ar program inserts and replaces members (.o files) in an archive (.a) file (and can also do other things). The ranlib program builds a table of contents for an archive file, so that its members can be accessed at random.

14.8.21 [E] (a) The text processing language used for writing UNIX™ man pages is troff. (b) Listings and other tabular material can be included in a man page (or other troff document) by using tbl, and mathematics can be included by using eqn.

14.8.22 `[E]` (a) The `catman` program builds a `whatis` database for use by `man -k` (also known as `apropos`). (b) Katmandu, the capital city of Nepal, is located at latitude 27.7° north and longitude 85.3° east.

14.8.23 `[E]` David has probably neglected to list the directory containing Sarah's `man` pages in his `MANPATH` shell variable.

14.8.24 `[E]` The `-k` option of the `man` command causes it to list all of the man pages in the user's `MANPATH` whose "One line description" (see textbook pages 349-351) contains a specified word or phrase. Here is an example of its use.

```
unix[1] man -k inver
dft (3)             - Return in-place the direct or inverse discrete Fourier transform of a sequence.
dft2 (3)            - Return in-place the direct or inverse discrete Fourier transform of a matrix.
fft (3)             - Return in-place the direct or inverse fast Fourier transform of a sequence.
fft2st (3)          - Return in-place the direct or inverse fast Fourier transform of a matrix.
fft2tr (3)          - Return in-place the direct or inverse fast Fourier transform of a matrix.
hist (3)            - Compute sample probability density and inverse cumulative distribution from data.
acosh (3)           - inverse hyperbolic cosine function
acoshf (3)          - inverse hyperbolic cosine function
acoshl (3)          - inverse hyperbolic cosine function
asinh (3)           - inverse hyperbolic sine function
asinhf (3)          - inverse hyperbolic sine function
asinhl (3)          - inverse hyperbolic sine function
atanh (3)           - inverse hyperbolic tangent function
atanhf (3)          - inverse hyperbolic tangent function
atanhl (3)          - inverse hyperbolic tangent function
indxbib (1)         - make inverted index for bibliographic databases
pnminvert (1)       - invert a portable anymap
timegm (3)          - inverses of gmtime and localtime
timelocal (3)       - inverses of gmtime and localtime
unix[2]
```

The `-t` option of the `man` command typesets a `man3` file for output to the device specified in the `TCAT` shell variable.

15.4.1 `[E]` Here are some ways that a program's execution time can be reduced without tuning its source code: use a faster computer; use compiler optimization; improve the algorithm; use high-quality library subprograms. Vector processing typically involves some tuning of the FORTRAN source code to facilitate vectorization; see §16.1.5 and §16.1.6 of the text. Parallel processing involves rewriting the code to insert `MPI` subroutine calls, and hand-coding in assembler language amounts to rewriting the FORTRAN source code into a different language entirely.

15.4.2 [E] The expansion factor is the total wallclock time that elapses in running a program divided by the CPU time that it consumes (see textbook page 365). In the example of §15.1.1 the `time` program reports

```
unix[113] time a.out
 17.6 real        15.9 user         0.6 sys
```

so the wallclock time elapsed was 17.6 seconds and the CPU time consumed was 15.9 seconds, for an expansion factor of $17.6/15.9 \approx 1.1$

15.4.3 [E] Code tuning is difficult and fraught with grief, so we want to focus our attention on those (hopefully few) parts of a program that use most of the CPU time. Profiling identifies those parts.

15.4.4 [H] Here is the `ETIME` example program of §15.1.3 modified to report total (user + system) CPU time.

```
      REAL*4 TARRAY(2)
      TSTART=ETIME(TARRAY)
        [code segment to be timed]
      TSTOP=ETIME(TARRAY)
      WRITE(6,901) TSTOP-TSTART
  901 FORMAT('Total CPU time = ',F6.2,' seconds')
      STOP
      END
```

15.4.5 [H] From §4 we recall that the largest value representable in an `INTEGER*4` is $2^{31}-1 = 2147483647$, so if each count represents a microsecond we can store times up to 2147.483647 seconds or about 36 minutes. A two-part value can store this time in its microseconds component, and an additional 2147483647 seconds in its seconds component, for a maximum value of 2147485794.483647 seconds or a little over 68 years. Storing the seconds value in a `REAL*8` preserves all 16 digits of this value (the floating-point representation is not exact, but gives the correct last digit when rounded to 16 digits).

15.4.7 [H] The number of comparisons is

$$(\mathtt{N}-1)+(\mathtt{N}-2)+\cdots+1=\sum_{i=1}^{\mathtt{N}-1} i=\frac{1}{2}\mathtt{N}(\mathtt{N}-1)$$

The number of exchanges depends on the data and could range from 0 to $\frac{1}{2}\mathtt{N}(\mathtt{N}-1)$. Because the number of exchanges depends on the data, it is not a useful measure of the amount of work in this program.

15.4.12 `[H]` The approximation is exact for constant, linear, and quadratic functions (see §25.6.2 in my *Introduction to Mathematical Programming*). It is easy to write down a function for which the approximation takes *more* work than evaluating the closed-form expression for the derivative; for example,

$$\frac{d}{dx}(2x+3) = 2$$

requires no work at all to evaluate, but approximating the derivative requires an addition, two subtractions, two function evaluations (each a multiplication and an addition), and a division (if 2δ is stored) or another multiplication (if $1/2\delta$ is stored). If it costs more than twice as much to evaluate the formula for $f'(x)$ as it does to compute a single value of $f(x)$, then using the approximation instead might take *less* work. For example,

$$\text{if} \quad f(x) = \sqrt{x}^{(\sqrt{x})} \quad \text{then} \quad f'(x) = \tfrac{1}{2}\sqrt{x}^{(\sqrt{x}-1)}(1 + \tfrac{1}{2}\ln(x)).$$

Finding the logarithm in the derivative is about as much work as raising one real number to another, so evaluating the formula for $f'(x)$ at one value of x might cost more than evaluating $f(x)$ at two different values of x.

15.4.15 [H] Testing whether `X**0.5D0` takes longer than `DSQRT(X)` is tricky because the compiler might recognize the exponent as one half and generate a call to the square root function. To rule out that possibility, we can use a variable for the exponent and read in its value at run time. These functions are really fast, so in order to measure reliable times for them, even using `TIMER`, we must time loops and divide by the number of repetitions.

```
      COMMON /EXPT/ TIMING,NONALG,NBIN,TOPBIN,NTMEAS,TOH,BINCPU
      LOGICAL*4 TIMING
      INTEGER*4 TOPBIN,TOH,BINCPU(2,22)
      REAL*8 E,X,T,TZ
      PARAMETER(NREPS=100)
C
C     read the value 0.5D0 for E, and initialize for timing
      READ(5,*) E
      TIMING=.TRUE.
      CALL TIMER(-1,1)
C
C     time evaluations of X**0.5D0
      CALL TIMER(1,2)
      X=2.D0
      DO 1 I=1,NREPS
           X=X**E
    1 CONTINUE
C
C     time invocations of DSQRT(X)
      CALL TIMER(2,3)
      X=2.D0
      DO 2 I=1,NREPS
           X=DSQRT(X)
    2 CONTINUE
C
C     time a loop that only makes the assignments
      CALL TIMER(3,4)
      X=2.D0
      DO 3 I=1,NREPS
           X=2.D0
    3 CONTINUE
C
C     report results
      CALL TIMER(NONALG,5)
      CALL TIMER(0,6)
      TZ=1.D+06*DFLOAT(BINCPU(1,3))+DFLOAT(BINCPU(2,3))
      T=1.D+06*DFLOAT(BINCPU(1,1))+DFLOAT(BINCPU(2,1))-TZ
      WRITE(6,901) T/DFLOAT(NREPS)
  901 FORMAT('X**E time =',1PD23.16,' us per call')
      T=1.D+06*DFLOAT(BINCPU(1,2))+DFLOAT(BINCPU(2,2))-TZ
      WRITE(6,902) T/DFLOAT(NREPS)
  902 FORMAT('DSQRT(X) time =',1PD23.16,' us per call')
      STOP
      END
```

When I compiled this program and ran it three times on my Linux laptop it produced the output shown at the top of the next page. The results from the three runs are consistent so the experiment shows each `X**0.5D0` taking about 1700 microseconds and each `DSQRT(X)` invocation taking about 60 microseconds. These findings are consistent with the recommendations on page 378 of the textbook.

```
unix[1] ftn sqrttime.f
unix[2] a.out
0.5D0
TIMER assumes processor speed is  800.000 MHz.
X**E time = 1.7050000000000000D+03 us per call
DSQRT(X) time = 5.5930000000000000D+01 us per call
unix[3] a.out
0.5D0
TIMER assumes processor speed is  800.000 MHz.
X**E time = 1.6928099999999999D+03 us per call
DSQRT(X) time = 6.0619999999999997D+01 us per call
unix[4] a.out
0.5D0
TIMER assumes processor speed is  800.000 MHz.
X**E time = 1.6725000000000000D+03 us per call
DSQRT(X) time = 5.5619999999999997D+01 us per call
unix[5]
```

15.4.19 [H] In the given code segment the elements of B are accessed at a stride of 7 doublewords. Exchanging the loops yields the code on the left where B is accessed at a stride of 1 doubleword.

```
      REAL*8 B(5,7),S
      :
      S=0.D0
      DO 1 I=1,7
           DO 2 J=1,5
                S=S+B(J,I)
    2      CONTINUE
    1 CONTINUE
```

```
      REAL*8 B(5,7),S
      :
      S=0.D0
      DO 1 I=1,7
           S=S+B(1,I)
           S=S+B(2,I)
           S=S+B(3,I)
           S=S+B(4,I)
           S=S+B(5,I)
    1 CONTINUE
```

This rearrangement assumes that the the accumulation of roundoff error is not significantly affected by changing the order in which the elements are added. The original nest required $1 + 5 = 6$ loop initializations, whereas this arrangement has $1 + 7 = 8$. The stride-1 nest will be faster if the improvement in memory reference pattern more than offsets the extra loop initializations. Unrolling the inner loop yields the code on the right above, which has only one loop initialization.

15.4.21 [E] The code segment uses an unformatted WRITE but transmits the array elements individually. Using an array operand will produce the same output faster.

```
      REAL*8 A(300,200)
      :
      WRITE(3) A
```

17.4.1 `[E]` (a) The new *statements* introduced in §17.1 are `SAVE`, `ALLOCATABLE`, `ALLOCATE`, `DEALLOCATE`, `INTERFACE`, and `END INTERFACE`. Adding these 5 to the 32 statements of Classical FORTRAN yields a combined census of 37, which is 44% of the 85 statements in full Fortran-90. (b) Three new *statements* are introduced in §17.2: `FORALL`, `EXTRINSIC`, and `PURE`. Six *directives* are introduced in §17.2: `PROCESSORS`, `DISTRIBUTE`, `ALIGN`, `INDEPENDENT`, `NEW`, and `SEQUENCE`.

17.4.2 `[E]` According to the introduction of textbook §17.1, these are the things that might keep a Classical FORTRAN program from working in Fortran-90:

- Fortran-90 compilers are not required to recognize Hollerith constants.
- Fortran-90 compilers are not required to recognize `$` for hanging prompts.
- A `SAVE` statement is required to guarantee that a local variable in a subprogram keeps its value from one invocation to the next.
- A Classical FORTRAN program might unwittingly use subprogram or `COMMON` block names that are the same as those of Fortran-90 built-in functions.

17.4.6 `[E]` (a) The parameter passed to `SUB` is `-X` but `X` is an array, so `g77` complains that the arithmetic operator (`-`) can only operate on a scalar. In Fortran-90 the unary minus can operate on the array variable `X`, yielding an array of the same dimensions each element of which is the negative of the corresponding element in `X`.

(b) When the given program is compiled with a Fortran-90 compiler and run it produces the following output.

```
unix[1] a.out
 -1.00000000000000000 -2.00000000000000000
unix[2]
```

17.4.8 `[E]` The array expression `1.D0/A` evaluates to a matrix the same size as `A` having each of its elements equal to the reciprocal of the corresponding element in `A`. This usually is not the same as the inverse of the matrix `A`.

The array expression `A*B` evaluates to a matrix the same size as `A` and `B` (which must be of equal size) and having each of its elements equal to the product of the corresponding elements in `A` and `B`. This usually is not the same as the matrix product of `A` with `B`.

The Fortran-90 built-in function `MATMUL` performs matrix multiplication.

17.4.9 [E] In Fortran-90 a FUNCTION subprogram can return an array result, and then an INTERFACE block *must* be used in any routine that invokes the subprogram to show the dimensions of the result array that the FUNCTION will return. An INTERFACE block *may* also be used in an invoking routine to provide the dimensions of array arguments passed to a subprogram, so that the dimensions need not be passed to the subprogram.

17.4.10 [H] An array assignment can replace an element in the left-hand side array more than once, as in this program.

```
      INTEGER*4 INDX(3)/1,1,2/,A(3)/-1,-1,-1/,B(3)/1,2,3/
      A(INDX)=B
      PRINT *,A
      STOP
      END
```

Here A(1) gets replaced twice, so that A ends up containing either [1,3,-1] or [2,3,-1] depending on which assignment into A(1) gets done last. This ambiguity violates the semantics of array assignment, in which all of the result elements are thought of as changing at once, so the assignment statement given above is wrong. When the program is compiled using either a Sun or an IBM Fortran-90 compiler and run, no error is reported and the following output is produced.

```
unix[1] a.out
 2 3 -1
unix[2]
```

17.4.11 [P] In the program below `MATPOW` is an array-valued function that returns the `K`th power of `A`.

```
      REAL*8 A(2,2)/1.D0,0.D0,1.D0,1.D0/,B(2,2)
      INTERFACE
        FUNCTION MATPOW(A,N,K)
        REAL*8 MATPOW(2,2),A(2,2)
        END
      END INTERFACE
      B=MATPOW(A,2,5)
      DO 1 I=1,2
           WRITE(6,901) (B(I,J),J=1,2)
  901      FORMAT(1X,F3.0,1X,F3.0)
    1 CONTINUE
      STOP
      END
C
      FUNCTION MATPOW(A,N,K)
      REAL*8 MATPOW(N,N),A(N,N),TEMP(N,N)
      IF(K.GT.1) THEN
         TEMP=A
         DO 1 J=1,K-1
              A=MATMUL(A,TEMP)
    1    CONTINUE
      ENDIF
      MATPOW=A
      RETURN
      END
```

The work array `TEMP` is an automatic array. To test `MATPOW` this main program uses a matrix `A` whose kth power is easy to figure out by hand.

$$\mathtt{A}^k = \begin{bmatrix} 1 & 1 \\ 0 & 1 \end{bmatrix}^k = \begin{bmatrix} 1 & k \\ 0 & 1 \end{bmatrix}.$$

The main program asks for the 5th power of `A`. When the main and function are compiled together and run, the following output is produced.

```
unix[1] a.out
  1.  5.
  0.  1.
unix[2]
```

17.4.12 [P] In the program below TRACE is a FUNCTION that computes the trace of a matrix by using array operations instead of a DO loop.

```
      REAL*8 A(3,3)
      DO 1 I=1,3
      DO 1 J=1,3
           A(I,J)=DFLOAT(I*J)
    1 CONTINUE
      DO 2 I=1,3
           WRITE(6,901) (A(I,J),J=1,3)
  901      FORMAT(3(1X,F5.0))
    2 CONTINUE
      WRITE(6,902) TRACE(A,3)
  902 FORMAT(' TRACE=',F5.0)
      STOP
      END
C
      FUNCTION TRACE(A,N)
      REAL*8 TRACE,A(*)
      TRACE=SUM(A(1:N*N:N+1))
      RETURN
      END
```

The main program puts some numbers in A, writes out the matrix, and then prints its trace. Because Fortran-90 stores array elements in contiguous locations in column-major order, we can treat A as a vector in TRACE. The array section A(1:N*N:N+1) consists of the diagonal elements (for this main program, A(1), A(5), and A(9)). The built-in Fortran-90 function SUM adds up those elements to produce the trace of the matrix.

When the program above is compiled with a Fortran-90 compiler and run it produces the following output.

```
unix[1] a.out
    1.    2.    3.
    2.    4.    6.
    3.    6.    9.
 TRACE=  14.
unix[2]
```

The A that the program generates has diagonals that add to 14.D0 so the correct value is returned by TRACE.

17.4.14 [H] Here is a program that illustrates one way of performing the calculation.

```
      REAL*8 A(3,3),X(3)
      DO 1 I=1,3
      DO 1 J=1,3
           A(I,J)=DFLOAT(I*J)
    1 CONTINUE
      X=0.D0
      DO 2 J=1,3
           X=X+A(:,J)
    2 CONTINUE
      DO 3 I=1,3
           WRITE(6,901) (A(I,J),J=1,3),X(I)
  901      FORMAT(3(1X,F5.0),6X,F5.0)
    3 CONTINUE
      STOP
      END
```

When the program is compiled with a Fortran-90 compiler and run, it produces the following output.

```
unix[1] a.out
    1.    2.    3.          6.
    2.    4.    6.         12.
    3.    6.    9.         18.
unix[2]
```

17.4.16 [E] If a program solves problems that vary in size, it is necessary with fixed memory allocation to dimension arrays for the largest problem anticipated (as discussed at length in §7.1 of the textbook). Using dynamic memory allocation allows arrays to be dimensioned at run time based on the size of the problem to be solved.

If a subprogram uses an array that varies in size from one invocation to the next, it is necessary with fixed memory allocation for the storage it occupies to be declared at fixed size in the calling routine and the address passed as a subprogram parameter. Using dynamic memory allocation (in the form of a Fortran-90 automatic array) allows such workspace to be allocated locally upon entry to the subprogram and deallocated upon return, which simplifies the calling sequence of the subprogram.

Dynamic memory allocation *cannot* be used to set the rank of an array (the number of dimensions) at run time. The rank of a dynamically-allocated array is the number of colons given in its `ALLOCATABLE` statement. Dynamic memory allocation can only be used to set the size (number of elements) in each dimension.

17.4.17 [H] Here is the code of §17.1.2 modified so that the main program calls SUB twice but the array A is allocated within DOT only once.

```
      REAL*8 DOT(2)
      CALL SUB(DOT)
      PRINT *,DOT
      CALL SUB(DOT)
      PRINT *,DOT
      STOP
      END
C
      SUBROUTINE SUB(DOT)
      SAVE
      ALLOCATABLE A(:)
      REAL*8 DOT(2),A
      IF(.NOT.ALLOCATED(A)) THEN
          READ *,N
          PRINT *,'allocating A(',N,')'
          ALLOCATE(A(N))
      ENDIF
      READ *,A
      DOT(1)=0.D0
      DO 1 I=1,N
           DOT(1)=DOT(1)+A(I)**2
    1 CONTINUE
      DOT(2)=DSQRT(DOT(1))
      RETURN
      END
```

When the program is compiled and run it produces the following output, showing that A is allocated only once.

```
unix[1] a.out
2
 allocating A( 2 )
3 4
 25.0000000000000000 5.00000000000000000
5 6
 61.0000000000000000 7.81024967590665398
unix[2]
```

If we wanted to be able to reset N each time SUB is called than it would be necessary to deallocate and reallocate A every time N increased.

17.4.19 [E] (a) Array A is dynamically allocated with a scope consisting of the main program, SUB1, and SUB2.
(b) Array B is dynamically allocated with a scope consisting of the main program only (where it is never actually used).
(c) Array C is statically allocated with a scope consisting of the main program and SUB1.
(d) Array D is dynamically allocated with a scope consisting of SUB1 and SUB2.
(e) Array E is dynamically allocated with a scope consisting of SUB2.

17.4.24 [E] Here is a program that does the job.

```
      READ *,I
      WRITE(6,901) I
  901 FORMAT(B32.32)
      STOP
      END
```

When it is compiled and run it produces output like that shown below.

```
unix[1] a.out
123456789
00000111010110111100110100010101
unix[2] a.out
-123456789
11111000101001000011001011101011
unix[3]
```

The program prints the bits of the binary representation, so if the input number `I` is negative it prints the 2's complement of `-I`, including the sign bit of `1`.

17.4.26 [E] The given code is a legal Fortran-90 program. Concatenating the statements makes it impossible to use indentation to show the structure of the loop, and that makes the program more difficult to understand than it would be if it were written with one statement per line.

17.4.27 [E] The given statement is legal in Fortran-90, and when used in the program below produces the output shown.

```
      B=3.D0
      A=2.D0
      C=1.D0
      X=(-B+DSQRT(B**2-4.D0*A*C)) ! this is the numerator
     ;  /(2.D0*A)                 ! this is the denominator
      PRINT *,X
      STOP
      END
```

```
unix[1] a.out
 -0.5000000000
unix[2]
```

This result is correct, because $(-3+\sqrt{3^2-4\times 2\times 1})/(2\times 2) = -\frac{1}{2}$.

17.4.28 [E] (a) The Fortran-90 features used in the given program are end-of-line and ! comments, the use of ; to concatenate multiple statements on a single line, the use of double quotes " to enclose a character string in the FORMAT statement, and the use of a B format field descriptor to write out the sum in binary. (b) The statement number "1" referred to in the GO TO is buried among several ! comment markers in the first column. (c) Here is the program rewritten to improve its typographical layout.

```
      INTEGER*4 SUM/0/ !initialize
!
!     get the next input value
    1 READ(5,*,END=2) I
        SUM=SUM+I
      GO TO 1
!
!     output total
    2 WRITE(6,901) SUM
  901 FORMAT("Sum=",B32.32)
      STOP
      END
```

17.4.29 [E] Here is a Fortran-90 program that prints the required output.

```
      WRITE(6,901)
  901 FORMAT("This line contains both single '",
     ;       ' and double " quotes.')
      STOP
      END
```

When it is compiled and run it produces the following output.

```
unix[1] a.out
This line contains both single ' and double " quotes.
unix[2]
```

17.4.31 [P] Here is a program that uses the Fortran-90 features mentioned in the problem statement to do the calculation required for Exercise 5.8.33:

```
      ALLOCATABLE U(:,:),UOLD(:,:)
      REAL*8 U,UOLD
      REAL*8 EPS/0.001D0/,DIFF
!
!     get the dimension and boundary values from the user
      WRITE(0,901,ADVANCE='NO')
  901 FORMAT('N=')
      READ(5,*) N
      ALLOCATE(U(N,N),UOLD(N,N)) ! just big enough
      U=0.D0
      UOLD=0.D0
      READ(5,*) U(1,:),U(N,:),U(2:N-1,1),U(2:N-1,N)
!
!     iterate to find steady-state temperature distribution
      K=0
    2 K=K+1
      DO 1 J=2,N-1
      DO 1 I=2,N-1
           U(I,J)=0.25D0*(SUM(U(I-1:I+1:2,J))+
     ;                    SUM(U(I,J-1:J+1:2)))
    1 CONTINUE
      DIFF=MAXVAL(DABS(U-UOLD))
      UOLD=U
      IF(DIFF.GT.EPS) GO TO 2
!
!     report results
      WRITE(6,902) K
  902 FORMAT("after",I3,' iterations')
      DO 3 I=1,N
           WRITE(6,903) U(I,:)
  903      FORMAT(15(1X,F4.1))
    3 CONTINUE
      DEALLOCATE(U,UOLD)
      STOP
      END
```

17.4.32 [E] (a) According to textbook §17.2.1, data parallel calculations are characterized by operation-level parallelism (i.e., they are SIMD), single thread of control (the operations can be thought of as taking place sequentially even though the results will actually be found in parallel), single logical memory (the data can be thought of as present everywhere even though each processor will actually use only part of it), and intermittent synchronization (while each processor is doing its part of the calculation it does not need to coordinate with the others).

(b) Decomposing a problem in a data-parallel way consists of "finding an array whose elements can be assigned in parallel and dividing that work, along with the data it requires, among the processors."

17.4.33 `[E]` HPF can be used to parallelize only calculations that are data-parallel, so MPI can be used to parallelize a wider variety of calculations. HPF is easier to use so it shortens program development time, but the speedup achieved using MPI is typically better. Error conditions are easier (though often not easy) to understand when they occur in an MPI program.

17.4.34 `[E]` The HPF directives discussed in textbook §17.2 are as follows. `ALIGN` tells HPF which data items must be present together on a processor. `DISTRIBUTE` tells HPF which result elements are to be computed on which processors. `INDEPENDENT` tells HPF that the iterations of a `DO` loop or `FORALL` can be done at the same time on different processors. The `NEW` clause of `INDEPENDENT` tells HPF that a variable is local to each iteration of a loop and can vary within the iterations independently. `PROCESSORS` tells HPF how many processors to use and how to think of them as being arranged. `SEQUENCE` tells HPF that an array is to be treated as in Classical Fortran rather than distributed.

17.4.35 `[E]` An array that is not distributed, or aligned with an array that is distributed, is copied by HPF to all of the processors.

17.4.38 [H] The first distribution of B to the 3 processors yields the following assignment of processors to array elements.

```
REAL*8 B(3,3)
PROCESSORS PROCS(3)
DISTRIBUTE B(CYCLIC,CYCLIC) ONTO PROCS
```

$$\left[\begin{array}{ccc} (1) & (2) & (3) \\ (2) & (3) & (1) \\ (3) & (1) & (2) \end{array}\right]$$

For example, processor 1 is responsible for finding B(1,1), B(3,2), and B(2,3). The loop calculates

```
B(1,1)=A(1,1)+A(1,2)+A(2,1)+A(2,2)
B(3,2)=A(3,2)+A(3,3)+A(4,2)+A(4,3)
B(2,3)=A(2,3)+A(2,4)+A(3,3)+A(3,4)
```

so processor 1 needs to know 11 elements of A to find the B elements allocated to it (A(3,3) is used twice). Because most of A is needed on each processor it might as well be present everywhere rather than distributed piecemeal, so it is not DISTRIBUTEd or ALIGNed. That means 16 elements are sent to each of 3 processors for each of the N different values of A, for a total of 48N values transmitted.

The second distribution of B yields this assignment of processors to array elements.

```
REAL*8 B(3,3)
PROCESSORS PROCS(3)
DISTRIBUTE B(*,BLOCK) ONTO PROCS
```

$$\left[\begin{array}{ccc} (1) & (2) & (3) \\ (1) & (2) & (3) \\ (1) & (2) & (3) \end{array}\right]$$

For example, processor 1 is responsible for finding B(1,1), B(2,1), and B(3,1). The loop calculates

```
B(1,1)=A(1,1)+A(1,2)+A(2,1)+A(2,2)
B(2,1)=A(2,1)+A(2,2)+A(3,1)+A(3,2)
B(3,1)=A(3,1)+A(3,2)+A(4,1)+A(4,2)
```

Now processor 1 needs to know only 8 elements of A to find the B elements allocated to it (several elements are used more than once). The ALIGN directives send columns J and J+1 of A to the processor responsible for computing column J of B, or 8 elements of A to each processor for a total of 8N values transmitted.

This exercise shows that the distribution of data can have an important effect on how much communication is required.

17.4.39 `[H]` The loop makes the following sequence of assignments.

```
X(1)=DSIN(ALPHA(1))
X(2)=X(1)
X(2)=DSIN(ALPHA(2))
X(3)=X(2)
X(3)=DSIN(ALPHA(3))
X(4)=X(3)
:
X(N-1)=DSIN(ALPHA(N-1))
X(N)=X(N-1)
```

Each vector element set by `X(I+1)=X(I)` is immediately overwritten in the next pass of the loop, so all of those assignments could be removed except for the last one. Thus the code segment can be rewritten as shown below.

```
      DO 1 I=1,N-1
           X(I)=DSIN(ALPHA(I))
    1 CONTINUE
      X(N)=X(N-1)
```

Now there is no data dependence.

17.4.40 `[H]` The loop makes the following sequence of assignments.

```
K(1)=K(1)+1        !   =  1
K(2)=K(1)+1        !   =  2
K(2)=K(2)+2        !   =  4
K(3)=K(2)+1        !   =  5
K(3)=K(3)+3        !   =  8
K(4)=K(3)+1        !   =  9
K(4)=K(4)+4        !   = 13
K(5)=K(4)+1        !   = 14
K(5)=K(5)+5        !   = 19
K(6)=K(5)+1        !   = 20
:
K(999)=K(998)+1
K(999)=K(999)+999
K(1000)=K(999)+1
```

The result is to assign these values to `K`.

```
K(1)= 1
K(2)= 4
K(3)= 8
K(4)=13
K(5)=19
:
K(999)=500498
K(1000)=500499
```

Here is the given code segment rewritten to do that without a data dependence.

```
      INTEGER*4 K(1000)/1000*0/
      DO 1 I=1,999
           K(I)=(I+1)*(I+2)/2 - 2
    1 CONTINUE
      K(1000)=K(999)+1
```

17.4.41 `[H]` A control dependence occurs when the result of one iteration of a loop affects the flow of control in another iteration. The first test in the given code segment does not introduce a control dependence, because the only element of `X` that might get changed is the one that was tested. The second test does introduce a control dependence, because if the condition is satisfied then `X(I+1)` is set to `0.D0` and that affects what happens in the next iteration. Because of this the iterations must be performed in sequence and the loop cannot be parallelized.

However, from calculus we know that the exponential function is *never* negative, so the condition `(DEXP(X(I)) .LT. 0.D0)` will never be satisfied and the code can be rewritten like this without a control dependence.

```
      DO 1 I=1,N
           IF(X(I).GE.0.D0) X(I)=X(I)-1.D0
    1 CONTINUE
```

17.4.42 [P] (a) In the algorithm for finding a sum in parallel, the number of partial sums is reduced by a factor of 2 at each stage, so the number of reductions required is NRED = $\lceil \log_2(\texttt{N}) \rceil$. Counting the original list and the result, there are $1 + \lceil \log_2(\texttt{N}) \rceil$ groups of partial sums. (b) Here is a program that implements the algorithm for N=17.

```
      PARAMETER(N=17,N6=N*6)
      REAL*8 S(N,6)/N6*0.D0/
      P=ALOG(FLOAT(N))/ALOG(2.)
      NRED=IFIX(P)
      IF(P.GT.FLOAT(NRED)) NRED=NRED+1
      :
      [put values to be added in S(*,1)]
      :
      K=1
      DO 1 J=2,NRED+1
           K=2*K
           DO 2 I=1,N,K
                S(I,J)=S(I,J-1)
                IF(I.LE.N-K/2) S(I,J)=S(I,J)+S(I+K/2,J-1)
    2      CONTINUE
    1 CONTINUE
      :
      [use the result, in S(1,NRED+1)
      :
      STOP
      END
```

The successive groups of partial sums are stored (with increasing sparsity) in columns 2...NRED+1 of the matrix S. (c) For each value of J, the innermost loop is independent and can be done in parallel, with one processor finding each S(I,J) for I=1, I=1+K, I=2+K, ...up to the last value that does not exceed N (perhaps not equal to N). None of the right-hand sides in the assignment depend on any of the left-hand sides for the current value of I. Unfortunately, using only the features of MPI that are discussed in the textbook it is not possible to distribute the elements of S to the processors so that each has all of the elements it needs, without sending the whole array everywhere. (d) Replacing this code by a call to SUM results in the program below.

```
      REAL*8 X(17),TOT,SUM
CHPF$ DISTRIBUTE X(BLOCK)
      :
      [put values to be added in X]
      :
      TOT=SUM(X)
      :
      [use the result, in TOT
      :
      STOP
      END
```

It is up to the compiler maker to decide how to parallelize the SUM function, so we have no way of knowing what algorithm it will use.

17.4.44 [H] Not all Fortran-90 array assignments can be rewritten as INDEPENDENT DO loops. For example, when the array assignment of Exercise 17.4.43 is rewritten as a DO loop it has a data dependence.

17.4.45 [E] (a) A FORALL differs from a DO loop or nest of DO loops in that the order of the assignments is up to HPF and unpredictable by us, and all the right-hand sides are evaluated before the left-hand sides get changed. (b) A FORALL differs from a Fortran-90 array assignment in that a mask can be used to further control which result elements get assigned.

17.4.46 [E] It might be necessary to assert that a FORALL is INDEPENDENT for either of these reasons: (1) indirection in the left-hand side of the assignment might risk the same element being assigned twice, and if we know that can't happen we can tell HPF the data dependence doesn't exist; (2) indirection in the right-hand side of the assignment might make it appear that synchronization is required between processors, and if we know that no overlap can occur in the calculation we can tell HPF that no synchronization is needed.

17.4.47 [H] We only need to examine K(1) before deciding what to do. (a) This code uses a DO loop.

```
      READ *,K(1)
      IF(K(1).GT.0) THEN
CHPF$    INDEPENDENT
         DO 3 L=1,M
              K(L)=L
    3    CONTINUE
      ENDIF
```

(b) This code uses a FORALL statement.

```
      READ *,K(1)
      IF(K(1).GT.0) THEN
         FORALL(L=1:M) K(L)=L
      ENDIF
```

17.4.49 [H] In asserting that a FORALL is independent we promise that there will be neither data dependencies from assigning a result value more than once, nor overlap between the parallel calculations. The given FORALL requires synchronization between the processors, so it is *not* independent. If we declare the FORALL to be INDEPENDENT anyhow, the answers come out wrong.

17.4.50 [H] (a) Here is the code segment of textbook §17.2.2 with the K loop replaced by an invocation of SUM.

```
      REAL*8 A(2,3),B(3,4),C(2,4)
CHPF$ PROCESSORS PROCS(2,4)
CHPF$ DISTRIBUTE C(BLOCK,BLOCK) ONTO PROCS
CHPF$ ALIGN A(I,*) WITH C(I,*)
CHPF$ ALIGN B(*,J) WITH C(*,J)
      :
CHPF$ INDEPENDENT,NEW(I)
      DO 1 J=1,4
      DO 1 I=1,2
           C(I,J)=SUM(A(I,:)*B(:,J))
    1 CONTINUE
```

(b) Here is the code segment of textbook §17.2.2 revised to use a FORALL that invokes SUM.

```
      REAL*8 A(2,3),B(3,4),C(2,4)
CHPF$ PROCESSORS PROCS(2,4)
CHPF$ DISTRIBUTE C(BLOCK,BLOCK) ONTO PROCS
CHPF$ ALIGN A(I,*) WITH C(I,*)
CHPF$ ALIGN B(*,J) WITH C(*,J)
      :
CHPF$ INDEPENDENT
      FORALL(I=1:4,J=1:2) C(I,J)=SUM(A(I,:)*B(:,J))
```

(c) Here is the code segment of textbook §17.2.2 revised to use the MATMUL built-in function.

```
      REAL*8 A(2,3),B(3,4),C(2,4)
CHPF$ PROCESSORS PROCS(2,4)
CHPF$ DISTRIBUTE C(BLOCK,BLOCK) ONTO PROCS
CHPF$ ALIGN A(I,*) WITH C(I,*)
CHPF$ ALIGN B(*,J) WITH C(*,J)
      :
      C=MATMUL(A,B)
```

17.4.53 [E] The prescriptive distribution of a subprogram parameter tells HPF how the array is to be distributed within the subprogram, without regard to its distribution in the invoking routine. If a prescriptive distribution disagrees with the one that is used in the invoking routine, HPF will remap the data upon entry and put it back again when the subprogram returns.

A transcriptive distribution is inherited from the calling routine, so no remapping takes place but the compiler must generate machine code for the subprogram in ignorance of how the array will be distributed.

17.4.54 [E] HPF abandons sequential storage so that it can organize array data in ways that minimize the overhead of passing messages. This means that to take advantage of parallel processing using HPF we must refrain from using the traditional memory-management techniques described in textbook §11, refrain from using adjustable dimensions in subprograms, pass array sections rather than starting addresses for subprogram parameters, and use special care when using COMMON and EQUIVALENCE.

17.4.56 `[E]` An `EXTRINSIC` routine is run once on each processor, typically to process all the array columns that are distributed to that processor. The `LBOUND` and `UBOUND` functions allow such a routine to find out what range of columns is present on the processor where the function is running.

17.4.57 `[H]` Imagine the crosshatched stencil shown in the statement of Exercise 5.8.33 repeated in an overlapping pattern so that the grid is colored like a chess board. Now if we think of the original values in the shaded squares as fixed data, one iteration of the update formula can be applied to all the open squares simultaneously without there being a data dependence in any of the calculations. At the same time, we can think of the original values in the open squares as data fixed for the iteration, and update all the shaded squares simultaneously. These calculations can be parallelized by HPF. After new values have been found for all of the open squares and all of the shaded squares, those values can be used to replace all the original values, and one iteration is complete.