# PHYS 410/555: Computational Physics
# Homework 5 Key

## Problem 1:

The canonical way of checking an $n$-th-order interpolation scheme is use as test input $(x_i, f_i)$ (i.e. the set of values in which we are to interpolate), values sampled from a degree $n-1$ polynomial. That is, we use as input

$$(x_i, p_{n-1}(x_i))$$

where $p_{n-1}(x)$ is some conveniently chosen polynomial of degree $n-1$:

$$p_{n-1}(x) \equiv \sum_{i=0}^{n-1} c_i\, x^i$$

Then, for arbitrary `xto` (modulo floating-point problems with overflow etc.), and smallish $n$, we should find that

**`dpint(xto,x,f,n)`**

returns "exactly" $p($`xto`$)$—i.e. $p($`xto`$)$ with an error of the order of machine epsilon (the error *will* generally increase significantly with increasing $n$.)
I used this technique as the basic way of checking your implementation of `dpint` and `tdpint`). Specifically, I checked that your `tdpint` returned the correct values for interpolation in the 5-th degree polynomial

$$(x+1)^5$$

evaluated at the points

$$x_i = 0, 1, 2, 3, 4, 5.$$

*Sample source code*—**`dpint.f`**:

```
c===========================================================
c     dpint:  Computes p(xto) where p(x) is the degree n-1
c     polynomial passing through (x(i),f(i)) , i = 1 , n.
c     Uses Neville's algorithm as discussed in class.
c     Return code 'rc' is set as follows:
c
c         rc = 0   ->   Normal interpolation
c         rc = 1   ->   Normal extrapolation
c         rc = 2   ->   Requested degree (n) too large
c         rc = 3   ->   Non-distinct x(i)
c===========================================================
      real*8 function dpint(xto,x,f,n,rc)

         implicit       none

         real*8         dvmin,   dvmax

         integer        n,       rc
         real*8         xto,     x(n),    f(n)

c-----------------------------------------------------------
c        Storage for constructing 'tableau'.
c-----------------------------------------------------------
         integer        maxn
         parameter    ( maxn = 20 )
         real*8         p(maxn)

c-----------------------------------------------------------
c        Locals.
c-----------------------------------------------------------
         real*8         den
         integer        i,       j,       k,       m

c-----------------------------------------------------------
c        Initialize return value arbitrarily, calling
c        routine must check 'rc' to see whether an error
c        has occured.
c-----------------------------------------------------------
         dpint = 0.0d0

c-----------------------------------------------------------
c        Check input.
c-----------------------------------------------------------
         if( n .gt. maxn ) then
            write(0,*) 'dpint: Requested polynomial degree ',
     &         n - 1, ' exceeds implementation maximum ',
     &         maxn - 1
            rc = 2
            return
         end if

c-----------------------------------------------------------
c        Is this an interpolation or extrapolation?
c        Functions 'dvmin' and 'dvmax', defined below,
c        return minimum and maximum, respectively, of a
c        real*8 vector.
c-----------------------------------------------------------
         if( dvmin(x,n) .le. xto   .and.
     &       xto .le. dvmax(x,n) ) then
            rc = 0
         else
            rc = 1
         end if

c-----------------------------------------------------------
c        Construct the interpolated value via Neville's
c        algorithm.
c-----------------------------------------------------------
         do i = 1 , n
            p(i) = f(i)
         end do

         do m = 1 , n - 1
            do i = 1 , n - m
               den = x(i) - x(i+m)
               if( den .eq. 0.0d0 ) then
                  write(0,*) 'dpint: x(i) are not all distinct'
                  rc = 3
                  return
               end if
               p(i) = ( (xto - x(i+m)) * p(i)  +
     &                  (x(i) - xto)   * p(i+1) ) /
     &                den
            end do
         end do
c-----------------------------------------------------------
c        Return the interpolated value.
c-----------------------------------------------------------
         dpint = p(1)

         return

      end

c===========================================================
c     dvmin: Returns minimum of real*8 vector.
c===========================================================
      real*8 function dvmin(v,n)

         implicit       none

         integer        n
         real*8         v(n)

         integer        i

         if( n .gt. 0 ) then
            dvmin = v(1)
            do i = 2 , n
               dvmin = min(dvmin,v(i))
            end do
         else
```

```
            dvmin = 0.0d0
         end if

         return

      end

c===========================================================
c     dvmax: Returns maximum of real*8 vector.
c===========================================================
      real*8 function dvmax(v,n)

         implicit       none

         integer        n
         real*8         v(n)

         integer        i

         if( n .gt. 0 ) then
            dvmax = v(1)
            do i = 2 , n
               dvmax = max(dvmax,v(i))
            end do
         else
            dvmax = 0.0d0
         end if

         return

      end
```

*Sample source code—***tdpint.f** *See Homework 3 key for source to* **ddvfrom.f** *and* **dvvto.f**.

```
c===========================================================
c     Tests polynomial interpolation routine 'dpint'.
c     Reads (x,f) pairs from standard input and gets
c     x-values (xto) to be interpolated to from command line.
c     Outputs pairs (xto,p(xto)) (where p(x) is the interp-
c     olating polynomial passing through the (x,f) pairs)
c     to standard output.
c===========================================================
      program        tdpint

      implicit       none
c-----------------------------------------------------------
c     Declaration of functions, including interpolator
c     'dpint'.
c-----------------------------------------------------------
      integer        iargc
      real*8         r8arg,        dpint

      real*8         r8_never
      parameter     ( r8_never = -1.0d-60 )
c-----------------------------------------------------------
c     Maximum number of (x,f) pairs (1 + max degree of
c     interpolating polynomial), actual number of pairs,
c     and storage for pairs.
c-----------------------------------------------------------
      integer        maxn
      parameter     ( maxn = 20 )
      integer        n
      real*8         x(maxn),      f(maxn)
c-----------------------------------------------------------
c     Maximum number of (xto,p(xto)) pairs (# of command
c     line arguments), actual number of pairs, and storage
c     for pairs.
c-----------------------------------------------------------
      integer        maxnto
      parameter     ( maxnto = 10 )
      integer        nto
      real*8         xto(maxnto),  fto(maxnto)

      integer        ito,          rc

c-----------------------------------------------------------
c     Argument parsing: extract 'xto' values.
c-----------------------------------------------------------
      nto = min(iargc(),maxnto)
      if( nto .lt. 1 ) go to 900
      do ito = 1 , nto
         xto(ito) = r8arg(ito,r8_never)
         if( xto(ito) .eq. r8_never ) go to 900
      end do

c-----------------------------------------------------------
c     Read input pairs using 'dvvfrom' from homework 3.
c-----------------------------------------------------------
      call dvvfrom('-',x,f,n,maxn)
      do ito = 1 , nto
c-----------------------------------------------------------
c        Compute the interpolated values.
c-----------------------------------------------------------
         fto(ito) = dpint(xto(ito),x,f,n,rc)
         if( rc .gt. 1 ) then
            write(0,*) 'tdpint: Invalid input.'
            stop
         end if
      end do
c-----------------------------------------------------------
c     Write output pairs using 'dvvto' from homework 3.
c-----------------------------------------------------------
      call dvvto('-',xto,fto,nto)

      stop

 900  continue
         write(0,*) 'tdpint: <xto> [<xto> ...]'
      stop

      end
```

**Problem 2:**

The equation of motion to be solved via finite-difference techniques is

$$\ddot{q}(t) = -\omega^2 q \qquad 0 \le t \le t_{\max} \qquad q(0) = q_0 \quad \dot{q}(0) = \dot{q}_0$$

where $q_0$ and $\dot{q}_0$ are the specified initial conditions. Discretizing the time domain via

$$t \to t^n \equiv n \triangle t, \quad n = 0, 1, \ldots, \texttt{nt} \quad \triangle t \equiv \frac{t_{\max}}{\texttt{nt} - 1}$$

and using the standard centred second-order ($O(\triangle t^2)$) finite difference approximation to a second derivative, we have:

$$\frac{q^{n+1} - 2q^n + q^{n-1}}{\triangle t^2} = -\omega^2 q^n, \quad n = 1, 2, \ldots, \texttt{nt} - 1$$

This equation can be solved explicitly for $q^{n+1}$:

$$q^{n+1} = \left(2 - \triangle t^2 \omega^2\right) q^n - q^{n-1} = c_0 q^n + c_1 q^{n-1}$$

where

$$c_0 \equiv 2 - \triangle t^2 \omega^2 \qquad c_1 \equiv -1$$

To initialize the difference scheme we need values for $q^0$ and $q^1$ which are accurate to at least $O(\triangle t^2)$ (i.e. we need $q^1 = q(\triangle t) + O(\triangle t^3)$). Specifically:

$$q^0 = q_0$$

$$q^1 = q(0) + \triangle t \dot{q}(0) + \frac{1}{2} \triangle t^2 \ddot{q}(0) = q_0 + \triangle t \dot{q}_0 - \frac{1}{2} \triangle t^2 \omega^2 q_0$$

In analogy with our discussion of the stability of difference schemes for time-dependent *partial* differential equations (such as the wave equation), we can perform a stability analysis of this scheme. To do so, we make the *ansatz*

$$q^n = \mu^n$$

where $\mu$ will, in general, be a complex-valued quantity. We then demand that all solutions satisfy $|\mu| \le 1$, since if $|\mu| > 1$, the difference solution clearly will "blow up". Substituting the *ansatz* in the difference equation, we find the characteristic equation

$$\mu^2 - 2\sigma\mu + 1 = 0$$

where $\sigma$, defined by

$$\sigma \equiv 1 - \frac{1}{2} \left(\omega \triangle t\right)^2$$

is a real quantity satisfying $\sigma \le 1$.
The characteristic equation has two roots

$$\mu = \frac{2\sigma \pm \sqrt{\left((2\sigma)^2 - 4\right)}}{2} = \sigma \pm \sqrt{\sigma^2 - 1}$$

There are now two separate cases to consider:

1. $-1 \le \sigma \le 1$: In this case, $\sqrt{\sigma^2 - 1}$ is purely imaginary and $|\sigma^2 - 1|^2 = 1 - \sigma^2$. Thus $|\mu|^2 = \Re(\mu)^2 + \Im(\mu)^2 = \sigma^2 + 1 - \sigma^2 = 1$—so in this case, *$\mu$ lies on the unit circle*, and presumably the scheme will be stable.
2. $\sigma < -1$: In this case $\sqrt{\sigma^2 - 1}$ is purely real, and $|\mu| = |\sigma - \sqrt{\sigma^2 - 1}| > 1$, which indicates that the scheme will be unstable.

Thus we expect the scheme to become unstable when $\sigma < -1$ which, from the definition of $\sigma$, occurs when $\frac{1}{2} \left(\omega \triangle t\right)^2 > 2$. We therefore have the following restriction on the time step:

$$\triangle t \le \frac{2}{\omega}$$

With the invocation

```
sho 1.0 0.0 1.0 513 8 8
```

the above stability criterion is violated, and the solution *does* "blow up". You were not expected (necessarily) to derive the above stability condition, but you should have suspected that the scheme was unstable from the observed behaviour.

*Sample source code—*`sho.f`*:*

```
c===========================================================
c     sho: Solves the simple harmonic oscillator equation
c
c        dq^2/dt^2 = -omwga^2 q
c
c     using O(h^2) finite-difference methods.
c===========================================================
      program         sho

      implicit        none

      real*8          r8_never
      parameter     ( r8_never = -1.0d-60 )

      real*8          r8arg
      integer         iargc,      i4arg

c-----------------------------------------------------------
c     Command-line arguments
c-----------------------------------------------------------
      real*8          q0,         qdot0,      omsq,
     &                tmax
      integer         level,      olevel

c-----------------------------------------------------------
c     Local variables:
c
c        q(2)          Maintains difference approximation of
c                      oscillator's position (need two time
c                      levels for explicit 3-level scheme)
c        c(-1:0)       Coefficients used in implementation
c                      of difference scheme.
c        dt            Time step (finite-difference scale, h).
c        t             Maintains integration time.
c        nt, it        Number of time steps, current time step.
c        nm1, n, np1   Set so that q(nm1), q(n), q(np1) refer
c                      to previous, current and next osc-
c                      illator positions, respectively.
c        ofreq         Output frequency
c-----------------------------------------------------------
      real*8          q(2),       c(-1:0)
      real*8          dt,         t
      integer         nt,         n,          np1,        nm1,
     &                it,         ofreq
```

```
c-------------------------------------------------------------
c     Argument parsing.
c-------------------------------------------------------------
      if( iargc() .lt. 1 ) go to 900
      q0    = r8arg(1,r8_never)
      if( q0 .eq. r8_never ) go to 900
      qdot0 = r8arg(2,0.0d0)
      omsq  = r8arg(3,1.0d0)
      if( omsq .lt. 0.0d0 ) omsq = 1.0d0
      tmax  = r8arg(4,8.0d0)
      if( tmax .lt. 0.0d0 ) tmax = 8.0d0
      level  = i4arg(5,8)
      if( level .lt. 2 ) level = 8
      olevel = i4arg(6,8)
      if( olevel .gt. level .or. olevel .lt. 1 ) olevel = level

c-------------------------------------------------------------
c     Compute output frequency and set up finite difference
c     parameters and coefficients.
c-------------------------------------------------------------
      ofreq = 2 ** (level - olevel)

      nt  = 2**level + 1
      dt  = tmax / (nt - 1)

      c(0)  = 2.0d0 - omsq * dt * dt
      c(-1) = -1.0d0

c-------------------------------------------------------------
c     Initialize difference scheme (q^0, q^1) to O(dt^2)
c     and output initial time and oscillator position.
c-------------------------------------------------------------
      n   = 1
      np1 = 2
      nm1 = 2
      q(nm1) = q0
      q(n)   = 0.5d0 * c(0) * q0 + dt * qdot0
      write(*,*) 0.0d0, q(nm1)
      t = dt
      if( ofreq .eq. 1 ) then
         write(*,*) t,    q(n)
      end if

c-------------------------------------------------------------
c     Main time-step loop: Update oscillator position
c     using difference equation, update time, and output
c     time and position every 'ofreq' steps.
c-------------------------------------------------------------
      do it = 2 , nt - 1
         q(np1) = c(0) * q(n) + c(-1) * q(nm1)
         t = t + dt
         if( mod(it,ofreq) .eq. 0 ) then
            write(*,*) t,   q(np1)
         end if
         np1 = n
         n   = nm1
         nm1 = np1
      end do

      stop

c-------------------------------------------------------------
c     Usage exit.
c-------------------------------------------------------------
 900  continue
         write(0,*) 'usage: sho <q0> [<qdot0> <omsq> '//
     &              '<tmax> <level> <olevel>]'
         write(0,*)
         write(0,*) ' defaults          0.0    1.0   '//
     &              ' 8.0      8       8'
      stop

      end
```

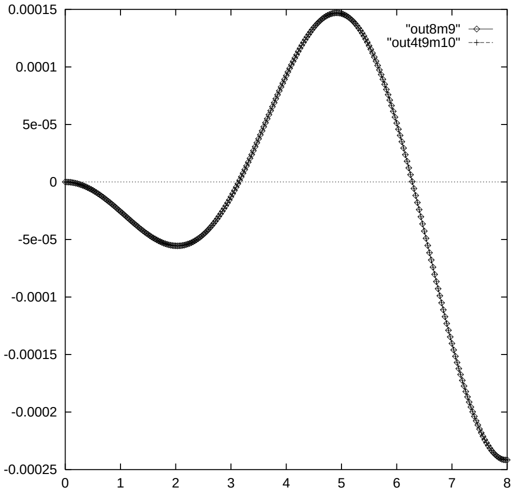
*Sample commands for preparing data for convergence test. Refer to class notes for information concerning* **nf** *and* **paste** *commands.*

```
einstein% sho 1.0 0.0 1.0 8.0  8 8 > out8
einstein% sho 1.0 0.0 1.0 8.0  9 8 > out9
einstein% sho 1.0 0.0 1.0 8.0 10 8 > out10

einstein% paste out8 out9  | nf _1 '(_2 - _4)'     > out8m9
einstein% paste out9 out10 | nf _1 '4 * (_2 - _4)' > out4t9m10
```

*Gnuplot commands for making convergence plot from data generated above.*

```
gnuplot> set terminal postscript portrait
gnuplot> set size 0.760,1.0
gnuplot> set output "ctest.ps"
gnuplot> plot "out8m9" with linespoints, \
              "out4t9m10" with linespoints
gnuplot> quit
```

*Results of convergence test on levels 8, 9, 10. The convergence test shows* $q_8(t) - q_9(t)$ *and* $4(q_9(t) - q_{10}(t))$ *graphed on the same plot. The close coincidence of the two curves provides strong evidence for second-order convergence of the solution.*



**Problem 3:**

The equation of motion to be solved using finite difference methods is

$$u_{tt} = u_{xx} \qquad 0 \le x \le 1 \qquad 0 \le t \le t_{\max}$$

subject to the initial and boundary conditions

$$u(x,0) = l(x) + r(x) \qquad u_t(x,0) = l'(x) - r'(x)$$

$$u(0,t) = u(1,t) = 0$$

where $l(x)$ and $r(x)$ are the initially left-moving and right-moving, respectively, parts of the solution. We discretize the problem domain as follows:

$$x \to x_j \equiv (j-1)\Delta x, \;\; j = 1, \ldots, \mathtt{nx} \quad \Delta x = \frac{1}{\mathtt{nx} - 1}$$

$$t \to t^n \equiv n\Delta t = n\lambda\Delta x, \;\; n = 0, 1, \ldots, \mathtt{nt} \quad \Delta t = \lambda \Delta x$$

$$u(x,t,) \to u\left(x_j, t^n\right) \equiv u_j^n$$

We can then construct an $O(h^2)$ ($h = \Delta x = \Delta t/\lambda$) approximation to the equation of motion as discussed in class:

$$\frac{u_j^{n+1} - 2u_j^n + u_j^{n-1}}{\Delta t^2} = \frac{u_{j+1}^n - 2u_j^n + u_{j-1}^n}{\Delta x^2}$$

Solving for $u_j^{n+1}$, we have

$$u_j^{n+1} = 2u_j^n - u_j^{n-1} + \lambda^2\left(u_{j+1}^n - 2u_j^n + u_{j-1}^n\right)$$

$$j = 2, \ldots, \mathtt{nx} - 1, \;\; n = 2, \ldots, \mathtt{nt} - \mathtt{1}$$

The boundary conditions are

$$u_1^{n+1} = u_{\mathtt{nx}}^{n+1} = 0$$

We also need initial values, $u_j^0$ and $u_j^1$. These are given (upto and including terms of $O(h^2)$ in the case of $u_j^1$) by

$$u_j^0 = l_j + u_j$$

$$u_j^1 = l_j + u_j + \Delta t\left(l_j' - u_j'\right) + \frac{1}{2}\Delta t^2\left(l_j'' + u_j''\right)$$

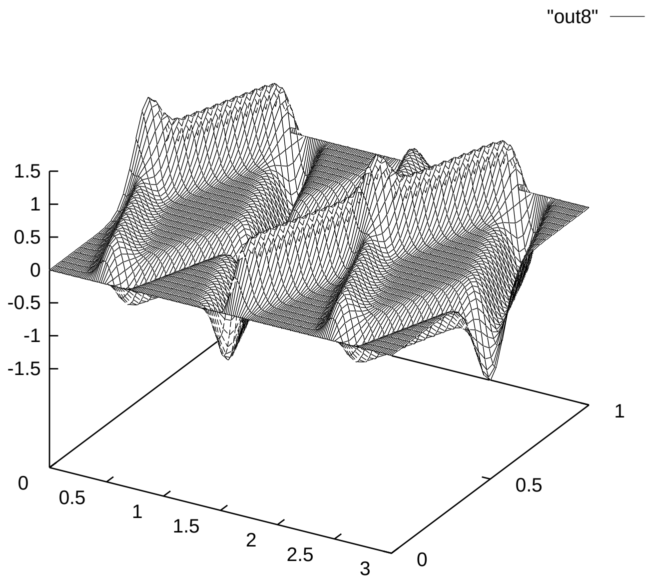
As discussed in "Notes on the 1-D Wave Equation" distributed in class, a *Von Neumann* stability analysis of the discrete equations of motion predicts that the above scheme will be unstable if $\lambda > 1$. This is an instance of the famous CFL (Courant-Friedrichs-Levy (1928)) condition on explicit difference-schemes for hyperbolic equations. The CFL condition is often paraphrased as "the numerical domain of dependence must contain the physical domain of dependence".

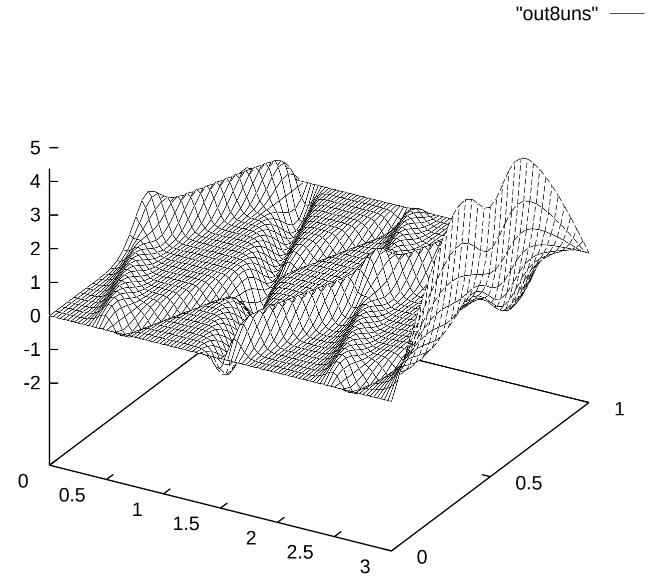
*Sample source code*—`wave1d.f`*:*

```
c===========================================================
c     Solves 1-dimensional wave equation
c
c        u_tt = u_xx     0 <= x <= 1   u(0,t) = u(1,t) = 0
c
c     using second-order finite difference techniques
c
c===========================================================
      program        wave1d

      implicit       none

      integer        iargc,      i4arg
      real*8         r8arg
c-----------------------------------------------------------
c     Storage for grid functions:
c
c     u(maxn,2)   Difference solution (two time levels)
c     lm(maxn,3)  Initial left-moving profile (lm(1:n,1))
c                 and first (lm(1:n,2)) and second
c                 (lm(1:n,3)) derivatives.
c     rm(maxn,3)  Initial right-moving profile (rm(1:n,1))
c                 and first (rm(1:n,2)) and second
c                 (rm(1:n,3)) derivatives.
c     x(maxn)     Difference mesh
c-----------------------------------------------------------
      integer        maxn
      parameter    ( maxn = 2**15 + 1 )

      real*8         u(maxn,2),   lm(maxn,3),   rm(maxn,3),
     &               x(maxn)
c-----------------------------------------------------------
c     Command-line arguments and related mesh parameters.
c-----------------------------------------------------------
      integer        level,       ncross,       olevel
      real*8         lambda,      alm,          arm

      integer        nt,          nx
c-----------------------------------------------------------
c     Locals.
c-----------------------------------------------------------
      integer        n,           np1,          nm1,
     &               j,           it,           ofreq
      real*8         dx,          dt,           lamsq,
     &               t
c-----------------------------------------------------------
c     Argument parsing.
c-----------------------------------------------------------
      if( iargc() .ne. 6 ) go to 900
      level  = i4arg(1,-1)
      if( level .lt. 1 ) go to 900
      lambda = r8arg(2,-1.0d0)
      if( lambda .lt. 0.0d0 ) go to 900
      ncross = i4arg(3,-1)
      if( ncross .lt. 1 ) go to 900
      alm    = r8arg(4,0.5d0)
      arm    = r8arg(5,0.5d0)

      olevel = i4arg(6,level)
      if( olevel .gt. level .or. olevel .lt. 1 ) olevel = level
      ofreq = 2 ** (level - olevel)

c-----------------------------------------------------------
c     Set up the finite-difference mesh.  Note that in our
c     units (where the speed of signal propagation is 1),
c     a "crossing time" is one unit of time, so 'ncross' is
c     synonymous with 'tmax'.   In the definition of 'nt',
c     we would have 'nt = ncross / dt + 1.0d0' if we knew
c     that 'dt' would always EXACTLY divide 'ncross'; to
c     guard against values such as 23.99999 being truncated
c     to 23, we add an extra 0.5d0.  We could also use the
c     'nearest-integer' function nint().
c-----------------------------------------------------------
      nx = 2 ** level + 1
      dx = 1.0d0 / (nx - 1)
      dt = lambda * dx
      nt = ncross / dt + 1.5d0
      do j = 1 , nx
```

```
          x(j) = (j - 1) * dx
       end do
       lamsq = lambda * lambda

c-----------------------------------------------------------
c      Define initial left-moving and right-moving pulses and
c      derivatives.
c-----------------------------------------------------------
       call dvgaussian(lm(1,1),lm(1,2),lm(1,3),x,nx,
     &                 alm,0.5d0,0.1d0)
       call dvgaussian(rm(1,1),rm(1,2),rm(1,3),x,nx,
     &                 arm,0.5d0,0.1d0)

c-----------------------------------------------------------
c      Define t=0, t=dt data, being careful to ensure that
c      the data at both time levels satisfy the boundary
c      conditions.
c-----------------------------------------------------------
       n    = 1
       np1  = 2
       nm1  = 2
       u(1,nm1)  = 0.0d0
       u(1,n  )  = 0.0d0
       do j = 2 , nx - 1
          u(j,nm1) = lm(j,1) + rm(j,1)
          u(j,n)   = u(j,nm1) + dt * (lm(j,2) - rm(j,2)) +
     &             0.5d0 * dt * dt * (lm(j,3) + rm(j,3))
       end do
       u(nx,nm1) = 0.0d0
       u(nx,n  ) = 0.0d0
c-----------------------------------------------------------
c      Output the t=0 data (always) and the t=dt data if
c      output is enabled for every time step.
c-----------------------------------------------------------
       call gnuout(u(1,nm1),x,nx,0.0d0,ofreq)
       t = dt
       if( ofreq .eq. 1 ) then
          vsrc = vsxynt('u'//itoc(level),t,x,u(1,n),nx)
       end if

c-----------------------------------------------------------
c      Main evolution loop.
c-----------------------------------------------------------
       do it = 2 , nt - 1
          do j = 2 , nx - 1
             u(j,np1) = 2.0d0 * u(j,n) - u(j,nm1) + lamsq * (
     &          u(j-1,n) - 2.0d0 * u(j,n) + u(j+1,n) )
          end do
          t = t + dt
c-----------------------------------------------------------
c         'gnuplot' style output every 'ofreq' steps
c-----------------------------------------------------------
          if( mod(it,ofreq) .eq. 0 ) then
             call gnuout(u(1,np1),x,nx,t,ofreq)
          end if
c-----------------------------------------------------------
c         Swap 'pointers'.
c-----------------------------------------------------------
          np1 = n
          n   = nm1
          nm1 = np1
       end do

       stop

c-----------------------------------------------------------
c      Usage.
c-----------------------------------------------------------
900    continue
          write(0,*) 'usage: wave1d <level> <dt/dx> '//
     &       '<ncross> <a left-mover> <a right-mover> '//
     &       '<olevel>'
       stop

       end
```

*Surface plots of results for stable (top) and unstable (bottom) cases.*




*Gnuplot commands for making a surface plot such as those displayed above.*

```
gnuplot> set terminal postscript landscape
gnuplot> set output "out8.ps"
gnuplot> set parametric
gnuplot> set hidden
gnuplot> splot "out8" with lines
gnuplot> quit
```