NUMERICAL RECIPES

Example Book (FORTRAN

Second Edition

William T. Vetterling
William H. Press
Saul A. Teukolsky
Brian P. Flannery

Numerical Recipes Example Book (FORTRAN)

Second Edition

William T. Vetterling

Polaroid Corporation

Saul A. Teukolsky

Department of Physics, Cornell University

William H. Press

Harvard-Smithsonian Center for Astrophysics

Brian P. Flannery

EXXON Research and Engineering Company

CAMBRIDGE
UNIVERSITY PRESS

Published by the Press Syndicate of the University of Cambridge
The Pitt Building, Trumpington Street, Cambridge CB2 1RP
40 West 20th Street, New York, NY 10011-4211, USA
10 Stamford Road, Oakleigh, Melbourne 3166, Australia

First edition originally published 1986
Second edition originally published 1992
Reprinted 1993 (twice), 1994, 1995, 1998

Printed in the United States of America
Typeset in TEX

Library of Congress Cataloging-in-Publication Data available.

A catalogue record for this book is available from the British Library.

ISBN 0-521-43721-0 Example book in FORTRAN (this book)
ISBN 0-521-43064-X Numerical Recipes in FORTRAN
ISBN 0-521-43717-2 FORTRAN diskette (IBM 5.25", 1.2M)
ISBN 0-521-43719-9 FORTRAN diskette (IBM 3.5", 720K)
ISBN 0-521-43716-4 FORTRAN diskette (Mac 3.5". 800K)

Preface

This *Numerical Recipes Example Book (FORTRAN)* is designed to accompany the text and reference book *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, Second Edition, by William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery (Cambridge University Press, 1992). In that volume, the algorithms and methods of scientific computation are developed in considerable detail, starting with basic mathematical analysis and working through to actual implementation in the form of `FORTRAN` subroutines. The routines in *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, numbering more than 300, are meant to be incorporated into user applications; they are subroutines (or functions), not stand-alone programs.

It often happens, when you want to incorporate somebody else's procedure into your own application program, that you first want to see the procedure demonstrated on a simple example. Prose descriptions of how to use a procedure (even those in *Numerical Recipes*) can occasionally be inexact. There is no substitute for an actual, `FORTRAN` demonstration program that shows exactly how data are fed to a procedure, how the procedure is called, and how its results are unloaded and interpreted.

Another not unusual case occurs when you have, for one seemingly good purpose or another, modified the source code in a "foreign" procedure. In such circumstances, you might well want to test the modified procedure on an example known previously to have worked correctly, *before* letting it loose on your own data. There is the related case where procedure source code may have become corrupted, e.g., lost some lines or characters in transmission from one machine to another, and a simple revalidation test is desirable.

These are the needs addressed by this *Numerical Recipes Example Book*. Divided into chapters identically with *Numerical Recipes in FORTRAN: The Art of Scientific Computing*, this book contains `FORTRAN` source programs that exercise and demonstrate all of the *Numerical Recipes* subroutines and functions. The programs are commented, and each is also prefaced by a short description of what it does, and of which *Numerical Recipes* routines it exercises. In many cases where the demonstration programs require input data, that data is also printed in this book. In some cases, where the demonstration programs are not "self-validating," sample output is also shown.

Necessarily, in the interests of clarity, the *Numerical Recipes* procedures and functions are demonstrated in simple ways. A consequence is that the demonstration programs in this book do not usually test all possible regimes of input data, or even all lines of procedure source code. The demonstration programs in this book were by no means the only validating tests that the *Numerical Recipes* procedures and functions

were required to pass during their development. The programs in this book *were* used during the later stages of the production of *Numerical Recipes in FORTRAN: The Art of Scientific Computing* to maintain integrity of the source code, and in this role were found to be invaluable.

Disclaimer of Warranty

THE PROGRAMS LISTED IN THIS BOOK ARE PROVIDED "AS IS" WITHOUT WARRANTY OF ANY KIND. WE MAKE NO WARRANTIES, EXPRESS OR IMPLIED, THAT THE PROGRAMS CONTAINED IN THIS VOLUME ARE FREE OF ERROR, OR ARE CONSISTENT WITH ANY PARTICULAR STANDARD OF MERCHANTABILITY, OR THAT THEY WILL MEET YOUR REQUIREMENTS FOR ANY PARTICULAR APPLICATION. THEY SHOULD NOT BE RELIED ON FOR SOLVING A PROBLEM WHOSE INCORRECT SOLUTION COULD RESULT IN INJURY TO A PERSON OR LOSS OF PROPERTY. IF YOU DO USE THE PROGRAMS IN SUCH A MANNER, IT IS AT YOUR OWN RISK. THE AUTHORS AND PUBLISHER DISCLAIM ALL LIABILITY FOR DIRECT OR CONSEQUENTIAL DAMAGES RESULTING FROM YOUR USE OF THE PROGRAMS.

CONTENTS

Chapter 1: Preliminaries

The routines in Chapter 1 of Numerical Recipes are introductory and less general in purpose than those in the remainder of the book. This chapter's routines serve primarily to expose the book's notational conventions, illustrate control structures, and perhaps to amuse. You may even find them useful. We hope that you will use `badluk` *for no serious purpose.*

* * * *

Subroutine `flmoon` calculates the phases of the moon, or more exactly, the Julian day and fraction thereof on which a given phase will occur or has occurred. The program `xflmoon` asks the present date and compiles a list of upcoming phases. We have compared the predictions to lunar tables, with happy results. Shown are the results of a test run, which you may replicate as a check. In this program, notice that we have set `TZONE` (the time zone) to -5.0 to signify the five hour separation of the Eastern Standard time zone from Greenwich, England. Our convention requires you to use negative values of `TZONE` if you are west of Greenwich, as we are. The Julian day results are converted to calendar dates through the use of `caldat`, which appears later in the chapter. The fractional Julian day and time zone combine to form a correction that can possibly change the calendar date by one day.

```
    Date          Time(EST)    Phase
12  9 1992          7 PM     full moon
12 16 1992          2 PM     last quarter
12 23 1992          8 PM     new moon
12 31 1992         10 PM     first quarter
 1  8 1993          8 AM     full moon
 1 14 1993         11 PM     last quarter
 1 22 1993          1 PM     new moon
 1 30 1993          6 PM     first quarter
 2  6 1993          7 PM     full moon
 2 13 1993         10 AM     last quarter
 2 21 1993          8 AM     new moon
 3  1 1993         11 AM     first quarter
 3  8 1993          5 AM     full moon
 3 14 1993         11 PM     last quarter
 3 23 1993          2 AM     new moon
 3 30 1993         11 PM     first quarter
 4  6 1993          2 PM     full moon
 4 13 1993          3 PM     last quarter
 4 21 1993          6 PM     new moon
 4 29 1993          8 AM     first quarter
```

```
      PROGRAM xflmoon
C     driver for routine flmoon
      REAL TZONE
      PARAMETER(TZONE=-5.0)
      REAL frac,timzon
      INTEGER i,im,id,iy,ifrac,istr,j1,j2,julday,n,nph
      CHARACTER phase(4)*15,timstr(2)*3
      DATA phase/'new moon','first quarter',
     *     'full moon','last quarter'/
      DATA timstr/' AM',' PM'/
      write(*,*) 'Date of the next few phases of the moon'
      write(*,*) 'Enter today''s date (e.g. 12,15,1992)'
      timzon=TZONE/24.0
      read(*,*) im,id,iy
C     approximate number of full moons since January 1900
      n=12.37*(iy-1900+(im-0.5)/12.0)
      nph=2
      j1=julday(im,id,iy)
      call flmoon(n,nph,j2,frac)
      n=n+nint((j1-j2)/29.53)
      write(*,'(/1x,t6,a,t19,a,t32,a)') 'Date','Time(EST)','Phase'
      do 11 i=1,20
        call flmoon(n,nph,j2,frac)
        ifrac=nint(24.*(frac+timzon))
        if (ifrac.lt.0) then
          j2=j2-1
          ifrac=ifrac+24
        endif
        if (ifrac.ge.12) then
          j2=j2+1
          ifrac=ifrac-12
        else
          ifrac=ifrac+12
        endif
        if (ifrac.gt.12) then
          ifrac=ifrac-12
          istr=2
        else
          istr=1
        endif
        call caldat(j2,im,id,iy)
        write(*,'(1x,2i3,i5,t20,i2,a,5x,a)') im,id,iy,
     *       ifrac,timstr(istr),phase(nph+1)
        if (nph.eq.3) then
          nph=0
          n=n+1
        else
          nph=nph+1
        endif
11    continue
      END
```

The function `julday`, our exemplar of the `if` control structure, converts calendar dates to Julian dates. Not many people know the Julian date of their birthday or any other convenient reference point, for that matter. To remedy this, we offer a list of checkpoints, which appears at the end of this chapter as the file `DATES.DAT`. The

program **xjulday** lists the Julian date for each historic event for comparison. Then it allows you to make your own choices for entertainment.

```
      PROGRAM xjulday
C     driver for julday
      INTEGER i,im,id,iy,julday,n
      CHARACTER txt*40,name(12)*15
      DATA name/'January','February','March','April','May','June',
     *     'July','August','September','October','November',
     *     'December'/
      open(7,file='DATES.DAT',status='OLD')
      read(7,'(a)') txt
      read(7,*) n
      write(*,'(/1x,a,t12,a,t17,a,t23,a,t37,a/)') 'Month','Day','Year',
     *     'Julian Day','Event'
      do 11 i=1,n
        read(7,'(i2,i3,i5,a)') im,id,iy,txt
        write(*,'(1x,a10,i3,i6,3x,i7,5x,a)') name(im),id,iy,
     *       julday(im,id,iy),txt
11    continue
      close(7)
      write(*,'(/1x,a/)') 'Month,Day,Year (e.g. 1,13,1905)'
      do 12 i=1,20
        write(*,*) 'MM,DD,YYYY'
        read(*,*) im,id,iy
        if (im.lt.0) stop
        write(*,'(1x,a12,i8/)') 'Julian Day: ',julday(im,id,iy)
12    continue
      END
```

The next program in *Numerical Recipes* is **badluk**, an infamous code that combines the best and worst instincts of man. We include no demonstration program for **badluk**, not just because we fear it, but also because it is self-contained, with sample results appearing in the text.

Chapter 1 closes with routine **caldat**, which illustrates no new points, but complements **julday** by doing conversions from Julian day number to the month, day, and year on which the given Julian day began. This offers an opportunity, grasped by the demonstration program **xcaldat**, to push dates through both **julday** and **caldat** in succession, to see if they survive intact. This, of course, tests only your authors' ability to make mistakes backwards as well as forwards, but we hope you will share our optimism that correct results here speak well for both routines. (We have checked them a bit more carefully in other ways.)

```
      PROGRAM xcaldat
C     driver for routine caldat
      INTEGER i,im,imm,id,idd,iy,iyy,iycopy,j,julday,n
      CHARACTER name(12)*10
C     check whether CALDAT properly undoes the operation of JULDAY
      DATA name/'January','February','March','April','May',
     *     'June','July','August','September','October',
     *     'November','December'/
      open(7,file='DATES.DAT',status='OLD')
      read(7,*)
      read(7,*) n
      write(*,'(/1x,a,t40,a)') 'Original Date:','Reconstructed Date:'
```

```
      write(*,'(1x,a,t12,a,t17,a,t25,a,t40,a,t50,a,t55,a/)')
     *     'Month','Day','Year','Julian Day','Month','Day','Year'
      do 11 i=1,n
        read(7,'(i2,i3,i5)') im,id,iy
        iycopy=iy
        j=julday(im,id,iycopy)
        call caldat(j,imm,idd,iyy)
        write(*,'(1x,a,i3,i6,4x,i9,6x,a,i3,i6)') name(im),id,
     *       iy,j,name(imm),idd,iyy
11    continue
      END
```

Appendix

File DATES.DAT:

```
List of dates for testing routines in Chapter 1
16 entries
12 31   -1 End of millennium
01 01    1 One day later
10 14 1582 Day before Gregorian calendar
10 15 1582 Gregorian calendar adopted
01 17 1706 Benjamin Franklin born
04 14 1865 Abraham Lincoln shot
04 18 1906 San Francisco earthquake
05 07 1915 Sinking of the Lusitania
07 20 1923 Pancho Villa assassinated
05 23 1934 Bonnie and Clyde eliminated
07 22 1934 John Dillinger shot
04 03 1936 Bruno Hauptman electrocuted
05 06 1937 Hindenburg disaster
07 26 1956 Sinking of the Andrea Doria
06 05 1976 Teton dam collapse
05 23 1968 Julian Day 2440000
```

Chapter 2: Linear Algebraic Equations

Numerical Recipes Chapter 2 begins the "true grit" of numerical analysis by considering the solution of linear algebraic equations. This is done first by Gauss-Jordan elimination (`gaussj`*), and then by LU decomposition with forward and backsubstitution (*`ludcmp` *and* `lubksb`*). Several linear systems of special form, represented by tridiagonal, band diagonal, cyclic, Vandermonde, and Toeplitz matrices, may be treated with subroutines* `tridag`, `bandec`, `cyclic`, `vander`, *and* `toeplz` *respectively. Cholesky decomposition (*`choldc` *and* `cholsl`*) is the preferred method for symmetric positive definite systems. QR decomposition (*`qrdcmp`*) is less efficient than LU decomposition in general; however, the ease with which it is updated (*`qrupdt`*) if the system is changed slightly makes it useful in certain applications. For singular or nearly singular matrices the best choice is singular value decomposition with backsubstitution (*`svdcmp` *and* `svbksb`*). Linear systems with relatively few non-zero coefficients, so-called "sparse" matrices, are handled by routine* `linbcg`*. A suite of routines for manipulating general sparse matrices,* `sprsax`, `sprstx`, *etc., is provided.*

* * * *

`gaussj` performs Gauss-Jordan elimination with full pivoting to find the solution of a set of linear equations for a collection of right-hand side vectors. The demonstration routine `xgaussj` checks its operation with reference to a group of test input matrices printed at the end of this chapter as file `MATRX1.DAT`. Each matrix is subjected to inversion by `gaussj`, and then multiplication by its own inverse to see that a unit matrix is produced. Then the solution vectors are each checked through multiplication by the original matrix and comparison with the right-hand side vectors that produced them.

```
      PROGRAM xgaussj
C     driver for routine gaussj
      INTEGER MP,NP
      PARAMETER(MP=20,NP=20)
      INTEGER j,k,l,m,n
      REAL a(NP,NP),b(NP,MP),ai(NP,NP),x(NP,MP)
      REAL u(NP,NP),t(NP,MP)
      CHARACTER dummy*3
      open(7,file='MATRX1.DAT',status='old')
10    read(7,'(a)') dummy
      if (dummy.eq.'END') goto 99
      read(7,*)
      read(7,*) n,m
      read(7,*)
```

```
      read(7,*) ((a(k,l), l=1,n), k=1,n)
      read(7,*)
      read(7,*) ((b(k,l), k=1,n), l=1,m)
C     save Matrices for later testing of results
      do 13 l=1,n
        do 11 k=1,n
          ai(k,l)=a(k,l)
11      continue
        do 12 k=1,m
          x(l,k)=b(l,k)
12      continue
13    continue
C     invert Matrix
      call gaussj(ai,n,NP,x,m,MP)
      write(*,*) 'Inverse of Matrix A : '
      do 14 k=1,n
        write(*,'(1h ,(6f12.6))') (ai(k,l), l=1,n)
14    continue
C     test Results
C     check Inverse
      write(*,*) 'A times A-inverse (compare with unit matrix)'
      do 17 k=1,n
        do 16 l=1,n
          u(k,l)=0.0
          do 15 j=1,n
            u(k,l)=u(k,l)+a(k,j)*ai(j,l)
15        continue
16      continue
        write(*,'(1h ,(6f12.6))') (u(k,l), l=1,n)
17    continue
C     check Vector Solutions
      write(*,*) 'Check the following vectors for equality:'
      write(*,'(t12,a8,t23,a12)') 'Original','Matrix*Sol''n'
      do 21 l=1,m
        write(*,'(1x,a,i2,a)') 'Vector ',l,':'
        do 19 k=1,n
          t(k,l)=0.0
          do 18 j=1,n
            t(k,l)=t(k,l)+a(k,j)*x(j,l)
18        continue
          write(*,'(8x,2f12.6)') b(k,l),t(k,l)
19      continue
21    continue
      write(*,*) '**********************************'
      write(*,*) 'Press RETURN for next problem:'
      read(*,*)
      goto 10
99    close(7)
      END
```

The demonstration program for routine **ludcmp** relies on the same package of test matrices, but just performs an LU decomposition of each. The performance is checked by multiplying the lower and upper matrices of the decomposition and comparing with the original matrix. The array **indx** keeps track of the scrambling done by **ludcmp** to effect partial pivoting. We had to do the unscrambling here, but you will normally not be called upon to do so, since **ludcmp** is used with the routine

lubksb, which knows how to do its own descrambling.

```
      PROGRAM xludcmp
C     driver for routine ludcmp
      INTEGER NP
      PARAMETER(NP=20)
      INTEGER j,k,l,m,n,indx(NP),jndx(NP)
      REAL d,dum,a(NP,NP),xl(NP,NP),xu(NP,NP),x(NP,NP)
      CHARACTER txt*3
      open(7,file='MATRX1.DAT',status='old')
      read(7,*)
10    read(7,*)
      read(7,*) n,m
      read(7,*)
      read(7,*) ((a(k,l), l=1,n), k=1,n)
      read(7,*)
      read(7,*) ((x(k,l), k=1,n), l=1,m)
C     print out a-matrix for comparison with product of lower
C     and upper decomposition matrices.
      write(*,*) 'Original matrix:'
      do 11 k=1,n
        write(*,'(1x,6f12.6)') (a(k,l), l=1,n)
11    continue
C     perform the decomposition
      call ludcmp(a,n,NP,indx,d)
C     compose separately the lower and upper matrices
      do 13 k=1,n
        do 12 l=1,n
          if (l.gt.k) then
            xu(k,l)=a(k,l)
            xl(k,l)=0.0
          else if (l.lt.k) then
            xu(k,l)=0.0
            xl(k,l)=a(k,l)
          else
            xu(k,l)=a(k,l)
            xl(k,l)=1.0
          endif
12      continue
13    continue
C     compute product of lower and upper matrices for
C     comparison with original matrix.
      do 16 k=1,n
        jndx(k)=k
        do 15 l=1,n
          x(k,l)=0.0
          do 14 j=1,n
            x(k,l)=x(k,l)+xl(k,j)*xu(j,l)
14        continue
15      continue
16    continue
      write(*,*) 'Product of lower and upper matrices (unscrambled):'
      do 17 k=1,n
        dum=jndx(indx(k))
        jndx(indx(k))=jndx(k)
        jndx(k)=dum
17    continue
```

```
      do 19 k=1,n
        do 18 j=1,n
          if (jndx(j).eq.k) then
            write(*,'(1x,6f12.6)') (x(j,l), l=1,n)
          endif
18      continue
19    continue
      write(*,*) 'Lower matrix of the decomposition:'
      do 21 k=1,n
        write(*,'(1x,6f12.6)') (xl(k,l), l=1,n)
21    continue
      write(*,*) 'Upper matrix of the decomposition:'
      do 22 k=1,n
        write(*,'(1x,6f12.6)') (xu(k,l), l=1,n)
22    continue
      write(*,*) '***********************************'
      write(*,*) 'Press RETURN for next problem:'
      read(*,*)
      read(7,'(a3)') txt
      if (txt.ne.'END') goto 10
      close(7)
      END
```

Our example driver for `lubksb` makes calls to both `ludcmp` and `lubksb` in order to solve the linear equation problems posed in file `MATRX1.DAT` (see discussion of `gaussj`). The original matrix of coefficients is applied to the solution vectors to check that the result matches the right-hand side vectors posed for each problem. We apologize for using routine `ludcmp` in a test of `lubksb`, but `ludcmp` has been tested independently, and anyway, `lubksb` is nothing without this partner program, so a test of the combination is more to the point.

```
      PROGRAM xlubksb
C     driver for routine lubksb
      INTEGER NP
      PARAMETER (NP=20)
      REAL p,a(NP,NP),b(NP,NP),c(NP,NP),x(NP)
      INTEGER j,k,l,m,n,indx(NP)
      CHARACTER txt*3
      open(7,file='MATRX1.DAT',status='old')
      read(7,*)
10    read(7,*)
      read(7,*) n,m
      read(7,*)
      read(7,*) ((a(k,l), l=1,n), k=1,n)
      read(7,*)
      read(7,*) ((b(k,l), k=1,n), l=1,m)
C     save matrix a for later testing
      do 12 l=1,n
        do 11 k=1,n
          c(k,l)=a(k,l)
11      continue
12    continue
C     do LU decomposition
      call ludcmp(c,n,NP,indx,p)
C     solve equations for each right-hand vector
      do 16 k=1,m
```

```
        do 13 l=1,n
          x(l)=b(l,k)
13      continue
        call lubksb(c,n,NP,indx,x)
C     test results with original matrix
        write(*,*) 'Right-hand side vector:'
        write(*,'(1x,6f12.6)') (b(l,k), l=1,n)
        write(*,*) 'Result of matrix applied to sol''n vector'
        do 15 l=1,n
          b(l,k)=0.0
          do 14 j=1,n
            b(l,k)=b(l,k)+a(l,j)*x(j)
14        continue
15      continue
        write(*,'(1x,6f12.6)') (b(l,k), l=1,n)
        write(*,*) '*********************************'
16    continue
      write(*,*) 'Press RETURN for next problem:'
      read(*,*)
      read(7,'(a3)') txt
      if (txt.ne.'END') goto 10
      close(7)
      END
```

Subroutine `tridag` solves linear equations with coefficients that form a tridiagonal matrix. We provide at the end of this chapter a second file of matrices `MATRIX2.DAT` for the demonstration driver. In all other respects, the demonstration program `xtridag` operates in the same fashion as `xlubksb`.

```
      PROGRAM xtridag
C     driver for routine tridag
      INTEGER NP
      PARAMETER (NP=20)
      INTEGER k,n
      REAL diag(NP),superd(NP),subd(NP),rhs(NP),u(NP)
      CHARACTER txt*3
      open(7,file='MATRX2.DAT',status='old')
10    read(7,'(a3)') txt
      if (txt.eq.'END') goto 99
      read(7,*)
      read(7,*) n
      read(7,*)
      read(7,*) (diag(k), k=1,n)
      read(7,*)
      read(7,*) (superd(k), k=1,n-1)
      read(7,*)
      read(7,*) (subd(k), k=2,n)
      read(7,*)
      read(7,*) (rhs(k), k=1,n)
C     carry out solution
      call tridag(subd,diag,superd,rhs,u,n)
      write(*,*) 'The solution vector is:'
      write(*,'(1x,6f12.6)') (u(k), k=1,n)
C     test solution
      write(*,*) '(matrix)*(sol''n vector) should be:'
      write(*,'(1x,6f12.6)') (rhs(k), k=1,n)
      write(*,*) 'Actual result is:'
```

```
      do 11 k=1,n
        if (k.eq.1) then
          rhs(k)=diag(1)*u(1) + superd(1)*u(2)
        else if (k.eq.n) then
          rhs(k)=subd(n)*u(n-1) + diag(n)*u(n)
        else
          rhs(k)=subd(k)*u(k-1) + diag(k)*u(k)
     *         + superd(k)*u(k+1)
        endif
11    continue
      write(*,'(1x,6f12.6)') (rhs(k), k=1,n)
      write(*,*) '*********************************'
      write(*,*) 'Press RETURN for next problem:'
      read(*,*)
      goto 10
99    close(7)
      END
```

The demonstration program `xbanmul` forms a banded matrix, multiplies if by a vector using `banmul`, and compares the answer to the result of carrying out a full matrix multiplication.

```
      PROGRAM xbanmul
C     driver for routine banmul
      INTEGER NP,M1,M2,MP
      PARAMETER (NP=7,M1=2,M2=1,MP=M1+1+M2)
      INTEGER i,j,k
      REAL a(NP,MP),aa(NP,NP),ax(NP),b(NP),x(NP)
      do 12 i=1,M1
        do 11 j=1,NP
          a(j,i)=10*j+i
11      continue
12    continue
C     lower band
      do 13 i=1,NP
        a(i,M1+1)=i
13    continue
C     diagonal
      do 15 i=1,M2
        do 14 j=1,NP
          a(j,M1+1+i)=0.1*j+i
14      continue
15    continue
C     upper band
      do 17 i=1,NP
        do 16 j=1,NP
          k=i-M1-1
          if (j.ge.max(1,1+k).and.j.le.min(M1+M2+1+k,NP)) then
            aa(i,j)=a(i,j-k)
          else
            aa(i,j)=0.0
          endif
16      continue
17    continue
      do 18 i=1,NP
        x(i)=i/10.0
18    continue
```

```
      call banmul(a,NP,M1,M2,NP,MP,x,b)
      do 21 i=1,NP
        ax(i)=0.0
        do 19 j=1,NP
          ax(i)=ax(i)+aa(i,j)*x(j)
19      continue
21    continue
      write(*,'(t8,a,t32,a)') 'Reference vector','banmul vector'
      do 22 i=1,NP
        write(*,'(t8,f12.4,t32,f12.4)') ax(i),b(i)
22    continue
      END
```

The sample program xbandec forms a random banded matrix $\mathbf{A}$, a random vector $\mathbf{x}$, and calculates $\mathbf{b} = \mathbf{A} \cdot \mathbf{x}$. It then supplies $\mathbf{A}$ and $\mathbf{b}$ to bandec and banbks and compares the solution to the saved copy of $\mathbf{x}$.

```
      PROGRAM xbandec
C     driver for routine bandec
      REAL ran1
      REAL a(7,4),x(7),b(7),al(7,2),d
      INTEGER indx(7)
      INTEGER i,idum,j
      idum=-1
      do 12 i=1,7
        x(i)=ran1(idum)
        do 11 j=1,4
          a(i,j)=ran1(idum)
11      continue
12    continue
      call banmul(a,7,2,1,7,4,x,b)
      do 13 i=1,7
        write(*,*) i,b(i),x(i)
13    continue
      call bandec(a,7,2,1,7,4,al,2,indx,d)
      call banbks(a,7,2,1,7,4,al,2,indx,b)
      do 14 i=1,7
        write(*,*) i,b(i),x(i)
14    continue
      stop
      END
```

mprove is a short routine for improving the solution vector for a set of linear equations, providing that an LU decomposition has been performed on the matrix of coefficients. Our test of this function is to use ludcmp and lubksb to solve a set of equations specified in the DATA statements at the beginning of the program. The solution vector is then corrupted by the addition of random values to each component. mprove works on the corrupted vector to recover the original.

```
      PROGRAM xmprove
C     driver for routine mprove
      INTEGER N,NP
      PARAMETER(N=5,NP=5)
      INTEGER i,j,idum,indx(N)
      REAL d,a(NP,NP),b(N),x(N),aa(NP,NP),ran3
      DATA a/1.0,2.0,1.0,4.0,5.0,2.0,3.0,1.0,5.0,1.0,
```