Pittsburgh Group Code : Gravitational Wave Extraction - A Benchmark?
(evolution of scalar fields from characteristic data)

Summary by Tomasz Haupt and Mark Miller

Last revision: Oct. 20, 1994 Work supported in part by NSF ASC 93 18152/ PHY 93 18152 (ARPA supplemented)

Introduction

Gravitational wave extraction is one of the important aspects of the computational grand challenge program. There are two schools of thought when it comes to implementing the extraction of gravitational waves from a numerically generated spacetime. One way is to simply let the waves propogate via standard Cauchy devolopment (evolving spatial surfaces using the standard initial value formulation). One of the major drawbacks with this method is that a very large computational grid is needed to allow the waves to propogate far from the source.

The other way to study radiation far from the source is to use an initial value formulation based on "characteristic surfaces." This is the technique developed by the Pittsburgh group [1], which will be outlined below. It should be stressed that the code does *not* deal with gravitational waves. The code evolves a scalar field on a flat, background, spacetime via the standard linear wave equation. It is the hope of the Pittsburgh group that the code will be generic enough to incorporate evolving gravitational waves later.

This paper is a summary of the paper by Gómez *et. al.* [1]. It is written for computer scientists working on the computational grand challenge program "Black Hole Binaries: Coalescence and Gravitational Radiation." Comments and suggestions are welcome.

1 Characteristic Initial Value Problem

The "characteristic initial value problem" (CIVP) as approached by the Pittsburgh group is based upon two novel ingredients:

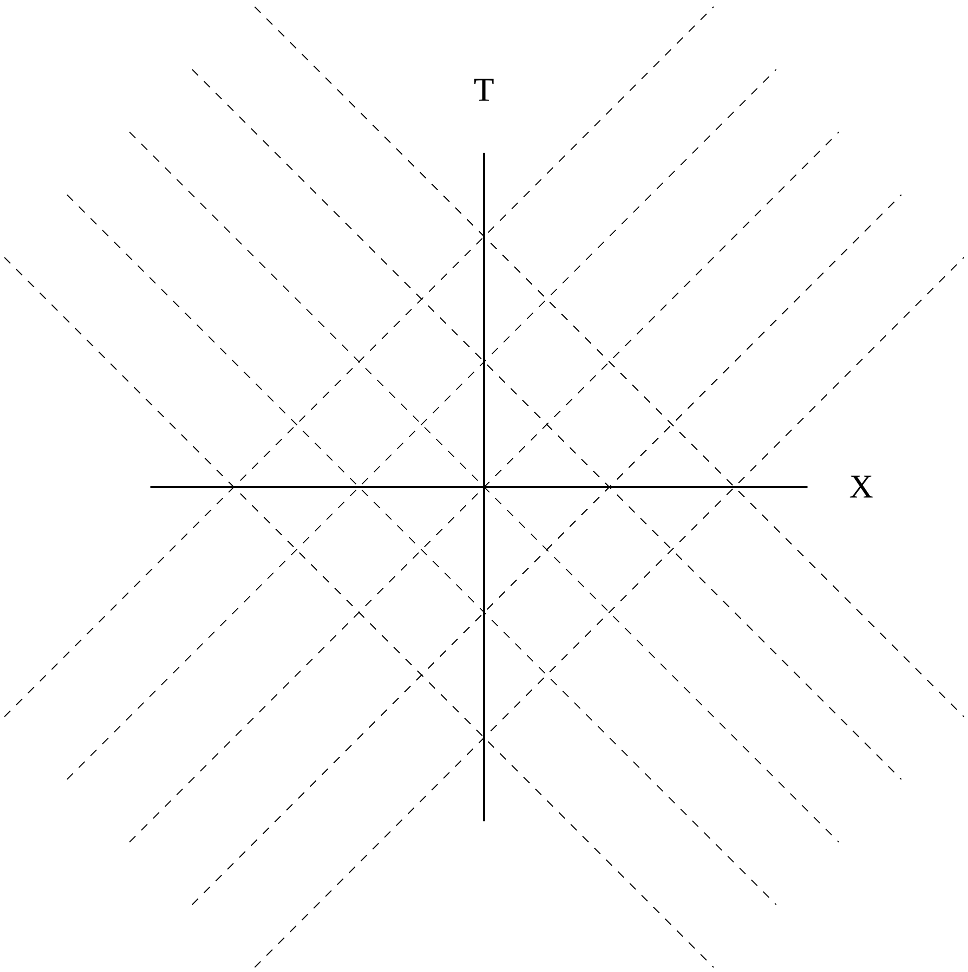
- using "characteristic surfaces" instead of spatial surfaces
- using "compactification methods" to describe asymptotic properties of radiation

Characteristic surfaces can be thought of intuitively as the surfaces in spacetime that are "traced out" by traveling waves. For instance, the scalar wave equation in two dimensions is

$$-\frac{\partial^2\Phi}{\partial t^2}+\frac{\partial^2\Phi}{\partial x^2}=0$$

Any function $f(x \pm t)$ is a solution to this equation, so the characteristic surfaces are given by

$$x \pm t = const$$



Two Dimensional Characteristic Surfaces

Remember, we have set the speed of light $c = 1$. Therefore all of the characteristic surfaces are 45^o from the x or t axis. The reader may recognize that the characteristic surfaces through any point is referred to as the "light cone" associated with that point.

The key idea of "compactification methods" is to replace a coordinate that ranges from 0 to ∞ with a new coordinate that ranges from 0 to 1. One example is the compactification of the two-dimensional plane. Let the plane be coordinatized by the usual polar coordinates (r, ϕ). The coordinate r ranges from 0 to ∞. We can replace this radial coordinate with a new radial coordinate,

ξ :

$$\xi = \frac{r/r_0}{1 + r/r_0}$$

where r_0 is some constant. We see that as r ranges from 0 to ∞, ξ ranges from 0 to 1. In this way, one can actually represent the point $r = \infty$ on a computational grid ($\xi = 1$)!!

2 Scalar Wave Equation

The scalar wave equation in 4 dimensions (using cartesian coordinates) is

$$-\frac{\partial^2 \Phi}{\partial t^2} + \frac{\partial^2 \Phi}{\partial x^2} + \frac{\partial^2 \Phi}{\partial y^2} + \frac{\partial^2 \Phi}{\partial z^2} = 0$$

where $\Phi = \Phi(t, x, y, z)$.

In spherical polar coordinates, (t, r, θ, ϕ), the scalar wave equation is

$$-\frac{\partial^2 \Phi}{\partial t^2} + \frac{1}{r}\frac{\partial^2}{\partial r^2}(r\Phi) + \frac{1}{r^2 \sin\theta}\frac{\partial}{\partial \theta}(\sin\theta \frac{\partial \Phi}{\partial \theta}) + \frac{1}{r^2 \sin^2\theta}\frac{\partial^2 \Phi}{\partial \phi^2} = 0$$

To cast this equation in a "characteristic surface form", we replace the time coordinate, t, with a coordinate that labels characteristic surfaces:

$$u = t - r$$

The u coordinate is called the "retarded time," and it labels outgoing characteristic surfaces. In (u, r, θ, ϕ) coordinates, called null spherical polar coordinates, the scalar wave equation takes the form

$$-2\frac{\partial^2 (r\Phi)}{\partial u \partial r} + \frac{\partial^2 (r\Phi)}{\partial r^2} + \frac{1}{r \sin\theta}\frac{\partial}{\partial \theta}(\sin\theta \frac{\partial \Phi}{\partial \theta}) + \frac{1}{r \sin^2\theta}\frac{\partial^2 \Phi}{\partial \phi^2} = 0$$

The thing one should notice about this equation is that it is first order in retarded time, u. This means that only the value of Φ on the initial surface need be specified as initial data. This is in contrast to the initial data for the scalar wave equation involving the regular time, t. In that case, the equation is second order in t, so that initial data consists of the value of Φ *and* its time derivative of the initial surface.

To carry out the compactification, we replace the radial coordinate, r, with a new coordinate ξ defined by

$$\xi = \frac{r/r_0}{1 + r/r_0}$$

We see that as r ranges from 0 to ∞, ξ ranges from 0 to 1. For numerical computations, the angle coordinates, θ and ϕ, are replaced by η and ζ:

$$\eta = -\cos\theta$$

$$\zeta = \phi$$

These are the coordinates used for numerical computations. The ranges for the coordinates are :

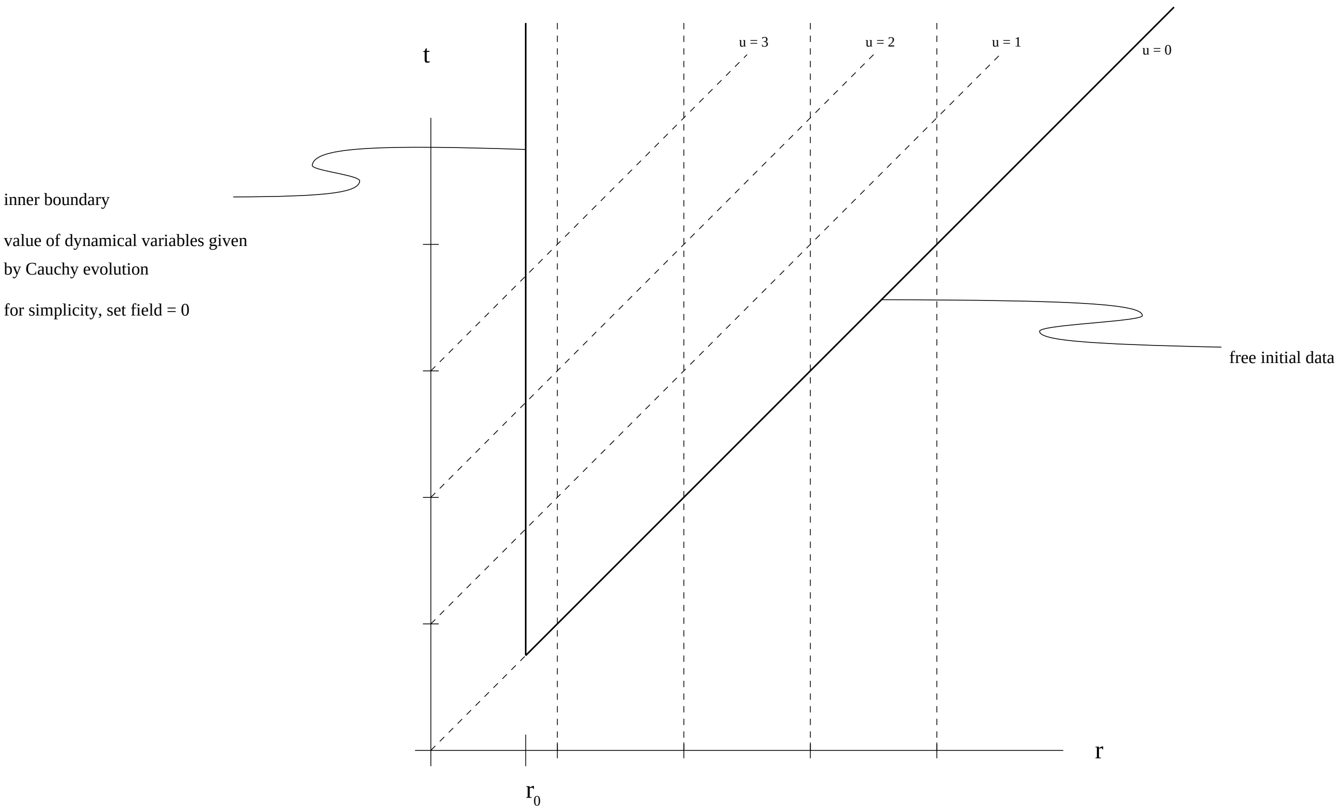
$$u_{initial} \leq u \leq u_{final}$$

$$1/2 \leq \xi \leq 1$$

$$-1 \leq \eta \leq 1$$

$$0 \leq \zeta \leq 2\pi$$

Note that ξ only ranges from 1/2 to 1. The point $\xi = 1$ corresponds to infinite radial distance, $r = \infty$. However, $\xi = 1/2$ corresponds to the radial distance $r = r_0$. In a realistic gravitational wave calculation, the spacetime can be intuitively be broken into two parts. In the part where $r < r_0$ (where the black holes are), evolution will be attained via Cauchy development (the standard initial value formulation). In the part where $r > r_0$ (the radiation part of the spacetime), evolution will be attained via the characteristic initial value formulation. The method used to transport information from the Cauchy development to the characteristic surface evolution is another big problem for this computational grand challange program. For simplicity, the Pittsburgh code sets the value of Φ to be 0 at the $r = r_0$ boundary (this is the boundary condition for a perfect reflector). The boundary to the computational domain is given in the r-t diagram below.



The Pittsburgh Group works with a new dynamical variable, which is just the old dynamical variable multiplied by the radial coordinate, r:

$$G = r\Phi$$

3 Numerical Algorithm

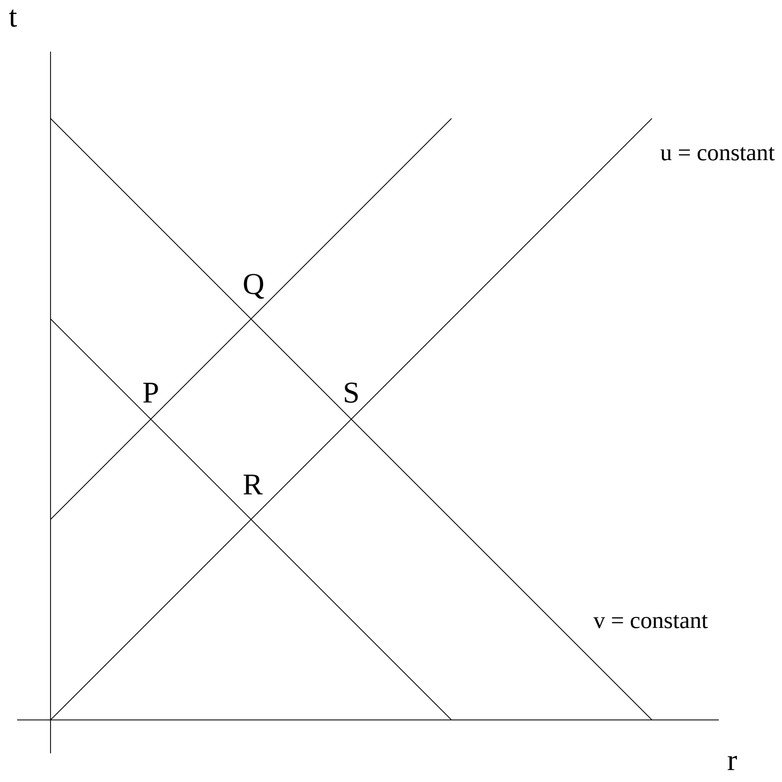
3.1 Physical Interpretation

Although the coordinates (u, ξ, η, ζ) are used as coordinates for the computational grid, the numerical method employs a mixture of two sets of coordinates. The other coordinates (we will call them "physical coordinates") have the same angular variables (η, ζ), but the (t, r) coordinates are replaced by (u, v):

$$u = t - r$$

$$v = t + r$$

The u coordinate is the previously defined "retarded time," which labels the outgoing characteristic surfaces. The v coordinate , called the "advanced time," labels the ingoing characteristic surfaces.



Instead of finite differencing the scalar wave equation directly, the Pittsburgh group integrates the equation over a "physical cell" (as apposed to a computational cell). An example of a physical cell is the cell formed by the vertices $QPRS$ in the figure above. If the scalar wave equation expressed in physical coordinates (u, v, η, ζ) is integrated over the physical cell $QPRS$, the resulting integral equation is remarkably simple:

$$G_Q = G_P + G_S - G_R + \int du\, dv[-\frac{L^2 G}{(u+v)^2}]$$

where the integration is over the cell $QPRS$. The term $L^2 G$ is just shorthand notation for various angle derivatives, the so-called " angular momentum operator," which, in η-ζ coordinates is

$$L^2 G = -(1-\eta^2)\frac{\partial^2 G}{\partial \eta^2} + 2\eta\frac{\partial G}{\partial \eta} - \frac{1}{(1-\eta^2)}\frac{\partial^2 G}{\partial \zeta^2}$$

3.2 The Finite Difference Equation

The coordinates that are actually used in the computation, (u, ξ, η, ζ), are discretized as follows:

$$u_n = n\,\Delta u$$

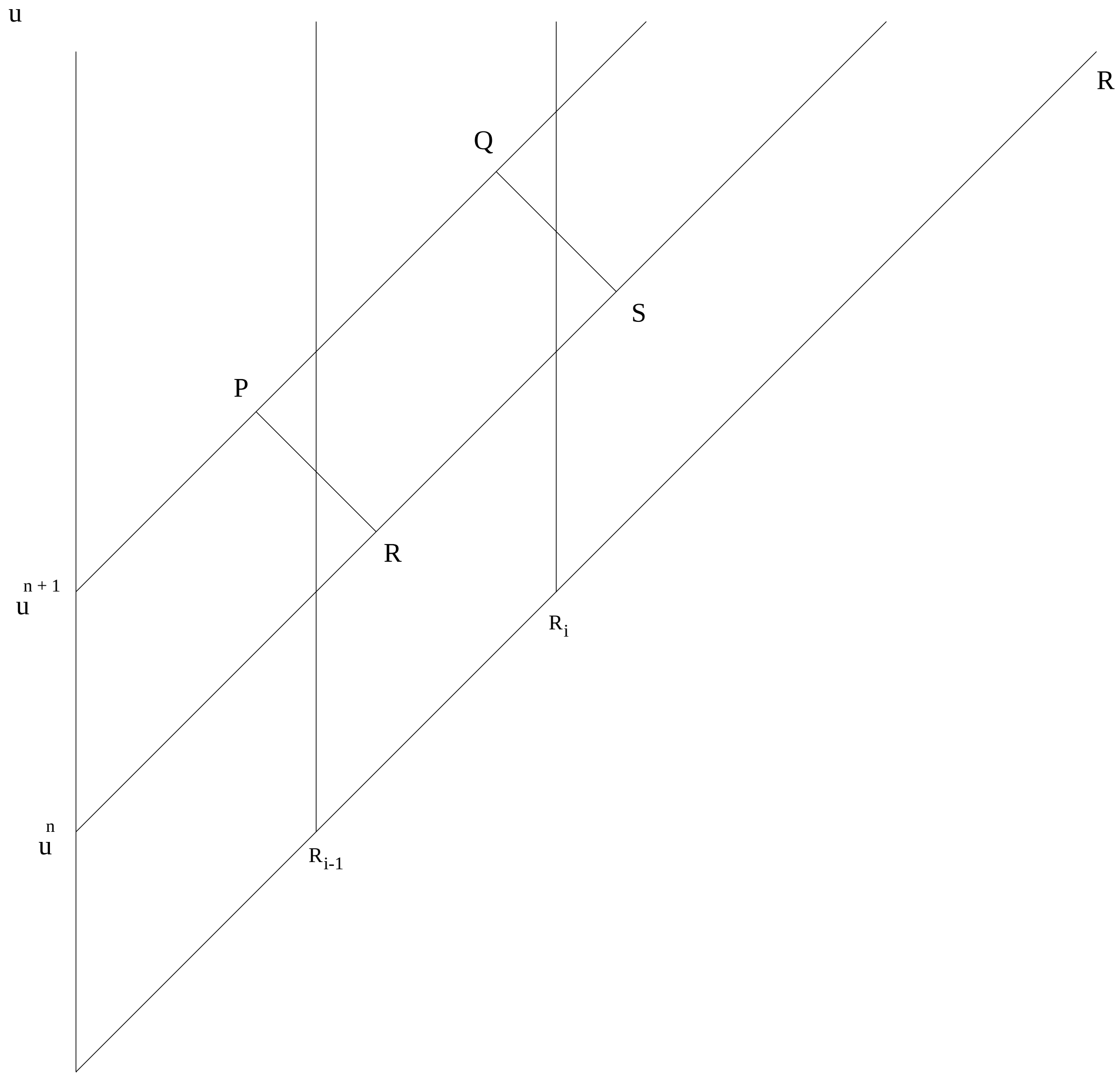
$$\xi_i = i\,\Delta\xi$$
$$\eta_j = j\,\Delta\eta$$
$$\zeta_k = k\,\Delta\zeta$$

The field value at these sites is denoted by

$$G^n_{ijk} = G(u_n, \xi_i, \eta_j, \zeta_k)$$

It turns out that the vertices of the physical cell cannot be placed at the coordinate grid points. The reason for this is that the computational coordinates employ the *compactified* (i.e. non linear in r) radial coordinate, ξ. Intervals of equal spacing in ξ will result in intervals of differing spacing in r. Because the retarded time is linear in t, there is no way to fit null curves to the coordinate grid. (Remember, null curves are the boundaries of the physical cells. For instance, the line segment QP is the path a photon would take through the r-t plane.)

The Pittsburgh group claims that detailed numerical experimentation suggests that a stable algorithm with high accuracy will result if the physical cell is situated on the computational grid in the following way:



That is, the sides formed by incoming characteristics intersect adjacent outgoing characteristic surfaces at equal but opposite ξ-displacements from the neighboring grid points. Expressions for the u-ξ coordinates of the vertices of the physical cell (Q,P,R, and S) given to second-order are

$$\xi_{i-1} - \xi_P = \xi_R - \xi_{i-1} = \Delta u \, (1 - \xi_{i-1})^2/4$$

$$\xi_i - \xi_Q = \xi_S - \xi_i = \Delta u \, (1 - \xi_i)^2/4$$

The field values at the vertices of the physical cell are obtained by quadratic interpolation, for example,

$$G_P = [G_{i-2}(\xi_P - \xi_{i-1})(\xi_P - \xi_i) - 2G_{i-1}(\xi_P - \xi_{i-2})(\xi_P - \xi_i) + G_i(\xi_P - \xi_{i-2})(\xi_P - \xi_{i-1})]/(\Delta\xi)^2$$

By doing a Taylor series expansion of G around the point at $(\xi_{i-1/2}, u_{n+1/2})$, it can be seen that the above expression for G_P is second order accurate to the analytic solution. That is,

$$G_P = [G_P]_{analytic} + \mathcal{O}((\Delta\xi)^2)$$

The integral is evaluated by treating the integrand as a constant over the physical cell, with value equal to the average at the center. The angular derivatives in the term L^2G are replaced with standard second-order-accurate finite difference approximations.

It turns out that the resulting finite difference equation is equivalent to the equation obtained by finite differencing the scalar wave equation directly using certain "angled derivative" approximations for the appearing derivatives (see below).

In (u, ξ, η, ζ) coordinates, the scalar wave equation takes the form

$$-\frac{\partial^2 G}{\partial u\,\partial\xi} + \frac{\partial}{\partial\xi}[(1-\xi)^2\frac{\partial G}{\partial\xi}] + \frac{1}{\xi^2}\frac{\partial}{\partial\eta}[(1-\eta^2)\frac{\partial G}{\partial\eta}] + \frac{1}{\xi^2(1-\eta^2)}\frac{\partial^2 G}{\partial\zeta^2} = 0$$

The "angled derivative" approximations used are the following second order approximations:

$$\frac{\partial^2 G}{\partial u\,\partial\xi} = (G^{n+1}_{i,j,k} - G^{n+1}_{i-1,j,k} - G^n_{i,j,k} + G^n_{i-1,j,k})\frac{1}{\Delta u\,\Delta\xi}$$

$$\frac{\partial^2 G}{\partial\xi^2} = (G^{n+1}_{i,j,k} - 2G^{n+1}_{i-1,j,k} + G^{n+1}_{i-2,j,k} + G^n_{i+1,j,k} - 2G^n_{i,j,k} + G^n_{i-1,j,k})\frac{1}{2\Delta\xi^2}$$

$$\frac{\partial^2 G}{\partial\eta^2} = (G^{n+1}_{i-1,j+1,k} - 2G^{n+1}_{i-1,j,k} + G^{n+1}_{i-1,j-1,k} + G^n_{i,j+1,k} - 2G^n_{i,j,k} + G^n_{i,j-1,k})\frac{1}{2\Delta\eta^2}$$

$$\frac{\partial^2 G}{\partial\zeta^2} = (G^{n+1}_{i-1,j,k+1} - 2G^{n+1}_{i-1,j,k} + G^{n+1}_{i-1,j,k-1} + G^n_{i,j,k+1} - 2G^n_{i,j,k} + G^n_{i,j,k-1})\frac{1}{2\Delta\zeta^2}$$

3.3 Stability Issues

A von Neumann stability analysis of the above finite differencing scheme, where the coordinate coefficients of the derivatives are treated as constants, reveals the fact that the modulus of the amplification factor is exactly 1 (For an introduction to basic concepts in finite differencing, including an introduction to von Neumann stability analysis, see [2]).

The Courant condition for the scalar wave equation in ordinary cartesian coordinates is simply

$$\Delta t \leq \frac{\Delta}{\sqrt{3}}$$

where $\Delta = min(\Delta x, \Delta y, \Delta z)$. (Remember, the speed of light is equal to 1). However, due to the fact that the Pittsburgh group uses a scalar wave equation that is first order in retarded time, the Courant condition is more restrictive:

$$\Delta u \leq f(\Delta\xi, \Delta\eta, \Delta\zeta) \sim \mathcal{O}(\Delta^2)$$

The exact structure of the function $f(\Delta\xi, \Delta\eta, \Delta\zeta)$ can be found in [1], but it should be pointed out that f is *second order* in the Δ's. This is the trade-off for dealing with a first order (in time) equation.

3.4 Evolution Scheme

The final finite difference equation connects G_i^{n+1} with those of neighboring points on the coordinate lattice which are either earlier in time ($G_{i-1}^n, G_i^n, G_{i+1}^n$), or else contemporary but located at smaller radius ($G_{i-2}^{n+1}, G_{i-1}^{n+1}$). Consequently, it is possible to move through the interior of the lattice, computing G_i^{n+1} explicitly by an orderly march. This is achieved by starting at $\xi = 1/2$ at time u_{n+1}. (It is assumed that all field values at the earlier time, u_n, are known via initial conditions or from previous evolution.) Proceed outward one radial step, using the finite difference equation to calculate all field values at all angular sites on the lattice. Step outward one radial step and repeat. Continue in this manner, until the the ξ coordinate reaches 1. Thus, all the field values out to radiative infinity along the new characteristic surface at u_{n+1} are updated, completing one evolutionary time step.

To completely specify the evolution algorithm we need to describe the treatment of G at the coordinate boundaries. Initial data is given at $u = u_0$ to start the method. Periodic boundary conditions are applied for the angle ϕ, so that $G(u, \xi, \eta, 0) = G(u, \xi, \eta, 2\pi)$. Angular derivatives at the axis $\eta = \pm 1$ are computed using second-order-accurate forward or backward differences. At the inner radial edge, a reflective boundary condition is used: $G(\xi = 1/2) = 0$ (see the discussion in section 2). At the outer edge, $\xi = 1$, the finite difference equation no longer applies, and it is necessary to reapply the basic integral identity. This is straightforward, since at radiative infinity, the points Q and S are grid points: $\xi_P = \xi_S = 1$.

4 The Code

The flow diagram of the code is simple:

GRID: initialize coordinates

GETPARAM: set step size du, u=0, u_final

SURDATA: initialize grid function g at u=0

do while (u. lt. u_final)

> **u=u+du**
> **COPY: g_old=g**
> **BOUNDARY: set boundary conditions**
> **ADVANCE: calculate new values of g**

end do

The subroutine **advance** implements the marching algorithm, computing new values of $G^{n+1}_{i,j,k}$ on the surface u_{n+1}. The i index corresponds to the compactified radial coordinate r, while j and k correspond to angular coordinates. Essentially, values of $G_{i,j,k}$ are evaluated in a sequential loop over i. However, because of boundary conditions, the first point, $G_{1,j,k}$, and the last point, $G_{nx,j,k}$ on u_{n+1} are calculated separately. The structure of the finite difference equations allows one to collect all computations involving angular coordinates (the L^2 operator) in a separate routine **mkl2**. Note that the L^2 operator does not depend on r (index i), thus **mkl2** returns values of the L^2 operator on a two dimensional mesh. For each i, **mkl2** is called twice: once with G^{n+1}_i, and once with G^n_i.

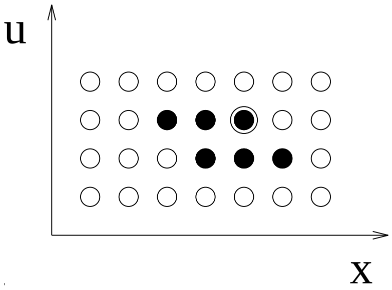
A simplified version of the subroutine **advance** looks as follows:

do i=1,nx
  do j=1,ny
    do k=1,nz

$$\begin{aligned} G^{n+1}_{i,j,k} = {} & C1 * ((L^2)^n_{i,j,k} + (L^2)^{n+1}_{i,j,k})/2 + \\ & C2 * G^{n+1}_{i-1,j,k} + C3 * G^{n+1}_{i-2,j,k} + \\ & C4 * G^n_{i+1,j,k} + C5 * G^n_{i,j,k} + C6 * G^n_{i-1,j,k} \end{aligned}$$

    enddo
  enddo
enddo

where $C1,...,C6$ are coefficients depending on du,dx and x(i). The neighbor dependence of the updating process can be symbolized as follows (x is the radial coordinate represented in the mesh by the index i):



The loop over i is sequential: one cannot proceed with the i-th iteration until the $(i-1)$th iteration is completed. However, for a given i, the loops over j and k can be executed in parallel. To express parallelism in HPF implementation, these loops are replaced by a **FORALL** statement:

do i = 1,nx
  FORALL (j=1:ny, k=1:nz)

$$G^{n+1}_{i,j,k} = C1 * ((L^2)^n_{i,j,k} + (L^2)^{n+1}_{i,j,k})/2 + C2 * G^{n+1}_{i-1,j,k} + C3 * G^{n+1}_{i-2,j,k} + C4 * G^n_{i+1,j,k} + C5 * G^n_{i,j,k} + C6 * G^n_{i-1,j,k}$$

enddo

A similar transformation is made within the subroutine **mkl2**.

The HPF code implements this calculation on a 2-dimensional grid of processors declared by the HPF directive

```
!HPF$ PROCESSORS PPP(2,2)
```

The declared size of the processor grid reflects the fact that the code has been run on the NPAC's alpha cluster (maximum physical nodes available is 8). On a 256-node system, one can declare PPP(16,16), for example. The 3-dimensional grid function G is distributed onto this processor arrangement in a BLOCK-BLOCK fashion in (j,k) subspace, while the first axis, i, is collapsed:

```
!HPF$ DISTRIBUTE G(*,BLOCK,BLOCK) ONTO PPP
```

(Note that for small data sets distribution along only one axis, say, (*,*,BLOCK), may lead to a better performance of the resulting code). To force the locality of data and thus to minimize communication, grid function G^n is aligned directly to G^{n+1}, and arrays with only angular indices (used in **mkl2**) are aligned to the angular indices of G

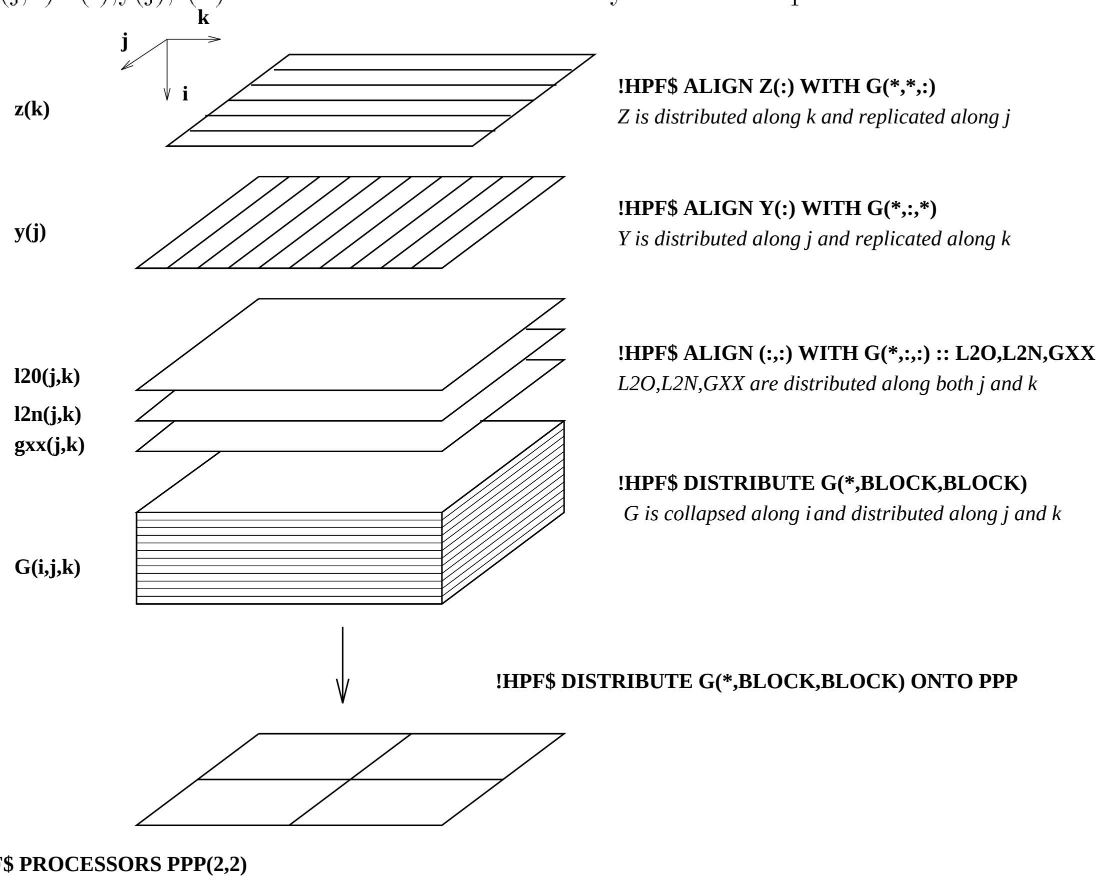
```
!HPF$ ALIGN GO(:,:,:)  with G(:,:,:)
!HPF$ ALIGN (:,:)  with G(*,:,:)  ::  gy,gyy,gzz,l2n,l2o,gx,gyy
```

Coordinate arrays y,z are correspondingly replicated row-wise and column-wise

```
!HPF$ ALIGN y(:)  WITH G(*,:,*)
!HPF$ ALIGN z(:)  WITH G(*,*,:)
```

and each processor has a complete replica of x.

As a result, for any combination of (i,j,k), the array elements $G^{n+1}_{i,j,k}$, $G^n_{i,j,k}$, l2n(j,k), l2o(j,k), ..., gyy(j,k) x(i),y(j),z(k) are stored in the local memory of the same processor.



The parallelism of the algorithm is expressed by FORALL statements and Fortran90 intrinsic function SUM. In many cases, Fortran 90 array assignments can be used instead of FORALL without any influence on both sematics of the program and its performace. The following loop is an example where the choice between Fortran 90 and FORALL syntax is a matter of a personal preference of the programmer:

```
        FORALL(j=0:ny, k=0:nz)
     &         gn(nx,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &        + (beta - 2 * alpha * alpha)       * gn(nx-1,j,k)
     &        + alpha * (alpha + 1.)             * gn(nx-2,j,k)
     &        + (beta + alpha * (2. * alpha - 3.)) * go(nx,j,k)
     &        + (-beta - alpha * (5. * alpha - 4.)) * go(nx-1,j,k)
     &        + alpha * (4. * alpha - 1.) * go(nx-2,j,k)
     &        + (-alpha * alpha) * go(nx-3,j,k)
     &        ) / (beta - alpha * (alpha - 1.))
```

and equivalent Fortran 90 statement:

```
              gn(nx,:,:) = (gamma * 0.5 * (l2o(:,:) + l2n(:,:))
     &        + (beta - 2 * alpha * alpha)       * gn(nx-1,:,:)
     &        + alpha * (alpha + 1.)             * gn(nx-2,:,:)
     &        + (beta + alpha * (2. * alpha - 3.)) * go(nx,:,:)
     &        + (-beta - alpha * (5. * alpha - 4.)) * go(nx-1,:,:)
     &        + alpha * (4. * alpha - 1.) * go(nx-2,:,:)
     &        + (-alpha * alpha) * go(nx-3,:,:)
     &        ) / (beta - alpha * (alpha - 1.))
```

Sometimes, Fortran 90 syntax is more elegant, for example compare:

```
     go = gn
```

with

```
     FORALL(i=0:nx,j=0:ny,k=0:nz) go(i,j,k)=gn(i,j,k)
```

However, Fortran90 arrays syntax introduces unpleasant limitation: objects at both sides of the assignment must be conformable (i.e., of the same shape). This code provides two examples of loops which can be easily expressed using the FORALL syntax, while they are awkward in Fortran90 implementation:

```
       FORALL(i=0:nx, j=0:ny, k=0:nz, (x(i).gt.x1) .and. (x(i).lt.x2))
     &           g(i,j,k) = ((x(i) - x1) * (x2 - x(i))) ** 2
     &                   * (1. - y(j) ** 2)
     &                   * cos(2. * z(k))
```

In Fortran90, one must declare three additional arrays:

```
      xwork=SPREAD(SPREAD(x,2,ny+1),3,nz+1)
      ywork=SPREAD(SPREAD(y,1,nx+1),3,nz+1)
      zwork=SPREAD(SPREAD(z,1,nx+1),2,ny+1)

      WHERE(xwork.gt.x1 .and. xwork.lt.x2)
     &  gn = ((xwork-x1)*(x2-xwork)**2)*(1.-ywork**2)*cos(2.*zwork)
```

or

```
         FORALL(j=1: ny-1, k=0:nz)
     &         l2(j,k) = - ((1. - y(j) ** 2) * gyy(j,k)
     &                     - 2. * y(j) * gy(j,k)
     &                     + gzz(j,k) / (1. - y(j) * y(j)))
```

which is equivalent to

```
          l2(1:ny-1,:) = - ((1. - SPREAD(y(1:ny-1),2,nz+1) ** 2)
     &                    * gyy(1:ny-1,:)
     &                    - 2. * SPREAD(y(1:ny-1),2,nz+1) * gy(1:ny-1,:)
     &                    + gzz(1:ny-1,:) /
     &                    (1. - SPREAD(y(1:ny-1),2,nz+1)**2))
```

Here, it is assumed that the compiler is smart enough to create only one copy of the temporary array SPREAD(y(1:ny-1),2,nz+1).

Finally, some deficiencies of the array assignment can be overcome by array constructors, like in the following example:

```
      x = 0.5 + (/ (i, i=0,nx) /) * dx
```

which is equivalent to

```
      FORALL(i=0:nx) x(i) = 0.5 + i * dx
```

For educational purposes, we developed a "pure" Fortran 90 implementation (i.e., without FORALL) as well.

A Pittsburgh Code: Fortran 77

```
*     nx, ny, nz:               grid sizes
*     x, y, z:                  grid arrays
*     gn, go:                   'new' and 'old' field arrays
*     gy, gyy, gzz:             angular derivatives
*     l2n, l2o:                 'new' and 'old' laplacian on the sphere

       integer nx, ny, nz
       parameter (nx = 128, ny = 64, nz = 64)
c       parameter (nx = 8, ny = 8, nz = 8)
       real x(0:nx), y(0:ny), z(0:nz)
       real gn(0:nx,0:ny,0:nz), go(0:nx,0:ny,0:nz)
       real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz)
       real l2n(0:ny,0:nz), l2o(0:ny,0:nz)

*     dx, dy, dz:               grid spacings
*     u, uf, du:                initial time, final time, time step
*     iter, iterskip:           iteration counter, no. of iter between dumps
*     islice, isliceno:         selects x, y or z slicing, slice number

       real dx, dy, dz
       real u, uf, du
       integer iter, iterskip
       integer islice, isliceno

ccc       open (unit = 11, file = 'plot.gnu', status = 'unknown')

       call grid(x, y, z, nx, ny, nz, dx, dy, dz, du)

       call getparam(u, uf, du, iterskip, islice, isliceno)

       call surfdata(gn, x, y, z, nx, ny, nz, u)

ccc       call wslice(gn, x, y, z, nx, ny, nz, islice, isliceno, u)

       iter = 0
       do while (u .lt. uf)
          iter = iter + 1
          u = u + du
          call copy(gn, go, nx, ny, nz)
```

```
          call boundary(gn, go, nx, ny, nz, dx, dy, dz, du, u)
          call advance(gn, go, gy, gyy, gzz, l2o, l2n, x, y, z,
     &                nx, ny, nz, dx, dy, dz, du)

ccc          if (mod(iter,iterskip) .eq. 0)
ccc     &      call wslice(gn, x, y, z, nx, ny, nz, islice, isliceno, u)

       end do

ccc       close (unit = 11)

       write(6,*) "gn(0,      0,   0)=", gn(0,0,0)
       write(6,*) "gn(1,      1,   1)=", gn(1,1,1)
       write(6,*) "gn(2,      2,   2)=", gn(2,2,2)
       write(6,*) "gn(1,   ny-5,   2)=", gn(1,ny-5,2)
       write(6,*) "gn(nx-2,   0,   0)=", gn(nx-2,0,0)
       write(6,*) "gn(nx-2,ny-5,nz-5)=", gn(nx-2,ny-5,nz-5)
       write(6,*) "gn(nx,  ny-1,nz-1)=", gn(nx,ny-1,nz-1)


       end
       subroutine grid(x, y, z, nx, ny, nz, dx, dy, dz, du)
       real x(0:nx), y(0:ny), z(0:nz)
       integer nx, ny, nz
       real dx, dy, dz, du

       real pi
       integer i, j, k

       pi = 3.14159265358979323846

       dx = 0.5 / float(nx)
       dy = 2. / float(ny)
       dz = 2. * pi / float(nz + 1)

       du = min(dx,dy,dz) / 2.

       do i = 0, nx
           x(i) = 0.5 + i * dx
       end do

       do j = 0, ny
```

```
      y(j) = -1. + j * dy
   end do

   do k = 0, nz
       z(k) = k * dz
   end do

   return
   end
   subroutine surfdata(g, x, y, z, nx, ny, nz, u)
   integer nx, ny, nz
   real x(0:nx), y(0:ny), z(0:nz)
   real g(0:nx,0:ny,0:nz)
   real u

   integer i, j, k
   real x1, x2

   x1 = 0.6
   x2 = 0.9

   do k = 0, nz
      do j = 0, ny
         do i = 0, nx
            if ((x(i) .gt. x1) .and. (x(i) .lt. x2)) then
               g(i,j,k) = ((x(i) - x1) * (x2 - x(i))) ** 2
&                      * (1. - y(j) ** 2)
&                      * cos(2. * z(k))
            else
               g(i,j,k) = 0.
            end if
         end do
      end do
   end do

   return
   end
   subroutine boundary(gn, go, nx, ny, nz, dx, dy, dz, du, u)
   real gn(0:nx,0:ny,0:nz), go(0:nx,0:ny,0:nz)
   integer nx, ny, nz
   real dx, dy, dz, du, u
```

```
*     ... for the time being, set the boundary to zero,
*         i.e. equivalent to having a perfect reflector at x=0.5

      integer j, k

      do k = 0, nz
         do j = 0, ny
            gn(0,j,k) = 0.
         end do
      end do

      return
      end
c      subroutine wslice(g, x, y, z, nx, ny, nz, islice, isliceno, u)
c      integer nx, ny, nz, islice, isliceno
c      real x(0:nx), y(0:ny), z(0:nz)
c      real g(0:nx,0:ny,0:nz)
c      real u
c
c      integer i, j, k
c      character filename*14
c
c      filename(1:2) = 'u='
c      write/home/B8D/haupt/BH/PITTSB/f77 (filename(3:14),'(e12.6)') u
c      open (unit = 10, file = filename, status = 'unknown')
c
c      if (islice .eq. 1) then
c         i = isliceno
c         do k = 0, nz
c            write (10,120) (y(j), z(k), g(i,j,k), j = 0, ny)
c            write (10,130)
c         end do
c      else if (islice .eq. 2) then
c         j = isliceno
c         do k = 0, nz
c            write (10,120) (x(i), z(k), g(i,j,k), i = 0, nx)
c            write (10,130)
c         end do
c      else if (islice .eq. 3) then
c         k = isliceno
c         do j = 0, ny
c            write (10,120) (x(i), y(j), g(i,j,k), i = 0, nx)
```

```
c            write (10,130)
c         end do
c      end if
c
c      close (unit = 10)
c
c      write (unit = 11, 150) 'splot "' // filename // '"'
c
c 120   format(3(5x,e12.5))
c 130   format('')
c 150   format(a)
c
c      return
c      end

      subroutine copy(gn, go, nx, ny, nz)
      integer nx, ny, nz
      real gn(0:nx,0:ny,0:nz), go(0:nx,0:ny,0:nz)

      integer i, j, k

      do k = 0, nz
         do j = 0, ny
            do i = 0, nx
               go(i,j,k) = gn(i,j,k)
            end do
         end do
      end do

      return
      end
      subroutine advance(gn, go, gy, gyy, gzz, l2o, l2n, x, y, z,
     &                  nx, ny, nz, dx, dy, dz, du)
      integer nx, ny, nz
      real gn(0:nx,0:ny,0:nz), go(0:nx,0:ny,0:nz)
      real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz)
      real l2o(0:ny,0:nz), l2n(0:ny,0:nz)
      real x(0:nx), y(0:ny), z(0:nz)
      real dx, dy, dz, du

      integer i, j, k
      real gx, gxx, xh
```

```
      real alpha, beta, gamma

*
*     ... at the inner boundary
*         evaluate the L^2 operator acting on g ...
*         on the new and old levels ...
*
          call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, 1)
          call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, 0)
*
*     ... and insert into a modified form of the evolution algorithm ...
*
      xh = x(0) + 0.5 * dx
      do k = 0, nz
         do j = 0, ny
            gx = (go(1,j,k) - go(0,j,k)) / dx
            gxx = (3. * go(0,j,k)
     &            - 7. * go(1,j,k)
     &            + 5. * go(2,j,k)
     &                -  go(3,j,k)
     &             ) / (2. * dx * dx)
            gn(1,j,k) = gn(0,j,k)
     &                + go(1,j,k)
     &                - go(0,j,k)
     &                + ((1. - xh) ** 2 * gxx - 2. * (1. - xh) * gx
     &                   - 0.5 * (l2o(j,k) + l2n(j,k)) / (xh * xh)
     &                  ) * (0.5 * du * dx)
         end do
      end do
*
*     ... over the bulk of the grid ...
*
      do i = 2, nx - 1

         alpha = (1. - (x(i) - 0.5 * dx)) / dx
         beta  = 4. / (du * dx)
         gamma = - 2. / (x(i) - 0.5 * dx) ** 2
*
*       ... evaluate the L^2 operator acting on g
*           on the new and old levels ...
*
         call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, i)
```

```
          call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, i-1)

*
*         ... and insert into the evolution algorithm ...
*
          do k = 0, nz
             do j = 0, ny
                gn(i,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &          + (beta - 2 * alpha * alpha)      * gn(i-1,j,k)
     &          + alpha * (alpha + 1.)            * gn(i-2,j,k)
     &          + alpha * (alpha - 1.)            * go(i+1,j,k)
     &          + (beta - 2 * alpha * alpha)      * go(i,j,k)
     &          + (- beta + alpha * (alpha + 1.)) * go(i-1,j,k)
     &          ) / (beta - alpha * (alpha - 1.))
             end do
          end do
         end do

*
*      ... at the outer boundary
*          evaluate the L^2 operator acting on g
*          on the old level ...
*
       call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, nx)
       call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, nx-1)

*
*      ... and insert into a modified form of the evolution algorithm ...
*
       do k = 0, nz
          do j = 0, ny
                gn(nx,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &          + (beta - 2 * alpha * alpha)        * gn(nx-1,j,k)
     &          + alpha * (alpha + 1.)              * gn(nx-2,j,k)
     &          + (beta + alpha * (2. * alpha - 3.)) * go(nx,j,k)
     &          + (-beta - alpha * (5. * alpha - 4.)) * go(nx-1,j,k)
     &          + alpha * (4. * alpha - 1.) * go(nx-2,j,k)
     &          + (-alpha * alpha) * go(nx-3,j,k)
     &          ) / (beta - alpha * (alpha - 1.))
          end do
       end do
```

```
      return
      end
      subroutine mkl2(g, gy, gyy, gzz, l2, y, nx, ny, nz, dy, dz, i)
      integer nx, ny, nz
      real g(0:nx,0:ny,0:nz)
      real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz), l2(0:ny,0:nz)
      real y(0:ny)
      real dy, dz
      integer i

      integer j, k
      real g_north,  g_south
      real gm_north, gm_south

*
*       calculate the 'phi' derivatives
*

         do j = 0, ny
            gzz(j,0) = (g(i,j,1)
     &           - 2. * g(i,j,0)
     &                + g(i,j,nz)
     &                  ) / (dz * dz)
            do k = 1, nz - 1
               gzz(j,k) = (g(i,j,k+1)
     &              - 2. * g(i,j,k)
     &                   + g(i,j,k-1)
     &                    ) / (dz * dz)
            end do
            gzz(j,nz) = (g(i,j,0)
     &            - 2. * g(i,j,nz)
     &                 + g(i,j,nz-1)
     &                  ) / (dz * dz)
         end do

*
*       ... estimate the average value of g on a small circle around
*           the poles
*
         g_north  = 0.
         g_south  = 0.
         gm_north = 0.
```

```
          gm_south = 0.
          do k = 0, nz
             g_north  = g_north  + g(i,ny,k)
             g_south  = g_south  + g(i,0,k)
             gm_north = gm_north + g(i,ny-1,k)
             gm_south = gm_south + g(i,1,k)
          end do
          g_north  = g_north / float(nz + 1)
          g_south  = g_south / float(nz + 1)
          gm_north = gm_north / float(nz + 1)
          gm_south = gm_south / float(nz + 1)

*
*        ... now the 'theta' derivatives
*

          do k = 0, nz
             gy(0,k) = (- 3. * g(i,0,k)
     &                 + 4. * g(i,1,k)
     &                      - g(i,2,k)
     &                ) / (2. * dy)
             gyy(0,k) = (2. * g(i,0,k)
     &                - 5. * g(i,1,k)
     &                + 4. * g(i,2,k)
     &                     - g(i,3,k)
     &                 ) / (dy * dy)
             do j = 1, ny - 1
                gy(j,k) = (g(i,j+1,k)
     &                  - g(i,j-1,k)
     &                   ) / (2. * dy)
                gyy(j,k) = (g(i,j+1,k)
     &              - 2. * g(i,j,k)
     &                   + g(i,j-1,k)
     &                    ) / (dy * dy)
             end do
             gy(ny,k) = (3. * g(i,ny,k)
     &                - 4. * g(i,ny-1,k)
     &                     + g(i,ny-2,k)
     &                 ) / (2. * dy)
             gyy(ny,k) = (2. * g(i,ny,k)
     &                 - 5. * g(i,ny-1,k)
     &                 + 4. * g(i,ny-2,k)
```

```
     &                       - g(i,ny-3,k)
     &                  ) / (dy * dy)
          end do

*
*        ... assemble the L^2 operator ...
*

          do k = 0, nz
             l2(0,k) = - 4. * (gm_south - g_south) / (dz * dz)
             do j = 1, ny - 1
                l2(j,k) = - ((1. - y(j) ** 2) * gyy(j,k)
     &                       - 2. * y(j) * gy(j,k)
     &                       + gzz(j,k) / (1. - y(j) * y(j)))
             end do
             l2(ny,k) = - 4. * (gm_north - g_north) / (dz * dz)
          end do

      return
      end
      subroutine getparam(ubegin, ufinal, du, iterskip, islice,
     &                   isliceno)
      real ubegin, ufinal, du
      integer iterskip
      integer islice, isliceno

      write (6,*) 'u initial: 0'
      ubegin = 0
      write (6,*) 'u final: 1'
      ufinal = 1
      write (6,*) 'estimated iters: ', int(ufinal / du)
c      write (*,'(a$)') 'iterskip: '
c      read (*,*) iterskip
c      write (*,'(a$)') 'slicing choice [1=x, 2=y, 3=z]: '
c      read (*,*) islice
c      write (*,'(a$)') 'slice #: '
c      read (*,*) isliceno


c      write (*,'(a$)') 'u initial: '
c      read (*,*) ubegin
c      write (*,'(a$)') 'u final: '
```

```
c        read (*,*) ufinal
c        write (*,'(a,i4)') 'estimated iters: ', int(ufinal / du)
c        write (*,'(a$)') 'iterskip: '
c        read (*,*) iterskip
c        write (*,'(a$)') 'slicing choice [1=x, 2=y, 3=z]: '
c        read (*,*) islice
c        write (*,'(a$)') 'slice #: '
c        read (*,*) isliceno

      return
      end
```

B Pittsburgh Code: HPF

```
      integer nx, ny, nz
      parameter (nx = 128, ny = 64, nz = 64)
c      parameter (nx = 8, ny = 8, nz = 8)
      real x(0:nx), y(0:ny), z(0:nz)
      real gn(0:nx,0:ny,0:nz), go(0:nx,0:ny,0:nz)
      real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz)
      real gx(0:ny,0:nz), gxx(0:ny,0:nz)
      real l2n(0:ny,0:nz), l2o(0:ny,0:nz)

*     dx, dy, dz:                 grid spacings
*     u, uf, du:                  initial time, final time, time step
*     iter, iterskip:             iteration counter, no. of iter between dumps
*     islice, isliceno:           selects x, y or z slicing, slice number

      real dx, dy, dz
      real u, uf, du
      integer iter, iterskip
      integer islice, isliceno

!HPF$ PROCESSORS PPP(2,2)
!HPF$ DISTRIBUTE GN(*,BLOCK,BLOCK) ONTO PPP
!HPF$ ALIGN (:,:,:) WITH GN(:,:,:) :: go
!HPF$ ALIGN (:,:)   WITH GN(*,:,:) :: gy, gyy, gzz, l2n, l2o, gx, gxx
!HPF$ ALIGN y(:) WITH GN(*,:,*)
!HPF$ ALIGN z(:) WITH GN(*,*,:)

      INTERFACE
      subroutine mkl2(g, gy, gyy, gzz, l2, y, nx, ny, nz, dy, dz, i)
c     ----------------------------------------------------------
      integer nx, ny, nz
      real g(0:nx,0:ny,0:nz)
      real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz), l2(0:ny,0:nz)
      real y(0:ny)
      real dy, dz
      integer i


!HPF$ PROCESSORS PPP(2,2)
!HPF$ DISTRIBUTE G *(*,BLOCK,BLOCK) ONTO *PPP
!HPF$ ALIGN (:,:)   WITH *G(*,:,:) :: gy, gyy, gzz, l2
```

```
!HPF$ ALIGN y(:) WITH *G(*,:,*)
      end subroutine mkl2
      END INTERFACE


ccc       open (unit = 11, file = 'plot.gnu', status = 'unknown')
c       ----------------------------------------------------------------
c        call grid(x, y, z, nx, ny, nz, dx, dy, dz, du)
c       -----------------------------------------------------------------
        pi = 3.14159265358979323846

        dx = 0.5 / float(nx)
        dy = 2. / float(ny)
        dz = 2. * pi / float(nz + 1)

        du = min(dx,dy,dz) / 2.

        FORALL(i=0:nx) x(i) = 0.5 + i * dx
        FORALL(j=0:ny) y(j) = -1. + j * dy
        FORALL(k=0:nz) z(k) = k * dz

c       ------------------------------------------------------------------
c       call getparam(u, uf, du, iterskip, islice, isliceno)
c       ------------------------------------------------------------------
        write (6,*) 'u initial: 0'
        u = 0
        write (6,*) 'u final: 1'
        uf = 1
        write (6,*) 'estimated iters: ', int(uf / du)

c       ------------------------------------------------------------------
c        call surfdata(gn, x, y, z, nx, ny, nz, u)
c       ------------------------------------------------------------------
        x1 = 0.6
        x2 = 0.9

        gn = 0
        FORALL(i=0:nx, j=0:ny, k=0:nz, (x(i).gt.x1) .and. (x(i).lt.x2))
     &            gn(i,j,k) = ((x(i) - x1) * (x2 - x(i))) ** 2
     &                    * (1. - y(j) ** 2)
     &                    * cos(2. * z(k))
```

```
c      -----------------------------------------------------------------
ccc       call wslice(gn, x, y, z, nx, ny, nz, islice, isliceno, u)
c      -----------------------------------------------------------------

       iter = 0
       do while (u .lt. uf)
          iter = iter + 1
          u = u + du
c      -----------------------------------------------------------------
c           call copy(gn, go, nx, ny, nz)
c      -----------------------------------------------------------------
           FORALL(i=0:nx, j=0:ny, k=0:nz) go(i,j,k) = gn(i,j,k)
c      -----------------------------------------------------------------
c           call boundary(gn, go, nx, ny, nz, dx, dy, dz, du, u)
c      -----------------------------------------------------------------
           FORALL(j=0:ny,k=0:nz) gn(0,j,k) = 0.

c      -----------------------------------------------------------------
c           call advance(gn, go, gy, gyy, gzz, l2o, l2n, x, y, z,
c     &                    nx, ny, nz, dx, dy, dz, du)
c      -----------------------------------------------------------------
c      ..........................................................
*
*     ... at the inner boundary
*         evaluate the L^2 operator acting on g ...
*         on the new and old levels ...
*
          call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, 1)
          call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, 0)
*
*     ... and insert into a modified form of the evolution algorithm ...
*
       xh = x(0) + 0.5 * dx

       FORALL(j=0:ny, k=0:nz) gx(j,k) = (go(1,j,k) - go(0,j,k)) / dx
       FORALL(j=0:ny, k=0:nz) gxx(j,k) = (3. * go(0,j,k)
     &           - 7. * go(1,j,k)
     &           + 5. * go(2,j,k)
     &               -  go(3,j,k)
     &            ) / (2. * dx * dx)

       FORALL(j=0:ny, k=0:nz) gn(1,j,k) = gn(0,j,k)
```

```
     &       + go(1,j,k)
     &         - go(0,j,k)
     &        + ((1. - xh) ** 2 * gxx(j,k) - 2. * (1. - xh) * gx(j,k)
     &                   - 0.5 * (l2o(j,k) + l2n(j,k)) / (xh * xh)
     &                  ) * (0.5 * du * dx)
*
*     ... over the bulk of the grid ...
*
c       stop
      do i = 2, nx - 1

         alpha = (1. - (x(i) - 0.5 * dx)) / dx
         beta  = 4. / (du * dx)
         gamma = - 2. / (x(i) - 0.5 * dx) ** 2

*
*        ... evaluate the L^2 operator acting on g
*            on the new and old levels ...
*
         call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, i)
         call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, i-1)

*
*        ... and insert into the evolution algorithm ...
*

        FORALL(j=0:ny, k=0:nz)
     &         gn(i,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &        + (beta - 2 * alpha * alpha)      * gn(i-1,j,k)
     &        + alpha * (alpha + 1.)            * gn(i-2,j,k)
     &        + alpha * (alpha - 1.)            * go(i+1,j,k)
     &        + (beta - 2 * alpha * alpha)      * go(i,j,k)
     &        + (- beta + alpha * (alpha + 1.)) * go(i-1,j,k)
     &        ) / (beta - alpha * (alpha - 1.))

      end do
*
*     ... at the outer boundary
*         evaluate the L^2 operator acting on g
*         on the old level ...
*
      call mkl2(go, gy, gyy, gzz, l2o, y, nx, ny, nz, dy, dz, nx)
```

```
      call mkl2(gn, gy, gyy, gzz, l2n, y, nx, ny, nz, dy, dz, nx-1)

*
*     ... and insert into a modified form of the evolution algorithm ...
*
        FORALL(j=0:ny, k=0:nz)
     &         gn(nx,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &        + (beta - 2 * alpha * alpha)       * gn(nx-1,j,k)
     &        + alpha * (alpha + 1.)             * gn(nx-2,j,k)
     &        + (beta + alpha * (2. * alpha - 3.)) * go(nx,j,k)
     &        + (-beta - alpha * (5. * alpha - 4.)) * go(nx-1,j,k)
     &        + alpha * (4. * alpha - 1.) * go(nx-2,j,k)
     &        + (-alpha * alpha) * go(nx-3,j,k)
     &        ) / (beta - alpha * (alpha - 1.))


ccc         if (mod(iter,iterskip) .eq. 0)
ccc    &      call wslice(gn, x, y, z, nx, ny, nz, islice, isliceno, u)

      end do

ccc      close (unit = 11)

      write(6,*) "gn(0,      0,   0)=", gn(0,0,0)
      write(6,*) "gn(1,      1,   1)=", gn(1,1,1)
      write(6,*) "gn(2,      2,   2)=", gn(2,2,2)
      write(6,*) "gn(1,   ny-5,   2)=", gn(1,ny-5,2)
      write(6,*) "gn(nx-2,   0,   0)=", gn(nx-2,0,0)
      write(6,*) "gn(nx-2,ny-5,nz-5)=", gn(nx-2,ny-5,nz-5)
      write(6,*) "gn(nx,  ny-1,nz-1)=", gn(nx,ny-1,nz-1)


      end




c     ----------------------------------------------------------
      subroutine mkl2(g, gy, gyy, gzz, l2, y, nx, ny, nz, dy, dz, i)
c     ----------------------------------------------------------
      integer nx, ny, nz
      real g(0:nx,0:ny,0:nz)
```

```
      real gy(0:ny,0:nz), gyy(0:ny,0:nz), gzz(0:ny,0:nz), l2(0:ny,0:nz)
      real y(0:ny)
      real dy, dz
      integer i


!HPF$ PROCESSORS PPP(2,2)
!HPF$ DISTRIBUTE G *(*,BLOCK,BLOCK) ONTO *PPP
!HPF$ ALIGN (:,:)   WITH *G(*,:,:) :: gy, gyy, gzz, l2
!HPF$ ALIGN y(:) WITH *G(*,:,*)

      integer j, k
      real g_north,  g_south
      real gm_north, gm_south

c     ..........................................................
*
*       calculate the 'phi' derivatives
*
      FORALL(j=0:ny) gzz(j,0) = (g(i,j,1)
     &          - 2. * g(i,j,0)
     &               + g(i,j,nz)
     &                 ) / (dz * dz)

      FORALL(j=0:ny, k=1:nz-1) gzz(j,k) = (g(i,j,k+1)
     &             - 2. * g(i,j,k)
     &                  + g(i,j,k-1)
     &                    ) / (dz * dz)
      FORALL(j=0:ny) gzz(j,nz) = (g(i,j,0)
     &           - 2. * g(i,j,nz)
     &                + g(i,j,nz-1)
     &                 ) / (dz * dz)

*
*       ... estimate the average value of g on a small circle around
*           the poles
*
         g_north  = SUM(g(i, ny,   0:nz)) / float(nz + 1)
         g_south  = SUM(g(i, 0,    0:nz)) / float(nz + 1)
         gm_north = SUM(g(i, ny-1, 0:nz)) / float(nz + 1)
         gm_south = SUM(g(i, 1,    0:nz)) / float(nz + 1)
```

```
*
*        ... now the 'theta' derivatives
*


         FORALL(k=0:nz) gy(0,k) = (- 3. * g(i,0,k)
     &                 + 4. * g(i,1,k)
     &                      - g(i,2,k)
     &                ) / (2. * dy)

         FORALL(k=0:nz) gyy(0,k) = (2. * g(i,0,k)
     &                - 5. * g(i,1,k)
     &                + 4. * g(i,2,k)
     &                     - g(i,3,k)
     &                 ) / (dy * dy)

         FORALL(j=1:ny-1, k=0:nz)    gy(j,k) = (g(i,j+1,k)
     &                  - g(i,j-1,k)
     &                   ) / (2. * dy)
         FORALL(j=1:ny-1, k=0:nz)    gyy(j,k) = (g(i,j+1,k)
     &              - 2. * g(i,j,k)
     &                   + g(i,j-1,k)
     &                    ) / (dy * dy)

         FORALL(k=0:nz) gy(ny,k) = (3. * g(i,ny,k)
     &                - 4. * g(i,ny-1,k)
     &                     + g(i,ny-2,k)
     &                 ) / (2. * dy)
         FORALL(k=0:nz) gyy(ny,k) = (2. * g(i,ny,k)
     &                 - 5. * g(i,ny-1,k)
     &                 + 4. * g(i,ny-2,k)
     &                      - g(i,ny-3,k)
     &                 ) / (dy * dy)


*
*        ... assemble the L^2 operator ...
*

         FORALL(k=0:nz)
     &       l2(0,k) = - 4. * (gm_south - g_south) / (dz * dz)
```

```
      FORALL(j=1: ny-1, k=0:nz)
&            l2(j,k) = - ((1. - y(j) ** 2) * gyy(j,k)
&                        - 2. * y(j) * gy(j,k)
&                        + gzz(j,k) / (1. - y(j) * y(j)))

      FORALL(k=0:nz)
&         l2(ny,k) = - 4. * (gm_north - g_north) / (dz * dz)


   return
   end
```

C Pittsburgh Code: Fortran 90

```
 module bh02

 type COORDS
 integer                                :: n_coord
 real, pointer, dimension(:)            :: coord
 end type COORDS


 type GF3
 character (len=64), pointer            :: name
 integer, pointer, dimension(:)         :: ddim
 real, pointer, dimension(:,:,:)        :: data
 type  (COORDS), pointer, dimension(:) :: c
 end type GF3


 type GF3T
 integer                                :: n_t
 real, pointer, dimension(:)            :: t
 type (GF3), pointer, dimension (:)     :: tlev
 end type GF3T

 type GF3TL
 character (len=256), pointer           :: name
 integer                                :: n_gfcn
 type (GF3T), pointer, dimension (:)    :: gfcn
 end type GF3TL

 contains

 subroutine allocate_grid_uniform(struct,struct_name,n_gfcn,
& ntlev,gnames,ndim,nx,ny,nz)
 type(GF3TL), pointer ::  struct
 character(256), target :: struct_name
 integer, dimension(:) :: ntlev
 character(64), dimension(:), target :: gnames
 integer n_gfcn, ndim, nx,ny,nz


  allocate(struct)
```

```
        struct % name => struct_name
        struct % n_gfcn = n_gfcn
        allocate(struct % gfcn(n_gfcn))

        do i = 1, struct % n_gfcn
          struct % gfcn(i) % n_t = ntlev(i)
          allocate(struct % gfcn(i) % t(ntlev(i)))
          allocate(struct % gfcn(i) % tlev(ntlev(i)))
        enddo

        do i = 1, n_gfcn
         do j= 1, struct % gfcn(i) % n_t
           struct % gfcn(i) % tlev(j) % name => gnames(i)
           allocate(struct % gfcn(i) % tlev(j) % ddim(ndim))
           struct % gfcn(i) % tlev(j) % ddim = (/nx,ny,nz/)
           allocate(struct % gfcn(i) % tlev(j) %
     +                                data(0:nx,0:ny,0:nz))
           allocate(struct % gfcn(i) % tlev(j) %
     +                                c(ndim))
        enddo
       enddo

       return
       end subroutine allocate_grid_uniform

       end module bh02

C      =================================================================

        module pittsburg
        character(256), target :: struct_name =
     &                    'structure for the Pittsburg code'
        integer, parameter :: n_gfcn = 1
        integer, dimension(n_gfcn) :: ntlev = (/2/)
        character(64), dimension(n_gfcn), target :: gnames = (/'g'/)
        integer, parameter :: ndim = 3
c        integer, parameter :: nx = 8, ny = 8, nz = 8
        integer, parameter :: nx = 128, ny = 64, nz = 64

        end module pittsburg

C       ================================================================
```

```
       program pittsburgh
       use bh02

       real dx, dy, dz
       real u, uf, du
       integer iter

       type(GF3TL), pointer                 :: wave

      interface
       subroutine timestep(WAVE,u,du,dx,dy,dz)
       use bh02
       type(GF3TL), pointer                 :: wave
       real u, du, dx, dy, dz
       end subroutine timestep

       subroutine coordinates(WAVE,dx,dy,dz,du,u,uf)
       use bh02
       type(GF3TL),pointer :: WAVE
       real dx,dy,dz,du,u,uf
       end subroutine coordinates

       subroutine grid(WAVE)
       use bh02
       type(GF3TL), pointer                 :: wave
       integer nx,ny,nz
       end subroutine grid

       subroutine initdata(gn, x, y, z, nx, ny, nz)
       real, dimension(0:,0:,0:)           :: gn
       real, dimension(0:)                 :: x,y,z
       integer nx,ny,nz
       end subroutine initdata

      end interface


c      ----------------------------------------------------------
C      DEFINE GRID
c      ----------------------------------------------------------
```

```
      call grid(WAVE)
      call coordinates(WAVE,dx,dy,dz,du,u,uf)

c     --------------------------------------------------------------------
c     INITIALIZE GRID FUNCTIONS
c     ---------------------------------------------------

      call initdata(WAVE%gfcn(1)%tlev(2)%data,
     &  WAVE%gfcn(1)%tlev(2)%c(1)%coord,
     &  WAVE%gfcn(1)%tlev(2)%c(2)%coord,
     &  WAVE%gfcn(1)%tlev(2)%c(3)%coord,
     &  WAVE%gfcn(1)%tlev(2)%c(1)%n_coord,
     &  WAVE%gfcn(1)%tlev(2)%c(2)%n_coord,
     &  WAVE%gfcn(1)%tlev(2)%c(3)%n_coord)
        WAVE % gfcn(1) % t(2) = u

      iter = 0
       do while (u .lt. uf)
          iter = iter + 1
          u = u + du
          call timestep(WAVE,u,du,dx,dy,dz)

       end do

c     the following ugly code is used to validate (naively) results

      nx =   WAVE%gfcn(1)%tlev(2)%c(1)%n_coord
      ny =   WAVE%gfcn(1)%tlev(2)%c(2)%n_coord
      nz =   WAVE%gfcn(1)%tlev(2)%c(3)%n_coord


      write(6,*) "gn(0,      0,   0)=",
     & WAVE%gfcn(1)%tlev(2)%data(0,0,0)
      write(6,*) "gn(1,      1,   1)=",
     & WAVE%gfcn(1)%tlev(2)%data(1,1,1)
      write(6,*) "gn(2,      2,   2)=",
     & WAVE%gfcn(1)%tlev(2)%data(2,2,2)
      write(6,*) "gn(1,   ny-5,   2)=",
     & WAVE%gfcn(1)%tlev(2)%data(1,ny-5,2)
      write(6,*) "gn(nx-2,   0,   0)=",
     & WAVE%gfcn(1)%tlev(2)%data(nx-2,0,0)
      write(6,*) "gn(nx-2,ny-5,nz-5)=",
```

```
     & WAVE%gfcn(1)%tlev(2)%data(nx-2,ny-5,nz-5)
       write(6,*) "gn(nx,  ny-1,nz-1)=",
     & WAVE%gfcn(1)%tlev(2)%data(nx,ny-1,nz-1)


       end




       subroutine grid(WAVE)
       use bh02
       use pittsburg
       type(GF3TL), pointer                  :: wave

       call allocate_grid_uniform(WAVE,struct_name,n_gfcn,
     & ntlev,gnames,ndim,nx,ny,nz)

       return
       end



       subroutine coordinates(WAVE,dx,dy,dz,du,u,uf)
       use bh02
       type(GF3TL),pointer :: WAVE
       real dx,dy,dz,du,u,uf

       type(COORDS)                 :: x,y,z
       integer nx,ny,nz

       nx=WAVE%gfcn(1)%tlev(2)%ddim(1)
       ny=WAVE%gfcn(1)%tlev(2)%ddim(2)
       nz=WAVE%gfcn(1)%tlev(2)%ddim(3)

c     allocate space for coordinates

      allocate(x % coord(0:nx))
      allocate(y % coord(0:ny))
      allocate(z % coord(0:nz))

c     initialize coordinates
```

```
      pi = 3.14159265358979323846

      dx = 0.5 / float(nx)
      dy = 2. / float(ny)
      dz = 2. * pi / float(nz + 1)

      FORALL(i=0:nx) x % coord(i) = 0.5 + i * dx
      FORALL(j=0:ny) y % coord(j) = -1. + j * dy
      FORALL(k=0:nz) z % coord(k) = k * dz

      x % n_coord = nx
      y % n_coord = ny
      z % n_coord = nz


      do i=1,1
       do j=1,2
        WAVE % gfcn(i) % tlev(j) % c(1) = x
        WAVE % gfcn(i) % tlev(j) % c(2) = y
        WAVE % gfcn(i) % tlev(j) % c(3) = z
       end do
      end do


      du = min(dx,dy,dz) / 2.

c     -----------------------------------------------------------------
c     call getparam(u, uf, du, iterskip, islice, isliceno)
c     -----------------------------------------------------------------
      write (6,*) 'u initial: 0'
      u = 0
      write (6,*) 'u final: 1'
      uf = 1
      write (6,*) 'estimated iters: ', int(uf / du)

      return
      end


     subroutine initdata(gn, x, y, z, nx, ny, nz)
     real, dimension(0:,0:,0:)           :: gn
```

```
      real, dimension(0:)                 :: x,y,z
      integer nx,ny,nz


       x1 = 0.6
       x2 = 0.9

       gn = 0
       FORALL(i=0:nx, j=0:ny, k=0:nz, (x(i).gt.x1) .and. (x(i).lt.x2))
     &             gn(i,j,k) = ((x(i) - x1) * (x2 - x(i))) ** 2
     &                     * (1. - y(j) ** 2)
     &                     * cos(2. * z(k))

c         WAVE % gfcn(1) % t(2) = u

       return
       end


      subroutine timestep(WAVE,u,du,dx,dy,dz)
      use bh02
      type(GF3TL), pointer                 :: wave
      real u, du,dx,dy,dz
      integer nx,ny,nz

      INTERFACE

      subroutine copy(a,b)
      real, dimension(:,:,:) :: a,b
      end subroutine copy

      subroutine advance(go,gn,dx,dy,dz,du)
      use bh02
      type(GF3) :: gn, go
      real dx,dy,dz,du
      end subroutine advance

      END INTERFACE


      nx = WAVE % gfcn(1) % tlev(2) % ddim(1)
      ny = WAVE % gfcn(1) % tlev(2) % ddim(2)
```

```
      nz = WAVE % gfcn(1) % tlev(2) % ddim(3)

      call copy(WAVE % gfcn(1) % tlev(2) % data,
     &           WAVE % gfcn(1) % tlev(1) % data)

      WAVE % gfcn(1) % t(1) = WAVE % gfcn(1) % t(1)
      WAVE % gfcn(1) % t(2) = u

c       ----------------------------------------------------------------
c           call boundary(gn, go, nx, ny, nz, dx, dy, dz, du, u)
c       ----------------------------------------------------------------
            FORALL(j=0:ny, k=0:nz)
     &       WAVE % gfcn(1) % tlev(2) % data(0,j,k) = 0.0

c       ----------------------------------------------------------------
            call advance(WAVE%gfcn(1)%tlev(1),WAVE%gfcn(1)%tlev(2),
     &      dx, dy, dz, du)
c       ----------------------------------------------------------------
      return
      end

c     ==================================================================
      subroutine copy(a,b)
      real, dimension(0:,0:,0:) :: a,b
      b=a
      return
      end

c     ==================================================================
      subroutine advance(go,gn,dx,dy,dz,du)
      use bh02
      type(GF3) :: gn, go
      real dx,dy,dz,du

      real, dimension(0:size(gn%data,2)-1,0:size(gn%data,3)-1)
     &  :: gx,gy,gxx,gyy,gzz,l2n,l2o
      real xh

       INTERFACE
       subroutine mkl2(g, gy, gyy, gzz, l2, y, nx, ny, nz, dy, dz, i)
       integer nx, ny, nz
       real g(0:,0:,0:)
```

```
      real gy(0:,0:), gyy(0:,0:), gzz(0:,0:), l2(0:,0:)
      real y(0:)
      real dy, dz
      integer i
      end subroutine mkl2
      END INTERFACE


     nx = gn % ddim(1)
     ny = gn % ddim(2)
     nz = gn % ddim(3)

     l2o = 0
     l2n = 0

     call mkl2(go%data,gy,gyy,gzz,l2o,go%c(2)%coord,nx,ny,nz,dy,dz,1)
     call mkl2(gn%data,gy,gyy,gzz,l2n,gn%c(2)%coord,nx,ny,nz,dy,dz,0)

*
*    ... and insert into a modified form of the evolution algorithm ...
*
      xh = gn % c(1) % coord(0) + 0.5 * dx


          FORALL(j=0:ny, k=0:nz)
    &       gx(j,k)  =
    &       (go % data(1,j,k)
    &      - go % data(0,j,k))/dx


          FORALL(j=0:ny, k=0:nz)
    &       gxx(j,k) =
    &       ( 3.0 * go % data(0,j,k)
    &      -  7.0 * go % data(1,j,k)
    &      +  5.0 * go % data(2,j,k)
    &      -        go % data(3,j,k)
    &       ) / (2.0 * dx * dx)

      FORALL(j=0:ny, k=0:nz) gn % data(1,j,k) = gn % data(0,j,k)
    &       + go % data(1,j,k)
    &         - go % data(0,j,k)
    &        + ((1. - xh) ** 2 * gxx(j,k) - 2. * (1. - xh) * gx(j,k)
```

```
     &                   - 0.5 * (l2o(j,k) + l2n(j,k)) / (xh * xh)
     &                  ) * (0.5 * du * dx)
*
*
*     ... over the bulk of the grid ...
*
      do i = 2, nx - 1

         alpha = (1. - (gn%c(1)%coord(i) - 0.5 * dx)) / dx
         beta  = 4. / (du * dx)
         gamma = - 2. /(gn%c(1)%coord(i) - 0.5 * dx) ** 2

*
*        ... evaluate the L^2 operator acting on g
*            on the new and old levels ...
*

      call mkl2(go%data,gy,gyy,gzz,l2o,go%c(2)%coord,nx,ny,nz,dy,dz,i)
      call mkl2(gn%data,gy,gyy,gzz,l2n,gn%c(2)%coord,nx,ny,nz,dy,dz,
     &i-1)

*
*        ... and insert into the evolution algorithm ...
*

        FORALL(j=0:ny, k=0:nz)
     &     gn%data(i,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &  + (beta - 2 * alpha * alpha)      * gn%data(i-1,j,k)
     &  + alpha * (alpha + 1.)            * gn%data(i-2,j,k)
     &  + alpha * (alpha - 1.)            * go%data(i+1,j,k)
     &  + (beta - 2 * alpha * alpha)      * go%data(i,j,k)
     &  + (- beta + alpha * (alpha + 1.)) * go%data(i-1,j,k))
     &  / (beta - alpha * (alpha - 1.))

      end do
*
*     ... at the outer boundary
*         evaluate the L^2 operator acting on g
*         on the old level ...
*
      call mkl2(go%data,gy,gyy,gzz,l2o,go%c(2)%coord,nx,ny,nz,dy,dz,nx)
      call mkl2(gn%data,gy,gyy,gzz,l2n,gn%c(2)%coord,nx,ny,nz,dy,dz,
```

```
     &nx-1)

c
*
*      ... and insert into a modified form of the evolution algorithm ...
*
         FORALL(j=0:ny, k=0:nz)
     &     gn%data(nx,j,k) = (gamma * 0.5 * (l2o(j,k) + l2n(j,k))
     &   + (beta - 2 * alpha * alpha)        * gn%data(nx-1,j,k)
     &   + alpha * (alpha + 1.)              * gn%data(nx-2,j,k)
     &   + (beta + alpha * (2. * alpha - 3.)) * go%data(nx,j,k)
     &   + (-beta - alpha * (5. * alpha - 4.)) * go%data(nx-1,j,k)
     &   + alpha * (4. * alpha - 1.) * go%data(nx-2,j,k)
     &   + (-alpha * alpha) * go%data(nx-3,j,k))
     &    / (beta - alpha * (alpha - 1.))



      return
      end



c      ----------------------------------------------------------
       subroutine mkl2(g, gy, gyy, gzz, l2, y, nx, ny, nz, dy, dz, i)
c      ----------------------------------------------------------
       integer nx, ny, nz
       real g(0:,0:,0:)
       real gy(0:,0:), gyy(0:,0:), gzz(0:,0:), l2(0:,0:)
       real y(0:)
       real dy, dz
       integer i

c      ..........................................................

*
*        calculate the 'phi' derivatives
*

c        do j=0,ny
c         write(6,*) 'j=',j
```

```
c         write(6,*) g(i,j,1),g(i,j,0),g(i,j,nz)
c         write(6,*) (g(i,j,1)
c     &           - 2. * g(i,j,0)
c     &                + g(i,j,nz)
c     &                  ) / (dz * dz)
c        enddo


        FORALL(j=0:ny) gzz(j,0) = (g(i,j,1)
     &           - 2. * g(i,j,0)
     &                + g(i,j,nz)
     &                  ) / (dz * dz)

        FORALL(j=0:ny, k=1:nz-1) gzz(j,k) = (g(i,j,k+1)
     &              - 2. * g(i,j,k)
     &                   + g(i,j,k-1)
     &                    ) / (dz * dz)
        FORALL(j=0:ny) gzz(j,nz) = (g(i,j,0)
     &            - 2. * g(i,j,nz)
     &                 + g(i,j,nz-1)
     &                  ) / (dz * dz)

*
*        ... estimate the average value of g on a small circle around
*            the poles
*
           g_north  = SUM(g(i, ny,   0:nz)) / float(nz + 1)
           g_south  = SUM(g(i, 0,    0:nz)) / float(nz + 1)
           gm_north = SUM(g(i, ny-1, 0:nz)) / float(nz + 1)
           gm_south = SUM(g(i, 1,    0:nz)) / float(nz + 1)

*
*        ... now the 'theta' derivatives
*

           FORALL(k=0:nz) gy(0,k) = (- 3. * g(i,0,k)
     &                 + 4. * g(i,1,k)
     &                      - g(i,2,k)
     &                ) / (2. * dy)

           FORALL(k=0:nz) gyy(0,k) = (2. * g(i,0,k)
     &                - 5. * g(i,1,k)
```

```
     &                 + 4. * g(i,2,k)
     &                      - g(i,3,k)
     &                  ) / (dy * dy)

          FORALL(j=1:ny-1, k=0:nz)    gy(j,k) = (g(i,j+1,k)
     &                   - g(i,j-1,k)
     &                    ) / (2. * dy)
          FORALL(j=1:ny-1, k=0:nz)    gyy(j,k) = (g(i,j+1,k)
     &               - 2. * g(i,j,k)
     &                    + g(i,j-1,k)
     &                     ) / (dy * dy)

          FORALL(k=0:nz) gy(ny,k) = (3. * g(i,ny,k)
     &                 - 4. * g(i,ny-1,k)
     &                      + g(i,ny-2,k)
     &                  ) / (2. * dy)
          FORALL(k=0:nz) gyy(ny,k) = (2. * g(i,ny,k)
     &                  - 5. * g(i,ny-1,k)
     &                  + 4. * g(i,ny-2,k)
     &                       - g(i,ny-3,k)
     &                  ) / (dy * dy)


*
*        ... assemble the L^2 operator ...
*

          FORALL(k=0:nz)
     &        l2(0,k) = - 4. * (gm_south - g_south) / (dz * dz)

          FORALL(j=1: ny-1, k=0:nz)
     &          l2(j,k) = - ((1. - y(j) ** 2) * gyy(j,k)
     &                       - 2. * y(j) * gy(j,k)
     &                       + gzz(j,k) / (1. - y(j) * y(j)))

          FORALL(k=0:nz)
     &       l2(ny,k) = - 4. * (gm_north - g_north) / (dz * dz)


       return
       end
```

References

[1] R. Gómez, R. Isaacson, and J. Winicour, J. Comp. Phys. 98, 11 (1992).

[2] L. Smarr *Basic Concepts in Finite Differencing of Parial Diferential Equations* in Sources of Gravitational Radiation, ed. Smarr, p. 139 - 159.