1-d Wave Equation Key

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:

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

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

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

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

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

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

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

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

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

$$u_j^0 = l_j + u_j$$

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

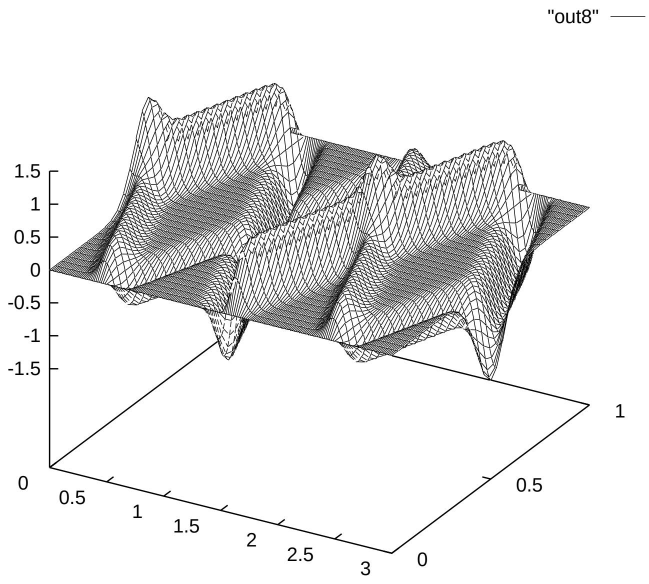
As discussed in the lectures, a *Von Neumann* stability analysis of the discrete equations of motion predicts that the above scheme will be unstable if $\lambda > 1$. This is an instance of the famous CFL (Courant-Friedrichs-Levy (1928)) condition on explicit difference-schemes for hyperbolic equations.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

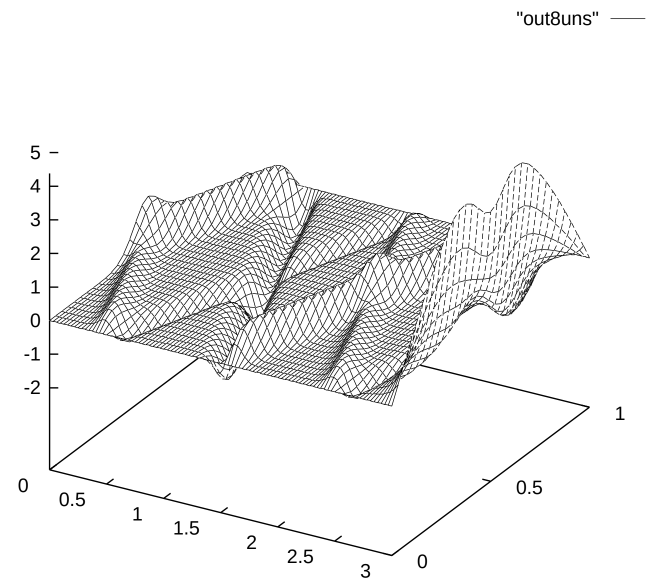
Sample source code—`wave1d.f`*:*

```
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===========================================================
      program         wave1d

      implicit        none

      integer         iargc,       i4arg
      real*8          r8arg

      integer         maxn
      parameter     ( maxn = 2**15 + 1 )

c-----------------------------------------------------------
c     Storage for grid functions:
c
c     u(maxn,3)   Difference solution (three 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-----------------------------------------------------------
      real*8          u(maxn,3),   lm(maxn,3),   rm(maxn,3),
     &                x(maxn)

c-----------------------------------------------------------
c     Command-line arguments.
c-----------------------------------------------------------
      integer         level,       ncross,       olevel
      real*8          lambda,      alm,          arm

      integer         nt,          nx

c-----------------------------------------------------------
c     Locals.
c-----------------------------------------------------------
      integer         n,           np1,          nm1,
     &                j,           it,           ofreq,
     &                ntemp
      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 * (nx - 1) / lambda + 1.5d0
      do j = 1 , nx
         x(j) = (j - 1) * dx
      end do

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     Initialize "pointers", define t=0, t=dt data being
c     careful to ensure that the data at both time levels
c     satisfy the boundary conditions.
c
c     Also, output the t = 0 data (always) and, optionally,
c     (i.e. if ofreq = 1) the t = dt data using the 'gnuout'
c     routine.
c-----------------------------------------------------------
      nm1  = 1
      n    = 2
      np1  = 3
      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

      t = 0.0d0
      call gnuout(u(1,nm1),x,nx,t,ofreq)
      t = dt
      if( ofreq .eq. 1 ) then
         call gnuout(u(1,nm1),x,nx,t,ofreq)
      end if
c-----------------------------------------------------------
c     Main evolution loop.  Note the upper bound on the
c     do loop.  Counting the first two time levels as
c     it = 0, it = 1, we want a total of nt time levels.
c     Indexing this way allows us to use
c
c        mod(it,ofreq) .eq. 0
c
c     to decide when it is time to output.
c-----------------------------------------------------------
      lamsq = lambda * lambda
      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-----------------------------------------------------------
         ntemp = nm1
         nm1   = n
         n     = np1
         np1   = ntemp
      end do

c-----------------------------------------------------------
c     Normal 'exit'.
c-----------------------------------------------------------
      stop

c-----------------------------------------------------------
c     Usage 'exit'.
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 portrait
gnuplot> set output "out8.ps"
gnuplot> set parametric
gnuplot> splot "out8" with lines
gnuplot> quit
```