User's Guide for TWPBVP: A Code for Solving Two-Point Boundary Value Problems

J. R. Cash
Department of Mathematics
Imperial College
London, England
e-mail: j.cash@ic.ac.uk

Margaret H. Wright
AT&T Bell Laboratories
Murray Hill, New Jersey
USA
e-mail: mhw@research.att.com

1 Introduction

The code TWPBVP (written in Fortran 77) is designed for the numerical solution of the two-point boundary value problem

$$\frac{du}{dx} = f(x, u), \qquad a \le x \le b, \tag{1}$$

$$g_i(u(a)) = 0, \quad i = 1, \ldots p; \qquad g_i(u(b)) = 0, \quad i = p+1, \ldots, n, \tag{2}$$

where $x \in \mathcal{R}$, $u \in \mathcal{R}^n$, $g_i \in \mathcal{R}$ for $1 \le i \le n$, and $0 \le p \le n$. The functions f and $\{g_i\}$ are assumed to be differentiable. The method used in TWPBVP is a deferred correction method based on mono-implicit Runge-Kutta formulas and adaptive mesh refinement. For details about the method, see [Cash86, Cash88, CW90, CW91].

Two aspects of the formulation (1)–(2) should be noted:

1. The problem must be posed as a first-order system. This requirement is not unduly restrictive, however, since standard techniques can be used to convert an nth-order equation to a system of n first-order equations. For example, the second-order problem

$$y'' = f(x, y, y'), \quad 0 \le x \le 1, \quad y(0) = \alpha, \quad y(1) = \beta,$$

can be converted to the following first-order system:

$$\begin{aligned} u' &= z \qquad & u(0) &= \alpha \\ z' &= f(x, u, z) \qquad & u(1) &= \beta. \end{aligned}$$

2. The boundary conditions must be separated. TWPBVP allows all the boundary conditions to be imposed at one end of the interval $[a, b]$, so that initial value problems are handled as a special case. It is hoped to extend TWPBVP to handle non-separated boundary conditions; see [AMR88] for a discussion of techniques that allow some non-separated boundary conditions to be converted to the separated form needed here.

2 Formal parameters

The calling sequence for TWPBVP is

```
      subroutine twpbvp(ncomp, nlbc, nucol, aleft, aright,
     *       nfxpnt, fixpnt, ntol, ltol, tol,
     *       linear, givmsh, giveu, nmsh,
     *       xx, nudim, u, nmax,
     *       lwrkfl, wrk, lwrkin, iwrk,
     *       fsub, dfsub, gsub, dgsub, iflbvp)
```

We now describe the formal parameters.

ncomp - Integer, input. ncomp is the number of first-order differential equations (n in (1)), and is the number of components of u at each mesh point. ncomp must be positive.

nlbc - Integer, input. nlbc is the number of boundary conditions at the left endpoint (p in (2)). nlbc must be nonnegative and must not exceed ncomp. If nlbc = ncomp, all the boundary conditions occur at the left endpoint, so the problem is an initial value problem. If nlbc is zero, all the boundary conditions occur at the right endpoint.

nucol - Integer, input. nucol is the declared column dimension of the two-dimensional array u used to store the computed solution (see the discussion below of the parameter u). nucol must be positive, and is an upper bound on nmax, the maximum allowed number of mesh points. In the specification of nmax, below, the actual formula for computing nmax is given. In order to make full use of the floating-point workspace, the user should select nucol equal to nmax.

aleft - Floating-point, input. The left endpoint of the interval of interest (the quantity a in (1)).

aright - Floating-point, input. The right endpoint of the interval of interest (the value b in (1)). aright must strictly exceed aleft.

nfxpnt - Integer, input. The number of "fixed points", i.e., points that must be included in every mesh in addition to the endpoints a and b. nfxpnt must be nonnegative, and may be zero. If nfxpnt = 0, only the two endpoints are required to appear in every mesh.

fixpnt - Floating-point array of size at least nfxpnt, input. If nfxpnt = 0, the fixpnt array must still be declared, say as dimension fixpnt(1), but is never accessed. If nfxpnt > 0, the fixpnt array contains nfxpnt fixed points that the user wishes to be included in every mesh. fixpnt(1) must strictly exceed aleft; for $1 \le i \le$ nfxpnt $- 1$, fixpnt($i+1$) must strictly exceed fixpnt(i); and fixpnt(nfxpnt) must be strictly less than aright.

ntol - Integer, input. The number of tolerances used to determine termination of twpbvp, i.e., the number of components whose estimated accuracy is to be tested. ntol must be positive.

ltol - Integer array of size ntol, input. For each i, ltol(i) gives the index of the component of the computed solution u controlled by the ith tolerance. For example, if ntol = 2, ltol(1) = 2 and ltol(2) = 3, then component 2 of u is controlled by tol(1) and component 3 is controlled by tol(2); see below for a description of tol. Each element of ltol must be an integer between 1 and ncomp.

tol - Floating-point array of size ntol, input. For each i, tol(i) gives the ith error tolerance used in performing termination tests. For each $i = 1, \ldots,$ ntol, the code attempts at each mesh point x_k to approximate the true (unknown) solution $u(x_k)$ by a quantity u_k such that

$$\frac{|u_k^j - u^j(x_k)|}{\max(1, |u_k^j|)} < \texttt{tol}\,(i) \tag{3}$$

for each $i = 1, \ldots,$ ntol and $j =$ ltol(i), where u_k^j denotes component j of u_k. Relation (3) is a mixed relative/absolute error criterion of the type normally used for solving differential equations.

`linear` - Logical, input. If `linear` is `.true.`, the problem is treated as linear. If `linear` is `.false`, the problem is solved as nonlinear. Note that different algorithmic strategies are used for linear and nonlinear problems; hence, if one solves the same linear problem twice, with `linear` set to `.true.` and then to `.false.`, the computed solutions and meshes may well be different, although both solutions should satisfy the same overall error criterion.

`givmsh` - Logical, input. If `givmsh` is `.true.`, the user must define two parameters: `nmsh` and the `xx` array; see below. If `givmsh` is `true.`, `nmsh` must contain a positive number of initial mesh points, and `xx` must contain these mesh points. If `givmsh` is `.false.`, the user need not specify `nmsh` or `xx`, which are chosen by default; see the explanations below of `nmsh` and `xx`.

`giveu` - Logical, input. If `giveu` is `.false`, the code sets the initial trial solution at all mesh points to the constant value `uval0`, which is contained in the labeled common area `algprs`; the default value of `uval0`, set in a block data routine, is zero. To change `uval0`, the user should include the labeled common area `algprs` in the calling routine and set `uval0` to the desired value; see Section 3.

If `giveu` is `.true.`, `givmsh` must also be set to `.true.` When `giveu` is `.true.`, the user must set `nmsh` to the initial number of mesh points, `xx` to the initial array of mesh points, and `u`(i, j), $i = 1, \ldots,$ `ncomp` and $j = 1, \ldots,$ `nmsh`, to the initial trial solution at these mesh points. However, if the first Newton procedure fails with these values for `xx` and `u`, the code reverts to setting all components of the initial trial solution to `uval0` for the next mesh.

`nmsh` - Integer, optional input and output. If the parameter `givmsh` is `.true.`, the user must set `nmsh` to the positive initial number of mesh points (including the two endpoints). If `givmsh` is `.false.`, the user need not set `nmsh`. If `givmsh` is `.false.`, the initial value of `nmsh` is set to the greater of `nfxpnt`+2 and `nminit`, an integer in the labeled common area `algprs`. (The default value of `nminit` is set in a block data routine to 7; see Section 3.) To specify another initial value for `nmsh` without specifying the initial mesh points, the user can include the labeled common `algprs` in the calling routine and set `nminit` to the desired value; see Section 3.

On output, `nmsh` contains the final number of mesh points.

`xx` - Floating-point array, of size `nmax` (see below; nmax cannot exceed `nucol`), optional input and output. When `givmsh` is `.false.`, the initial mesh `xx` is defined as follows. If `nfxpnt` $= 0$, the initial mesh contains `nmsh` points equally spaced in [`aleft`, `aright`]. If `nfxpnt` > 0, the initial mesh contains (if possible) a total of `nmsh` points. These points are equally spaced in each subinterval [`aleft`, `fixpnt`(1)], [`fixpnt`(1), `fixpnt`(2)], …, [`fixpnt`(`nfxpnt`), `aright`].

If `givmsh` is `.true.`, the user must define the `xx` array on input as an initial mesh of strictly increasing values, with `xx`(1) = `aleft` and `xx`(`nmsh`) = `aright`. If `nfxpnt` > 0 and the user sets the initial mesh, the desired fixed points *must* be included in the `xx` array, since the code performs no tests to check for their inclusion! If `givmsh` is `.false.`, `nmsh` is set to the maximum of `nfxpnt`+2 and `nminit` as defined in the labeled common area `algprs`.

On output, `xx` contains the final array of `nmsh` mesh points.

`nudim` - Integer, input. `nudim` is the declared row dimension of the array `u`. `nudim` must be greater than or equal to `ncomp`.

`u` - Floating-point array, of declared size `(nudim, nucol)`, and of conceptual size `(ncomp, nmsh)`, where `ncomp` is the number of components and `nmsh` is the number of mesh points. The `u` array is an optional input parameter, and an output parameter. If `giveu` (see above) is `.false.`, the `u` array need not be set by the user.

If `giveu` is .true., the user must provide an initial array `u`(i, j), $i = 1, \ldots$ `ncomp` and $j = 1, \ldots$ `nmsh`, corresponding to the desired initial trial solution, where `nmsh` is the user-specified number of initial mesh points. In this case, `u`$(*, 1)$ corresponds to the estimated solution at `aleft`, and `u`$(*,$`nmsh`$)$ corresponds to the estimated solution at `aright`.

`nmax` - Integer, output. `nmax` is the maximum number of mesh points possible with the given values of `ncomp`, `ntol`, `lwrkfl` (size of floating-point workspace), `lwrkin` (size of integer workspace) and `nucol`. (See below for the descriptions of `lwrkfl` and `lwrkin`.) `nmax` can be no larger than the input parameter `nucol`. When the value of `lwrkfl` is much greater than `ncomp` $*$ `ncomp`, the maximum number of mesh points allowed by the floating-point workspace limit is

$$\frac{\texttt{lwrkfl} - 5(\texttt{ncomp} * \texttt{ncomp}) - 2\texttt{ntol} - 9\texttt{ncomp}}{4(\texttt{ncomp} * \texttt{ncomp}) + 12\texttt{ncomp} + 3}$$

When `lwrkin` is much greater than `ncomp`, the maximum number of mesh points allowed by the integer workspace limit is $(\texttt{lwrkin} - \texttt{ncomp})/(\texttt{ncomp} + 2)$.

The value of `nmax` is calculated in `twpbvp`, and is not allowed to exceed `nucol`. The user can determine the exact maximum number of mesh points corresponding to specified values of `ncomp`, `lwrkfl` and `lwrkin` by calling `twpbvp` with `nucol` $= 1$ and the parameter `iprint` set to 0 (its default value) or 1; see below for a discussion of `iprint`.

`lwrkfl` - Integer, input. The size of the user-provided floating-point workspace array `wrk`.

`wrk` - Floating-point array of size `lwrkfl`, input. User-provided floating-point workspace, used to store intermediate values.

`lwrkin` - Integer, input. The size of the user-provided integer workspace array `iwrk`.

`iwrk` - Integer array of size `lwrkin`, input. User-provided integer workspace, used to store intermediate values.

`fsub` - Name of user-provided subroutine. `fsub` *must* be declared `external` in the calling routine. `fsub` defines the function f in the formulation (1) of the system of first-order differential equations. `fsub` must have the following declaration:

```
subroutine fsub(ncomp, x, u, f)
```

`ncomp` (input to `fsub`) is the number of components of `u`;

`x` (input to `fsub`) is a floating-point scalar;

`u` (input to `fsub`) is a floating-point array of size `ncomp`;

`f` (output from `fsub`) is a floating-point array of dimension `ncomp`. On exit from `fsub`, the array `f` must contain the `ncomp`-dimensional vector `f` whose ith component is the value of the ith component of the vector f of (1) evaluated at `x` and `u`.

`dfsub` - Name of user-provided subroutine. `dfsub` *must* be declared `external` in the calling routine. `dfsub` calculates the Jacobian matrix of `f` as defined by the subroutine `fsub`. `dfsub` must have the following declaration:

```
subroutine dfsub(ncomp, x, u, df)
```

`ncomp` (input to `dfsub`) is the number of components of `u`;

`x` (input to `dfsub`) is a floating-point scalar;

`u` (input to `dfsub`) is a floating-point array of dimension `ncomp`;

`df` (output from `dfsub`) is an `ncomp` by `ncomp` floating-point array whose declared row dimension in the routine `dfsub` must be `ncomp`. On exit from `dfsub`, the array `df` must contain the `ncomp` by `ncomp` Jacobian matrix of f from (1) (`f` of `fsub`) with respect to u, namely `df`(i, j) is the partial derivative of the ith function f_i with respect to the jth component of u.

`gsub` - Name of user-provided subroutine. `gsub` *must* be declared `external` in the calling routine. `gsub` is called `ncomp` times each time the boundary conditions are calculated; the ith call to `gsub` defines the ith function $g_i(\cdot)$ in the boundary conditions of (2). `gsub` must have the following declaration:

```
subroutine gsub(i, ncomp, u, g)
```

`i` (input to `gsub`) is an integer ranging from 1 to `ncomp`;

`ncomp` (input to `gsub`) is the number of components of `u`;

`u` (input to `gsub`) is a floating-point array of dimension `ncomp`;

`g` (output from `gsub`) is a floating-point scalar. On exit from the ith call to `gsub`, the value of `g` must contain the value of the function g_i from (2) that defines the ith boundary condition evaluated at u.

`dgsub` - Name of user-provided subroutine. `dgsub` *must* be declared `external` in the calling routine. `dgsub` calculates the Jacobian matrices corresponding to the functions g_i of (2) and `gsub` that define the boundary conditions. `dgsub` must have the following declaration:

```
subroutine dgsub(i, ncomp, u, dg)
```

`i` (input to `dgsub`) is an integer ranging from 1 to `ncomp`;

`ncomp` (input to `dgsub`) is the number of components of `u`;

`u` (input to `dgsub`) is a floating-point array of dimension `ncomp`;

`dg` (output from `dgsub`) is a floating-point array of dimension `ncomp`. On exit from the ith call of `dgsub`, the vector `dg` must contain the `ncomp` partial derivatives of the ith function g_i of (2) with respect to u.

`iflbvp` - Integer, output. `iflbvp` indicates the result of `twpbvp`. If the routine has terminated with apparent success, `iflbvp` $= 0$. If one of the input parameters is invalid, `iflbvp` $= -1$. If the number of mesh points needed for the next iteration would exceed `nmax`, then `iflbvp` $= 1$.

3 Related parameters

Some other parameters that may be of interest to the user, but do not occur in the formal declaration of `twpbvp`, are stored in the labeled common area `algprs`, whose declaration is

```
logical pdebug
common/algprs/ nminit, pdebug, iprint, idum, uval0
```

Any of the parameters in `algprs` may be changed by the user by including this labeled common in the calling routine and setting the value as desired.

The value of the integer `nminit` is discussed above under the formal parameter `nmsh`. `nminit` specifies the default initial number of mesh points, and, if not altered, is set to 7.

`pdebug` is a logical variable whose default value is `.false.`; if `pdebug` is set to `.true.`, an extensive debug printout will occur during execution of `twpbvp`.

`iprint` is an integer variable controlling the amount of non-debug printout. Its default value is zero, which means that some intermediate printing occurs (each Newton iteration, each change of mesh, and so on; see the example output in Section 5.3). If `iprint` is set to 1, more printing occurs. If `iprint` is set to -1, no printing at all occurs in `twpbvp`. Values of `iprint` other than 0, 1 and -1 are invalid.

`idum` is a dummy integer value needed only for common alignment. It serves no purpose in the algorithm.

`uval0` is a floating-point value that defines the initial value of the trial solution when `giveu` is `.false.`, or in some instances when an intermediate Newton process fails. Unless changed by the user, the default value of `uval0` is zero. The user may wish to change `uval0` if $u = 0$ is inappropriate for some reason—for example, the function f in (1) is undefined for $u = 0$.

The user should also look at the routine d1mach (which is provided automatically via netlib). The variables specified in this routine describe machine precision and may need to be changed depending on which machine is being used.

4 Suggestions for difficult problems

For difficult, nonlinear singular perturbation problems, the code TWPBVP may experience severe difficulties in obtaining convergence for the Newton iteration scheme. In such circumstances two approaches may be helpful. The first is to experiment with different initial meshes. This can involve using a different number of equally spaced points (by changing the default value of the parameter `nminit`; see Section 3), or, alternatively, trying a graded mesh in which extra mesh points are placed in regions of known or suspected difficulty.

A second possibility is to use *continuation*, which is particularly appropriate if a small parameter is associated with the given problem. For example, a large class of important singular perturbation problems appear in the form

$$\epsilon y'' = f(x, y, y'),$$

where ϵ is a very small parameter. An often-successful strategy for solving such problems is to begin with a relatively large value of ϵ, and then to solve a sequence of problems with ϵ steadily decreasing to the required value. The power of this approach comes from the fact that information regarding meshes and solution values can be passed from one problem to the next. Continuation is fully described in [AMR88]. J. R. Cash, one of the authors of TWPBVP, and R. W. Wright have developed an automatic continuation code that can be made available (with no guarantees) to interested users. For a copy of this continuation code e-mail either j.cash or r.wright@ic.ac.uk.

5 A sample problem

5.1 Mathematical formulation

Consider a second-order boundary value problem parameterized in terms of ϵ:

$$\begin{aligned} \epsilon u'' &= -e^u u' + \tfrac{1}{2}\pi \sin(\tfrac{1}{2}\pi x) e^{2u}, \quad 0 \le x \le 1, \\ u(0) &= 0, \quad u(1) = 0. \end{aligned} \tag{4}$$

(As ϵ decreases toward zero, (4) becomes more difficult to solve numerically.) In order to apply TWPBVP to (4), it must be converted to a system of two first-order equations:

$$u' = z, \qquad z' = \frac{1}{\epsilon}\left(-e^u z + \tfrac{1}{2}\pi\ \sin(\tfrac{1}{2}\pi x)\, e^{2u}\right), \tag{5}$$

with $u(0) = 0$ and $u(1) = 0$.

Suppose now that we wish to solve (5) with $\epsilon = 10^{-2}$, and to achieve an accuracy of approximately 10^{-6} in the value of u only. We shall provide neither an initial mesh nor an initial guess for the solution. The problem is to be solved on a machine with IEEE arithmetic, and hence double precision is used (approximately 16 decimal digits).

5.2 Sample Fortran routines

The following Fortran 77 main program calls `twpbvp` with the user-specified parameters given above. The subroutines `fsub`, `dfsub`, `gsub` and `dgsub` define the function f and the boundary conditions g_1 and g_2 for problem (5).

```
      implicit double precision (a-h,o-z)

      dimension  fixpnt(2), ltol(2), tol(2)
      dimension  u(2, 1000), xx(1000)
      dimension wrk(10000), iwrk(6000)
      character*30 pname

      logical linear, giveu, givmsh
      external fsub, dfsub, gsub, dgsub

*  The value of eps represents the parameter epsilon
*  appearing in the test problem.

      common/probs/eps, pi

      logical pdebug
      common/algprs/ nminit, pdebug, iprint, idum, uval0

*  blas: dload
*  double precision datan

      parameter (one = 1.0d+0, zero = 0.0d+0, ten = 10.0d+0)

      pi = 4.0d+0*datan(one)

      pname = 'Test problem with eps = .01'

      eps = 1.0d-2

      nudim = 2
      lwrkfl = 10000
      lwrkin = 6000
      nucol = 1000

      ncomp = 2
      nlbc = 1
      ntol = 1
      ltol(1) = 1
      tol(1) = 1.0d-2
      nfxpnt = 0

      aleft = zero
      aright = one

      linear = .false.
      giveu = .false.
      givmsh = .false.


      call twpbvp(ncomp, nlbc, nucol, aleft, aright,
     *       nfxpnt, fixpnt, ntol, ltol, tol,
     *       linear, givmsh, giveu, nmsh,
     *       xx, nudim, u, nmax,
     *       lwrkfl, wrk, lwrkin, iwrk,
     *       fsub, dfsub, gsub, dgsub, iflbvp)
```

```
      write(6,771) pname
      write(6,772) ncomp, nlbc, ntol
      write(6,773) aleft, aright
      if (linear) write (6,774)
      if (.not. linear) write(6,775)
      if (nfxpnt .gt. 0) write(6,776) nfxpnt
      write(6,777) (ltol(i), i=1,ntol)
      write(6,778) (tol(i), i=1,ntol)
      if (iflbvp .eq. 0) then
         write(6,901) nmsh
         if (nmsh .lt. 25) then
            iminc = 1
         elseif (nmsh .lt. 75) then
            iminc = 5
         elseif (nmsh .lt. 200) then
            iminc = 10
         elseif (nmsh .lt. 1000) then
            iminc = 20
         else
            iminc = 50
         endif
         do 100 i = 1, nmsh-1, iminc
            write(6,900) i, xx(i), (u(j,i), j=1,ncomp)
  100    continue
         write(6,900) nmsh, xx(nmsh), (u(j,nmsh), j=1,ncomp)
      endif

      stop
  771 format(1h ,a30)
  772 format(1h ,'ncomp=',i5,5x,'nlbc=',i5,5x,'ntol=',i5)
  773 format(1h ,'aleft =',1pe11.3,5x,'aright=',1pe11.3)
  774 format(1h ,'solved as a linear problem')
  775 format(1h ,'solved as a nonlinear problem')
  776 format(1h ,'nfxpnt =',i5)
  777 format(1h ,'ltol',5x,5i5)
  778 format(1h ,'tolerances',5(1pe11.3))
  900 format(1h ,i4,5(1pe14.6))
  901 format(1h ,'final solution, nmsh =',i8)
      end

      subroutine fsub(ncomp, x, u, f)
      implicit double precision (a-h,o-z)
      dimension u(*), f(*)
      common/probs/eps, pi

      parameter (half = 0.5d+0, two = 2.0d+0)

*  double precision dexp, dsin

*  nonlinear problem 1

      f(1) = u(2)
      f(2) = (-dexp(u(1))*u(2) +
     *          half*pi*dsin(half*pi*x)*dexp(two*u(1)))/eps
```

```
      return
      end

      subroutine dfsub(ncomp, x, u, df)
      implicit double precision (a-h,o-z)
      dimension u(*), df(ncomp,*)
      common/probs/eps, pi
      parameter (half = 0.5d+0, one = 1.0d+0, zero = 0.0d+0)
      parameter (two = 2.0d+0)

*  double precision dexp, dsin

      df(1,1) = zero
      df(1,2) = one
      df(2,1) = -(dexp(u(1))*u(2)
     *            - pi*dsin(pi*half*x)*dexp(two*u(1)))/eps
      df(2,2) = -dexp(u(1))/eps
      return
      end

      subroutine gsub(i, ncomp, u, g)
      implicit double precision (a-h,o-z)
      dimension u(*)
      g = u(1)
      return
      end


      subroutine dgsub(i, ncomp, u, dg)
      implicit double precision (a-h,o-z)
      dimension u(*), dg(*)
      parameter (one = 1.0d+0, zero = 0.0d+0)

      dg(1) = one
      dg(2) = zero
      return
      end
```

5.3 Sample output

The complete output produced by running this example on an SGI 4D/240S with 25 MHz MIPS R3000 cpu chip, under IRIX System V release 3.3.1, with binary IEEE arithmetic, is given next.

```
 Test problem with eps = .01
 ncomp =    2     nlbc =    1     ntol =    1
 aleft =  0.000E+00     aright =  1.000E+00
 solved as a nonlinear problem
 ltol         1
 tolerances  1.000E-02
 nmax from workspace =     231
 unimsh.  nmsh =    7
 initu, uval0    0.00000D+00
  iter       alfa      merit       rnsq
     0  2.315E-01  1.043E+03  2.122E+03
     1  1.558E-02  1.722E+04  2.060E+03
```

```
    2  4.499E-03  1.044E+05  2.043E+03
 dblmsh.  the doubled mesh has       13 points.
 iter       alfa      merit       rnsq
    0  2.404E-01  5.077E+01  4.582E+03
    1  4.809E-01  1.320E+02  1.488E+03
    2  9.617E-01  7.193E+02  1.743E+02
    3  1.000E-02  8.279E+04  1.722E+02
    4  3.271E-03  4.623E+05  1.719E+02
smpmsh, new nmsh =     40
 iter       alfa      merit       rnsq
    0  1.000E+00  1.506E+02  6.596E+01
    1  1.000E+00  5.433E+00  4.495E+00
    2  1.000E+00  2.004E-03  8.292E-04
    3  1.000E+00  1.627E-10  6.306E-11
Convergence, iter =    4    rnsq =   6.306E-11
fixed Jacobian convergence    1  1.187E-10

final solution, nmsh =      40

   i  mesh point        u(1)          u(2)
   1  0.000000E+00  0.000000E+00 -4.904288E+01
   6  4.166667E-02 -6.130136E-01 -3.138167E+00
  11  8.333333E-02 -6.638966E-01 -2.759269E-01
  16  1.250000E-01 -6.655629E-01  9.117284E-02
  21  1.666667E-01 -6.596432E-01  1.807697E-01
  26  2.083333E-01 -6.508858E-01  2.382272E-01
  31  2.500000E-01 -6.398296E-01  2.923964E-01
  36  6.666667E-01 -3.986648E-01  8.886575E-01
  40  1.000000E+00 -2.509872E-30  1.546497E+00
```

References

[AMR88] U. Ascher, R. M. M. Mattheij, and R. D. Russell, *Numerical Solution of Boundary Value Problems for Ordinary Differential Equations*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[Cash86] J. R. Cash (1986), On the numerical integration of nonlinear two-point boundary value problems using iterated deferred corrections, Part 1: A survey and comparison of some one-step formulae, *Comput. Math. Appl.* 12a, 1029–1048.

[Cash88] J. R. Cash (1988), On the numerical integration of nonlinear two-point boundary value problems using iterated deferred corrections, Part 2: The development and analysis of highly stable deferred correction formulae, *SIAM J. Numer. Anal.* 25, 862–882.

[CW90] J. R. Cash and M. H. Wright (1990), Implementation issues in solving nonlinear equations for two-point boundary value problems, *Computing* 45, 17–37.

[CW91] J. R. Cash and M. H. Wright (1991), A deferred correction method for nonlinear two-point boundary value problems: Implementation and numerical evaluation, *SIAM J. Sci. Stat. Comput.* 12, 971–989.