Numerical Solutions of Differential Equations
A Tutorial

Henrique Fleming

27-2-2002

1 Introduction

This is a tutorial on solving the ordinary differential equations of mechanics by numerical methods. I will be very straightful: we will learn how to solve second order ODE (ordinary differential equations) using a very good routine developed, years ago, by Shampine and Gordon. It is based on the Adams method, but we couldn't care less. Suffice it to say that the routine is of excellent reliability. It is in Fortran. If you don't know Fortran, it's OK. Fortran is the best language to ignore, as you can learn it in two days: I know, I learnt it in three days, and I am really slow!

Don't try to understand the routine: it is enormous and sophisticated. It does the following: you supply the differential equation, the initial conditions, the initial and end values of the independent variable (the time, in our case) and the precision you want. The routine then calculates the integration step so that the required precision is attained, and produces the position and velocity at the end point. (Actually, the step varies along the computation, being determined anew at each step).

In case you want to know more about the numerical treatment of ODE's, take a good look at Hamming, *Numerical Methods for Scientists and Engineers*. This is a great book! Its chapter on differential equations is illuminating, even if you don't care about numerical methods.

2 Basics of the method

Suppose the equation is

$$\frac{dy}{dt} = f(y,t) \tag{1}$$

There are four things to be done:

1.Write a Fortran routine informing the program which ODE is supposed to be integrated. Call it *derivs.f*, say.

2.Fetch the Shampine-Gordon routine *ode.f*.

3.Write the Fortran program (what do you want to do with the equation). Call it *main.f*, say.

4.Compile everything, with a command such as this:

g77 main.f ode.f derivs.f -o main.out

The executable will be called *main.out*. This is, of course, the optimistic version. Probably error messages will crop up, as always... You'll have to debug. (I think it was Marvin Minsky who said "Programming is debugging a white sheet of paper!").

Let us proceed by parts.

2.1 derivs.f

Suppose the equation is

$$\frac{dy}{dt} = f(y,t) \tag{2}$$

with $f(y,t) = y * y$.

We have to write this equation in the way required by *ode.f*. This means writing a routine like this:

```
subroutine derivs(c,a,b)
integer nt
parameter(nt=1)
double precision a(nt),b(nt),c

b(1) = a(1)*a(1)

return
end
```

In this subroutine, y is represented by the array a(nt), and $\frac{dy}{dt}$ by the array b(nt). In this case , as nt=1, this seems ridiculous, but we will soon see that the normal case requires nt¿ 1. So, the line

```
b(1) = a(1)*a(1)
```

is telling that $\frac{dy}{dt} = y^2$, as it should.

Consider now the second order equation

$$\frac{d^2y}{dt^2} = f(y, \frac{dy}{dt}, t) \tag{3}$$

The best way to solve it is transforming it into a system of two first order equations. Suppose $f(y, \frac{dy}{dt}, t) = y^2$. The trick is defining an auxiliary variable through

$$w = \frac{dy}{dt} \tag{4}$$

which allows Eq.(3) to be written

$$\frac{dy}{dt} = w \tag{5}$$

$$\frac{dw}{dt} = f(y, w, t) \tag{6}$$

so that, in our particular case, we would have

$$\frac{dy}{dt} = w \tag{7}$$

$$\frac{dw}{dt} = y^2 \tag{8}$$

Let's put this in the language required by *ode.f.* Here we will have two equations, so nt=2. The array a(2) will have two elements,

$$a(1) = y \tag{9}$$

$$a(2) = w \tag{10}$$

whereas the array b(2) will be given by

$$b(1) = \frac{dy}{dt} \tag{11}$$

$$b(2) = \frac{dw}{dt} \tag{12}$$

The subroutine *derivs.f* will then be written:

```
subroutine derivs (c,a,c)
integer nt
parameter (nt=2)
double precision a(nt), b(nt), c
b(1)= a(2)
b(2) = a(1)*a(1)
return
end
```

Let us now move to a more concrete example: a particle moving under the action of the gravitational field. We treat the problem as a 3-dimensional one for generality (though it is 2-dimensional, the orbits being plane curves). The equationas are:

$$m\frac{d^2\vec{r}}{dt^2} = -G\frac{mM}{r^2}\frac{\vec{r}}{r} \tag{13}$$

or, written in components,

$$m\frac{d^2x}{dt^2} = -G\frac{mM}{r^2}\frac{x}{r} \tag{14}$$

$$m\frac{d^2y}{dt^2} = -G\frac{mM}{r^2}\frac{y}{r} \tag{15}$$

$$m\frac{d^2z}{dt^2} = -G\frac{mM}{r^2}\frac{z}{r} \tag{16}$$

So we have three second-order differential equations. In terms of first-order ones, we will have six. So, in this case nt=6, and the arrays a(*) and b(*) will have six elements each. Without more ado, we write the subroutine *derivs.f* for this case.

```
 subroutine derivs(c, a, b)
      integer nt
      parameter (nt=6)
      double precision a(nt), b(nt), c
      double precision G, r
      parameter (G=1.D0)
      r=dsqrt(a(1)**2+a(3)**2+a(5)**2)
      b(1)=a(2)
      b(2)=-G*a(1)/r**3
      b(3)=a(4)
      b(4)=-G*a(3)/r**3
      b(5)=a(6)
      b(6)=-G*a(5)/r**3

      return
      end
```

where an arbitrary value was given to G. This is double-precision Fortran, what explains de *dsqrt*, which is the double-precision version of the square-root function. So much for the *derivs.f* routine.

3 The main routine

This is called *ode.f*, and does all the dirty job. Don't try to hack it: as it is it was so much tested as to be considered the most reliable routine available, being used to contro critical instruments in hospitals, for instance. You can use it as it is.

```
      subroutine ode(f,neqn,y,t,tout,relerr,abserr,iflag,work,iwork)
      implicit real*8(a-h,o-z)
c
c   double precision subroutine ode integrates a system of neqn
c   first order ordinary differential equations of the form:
c             dy(i)/dt = f(t,y(1),y(2),...,y(neqn))
c             y(i) given at  t .
c   the subroutine integrates from  t  to  tout .  on return the
c   parameters in the call list are set for continuing the integration.
c   the user has only to define a new value  tout  and call  ode  again.
c
c   the differential equations are actually solved by a suite of codes
c   de ,  step , and  intrp .  ode  allocates virtual storage in the
```

```
c   arrays  work  and  iwork  and calls  de .  de  is a supervisor which
c   directs the solution.  it calls on the routines  step  and  intrp
c   to advance the integration and to interpolate at output points.
c   step  uses a modified divided difference form of the adams pece
c   formulas and local extrapolation.  it adjusts the order and step
c   size to control the local error per unit step in a generalized
c   sense.  normally each call to  step  advances the solution one step
c   in the direction of  tout .  for reasons of efficiency  de
c   integrates beyond  tout  internally, though never beyond
c   t+10*(tout-t), and calls  intrp  to interpolate the solution at
c   tout .  an option is provided to stop the integration at  tout  but
c   it should be used only if it is impossible to continue the
c   integration beyond  tout .
c
c   this code is completely explained and documented in the text,
c   computer solution of ordinary differential equations:  the initial
c   value problem  by l. f. shampine and m. k. gordon.
c
c   the parameters represent:
c      f -- double precision subroutine f(t,y,yp) to evaluate
c                derivatives yp(i)=dy(i)/dt
c      neqn -- number of equations to be integrated (integer*4)
c      y(*) -- solution vector at t                 (real*8)
c      t -- independent variable                    (real*8)
c      tout -- point at which solution is desired   (real*8)
c      relerr,abserr -- relative and absolute error tolerances for local
c           error test (real*8).  at each step the code requires
c             dabs(local error) .le. dabs(y)*relerr + abserr
c           for each component of the local error and solution vectors
c      iflag -- indicates status of integration     (integer*4)
c      work(*)  (real*8)  -- arrays to hold information internal to
c      iwork(*) (integer*4)    which is necessary for subsequent calls
c
c   first call to ode --
c
c   the user must provide storage in his calling program for the arrays
c   in the call list,
c      y(neqn), work(100+21*neqn), iwork(5),
c   declare  f  in an external statement, supply the double precision
c   subroutine f(t,y,yp)  to evaluate
c      dy(i)/dt = yp(i) = f(t,y(1),y(2),...,y(neqn))
c   and initialize the parameters:
c      neqn -- number of equations to be integrated
c      y(*) -- vector of initial conditions
c      t -- starting point of integration
c      tout -- point at which solution is desired
```

```
c      relerr,abserr -- relative and absolute local error tolerances
c      iflag -- +1,-1.  indicator to initialize the code.  normal input
c           is +1.  the user should set iflag=-1 only if it is
c           impossible to continue the integration beyond  tout .
c   all parameters except  f ,  neqn  and  tout  may be altered by the
c   code on output so must be variables in the calling program.
c
c   output from  ode  --
c
c      neqn -- unchanged
c      y(*) -- solution at  t
c      t -- last point reached in integration.  normal return has
c           t = tout .
c      tout -- unchanged
c      relerr,abserr -- normal return has tolerances unchanged.  iflag=3
c           signals tolerances increased
c      iflag = 2 -- normal return.  integration reached  tout
c            = 3 -- integration did not reach  tout  because error
c                   tolerances too small.  relerr ,  abserr  increased
c                   appropriately for continuing
c            = 4 -- integration did not reach  tout  because more than
c                   500 steps needed
c            = 5 -- integration did not reach  tout  because equations
c                   appear to be stiff
c            = 6 -- invalid input parameters (fatal error)
c           the value of  iflag  is returned negative when the input
c           value is negative and the integration does not reach  tout ,
c           i.e., -3, -4, -5.
c      work(*),iwork(*) -- information generally of no interest to the
c           user but necessary for subsequent calls.
c
c   subsequent calls to  ode --
c
c   subroutine  ode  returns with all information needed to continue
c   the integration.  if the integration reached  tout , the user need
c   only define a new  tout  and call again.  if the integration did not
c   reach  tout  and the user wants to continue, he just calls again.
c   the output value of  iflag  is the appropriate input value for
c   subsequent calls.  the only situation in which it should be altered
c   is to stop the integration internally at the new  tout , i.e.,
c   change output  iflag=2  to input  iflag=-2 .  error tolerances may
c   be changed by the user before continuing.  all other parameters must
c   remain unchanged.
c
c***********************************************************************
c*  subroutines  de  and  step  contain machine dependent constants. *
```

```
c*  be sure they are set before using  ode .                          *
c*********************************************************************
c
      logical start,phase1,nornd
      dimension y(neqn),work(1),iwork(5)
      external f
      data ialpha,ibeta,isig,iv,iw,ig,iphase,ipsi,ix,ih,ihold,istart,
     1  itold,idelsn/1,13,25,38,50,62,75,76,88,89,90,91,92,93/
      iyy = 100
      iwt = iyy + neqn
      ip = iwt + neqn
      iyp = ip + neqn
      iypout = iyp + neqn
      iphi = iypout + neqn
      if(iabs(iflag) .eq. 1) go to 1
      start = work(istart) .gt. 0.0d0
      phase1 = work(iphase) .gt. 0.0d0
      nornd = iwork(2) .ne. -1
    1 call de(f,neqn,y,t,tout,relerr,abserr,iflag,work(iyy),
     1  work(iwt),work(ip),work(iyp),work(iypout),work(iphi),
     2  work(ialpha),work(ibeta),work(isig),work(iv),work(iw),work(ig),
     3  phase1,work(ipsi),work(ix),work(ih),work(ihold),start,
     4  work(itold),work(idelsn),iwork(1),nornd,iwork(3),iwork(4),
     5  iwork(5))
      work(istart) = -1.0d0
      if(start) work(istart) = 1.0d0
      work(iphase) = -1.0d0
      if(phase1) work(iphase) = 1.0d0
      iwork(2) = -1
      if(nornd) iwork(2) = 1
      return
      end
      subroutine de(f,neqn,y,t,tout,relerr,abserr,iflag,
     1  yy,wt,p,yp,ypout,phi,alpha,beta,sig,v,w,g,phase1,psi,x,h,hold,
     2  start,told,delsgn,ns,nornd,k,kold,isnold)
      implicit real*8(a-h,o-z)
c
c   ode  merely allocates storage for  de  to relieve the user of the
c   inconvenience of a long call list.  consequently  de  is used as
c   described in the comments for  ode .
c
c   this code is completely explained and documented in the text,
c   computer solution of ordinary differential equations:  the initial
c   value problem  by l. f. shampine and m. k. gordon.
c
      logical stiff,crash,start,phase1,nornd
```

```
      dimension y(neqn),yy(neqn),wt(neqn),phi(neqn,16),p(neqn),yp(neqn),
     1  ypout(neqn),psi(12),alpha(12),beta(12),sig(13),v(12),w(12),g(13)
      external f
c
c***********************************************************************
c*  the only machine dependent constant is based on the machine unit   *
c*  roundoff error  u  which is the smallest positive number such that *
c*  1.0+u .gt. 1.0 .  u  must be calculated and  fouru=4.0*u  inserted *
c*  in the following data statement before using  de .  the routine    *
c*  machin  calculates  u .  fouru  and  twou=2.0*u  must also be      *
c*  inserted in subroutine  step  before calling  de .                 *
      data fouru/.888d-15/
c***********************************************************************
c
c   the constant  maxnum  is the maximum number of steps allowed in one
c   call to  de .  the user may change this limit by altering the
c   following statement
      data maxnum/500/
c
c            ***            ***            ***
c   test for improper parameters
c
c      fouru = 4.0 * d1mach(4)                                          ***
      if(neqn .lt. 1) go to 10
      if(t .eq. tout) go to 10
      if(relerr .lt. 0.0d0  .or.  abserr .lt. 0.0d0) go to 10
      eps = dmax1(relerr,abserr)
      if(eps .le. 0.0d0) go to 10
      if(iflag .eq. 0) go to 10
      isn = isign(1,iflag)
      iflag = iabs(iflag)
      if(iflag .eq. 1) go to 20
      if(t .ne. told) go to 10
      if(iflag .ge. 2  .and.  iflag .le. 5) go to 20
   10 iflag = 6
      return
c
c   on each call set interval of integration and counter for number of
c   steps.  adjust input error tolerances to define weight vector for
c   subroutine  step
c
   20 del = tout - t
      absdel = dabs(del)
      tend = t + 10.0d0*del
      if(isn .lt. 0) tend = tout
      nostep = 0
```

```
      kle4 = 0
      stiff = .false.
      releps = relerr/eps
      abseps = abserr/eps
      if(iflag .eq. 1) go to 30
      if(isnold .lt. 0) go to 30
      if(delsgn*del .gt. 0.0d0) go to 50
c
c   on start and restart also set work variables x and yy(*), store the
c   direction of integration and initialize the step size
c
   30 start = .true.
      x = t
      do 40 l = 1,neqn
   40   yy(l) = y(l)
      delsgn = dsign(1.0d0,del)
      h = dsign(dmax1(dabs(tout-x),fouru*dabs(x)),tout-x)
c
c   if already past output point, interpolate and return
c
   50 if(dabs(x-t) .lt. absdel) go to 60
      call intrp(x,yy,tout,y,ypout,neqn,kold,phi,psi)
      iflag = 2
      t = tout
      told = t
      isnold = isn
      return
c
c   if cannot go past output point and sufficiently close,
c   extrapolate and return
c
   60 if(isn .gt. 0  .or.  dabs(tout-x) .ge. fouru*dabs(x)) go to 80
      h = tout - x
      call f(x,yy,yp)
      do 70 l = 1,neqn
   70   y(l) = yy(l) + h*yp(l)
      iflag = 2
      t = tout
      told = t
      isnold = isn
      return
c
c   test for too many steps
c
   80 if(nostep .lt. maxnum) go to 100
      iflag = isn*4
```

```
      if(stiff) iflag = isn*5
      do 90 l = 1,neqn
   90   y(l) = yy(l)
      t = x
      told = t
      isnold = 1
      return
c
c   limit step size, set weight vector and take a step
c
  100 h = dsign(dmin1(dabs(h),dabs(tend-x)),h)
      do 110 l = 1,neqn
  110   wt(l) = releps*dabs(yy(l)) + abseps
      call step(x,yy,f,neqn,h,eps,wt,start,
     1  hold,k,kold,crash,phi,p,yp,psi,
     2  alpha,beta,sig,v,w,g,phase1,ns,nornd)
c
c   test for tolerances too small
c
      if(.not.crash) go to 130
      iflag = isn*3
      relerr = eps*releps
      abserr = eps*abseps
      do 120 l = 1,neqn
  120   y(l) = yy(l)
      t = x
      told = t
      isnold = 1
      return
c
c   augment counter on number of steps and test for stiffness
c
  130 nostep = nostep + 1
      kle4 = kle4 + 1
      if(kold .gt. 4) kle4 = 0
      if(kle4 .ge. 50) stiff = .true.
      go to 50
      end
      subroutine step(x,y,f,neqn,h,eps,wt,start,
c    1  hold,k,kold,crash,phi,p,yp,psi)
     1  hold,k,kold,crash,phi,p,yp,psi,
     2  alpha,beta,sig,v,w,g,phase1,ns,nornd)
      implicit real*8(a-h,o-z)
c
c   double precision subroutine  step
c   integrates a system of first order ordinary
```

```
c   differential equations one step, normally from x to x+h, using a
c   modified divided difference form of the adams pece formulas.  local
c   extrapolation is used to improve absolute stability and accuracy.
c   the code adjusts its order and step size to control the local error
c   per unit step in a generalized sense.  special devices are included
c   to control roundoff error and to detect when the user is requesting
c   too much accuracy.
c
c   this code is completely explained and documented in the text,
c   computer solution of ordinary differential equations:  the initial
c   value problem  by l. f. shampine and m. k. gordon.
c
c
c   the parameters represent:
c      x -- independent variable             (real*8)
c      y(*) -- solution vector at x          (real*8)
c      yp(*) -- derivative of solution vector at  x  after successful
c           step                             (real*8)
c      neqn -- number of equations to be integrated (integer*4)
c      h -- appropriate step size for next step.  normally determined by
c           code                             (real*8)
c      eps -- local error tolerance.  must be variable  (real*8)
c      wt(*) -- vector of weights for error criterion   (real*8)
c      start -- logical variable set .true. for first step,  .false.
c           otherwise                        (logical*4)
c      hold -- step size used for last successful step  (real*8)
c      k -- appropriate order for next step (determined by code)
c      kold -- order used for last successful step
c      crash -- logical variable set .true. when no step can be taken,
c           .false. otherwise.
c   the arrays  phi, psi  are required for the interpolation subroutine
c   intrp.  the array p is internal to the code.  all are real*8
c
c   input to  step
c
c      first call --
c
c   the user must provide storage in his driver program for all arrays
c   in the call list, namely
c
c     dimension y(neqn),wt(neqn),phi(neqn,16),p(neqn),yp(neqn),psi(12)
c
c   the user must also declare  start  and  crash  logical variables
c   and  f  an external subroutine, supply the subroutine  f(x,y,yp)
c   to evaluate
c      dy(i)/dx = yp(i) = f(x,y(1),y(2),...,y(neqn))
```

```
c   and initialize only the following parameters:
c      x -- initial value of the independent variable
c      y(*) -- vector of initial values of dependent variables
c      neqn -- number of equations to be integrated
c      h -- nominal step size indicating direction of integration
c           and maximum size of step.  must be variable
c      eps -- local error tolerance per step.  must be variable
c      wt(*) -- vector of non-zero weights for error criterion
c      start -- .true.
c
c   step  requires the l2 norm of the vector with components
c   local error(l)/wt(l)  be less than  eps  for a successful step.  the
c   array  wt  allows the user to specify an error test appropriate
c   for his problem.  for example,
c      wt(l) = 1.0  specifies absolute error,
c            = dabs(y(l))  error relative to the most recent value of
c                 the l-th component of the solution,
c            = dabs(yp(l))  error relative to the most recent value of
c                 the l-th component of the derivative,
c            = dmax1(wt(l),dabs(y(l)))  error relative to the largest
c                 magnitude of l-th component obtained so far,
c            = dabs(y(l))*relerr/eps + abserr/eps  specifies a mixed
c                 relative-absolute test where  relerr  is relative
c                 error,  abserr  is absolute error and  eps =
c                 dmax1(relerr,abserr) .
c
c      subsequent calls --
c
c   subroutine  step  is designed so that all information needed to
c   continue the integration, including the step size  h  and the order
c   k , is returned with each step.  with the exception of the step
c   size, the error tolerance, and the weights, none of the parameters
c   should be altered.  the array  wt  must be updated after each step
c   to maintain relative error tests like those above.  normally the
c   integration is continued just beyond the desired endpoint and the
c   solution interpolated there with subroutine  intrp .  if it is
c   impossible to integrate beyond the endpoint, the step size may be
c   reduced to hit the endpoint since the code will not take a step
c   larger than the  h  input.  changing the direction of integration,
c   i.e., the sign of  h , requires the user set  start = .true. before
c   calling  step  again.  this is the only situation in which  start
c   should be altered.
c
c   output from  step
c
c      successful step --
```

```
c
c   the subroutine returns after each successful step with  start  and
c   crash  set .false. .  x  represents the independent variable
c   advanced one step of length  hold  from its value on input and  y
c   the solution vector at the new value of  x .  all other parameters
c   represent information corresponding to the new  x  needed to
c   continue the integration.
c
c      unsuccessful step --
c
c   when the error tolerance is too small for the machine precision,
c   the subroutine returns without taking a step and  crash = .true. .
c   an appropriate step size and error tolerance for continuing are
c   estimated and all other information is restored as upon input
c   before returning.  to continue with the larger tolerance, the user
c   just calls the code again.  a restart is neither required nor
c   desirable.
c
      logical start,crash,phase1,nornd
      dimension y(neqn),wt(neqn),phi(neqn,16),p(neqn),yp(neqn),psi(12)
      dimension alpha(12),beta(12),sig(13),w(12),v(12),g(13),
     1  gstr(13),two(13)
      external f
c*********************************************************************
c*  the only machine dependent constants are based on the machine unit *
c*  roundoff error  u  which is the smallest positive number such that *
c*  1.0+u .gt. 1.0  .  the user must calculate  u  and insert         *
c*  twou=2.0*u  and  fouru=4.0*u  in the data statement before calling *
c*  the code.  the routine  machin  calculates  u .                    *
      data twou,fouru/.444d-15,.888d-15/
c*********************************************************************
      data two/2.0d0,4.0d0,8.0d0,16.0d0,32.0d0,64.0d0,128.0d0,256.0d0,
     1  512.0d0,1024.0d0,2048.0d0,4096.0d0,8192.0d0/
      data gstr/0.500d0,0.0833d0,0.0417d0,0.0264d0,0.0188d0,0.0143d0,
     1  0.0114d0,0.00936d0,0.00789d0,0.00679d0,0.00592d0,0.00524d0,
     2  0.00468d0/
c     data g(1),g(2)/1.0,0.5/,sig(1)/1.0/
c
c
c      twou = 2.0 * d1mach(4)                                           ***
c      fouru = 2.0 * twou                                               ***
c       ***     begin block 0     ***
c   check if step size or error tolerance is too small for machine
c   precision.  if first step, initialize phi array and estimate a
c   starting step size.
c                    ***
```

```
c
c   if step size is too small, determine an acceptable one
c
      crash = .true.
      if(dabs(h) .ge. fouru*dabs(x)) go to 5
      h = dsign(fouru*dabs(x),h)
      return
    5 p5eps = 0.5d0*eps
c
c   if error tolerance is too small, increase it to an acceptable value
c
      round = 0.0d0
      do 10 l = 1,neqn
   10   round = round + (y(l)/wt(l))**2
      round = twou*dsqrt(round)
      if(p5eps .ge. round) go to 15
      eps = 2.0*round*(1.0d0 + fouru)
      return
   15 crash = .false.
      g(1)=1.0d0
      g(2)=0.5d0
      sig(1)=1.0d0
      if(.not.start) go to 99
c
c   initialize.  compute appropriate step size for first step
c
      call f(x,y,yp)
      sum = 0.0d0
      do 20 l = 1,neqn
        phi(l,1) = yp(l)
        phi(l,2) = 0.0d0
   20   sum = sum + (yp(l)/wt(l))**2
      sum = dsqrt(sum)
      absh = dabs(h)
      if(eps .lt. 16.0d0*sum*h*h) absh = 0.25d0*dsqrt(eps/sum)
      h = dsign(dmax1(absh,fouru*dabs(x)),h)
      hold = 0.0d0
      k = 1
      kold = 0
      start = .false.
      phase1 = .true.
      nornd = .true.
      if(p5eps .gt. 100.0d0*round) go to 99
      nornd = .false.
      do 25 l = 1,neqn
   25   phi(l,15) = 0.0d0
```

```
   99 ifail = 0
c       ***     end block 0     ***
c
c       ***     begin block 1     ***
c   compute coefficients of formulas for this step.  avoid computing
c   those quantities not changed when step size is not changed.
c                   ***
c
  100 kp1 = k+1
      kp2 = k+2
      km1 = k-1
      km2 = k-2
c
c   ns is the number of steps taken with size h, including the current
c   one.  when k.lt.ns, no coefficients change
c
      if(h .ne. hold) ns = 0
      if(ns.le.kold)   ns=ns+1
      nsp1 = ns+1
      if (k .lt. ns) go to 199
c
c   compute those components of alpha(*),beta(*),psi(*),sig(*) which
c   are changed
c
      beta(ns) = 1.0d0
      realns = ns
      alpha(ns) = 1.0d0/realns
      temp1 = h*realns
      sig(nsp1) = 1.0d0
      if(k .lt. nsp1) go to 110
      do 105 i = nsp1,k
        im1 = i-1
        temp2 = psi(im1)
        psi(im1) = temp1
        beta(i) = beta(im1)*psi(im1)/temp2
        temp1 = temp2 + h
        alpha(i) = h/temp1
        reali = i
  105   sig(i+1) = reali*alpha(i)*sig(i)
  110 psi(k) = temp1
c
c   compute coefficients g(*)
c
c   initialize v(*) and set w(*).  g(2) is set in data statement
c
      if(ns .gt. 1) go to 120
```

```
      do 115 iq = 1,k
        temp3 = iq*(iq+1)
        v(iq) = 1.0d0/temp3
  115   w(iq) = v(iq)
      go to 140
c
c   if order was raised, update diagonal part of v(*)
c
  120 if(k .le. kold) go to 130
      temp4 = k*kp1
      v(k) = 1.0d0/temp4
      nsm2 = ns-2
      if(nsm2 .lt. 1) go to 130
      do 125 j = 1,nsm2
        i = k-j
  125   v(i) = v(i) - alpha(j+1)*v(i+1)
c
c   update v(*) and set w(*)
c
  130 limit1 = kp1 - ns
      temp5 = alpha(ns)
      do 135 iq = 1,limit1
        v(iq) = v(iq) - temp5*v(iq+1)
  135   w(iq) = v(iq)
      g(nsp1) = w(1)
c
c   compute the g(*) in the work vector w(*)
c
  140 nsp2 = ns + 2
      if(kp1 .lt. nsp2) go to 199
      do 150 i = nsp2,kp1
        limit2 = kp2 - i
        temp6 = alpha(i-1)
        do 145 iq = 1,limit2
  145     w(iq) = w(iq) - temp6*w(iq+1)
  150   g(i) = w(1)
  199   continue
c       ***     end block 1     ***
c
c       ***     begin block 2     ***
c   predict a solution p(*), evaluate derivatives using predicted
c   solution, estimate local error at order k and errors at orders k,
c   k-1, k-2 as if constant step size were used.
c                   ***
c
c   change phi to phi star
```

```
c
      if(k .lt. nsp1) go to 215
      do 210 i = nsp1,k
        temp1 = beta(i)
        do 205 l = 1,neqn
  205     phi(l,i) = temp1*phi(l,i)
  210   continue
c
c   predict solution and differences
c
  215 do 220 l = 1,neqn
        phi(l,kp2) = phi(l,kp1)
        phi(l,kp1) = 0.0d0
  220   p(l) = 0.0d0
      do 230 j = 1,k
        i = kp1 - j
        ip1 = i+1
        temp2 = g(i)
        do 225 l = 1,neqn
          p(l) = p(l) + temp2*phi(l,i)
  225     phi(l,i) = phi(l,i) + phi(l,ip1)
  230   continue
      if(nornd) go to 240
      do 235 l = 1,neqn
        tau = h*p(l) - phi(l,15)
        p(l) = y(l) + tau
  235   phi(l,16) = (p(l) - y(l)) - tau
      go to 250
  240 do 245 l = 1,neqn
  245   p(l) = y(l) + h*p(l)
  250 xold = x
      x = x + h
      absh = dabs(h)
      call f(x,p,yp)
c
c   estimate errors at orders k,k-1,k-2
c
      erkm2 = 0.0d0
      erkm1 = 0.0d0
      erk = 0.0d0
      do 265 l = 1,neqn
        temp3 = 1.0d0/wt(l)
        temp4 = yp(l) - phi(l,1)
        if(km2)265,260,255
  255   erkm2 = erkm2 + ((phi(l,km1)+temp4)*temp3)**2
  260   erkm1 = erkm1 + ((phi(l,k)+temp4)*temp3)**2
```

```
  265   erk = erk + (temp4*temp3)**2
      if(km2)280,275,270
  270 erkm2 = absh*sig(km1)*gstr(km2)*dsqrt(erkm2)
  275 erkm1 = absh*sig(k)*gstr(km1)*dsqrt(erkm1)
  280 temp5 = absh*dsqrt(erk)
      err = temp5*(g(k)-g(kp1))
      erk = temp5*sig(kp1)*gstr(k)
      knew = k
c
c   test if order should be lowered
c
      if(km2)299,290,285
  285 if(dmax1(erkm1,erkm2) .le. erk) knew = km1
      go to 299
  290 if(erkm1 .le. 0.5d0*erk) knew = km1
c
c   test if step successful
c
  299 if(err .le. eps) go to 400
c       ***     end block 2     ***
c
c       ***     begin block 3     ***
c   the step is unsuccessful.  restore  x, phi(*,*), psi(*) .
c   if third consecutive failure, set order to one.  if step fails more
c   than three times, consider an optimal step size.  double error
c   tolerance and return if estimated step size is too small for machine
c   precision.
c                   ***
c
c   restore x, phi(*,*) and psi(*)
c
      phase1 = .false.
      x = xold
      do 310 i = 1,k
        temp1 = 1.0d0/beta(i)
        ip1 = i+1
        do 305 l = 1,neqn
  305     phi(l,i) = temp1*(phi(l,i) - phi(l,ip1))
  310   continue
      if(k .lt. 2) go to 320
      do 315 i = 2,k
  315   psi(i-1) = psi(i) - h
c
c   on third failure, set order to one.  thereafter, use optimal step
c   size
c
```

```
  320 ifail = ifail + 1
      temp2 = 0.5d0
      if(ifail - 3) 335,330,325
  325 if(p5eps .lt. 0.25d0*erk) temp2 = dsqrt(p5eps/erk)
  330 knew = 1
  335 h = temp2*h
      k = knew
      if(dabs(h) .ge. fouru*dabs(x)) go to 340
      crash = .true.
      h = dsign(fouru*dabs(x),h)
      eps = eps + eps
      return
  340 go to 100
c       ***     end block 3     ***
c
c       ***     begin block 4     ***
c   the step is successful.  correct the predicted solution, evaluate
c   the derivatives using the corrected solution and update the
c   differences.  determine best order and step size for next step.
c                   ***
  400 kold = k
      hold = h
c
c   correct and evaluate
c
      temp1 = h*g(kp1)
      if(nornd) go to 410
      do 405 l = 1,neqn
        rho = temp1*(yp(l) - phi(l,1)) - phi(l,16)
        y(l) = p(l) + rho
  405   phi(l,15) = (y(l) - p(l)) - rho
      go to 420
  410 do 415 l = 1,neqn
  415   y(l) = p(l) + temp1*(yp(l) - phi(l,1))
  420 call f(x,y,yp)
c
c   update differences for next step
c
      do 425 l = 1,neqn
        phi(l,kp1) = yp(l) - phi(l,1)
  425   phi(l,kp2) = phi(l,kp1) - phi(l,kp2)
      do 435 i = 1,k
        do 430 l = 1,neqn
  430     phi(l,i) = phi(l,i) + phi(l,kp1)
  435   continue
c
```

```
c   estimate error at order k+1 unless:
c     in first phase when always raise order,
c     already decided to lower order,
c     step size not constant so estimate unreliable
c
      erkp1 = 0.0d0
      if(knew .eq. km1  .or.  k .eq. 12) phase1 = .false.
      if(phase1) go to 450
      if(knew .eq. km1) go to 455
      if(kp1 .gt. ns) go to 460
      do 440 l = 1,neqn
  440   erkp1 = erkp1 + (phi(l,kp2)/wt(l))**2
      erkp1 = absh*gstr(kp1)*dsqrt(erkp1)
c
c   using estimated error at order k+1, determine appropriate order
c   for next step
c
      if(k .gt. 1) go to 445
      if(erkp1 .ge. 0.5d0*erk) go to 460
      go to 450
  445 if(erkm1 .le. dmin1(erk,erkp1)) go to 455
      if(erkp1 .ge. erk  .or.  k .eq. 12) go to 460
c
c   here erkp1 .lt. erk .lt. dmax1(erkm1,erkm2) else order would have
c   been lowered in block 2.  thus order is to be raised
c
c   raise order
c
  450 k = kp1
      erk = erkp1
      go to 460
c
c   lower order
c
  455 k = km1
      erk = erkm1
c
c   with new order determine appropriate step size for next step
c
  460 hnew = h + h
      if(phase1) go to 465
      if(p5eps .ge. erk*two(k+1)) go to 465
      hnew = h
      if(p5eps .ge. erk) go to 465
      temp2 = k+1
      r = (p5eps/erk)**(1.0d0/temp2)
```

```
      hnew = absh*dmax1(0.5d0,dmin1(0.9d0,r))
      hnew = dsign(dmax1(hnew,fouru*dabs(x)),h)
  465 h = hnew
      return
c        ***     end block 4     ***
      end
      subroutine intrp(x,y,xout,yout,ypout,neqn,kold,phi,psi)
      implicit real*8(a-h,o-z)
c
c   the methods in subroutine  step  approximate the solution near  x
c   by a polynomial.  subroutine  intrp  approximates the solution at
c   xout  by evaluating the polynomial there.  information defining this
c   polynomial is passed from  step  so  intrp  cannot be used alone.
c
c   this code is completely explained and documented in the text,
c   computer solution of ordinary differential equations:  the initial
c   value problem  by l. f. shampine and m. k. gordon.
c
c   input to intrp --
c
c   all floating point variables are double precision
c   the user provides storage in the calling program for the arrays in
c   the call list
       dimension y(neqn),yout(neqn),ypout(neqn),phi(neqn,16),psi(12)
c   and defines
c      xout -- point at which solution is desired.
c   the remaining parameters are defined in  step  and passed to  intrp
c   from that subroutine
c
c   output from  intrp --
c
c      yout(*) -- solution at  xout
c      ypout(*) -- derivative of solution at  xout
c   the remaining parameters are returned unaltered from their input
c   values.  integration with  step  may be continued.
c
      dimension g(13),w(13),rho(13)
      data g(1)/1.0d0/,rho(1)/1.0d0/
c
      hi = xout - x
      ki = kold + 1
      kip1 = ki + 1
c
c   initialize w(*) for computing g(*)
c
      do 5 i = 1,ki
```

```
          temp1 = i
    5     w(i) = 1.0d0/temp1
        term = 0.0d0
c
c   compute g(*)
c
        do 15 j = 2,ki
          jm1 = j - 1
          psijm1 = psi(jm1)
          gamma = (hi + term)/psijm1
          eta = hi/psijm1
          limit1 = kip1 - j
          do 10 i = 1,limit1
   10       w(i) = gamma*w(i) - eta*w(i+1)
          g(j) = w(1)
          rho(j) = gamma*rho(jm1)
   15     term = psijm1
c
c   interpolate
c
        do 20 l = 1,neqn
          ypout(l) = 0.0d0
   20     yout(l) = 0.0d0
        do 30 j = 1,ki
          i = kip1 - j
          temp2 = g(i)
          temp3 = rho(i)
          do 25 l = 1,neqn
            yout(l) = yout(l) + temp2*phi(l,i)
   25       ypout(l) = ypout(l) + temp3*phi(l,i)
   30     continue
        do 35 l = 1,neqn
   35     yout(l) = y(l) + hi*yout(l)
        return
        end
```

4 The calling routine

Now comes the part of the program that is entirely up to you. I describe here a program I wrote to draw a Kepler orbit.

```
      PROGRAM kepler
c-----------------------------------------------------------------
c  Calcula uma orbita de Kepler usando shampine-gordon's ode.f
c-----------------------------------------------------------------
      INTEGER nneqn, i
```

```
      PARAMETER(nneqn=6)
      DOUBLE PRECISION yy(nneqn),tt,ttout,rrelerr,aabserr
      REAL*8 wwork(100+21*nneqn)
      INTEGER*4 iiflag,iiwork(5)
      EXTERNAL f
      OPEN (unit=12, file='kepler.dat')
      iiflag=1
      rrelerr=1.0d-8
      aabserr=1.0d-15
      tt=0.D0
      ttout=tt+0.1D0
      yy(1)=1.D0
      yy(2)=0.D0
      yy(3)=0.D0
      yy(4)=1.D0
      yy(5)=0.D0
      yy(6)=0.D0

      do i=1,1000
        call ode(derivs, nneqn, yy, tt, ttout, rrelerr, aabserr, iiflag,
     &    wwork,iiwork)

ttout=tt+0.1D-1
        write(12,*) yy(3),yy(1)
      end do

      CLOSE(unit=12)
      stop
      END
```

Together with the comments in the subroutine *ode.f*, it should be self-explanatory. I use the same names as the arguments of *ode()*, only doubling the first letter. The exception is the variable f, which I call *derivs*. For example,the argument y in the definition of *ode* is called, in the calling routine, *yy*. *relerr* becomes *rrelerr*, and so on.

5 What to do?

1. Using copy and paste, create the files *derivs.f*, *ode.f* and, in this case, *kepler.f*, in the same directory.
2.Execute the command: *g77 kepler.f ode.f derivs.f -o kepler.out*
3.Execute the command: *kepler.out*
4.Execute the command: *gnuplot*
5.In gnuplot, execute: *plot "kepler.dat"*

That's it. A beautiful ellipse should be exhibited.
If you need to freshen up your Fortran, there is a good link in my site.