Source file: `bisect.f`

```
c===========================================================
c     bisect: Uses bisection to find approximate root
c     of f(x) on interval [xmin .. xmax].  Return value is
c     root located to (relative) tolerance 'xtol'.  Return code
c     'rc' is set to 0 on success, non-zero on failure
c     and routine succeeds (by definition) as long as initial
c     interval *does* bracket at least one root.  Routine
c     performs tracing of algorithm (on stderr) if input
c     argument 'trace' is .true.
c===========================================================
      real*8 function bisect(f,xmin,xmax,xtol,trace,rc)

         implicit     none

         real*8       drelabs

         real*8       f
         external     f

         real*8       xmin,          xmax,         xtol
         logical      trace
         integer      rc
c-----------------------------------------------------------
c        Other variables needed for search.
c-----------------------------------------------------------
         integer      mxiter
         parameter  ( mxiter = 50 )

         real*8       xlo,           dx,           sgn
         integer      iter
c-----------------------------------------------------------
c        Check that input interval is specified correctly
c        and that it manifestly brackets at least one root:
c        (i.e. the fcn changes sign).
c-----------------------------------------------------------
         if( xmax .le. xmin  .or.
     &       f(xmin) * f(xmax) .gt. 0.0d0 ) then
            write(0,*) 'bisect:  Input interval is not '//
     &                 'bracketing'
            rc = 1
c-----------------------------------------------------------
c           Returned value is meaningless in this case,
c           but have to return *some* value.
c-----------------------------------------------------------
            bisect = xmin
            return
         end if
c-----------------------------------------------------------
c        Compute 'sgn' such that sgn * f(xmin) < 0, and
c        intialize bracketing interval
c-----------------------------------------------------------
         sgn  = 1.0d0
         if( f(xmin) .le. 0.0d0 ) then
            sgn =  1.0d0
         else
            sgn = -1.0d0
         end if
         xlo  = xmin
         dx   = xmax - xmin
c-----------------------------------------------------------
c        Bisection loop: continue until root found to
c        specfied tolerance or until maximum number of
c        iterations taken
c-----------------------------------------------------------
         do iter = 1 , mxiter
            bisect = xlo + 0.5d0 * dx
            if( trace ) then
               write(0,*) xlo, xlo + dx, f(bisect)
            end if
            if( sgn * f(bisect) .lt. 0.0d0 ) then
               xlo = bisect
            end if
            if( drelabs(dx,bisect,1.0d-10) .le. xtol ) go to 900
            dx = 0.5d0 * dx
         end do

900      continue
         rc = 0
         if( trace ) write(0,*)
         return

      end

c===========================================================
c     drelabs:  Function useful for 'relativizing' quantity
c     being monitored for detection of convergence.
c===========================================================
      real*8 function drelabs(dx,x,xfloor)
         implicit      none

         real*8        dx,     x,    xfloor

         if( abs(x) .lt. abs(xfloor) ) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return
      end
```

Source file: `tbisect.f`

```
c===========================================================
c     tbisect: Illustrates root finding using bisection
c     routine 'bisect'.
c
c     Initial bracketing interval must be specified via the
c     command-line, along with optional convergence criteria
c     and output option.
c
c     This program also illustrates the general Fortran
c     techniques (briefly discussed previously) for:
c
c     (1) Writing and using routines which take other routines
c     as arguments.
c     (2) Using a COMMON block to communicate information to
c     a routine in cases where the information cannot be
c     passed via the argument list.
c     (3) Using an "INCLUDE" file (in this case 'comf.inc')
c     to ensure that the same common block structure is defined
c     in all program units.
c
c     Currently set up for computing square roots i.e.
c     solves
c
c         f(x; a) = x**2 - a = 0
c
c     for 'a' specifed on command-line
c
c     Outputs a, approximate root (x*) and f(x*; a) on stdout.
c===========================================================
      program         tbisect
      implicit        none
c-----------------------------------------------------------
c     Declaration of the bisection routine.
c-----------------------------------------------------------
      real*8          bisect
c-----------------------------------------------------------
c     Name of the specific function whose root we seek.
c     Note use of 'external' to let compiler know 'fsqr'
c     is the name of a function, not a variable.
c-----------------------------------------------------------
      real*8          fsqr
      external        fsqr

      integer         i4arg,         iargc
      real*8          r8arg
c-----------------------------------------------------------
c     For use in detecting bad real*8 command-line value.
c-----------------------------------------------------------
      real*8          r8_never
      parameter     ( r8_never = -1.0d-60 )
c-----------------------------------------------------------
c     Use a common block to pass number whose square root
c     is sought to external function 'fsqr'.
c-----------------------------------------------------------
      include       'comf.inc'
c-----------------------------------------------------------
```

```
c     Initial bracket, convergence tolerance and output
c     option from command-line; default value for conv.
c     tolerance.
c-----------------------------------------------------------
      real*8          xmin,          xmax,        xtol
      logical         trace

      real*8          default_xtol
      parameter     ( default_xtol = 1.0d-8 )
c-----------------------------------------------------------
c     Root and return code from bisection routine.
c-----------------------------------------------------------
      real*8          root
      integer         rc
c-----------------------------------------------------------
c     Argument parsing.
c-----------------------------------------------------------
      if( iargc() .lt. 3 ) go to 900
      a     = r8arg(1,r8_never)
      xmin  = r8arg(2,r8_never)
      xmax  = r8arg(3,r8_never)
      if( a .eq. r8_never  .or.  xmin .eq. r8_never  .or.
     &    xmax .eq. r8_never ) go to 900

      xtol  = r8arg(4,default_xtol)
      trace = iargc() .gt. 4
c-----------------------------------------------------------
c     Invoke root finder then write a, sqrt(a), and residual
c     to standard output.
c-----------------------------------------------------------
      root = bisect(fsqr,xmin,xmax,xtol,trace,rc)
      if( rc .eq. 0 ) then
         write(*,*) a, root, fsqr(root)
      else
         write(0,*) 'tbisect: Bisection failed.'
      end if
c-----------------------------------------------------------
c     Normal exit.
c-----------------------------------------------------------
      stop

c-----------------------------------------------------------
c     Usage exit.
c-----------------------------------------------------------
 900  continue
         write(0,*) 'usage: tbisect <a> <xmin> <xmax> '//
     &              '[<xtol> <trace>]'
      stop

      end

c===========================================================
c     Function whose root is sought.  Again, note use of
c     COMMON block to pass additional information (in this
c     case 'a') to the routine.
c===========================================================
      real*8 function fsqr(x)
         implicit      none

         real*8        x

         include      'comf.inc'

         fsqr = x**2 - a

         return
      end
```

Source file: `comf.inc`

```
c-----------------------------------------------------------
c     Common block for communicating value of 'a' from main
c     to 'fsqr'.
c-----------------------------------------------------------
      real*8              a
      common    / comf /   a
```

Source file: `Output on lnx1`

```
############################################################
# Building 'tbisect' and sample output on lnx1
#
# 'tbisect' is set up to compute sqrt(a) via bisection.
############################################################
lnx1% pwd; ls
/home/phys410/nonlin/bisect
Makefile  bisect.f  comf.inc  tbisect.f

lnx1% make
pgf77 -g -c tbisect.f
pgf77 -g -c bisect.f
pgf77 -g -L/usr/local/PGI/lib tbisect.o bisect.o -lp410f \
      -o tbisect

Linking:

############################################################
# Compute +sqrt(2) to default tolerance (1.0d-8)
#
# Note: Exact value to 16 digits is 1.414 2135 6237 3095
############################################################
lnx1% tbisect 2.0 1.0 2.0
  2.000000000000000   1.414213564246893   5.2999007543741428E-009

############################################################
# Recompute with higher tolerance (1.0d-12)
############################################################
lnx1% tbisect 2.0 1.0 2.0 1.0e-12
  2.000000000000000   1.414213562372879  -6.1080654423228964E-013

############################################################
# Enable tracing output by supplying 5th argument.  Note
# supplying a '.' as an argument parsed by 'i4arg' or 'r8arg'
# is equivalent to specifying the default value.
############################################################
lnx1% tbisect 2.0 1.0 2.0 . 1
  1.000000000000000   2.000000000000000   0.2500000000000000
  1.000000000000000   1.500000000000000  -0.4375000000000000
  1.250000000000000   1.500000000000000  -0.1093750000000000
  1.375000000000000   1.500000000000000   6.6406250000000000E-002
  1.375000000000000   1.437500000000000  -2.2460937500000000E-002
  1.406250000000000   1.437500000000000   2.1728515625000000E-002
  1.406250000000000   1.421875000000000  -4.2724609375000000E-004
  1.414062500000000   1.421875000000000   1.0635375976562500E-002
  1.414062500000000   1.417968750000000   5.1002502441406250E-003
  1.414062500000000   1.416015625000000   2.3355484008789063E-003
  1.414062500000000   1.415039062500000   9.5391273498535156E-004
  1.414062500000000   1.414550781250000   2.6327371597290039E-004
  1.414062500000000   1.414306640625000  -8.2001090049743652E-005
  1.414184570312500   1.414306640625000   9.0632587671279907E-005
  1.414184570312500   1.414245605468750   4.3148174881935120E-006
  1.414184570312500   1.414215087890625  -3.8843369111418724E-005
  1.414199829101563   1.414215087890625  -1.7264334019273520E-005
  1.414207458496094   1.414215087890625  -6.4747728174552321E-006
  1.414211273193359   1.414215087890625  -1.0799813026096672E-006
  1.414213180541992   1.414215087890625   1.6174171832972206E-006
  1.414213180541992   1.414214134216309   2.6871771297010127E-007
  1.414213180541992   1.414213657379150  -4.0563185166320181E-007
  1.414213418960571   1.414213657379150  -6.8457083557404985E-008
  1.414213538169861   1.414213657379150   1.0013031115363447E-007
  1.414213538169861   1.414213597774506   1.5836612909936321E-008
  1.414213538169861   1.414213567972183  -2.6310235545778937E-008
  1.414213553071022   1.414213567972183  -5.2368113734324595E-009
  1.414213560521603   1.414213567972183   5.2999007543741428E-009

  2.000000000000000   1.414213564246893   5.2999007543741428E-009
```

Source file: `newtsqrt.f`

```
c===========================================================
c     newtsqrt:  Uses Newton's method to find (positive)
c     square root of number supplied on command line, i.e.
c     solves
c
c     f(x) = x^2 - a = 0
c
c     for given 'a'.  Optional second argument specifies
c     convergence criteria (relative dx).
c
c     Tracing output (written to standard error)
c     includes iteration number, estimated root (xn),
c     change in estimate (dxn), log10(dxn), residual and
c     log10(residual).
c===========================================================
      program        newtsqrt

      implicit       none

      integer        iargc
      real*8         r8arg,        drelabs

      real*8         r8_never
      parameter     ( r8_never = -1.0d-60 )
c-----------------------------------------------------------
c     Default convergence tolerance.
c-----------------------------------------------------------
      real*8         default_xtol
      parameter     ( default_xtol = 1.0d-8 )
c-----------------------------------------------------------
c     Maximum allowed number of Newton iterations.
c-----------------------------------------------------------
      integer        mxiter
      parameter     ( mxiter = 50 )
c-----------------------------------------------------------
c     Command-line arguments (see above).
c-----------------------------------------------------------
      real*8         a,            xtol
c-----------------------------------------------------------
c     Locals used in Newton iteration.
c-----------------------------------------------------------
      integer        iter
      real*8         xn,           resn,          dxn

c-----------------------------------------------------------
c     Argument parsing.
c-----------------------------------------------------------
      if( iargc() .lt. 1 ) go to 900
      a     = r8arg(1,r8_never)
      if( a .eq. r8_never .or. a .lt. 0.0d0 ) go to 900
      xtol  = r8arg(2,1.0d-8)
      if( xtol .le. 0.0d0 ) xtol = 1.0d-8

c-----------------------------------------------------------
c     Un-inspired initial guess: x^(0) = a / 2.
c-----------------------------------------------------------
      xn = 0.5d0 * a

c-----------------------------------------------------------
c     Newton loop.
c-----------------------------------------------------------
      write(0,*) 'Iter          xn                 '//
     &           'dxn     log10(dxn)    rn       log10(rn)'
      write(0,*)
      do iter = 1 , mxiter
         resn = xn**2 - a
         dxn  = resn / (2.0d0 * xn)
         xn   = xn - dxn
         write(0,1000) iter, xn, dxn, log10(abs(dxn)),
     &                 resn, log10(abs(resn))
1000     format(i2,1p,e26.16,e12.3,0p,f10.2,1p,e12.3,0p,f10.2)
c-----------------------------------------------------------
c        Jump out of Newton loop if soln has converged.
c-----------------------------------------------------------
         if( drelabs(dxn,xn,1.0d-10) .le. xtol ) go to 100
      end do
c-----------------------------------------------------------
c     No-convergence exit.
c-----------------------------------------------------------
      write(0,*) 'No convergence after ', mxiter,
     &           ' iterations'
      stop

c-----------------------------------------------------------
c     Normal exit, write input and estimated square root
c     to standard output.
c-----------------------------------------------------------
 100  continue
      write(0,*)
      write(*,*) a, xn
      stop

c-----------------------------------------------------------
c     Usage exit.
c-----------------------------------------------------------
 900  continue
         write(0,*) 'usage: newtsqrt <a> [<xtol>]'
      stop

      end

c===========================================================
c     drelabs:  Function useful for 'relativizing' quantity
c     being monitored for detection of convergence.
c===========================================================
      real*8 function drelabs(dx,x,xfloor)

         implicit      none

         real*8        dx,     x,     xfloor

         if( abs(x) .lt. abs(xfloor) ) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return

      end
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
	$(F77_COMPILE) $*.f

EXECUTABLES = newtsqrt

all: $(EXECUTABLES)

newtsqrt: newtsqrt.o
	$(F77_LOAD) newtsqrt.o -lp410f -o newtsqrt

clean:
	rm *.o
	rm $(EXECUTABLES)
```

Source file: `Output on lnx1`

```
#############################################################
# Building 'newtsqrt' and sample output on lnx1
#############################################################
/home/phys410/nonlin/newtsqrt
Makefile  newtsqrt.f

lnx1% make
pgf77 -g -c newtsqrt.f
pgf77 -g -L/usr/local/PGI/lib newtsqrt.o -lp410f -o newtsqrt
Linking:

#############################################################
# Compute +sqrt(10) to default tolerance (1.0d-8)
#
# Note: Exact value to 16 digits is 3.162 2776 6016 8379
#############################################################
% newtsqrt 10

 Iter          xn                 dxn     log10(dxn)    rn       log10(rn)

 1    3.5000000000000000E+00   1.500E+00      0.18   1.500E+01      1.18
 2    3.1785714285714284E+00   3.214E-01     -0.49   2.250E+00      0.35
 3    3.1623194221508828E+00   1.625E-02     -1.79   1.033E-01     -0.99
 4    3.1622776604441363E+00   4.176E-05     -4.38   2.641E-04     -3.58
 5    3.1622776601683795E+00   2.758E-10     -9.56   1.744E-09     -8.76

    10.00000000000000        3.162277660168380

#############################################################
# Recompute with higher tolerance---an extra Newton step
# is taken, but the solution was already accurate to
# roughly machine epsilon, so there is very little change
# in the output.
#############################################################
% newtsqrt 10 1.0e-12

 Iter          xn                 dxn     log10(dxn)    rn       log10(rn)

 1    3.5000000000000000E+00   1.500E+00      0.18   1.500E+01      1.18
 2    3.1785714285714284E+00   3.214E-01     -0.49   2.250E+00      0.35
 3    3.1623194221508828E+00   1.625E-02     -1.79   1.033E-01     -0.99
 4    3.1622776604441363E+00   4.176E-05     -4.38   2.641E-04     -3.58
 5    3.1622776601683795E+00   2.758E-10     -9.56   1.744E-09     -8.76
 6    3.1622776601683795E+00   1.908E-16    -15.72   1.207E-15    -14.92

    10.00000000000000        3.162277660168380

#############################################################
# Compute +sqrt(1/2) to default tolerance (1.0d-8)
#
# Note: Exact value to 16 digits is 0.7071 0678 1186 5475
#############################################################
lnx1% newtsqrt 0.5

 Iter          xn                 dxn     log10(dxn)    rn       log10(rn)

 1    1.1250000000000000E+00  -8.750E-01     -0.06  -4.375E-01     -0.36
 2    7.8472222222222223E-01   3.403E-01     -0.47   7.656E-01     -0.12
 3    7.1094518190757130E-01   7.378E-02     -1.13   1.158E-01     -0.94
 4    7.0711714297003674E-01   3.828E-03     -2.42   5.443E-03     -2.26
 5    7.0710678126246602E-01   1.036E-05     -4.98   1.465E-05     -4.83
 6    7.0710678118654755E-01   7.592E-11    -10.12   1.074E-10     -9.97

   0.5000000000000000       0.7071067811865476
```

Source file: `newt2.f`

```
c===========================================================
c     newt2:  Uses multi-dimensional Newton's method
c     to compute a root of simple non-linear system
c     discussed in class
c
c        sin(xy) - 1/2 = 0
c        y^2 - 6x - 2  = 0
c
c     Command line input is initial guess (two numbers)
c     for root, and optional convergence criteria.
c     Estimated root written to standard output.
c     Tracing output similar to that from 'newtsqrt'.
c===========================================================
      program        newt2

      implicit       none

      integer        iargc
      real*8         r8arg,       drelabs,     dvl2norm

      real*8         r8_never
      parameter    ( r8_never = -1.0d-60 )
c-----------------------------------------------------------
c     Size of system.
c-----------------------------------------------------------
      integer        neq
      parameter    ( neq = 2 )
c-----------------------------------------------------------
c     Command-line arguments:  Initial guess will be
c     input directly into 'x' array.
c-----------------------------------------------------------
      real*8         tol
c-----------------------------------------------------------
c     Variables used in Newton iteration and solution of
c     linear systems via LAPACK routine 'dgesv'.
c-----------------------------------------------------------
      real*8         J(neq,neq),   res(neq),
     &               x(neq)
      integer        ipiv(neq)
      integer        ieq,          info

      integer        mxiter,       nrhs
      parameter    ( mxiter = 50,  nrhs = 1 )

      integer        iter
      real*8         nrm2res,      nrm2dx,      nrm2x
c-----------------------------------------------------------
c     Default convergence tolerance.
c-----------------------------------------------------------
      real*8         default_tol
      parameter    ( default_tol = 1.0d-8 )
c-----------------------------------------------------------
c     Argument parsing.
c-----------------------------------------------------------
      if( iargc() .lt. neq ) go to 900
      do ieq = 1 , neq
         x(ieq) = r8arg(ieq,r8_never)
         if( x(ieq) .eq. r8_never ) go to 900
      end do
      tol  = r8arg(neq+1,default_tol)
      if( tol .le. 0.0d0 ) tol = default_tol

c-----------------------------------------------------------
c     Newton loop.
c-----------------------------------------------------------
      write(0,*) 'Iter             x                       y '//
     &          '           log10(dx) log10(res)'
      write(0,*)
      do iter = 1 , mxiter
c-----------------------------------------------------------
c        Evaluate residual vector.
c-----------------------------------------------------------
         res(1) = sin(x(1)*x(2)) - 0.5d0
         res(2) = x(2)**2 - 6.0d0 * x(1) - 2.0d0
         nrm2res = dvl2norm(res,2)
c-----------------------------------------------------------
c        Set up Jacobian.
c-----------------------------------------------------------
         J(1,1) = x(2) * cos(x(1) * x(2))
         J(1,2) = x(1) * cos(x(1) * x(2))
         J(2,1) = -6.0d0
         J(2,2) = 2.0d0 * x(2)
c-----------------------------------------------------------
c        Solve linear system (J dx = res) for update
c        dx.  Update returned in 'res' vector.
c-----------------------------------------------------------
         call dgesv( neq, nrhs, J, neq, ipiv, res, neq, info )
         if( info .eq. 0 ) then
c-----------------------------------------------------------
c           Update solution.
c-----------------------------------------------------------
            nrm2x  = dvl2norm(x,neq)
            nrm2dx = dvl2norm(res,neq)
            do ieq = 1 , neq
               x(ieq) = x(ieq) - res(ieq)
            end do
c-----------------------------------------------------------
c           Tracing output: note use of max to prevent
c           taking log10 of 0.
c-----------------------------------------------------------
            write(0,1000) iter, x(1), x(2),
     &                    log10(max(nrm2dx,1.0d-60)),
     &                    log10(max(nrm2res,1.0d-60))
1000        format(i2,1p,2e24.16,0p,2f8.2)
c-----------------------------------------------------------
c           Check for convergence.
c-----------------------------------------------------------
            if( drelabs(nrm2dx,nrm2x,1.0d-6) .le. tol ) go to 100
         else
            write(0,*) 'newt2: dgesv failed.'
            stop
         end if
      end do
c-----------------------------------------------------------
c     No-convergence exit.
c-----------------------------------------------------------
      write(0,*) 'No convergence after ', mxiter,
     &           ' iterations'
      stop

c-----------------------------------------------------------
c     Normal exit, write input and estimated square root
c     to standard output.
c-----------------------------------------------------------
 100  continue
      write(0,*)
      write(*,*) x
      stop

c-----------------------------------------------------------
c     Usage exit.
c-----------------------------------------------------------
 900  continue
         write(0,*) 'usage: newt2 <x0> <y0> [<tol>]'
      stop

      end

c===========================================================
c     dvl2norm:  Returns l2-norm of double precision vector.
c===========================================================
      real*8 function dvl2norm(v,n)

         implicit      none

         integer       n
         real*8        v(n)
         integer       i

         dvl2norm = 0.0d0
         do i = 1 , n
            dvl2norm = dvl2norm + v(i) * v(i)
         end do
         if( n .gt. 0 ) then
            dvl2norm = sqrt(dvl2norm / n)
         end if

         return

      end
```

```
c===========================================================
c     drelabs:  Function useful for 'relativizing' quantity
c     being monitored for detection of convergence.
c===========================================================
      real*8 function drelabs(dx,x,xfloor)

         implicit      none

         real*8        dx,     x,    xfloor

         if( abs(x) .lt. abs(xfloor) ) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return

      end
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
   $(F77_COMPILE) $*.f

EXECUTABLES = newt2

all: $(EXECUTABLES)

newt2: newt2.o
   $(F77_LOAD) newt2.o -lp410f -llapack $(LIBBLAS) -o newt2

clean:
   rm *.o
   rm $(EXECUTABLES)
```

Source file: `Output on lnx1`

```
#################################################################
# Building 'newt2' and sample output on lnx1.
#
# Note how different roots are found depending on the initial
# guess and how, in each case, convergence of both dx and
# the residual is quadratic as the solution is approached.
#################################################################
lnx1% pwd; ls
/home/phys410/nonlin/newt2
Makefile  newt2.f

lnx1% make
pgf77 -g -c newt2.f
pgf77 -g -L/usr/local/PGI/lib newt2.o \
      -lp410f -llapack -lblas -o newt2

Linking:
lnx1% newt2
 usage: newt2 <x0> <y0> [<tol>]

#################################################################
# Start with initial guess (1.0,1.0) and use default tolerance
#################################################################
lnx1% newt2 1.0 1.0
 Iter           x                      y           log10(dx) log10(res)

 1 -3.2999966453609808E-02  1.4010001006391706E+00   -0.11    0.70
 2  3.7660093320946681E-01  2.2207017966697333E+00   -0.19   -0.40
 3  2.6508349149835868E-01  1.9187667230922997E+00   -0.64   -0.30
 4  2.7416951525985471E-01  1.9092166705387069E+00   -2.03   -1.19
 5  2.7423631305849172E-01  1.9092977465351673E+00   -4.13   -3.95
 6  2.7423631371214592E-01  1.9092977458408303E+00   -9.17   -8.33

   0.2742363137121459       1.909297745840830

#################################################################
# Start with initial guess (10.0,10.0)
#################################################################
lnx1% newt2 10.0 10.0
 Iter           x                      y           log10(dx) log10(res)

 1  1.1551311217431483E+01  8.5653933652294452E+00    0.17    1.43
 2  5.2821340061726980E+00  6.2494950887340224E+00    0.67    0.26
 3  7.9156169058357619E+00  7.0845635560826592E+00    0.29    0.58
 4  8.0553488925966921E+00  7.0945184795080038E+00   -1.00   -0.08
 5  8.0478800969985382E+00  7.0913532277563132E+00   -2.24   -1.34
 6  8.0480621354266226E+00  7.0914295327798467E+00   -3.86   -2.93
 7  8.0480622340064549E+00  7.0914295740731097E+00   -7.12   -6.20

    8.048062234006455       7.091429574073110

#################################################################
# Start with initial guess (100.0,100.0)
#################################################################
lnx1% newt2 100.0 100.0
 Iter           x                      y           log10(dx) log10(res)

 1  1.4561314470371519E+02  5.4378394341111459E+01    1.66    3.82
 2  1.9021837653952545E+02  3.7701738714769562E+01    1.53    3.17
 3  2.0349983567820647E+02  3.5070267397907138E+01    0.98    2.29
 4  2.0392234856561166E+02  3.5007684984188501E+01   -0.52    0.70
 5  2.0390326095147370E+02  3.5005993323580434E+01   -1.87   -0.53
 6  2.0391023928640129E+02  3.5006591323292412E+01   -2.31   -0.59
 7  2.0391061250942664E+02  3.5006623302706338E+01   -3.58   -1.92
 8  2.0391061457091234E+02  3.5006623479357074E+01   -5.83   -4.18
 9  2.0391061457097669E+02  3.5006623479362588E+01  -10.34   -8.68

    203.9106145709767       35.00662347936259

#################################################################
# Start with initial guess (0.0,0.0), generates singular
# Jacobian
#################################################################
lnx1% newt2 0.0 0.0
 Iter           x                      y           log10(dx) log10(res)

 newt2: dgesv failed.

#################################################################
# Start with initial guess (1.0,1.0) but use more stringent
# tolerance
#################################################################
lnx1% newt2 1.0 1.0 1.0e-15
 Iter           x                      y           log10(dx) log10(res)

 1 -3.2999966453609808E-02  1.4010001006391706E+00   -0.11    0.70
 2  3.7660093320946681E-01  2.2207017966697333E+00   -0.19   -0.40
 3  2.6508349149835868E-01  1.9187667230922997E+00   -0.64   -0.30
 4  2.7416951525985471E-01  1.9092166705387069E+00   -2.03   -1.19
 5  2.7423631305849172E-01  1.9092977465351673E+00   -4.13   -3.95
 6  2.7423631371214592E-01  1.9092977458408303E+00   -9.17   -8.33
 7  2.7423631371214592E-01  1.9092977458408303E+00  -16.28  -16.07

   0.2742363137121459       1.909297745840830
```

Source file: `Maple verification of computations`

```
################################################################
# Checking 'newt2' using numerical root finding capablities
# of Maple.
################################################################
lnx1% maple
    |\^/|     Maple 6 (IBM INTEL LINUX)
._|\|   |/|_. Copyright (c) 2000 by Waterloo Maple Inc.
 \  MAPLE  /  All rights reserved. Maple is a registered trademark of
 <____ ____>  Waterloo Maple Inc.
      |       Type ? for help.
> Digits := 20;
                                 Digits := 20

> f1 := sin(x*y) - 1/2;
                             f1 := sin(x y) - 1/2

> f2 := y^2 - 6*x - 2;
                                       2
                              f2 := y  - 6 x - 2


################################################################
# Locates root found by 'newt2 1.0 1.0'
################################################################
> ans := fsolve( {f1,f2}, {x,y}, {x=0.25..0.30, y=1.8..2.0});
         ans := {x = .27423631371214588082, y = 1.9092977458408301606}


################################################################
# Compute residuals of root
################################################################
> r1 := evalf(subs(ans,f1)); r2 := evalf(subs(ans,f2));
                                           -19
                               r1 := -.1 10

                                           -18
                               r2 := -.1 10


################################################################
# Locates root found by 'newt2 10.0 10.0'
################################################################
> ans := fsolve( {f1,f2}, {x,y}, {x=7..9, y=6..8});
         ans := {x = 8.0480622340064835835, y = 7.0914295740731220704}

> r1 := evalf(subs(ans,f1)); r2 := evalf(subs(ans,f2));
                                           -18
                              r1 := -.35 10

                                   r2 := 0


################################################################
# Locates root found by 'newt2 100.0 100.0'
################################################################
> ans := fsolve( {f1,f2}, {x,y}, {x=203.9..203.95, y=35.0..35.01});
         ans := {x = 203.91061457097670060, y = 35.006623479362590528}

>  r1 := evalf(subs(ans,f1)); r2 := evalf(subs(ans,f2));
                                            -16
                             r1 := -.5214 10

                                   r2 := 0


################################################################
# Another nearby, but distinct, root
################################################################
> ans := fsolve( {f1,f2}, {x,y}, {x=203..204, y=35.0..35.1});
         ans := {x = 203.95052002180667001, y = 35.010043132376172782}

> r1 := evalf(subs(ans,f1)); r2 := evalf(subs(ans,f2));
                                            -16
                              r1 := .4548 10

                                   r2 := 0

> quit;
```

Source file: nlbvp1d.f

```
c===========================================================
c     Solves 1-d non-linear boundary value problem
c
c         u''(x) + (u u')^2 + sin(u) = f(x)
c
c         on  x = [0,1]; u(0) = 0, u(1) = 0
c
c     using second-order finite difference technique,
c     Newton's method and LAPACK tridiagonal solver DGTSV.
c
c     Currently set-up for solution
c
c         u(x) = sin(4 Pi x)
c===========================================================
      program         nlbvp1d

      implicit        none

      integer         i4arg
      real*8          r8arg,        drelabs,      dvl2norm

      real*8          r8_never
      parameter     ( r8_never = -1.0d-60 )
c-----------------------------------------------------------
c     Extrema of problem domain.
c-----------------------------------------------------------
      real*8          xmin,         xmax
      parameter     ( xmin = 0.0d0,  xmax = 1.0d0 )
c-----------------------------------------------------------
      integer         maxn
      parameter     ( maxn = 32 769 )
c-----------------------------------------------------------
c     Storage for discrete x-values, unknowns, coefficient
c     exact solution and right hand side values.
c-----------------------------------------------------------
      real*8          x(maxn),      u(maxn),
     &                uexact(maxn), f(maxn)
c-----------------------------------------------------------
c     Storage for main, upper and lower diagonals of
c     tridiagonal system (Jacobian matrix) and
c     right-hand-side vector (residual vector) for use with
c     LAPACK routine DGTSV.  Other parameters needed for
c     call to DGTSV.
c-----------------------------------------------------------
      real*8          d(maxn),      du(maxn),
     &                dl(maxn),     rhs(maxn)
      integer         nrhs
      parameter     ( nrhs = 1 )
      integer         info

c-----------------------------------------------------------
c     Discretization level and size of system (# of discrete
c     unknowns)
c-----------------------------------------------------------
      integer         level,        n,            i,
     &                option
c-----------------------------------------------------------
c     Variables used in Newton iteration.
c-----------------------------------------------------------
      integer         mxiter
      parameter     ( mxiter = 50 )

      integer         iter
      real*8          guess_factor,               tol,
     &                nrm2res,      nrm2du,       nrm2u

c-----------------------------------------------------------
c     Enable following parameter for full tracing of
c     Newton iteration.
c-----------------------------------------------------------
      logical         newton_trace
      parameter     ( newton_trace = .false. )

c-----------------------------------------------------------
c     Mesh spacing and related constants
c-----------------------------------------------------------
      real*8          h,            hm2,          m2hm2,
     &                hm1by2,       hhm2,         qhm2
```

```
      real*8          rmserr
c-----------------------------------------------------------
c     Argument parsing.
c-----------------------------------------------------------
      level  = i4arg(1,-1)
      if( level .lt. 0 ) go to 900
      n = 2 ** level + 1
      if( n .gt. maxn ) then
         write(0,*) 'Insufficient internal storage'
         stop
      end if
      guess_factor = r8arg(2,r8_never)
      if( guess_factor .eq. r8_never ) go to 900
      option  = i4arg(3,0)
      tol     = r8arg(4,1.0d-8)
c-----------------------------------------------------------
c     Set up finite-difference 'mesh' (discrete x-values)
c     and define some useful constants.
c-----------------------------------------------------------
      h     =  1.0d0 / (n - 1)
      do i = 1 , n
         x(i) = xmin + (i - 1) * h
      end do
      hm2     =  1.0d0 / (h * h)
      m2hm2   = -2.0d0 / (h * h)
      hm1by2  =  0.50d0 / h
      hhm2    =  0.50d0 * hm2
      qhm2    =  0.25d0 * hm2
c-----------------------------------------------------------
c     This only ensures that x(n) = xmax EXACTLY and is not
c     essential.
c-----------------------------------------------------------
      x(n) = xmax

c-----------------------------------------------------------
c     Set up exact solution, coefficient functions and right
c     hand side vector.
c-----------------------------------------------------------
      call exact(uexact,f,x,n)

c-----------------------------------------------------------
c     Initialize unknown (u) to constant (guess_factor)
c     times exact solution.
c-----------------------------------------------------------
      do i = 1 , n
         u(i) = guess_factor * uexact(i)
      end do

c===========================================================
c     N E W T O N   L O O P
c===========================================================
      do iter = 1 , mxiter

c-----------------------------------------------------------
c        Set up tridiagonal Jacobian matrix and evaluate
c        right-hand-side (residuals)
c-----------------------------------------------------------

c-----------------------------------------------------------
c        Left boundary: Dirichlet boundary condition has
c        0 residual.
c-----------------------------------------------------------
         d(1)      = 1.0d0
         du(1)     = 0.0d0
         rhs(1)    = 0.0d0
c-----------------------------------------------------------
c        Interior: J[i,j] = d(F_i)/d_(u_j) and has non-zero
c        elements only for j = i-1, i and i+1.
c-----------------------------------------------------------
         do i = 2 , n - 1
            dl(i-1) = hm2 - hhm2 * u(i)**2 * (u(i+1) - u(i-1))
            d(i)    = m2hm2
     &                + hhm2 * u(i) * (u(i+1) - u(i-1))**2
     &                + cos(u(i))
            du(i)   = hm2 + hhm2 * u(i)**2 * (u(i+1) - u(i-1))
            rhs(i)  = hm2 * (u(i+1) - 2.0d0 * u(i) + u(i-1))
     &              + qhm2 * u(i)**2 * (u(i+1) - u(i-1))**2
     &              + sin(u(i)) - f(i)
         end do
c-----------------------------------------------------------
c        Right boundary: Dirichlet boundary condition has
```

```
c        0 residual.
c------------------------------------------------------------
         dl(n-1)    = 0.0d0
         d(n)       = 1.0d0
         rhs(n)     = 0.0d0
c------------------------------------------------------------
c        Compute l-2 norm of residuals.
c------------------------------------------------------------
         nrm2res = dvl2norm(rhs,n)
         if( newton_trace ) then
            write(0,*) 'iter = ', iter
            write(0,*) 'res  = ', nrm2res
         end if
c------------------------------------------------------------
c        Solve tridiagonal system for Newton update, delu,
c        which satisfies
c
c        J delu = residuals
c------------------------------------------------------------

         call dgtsv( n, nrhs, dl, d, du, rhs, n, info )

         if( info .eq. 0 ) then
c------------------------------------------------------------
c           Solver successful: compute norms of u and delu,
c           update solution and check for convergence.
c------------------------------------------------------------
            nrm2u  = dvl2norm(u,n)
            nrm2du = dvl2norm(rhs,n)
            if( newton_trace ) then
               write(0,*) 'du   = ', nrm2du
               write(0,*) 'u    = ', nrm2u
            end if
            do i = 1 , n
               u(i) = u(i) - rhs(i)
            end do
            if( drelabs(nrm2du,nrm2u,1.0d-6) .le. tol )
     &          go to 500
         else
c------------------------------------------------------------
c           Solver failed, write error message and exit.
c------------------------------------------------------------
            write(0,*) 'nlbvp1d: dgtsv() failed, info = ',
     &                 info
         end if
      end do
c------------------------------------------------------------
c     Newton iteration failed to converge: write error
c     message and exit.
c------------------------------------------------------------
      write(0,*) 'nlbvp1d: No convergence after ', mxiter,
     &           ' iterations'
      stop

c------------------------------------------------------------
c     Newton iteration converged: output solution or error
c     to stdout, depending on output option.  Also compute
c     rms error and output to stderr.
c------------------------------------------------------------
500   continue

      rmserr = 0.0d0
      do i = 1 , n
         if( option .eq. 0 ) then
            write(*,*) x(i), u(i)
         else
            write(*,*) x(i), (uexact(i) - u(i))
         end if
         rmserr = rmserr + (uexact(i) - u(i)) ** 2
      end do
      rmserr = sqrt(rmserr / n)
      write(0,*) 'rmserr = ', rmserr

      stop

 900  continue
         write(0,*) 'usage: nlbvp1d <level> <guess_factor> '//
     &              '[<option> <tol>]'
         write(0,*)
         write(0,*) '       Specify option .ne. 0 for output'
         write(0,*) '       of error instead of solution'
      stop

      end

c============================================================
c     Computes exact values for u(x) (unknown function)
c     and f(x) (right hand side function).  x array must
c     have been previously defined.
c============================================================
      subroutine exact(u,f,x,n)

         implicit        none
         integer         n
         real*8          u(n),     f(n),      x(n)

         real*8          pi4
         integer         i

         pi4 = 16.0d0 * atan(1.0d0)
         do i = 1 , n
            u(i) = sin(pi4 * x(i))
            f(i) = -pi4**2 * sin(pi4 * x(i)) +
     &             pi4**2 * (sin(pi4 * x(i)) *
     &                       cos(pi4 * x(i)))**2 +
     &             sin(sin(pi4 * x(i)))
         end do

         return

      end

c============================================================
c     dvl2norm:  Returns l2-norm of double precision vector.
c============================================================
      real*8 function dvl2norm(v,n)

         implicit      none

         integer       n
         real*8        v(n)
         integer       i

         dvl2norm = 0.0d0
         do i = 1 , n
            dvl2norm = dvl2norm + v(i) * v(i)
         end do
         if( n .gt. 0 ) then
            dvl2norm = sqrt(dvl2norm / n)
         end if

         return

      end

c============================================================
c     drelabs:  Function useful for 'relativizing' quantity
c     being monitored for detection of convergence.
c============================================================
      real*8 function drelabs(dx,x,xfloor)

         implicit      none

         real*8        dx,     x,    xfloor

         if( abs(x) .lt. abs(xfloor) ) then
            drelabs = abs(dx)
         else
            drelabs = abs(dx/x)
         end if

         return

      end
```

Source file: `Makefile`

```
.IGNORE:

F77_COMPILE  = $(F77) $(F77FLAGS) $(F77CFLAGS)
F77_LOAD     = $(F77) $(F77FLAGS) $(F77LFLAGS)

.f.o:
	$(F77_COMPILE) $*.f

EXECUTABLES = nlbvp1d

all: $(EXECUTABLES)

nlbvp1d: nlbvp1d.o
	$(F77_LOAD) nlbvp1d.o \
	            -lp329f -llapack $(LIBBLAS) -o nlbvp1d
clean:
	rm *.o
	rm out*
	rm err[0-9]
	rm $(EXECUTABLES)

vclean: clean
	rm *.ps
```

Source file: `Nlbvp1d`

```
#!/bin/sh -x
P=`basename $0`

#-----------------------------------------------------------
# Nlbvp1d: script which runs 'nlbvp1d' and plots results.
#-----------------------------------------------------------

# Test for executable and make it if it doesn't exist.
test -f nlbvp1d || make

# Generate level-6 solution with guess_factor = 1.0

nlbvp1d 6 1.0  > out6

gnuplot<<END
set terminal postscript portrait
set output "soln6.ps"
set size square
set title "Solution of Nonlinear BVP\nguess_factor = 1.0"
set xlabel "x"
set ylabel "u(x)"
plot [0:1] [-1:1]  \
     sin(12.56637061435917*x) title "exact solution", \
  "out6" notitle
quit
END

# Perform convergence test for guess_factor = 1.0

nlbvp1d 5 1.0 1 > err5
nlbvp1d 6 1.0 1 | nf _1 '4 * _2' > err6
nlbvp1d 7 1.0 1 | nf _1 '16 * _2' > err7

gnuplot<<END
set terminal postscript portrait
set output "err567.ps"
set size square
set title "Convergence Test of Solution \
of Nonlinear BVP\nguess_factor = 1.0"
set xlabel "x"
set ylabel " "
set key 0.725,0.014
plot \
     "err5" title "Level-5 error", \
     "err6" title "4 * Level-6 error", \
     "err7" title "16 * Level-7 error"
quit
END

# Generate 3 distinct solutions and plot the results

nlbvp1d 6 0.7 > out6-0.7
nlbvp1d 6 1.0 > out6-1.0
nlbvp1d 6 1.1 > out6-1.1

gnuplot<<END
set terminal postscript portrait
set output "allsolns.ps"
set size square
set title "Three Distinct Solutions \
of Nonlinear BVP"
set xlabel "x"
set ylabel "u(x)"
plot \
     "out6-0.7" title "guess_factor = 0.7" with lines, \
     "out6-1.0" title "guess_factor = 1.0" with lines, \
     "out6-1.1" title "guess_factor = 1.1" with lines
quit
END

gnuplot<<END
set terminal postscript portrait
set output "allsolnsz.ps"
set size square
set title "Three Distinct Solutions \
of Nonlinear BVP - Detail"
set xlabel "x"
set ylabel "u(x)"
plot [0.025:0.25] [0.5:1.1] \
     "out6-0.7" title "guess_factor = 0.7" with lines, \
     "out6-1.0" title "guess_factor = 1.0" with lines, \
     "out6-1.1" title "guess_factor = 1.1" with lines
quit
END

ls -lt *.ps
```

Figure file: `soln6.ps`

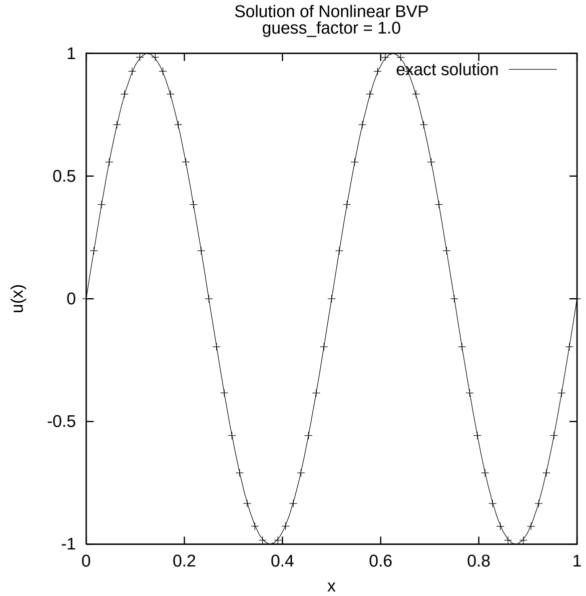

Figure file: `allsolns.ps`

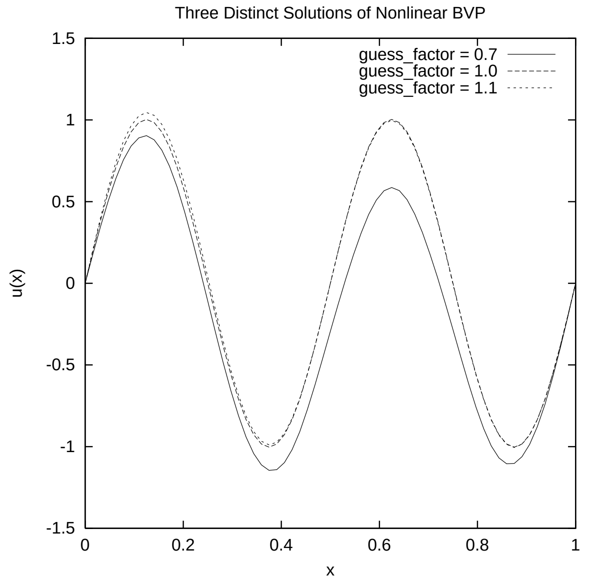

Figure file: `err567.ps`

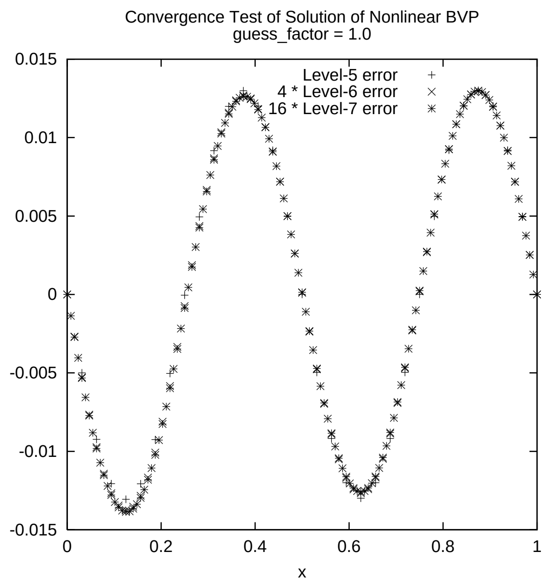

Figure file: `allsolnsz.ps`