Gravitational Wave Extraction - A Benchmark?

1 Overview

The simplified model we propose to build intends to serve as a benchmark for the problem of gravitational wave extraction. This will be an important component of all numerical codes being developed as part of the Black Hole Binaries collaboration. We will follow the line of research of the Pittsburgh group [1, 2, 3, 4] and describe the physical problem as a characteristic initial value problem (CIVP), although different approaches have been suggested [5]. We hope that the numerical codes we develop will be generic enough that alternative approaches might be incorporated later.

In the CIVP approach, space-time is described in terms of a metric in null spherical polar coordinates, (u, r, θ, ϕ), where retarded time u is given by $u = t - r$ in flat space time, and initial data is given on a null cone (the locus of the set of points at constant retarded time $u = t - r$). Supplementary boundary conditions are required at an inner boundary. This inner boundary can be taken to be a spacelike surface, such as a sphere at constant r, or alternatively, a null surface determined by radially incoming characterics. We will concern ourselves with the first alternative in the scope of this work. The equations to be solved take the form of a hyperbolic evolution equation which determines the rate of change ∂_u of a metric field, and a hierarchy of hypersurface equations which do not involve time derivatives and can be solved in a predetermined order.

Starting from the initial data at $u = u_0$ and the conditions on the inner boundary, the evolution equation and the hypersurface equations can be integrated outwards from the origin, yielding the metric functions on a null cone at $u = u_0 + \Delta u$. This march proceeds to the edge of the computational grid, which corresponds to the set of points for which $r \to \infty$. In the terminology of general relativity this set is refered to as (future) null infinity, $\mathcal{J}^+$.

The inner boundary could be taken to be the origin of the coordinate system, however, to take advantage of the numerical codes being developed by other groups on the collaboration, we propose a different approach.

We will specify the boundary conditions in terms of the values of the metric functions and their derivatives on a timelike surface Σ. This values will be the result of the numerical evolution of a Cauchy code. For simplicity we might think of Σ as a sphere, with the initial data for the exterior CIVP given on a null cone exterior to Σ. We will come back later to the question of what coordinate and gauge transformations might be necessary to translate from the variables used in the Cauchy codes being developed to variables suitable for our code.

We propose to build this exterior CIVP code in several stages. In the first stage we will solve the scalar wave equation with source $\Box\Phi = S$ on the domain defined by the region exterior to a given surface Σ. We will assume the solution has been given on the boundary Σ. The exterior region will extend from the surface Σ to future null infinity $\mathcal{J}^+$. For simplicity, we will assume that the slicing of the surface Σ at constant retarded time has the topology of a sphere. We will use (null) spherical polar coordinates to express the equations but we

will make no assumption of symmetry, i.e. the problem will be fully 3-D.

We will implement the equations using finite differences (FDE's) on structured, rectangular meshes. The implementation of the FDE's will need to be flexible enough to allow for the eventual use of multilevel adaptive grids, therefore special care will have to be put in designing the basic data structures in which the algorithm will operate. On this first stage however, the grids will have fixed spatial spacings that will not change with time. For the time being, we will restrict our attention to explicit schemes.

On subsequent stages, the equation will be modified to allow for source terms of the type found in the linearized Einstein equations in a Bondi gauge. We know from experience with the axysmmetric linearized Einstein equations, that the stability properties of the algorithm are likely to be different from those of the scalar algorithm. Later on additional terms will be incorporated, to try to mimic the nonlinear parts of the fully nonlinear vacuum Einstein equations. Lastly, once we have worked out what the supplementary conditions are on the inner boundary (the cilinder at constant r), the boundary conditions will be incorporated. We would expect that the structure of the equations and the resulting codes will remain general enough that the codes will be useful even for different metric elements (provided they are null), and hence for different forms of the equations which other workers might use.

2 Mathematical Definition of the Model Problem

We want to solve the scalar wave equation (SWE) in null polar coordinates

$$G_{,ur} - G_{,rr} - \frac{1}{r^2}\frac{(\sin\theta\, G_{,\theta})_{,\theta}}{\sin\theta} - \frac{1}{r^2}G_{,\phi\phi} = rS(G), \tag{1}$$

where $G = r\Phi$, on the domain exterior to a surface Σ (which we take to be given by $r = r_0 = constant$). The null spherical coordinates span the range $u \geq 0$, $r \geq r_0$, $0 \leq \theta \leq \pi$ and $0 \leq \phi \leq 2\pi$. The free initial data consists of the value of G on the initial null cone at $u = constant$. The required boundary condition is the value of G on the surface defined by $r = r_0$ for all values of u. Together with the initial data they completely determine the solution. Unlike in the more common Cauchy approach, the time derivative of G is not part of the initial data, since the equation has only a first retarded time derivative.

We introduce the auxiliary coordinates $x = (r/r_0)/(1 + (r/r_0))$, $y = -\cos\theta$ and $z = \phi$. The inner boundary of the computational domain S is at $x = \frac{1}{2}$, while the outer boundary, given by the set of points for which $r \to \infty$, corresponds to the edge $x = 1$. The coordinate ranges are then $\frac{1}{2} \leq x \leq 1$, $-1 \leq y \leq 1$ and $0 \leq z \leq 2\pi$. In this coordinates, the SWE takes the form

$$G_{,ux} - [(1-x)^2\, G_{,x}]_{,x} - \frac{1}{x^2}[(1-y^2)G_{,y}]_{,y} - \frac{1}{x^2}G_{,zz} = \frac{x}{(1-x)^3}S(G). \tag{2}$$

3 Geometrical Interpretation

It is possible to write a simple evolution algorithm for scalar waves based on an integral identity which the solution must satisfy at the corners of a null parallelogram lying on the (u, r) plane [3]. The wave equation with source, $\Box\Phi = S$, can be reexpressed in the form

$$\Box^{(2)} G = -\frac{L^2 G}{r^2} + rS, \tag{3}$$

where $\Box^{(2)}$ is the 2-dimensional wave operator intrinsic to the (u, r) plane. Integration over the null parallelogram then leads to the integral equation

$$G_Q = G_P + G_S - G_R + \int_\Sigma du dr [-\frac{L^2 G}{r^2} + rS], \tag{4}$$

where P, Q, R and S are the corners and Σ the area of the parallelogram.

¿From a numerical point of view, this approach and that discussed in Sec. 4 are entirely equivalent, in that the discretization of Eq. (4) yields the same FDE's, but the point of view taken here has proven to be extremely useful in the design of numerical algorithms to solve the (vacuum) Einstein equations.

4 Numerical issues

For the purpose of discussing the stability properties of the algorithm, we look at the equation at constant coefficients

$$G_{,ux} - A\, G_{,xx} - B\, G_{,yy} - C\, G_{,zz} = 0, \tag{5}$$

which may be considered as the limit of Eq. (2) obtained by "freezing" any explicit dependence on the coordinates. (For the time being we have dropped the source terms in the RHS of the equation. We will come back to that later.)

As stated previously, we use rectangular meshes. Given a region D defined by $x_l \le x \le x_r$, $y_l \le y \le y_r$, $z_l \le z \le z_r$, the mesh associated with that region is fully specified by the value of the coordinates at the region's boundaries x_l, x_r, y_l, y_r, z_l, z_r, and the number of grid points which lie inside the region in each coordinate direction, N_x, N_y, N_z. The internal grid point themselves are given by $x_i = x_l + i\Delta x, i = 1, \ldots, N_x$ where $\Delta x = (x_r - x_l)/(N_x + 1)$, and similarly for the coordinates y and z. The time slicing is represented by the discrete levels $u^n = n\Delta u$, and we denote by $G^n_{i\,j\,k}$ the value of the field G at the point(u^n, x_i, y_j, z_k).

We can discretize Eq. (5), on points interior to a region D, to second order accuracy in the grid spacings Δu, Δx, Δy and Δz, by the following FD approximation to the derivatives:

$$\begin{aligned} G_{,ux} &= \left(G^{n+1}_{i\,j\,k} - G^{n+1}_{i-1\,j\,k} - G^{n}_{i\,j\,k} + G^{n}_{i-1\,j\,k}\right) \frac{1}{\Delta u \Delta x} \\ G_{,xx} &= \left(G^{n+1}_{i\,j\,k} - 2\,G^{n+1}_{i-1\,j\,k} + G^{n+1}_{i-2\,j\,k} + G^{n}_{i+1\,j\,k} - 2\,G^{n}_{i\,j\,k} + G^{n}_{i-1\,j\,k}\right) \frac{1}{2(\Delta x)^2} \end{aligned}$$

$$
\begin{aligned}
G_{,yy} &= \left(G^{n+1}_{i-1\,j+1\,k} - 2\,G^{n+1}_{i-1\,j\,k} + G^{n+1}_{i-1\,j-1\,k} + G^{n}_{i\,j+1\,k} - 2\,G^{n}_{i\,j\,k} + G^{n}_{i\,j-1\,k}\right)\frac{1}{2(\Delta y)^2} \\
G_{,zz} &= \left(G^{n+1}_{i-1\,j\,k+1} - 2\,G^{n+1}_{i-1\,j\,k} + G^{n+1}_{i-1\,j\,k-1} + G^{n}_{i\,j\,k+1} - 2\,G^{n}_{i\,j\,k} + G^{n}_{i\,j\,k-1}\right)\frac{1}{2(\Delta z)^2} \qquad (6)
\end{aligned}
$$

Note the angled derivative approach used in the computation of the spatial derivatives, $G_{,xx}$, $G_{,yy}$ and $G_{,zz}$. We will see later that this is essential to guarantee stability of the algorithm. We then insert Eqn. (6) in Eq. (5) and solve for $G^n_{i\,j\,k}$. Note that the relevant numerical coefficients in the resulting explicit algorithm are $a = A\Delta u/(2\Delta x)$, $b = B\Delta u\,\Delta x/\,(2(\Delta y)^2)$ and $c = C\Delta u\,\Delta x/\,(2(\Delta z)^2)$.

A von Neumann stability analysis of the algorithm can be carried out by introducing the Fourier modes $G^n_{k\,l\,m} = \xi^n e^{ikx} e^{ily} e^{imz}$ (with real k, l and m) in Eq. (6), which yields:

$$
\begin{aligned}
\Delta u\Delta x\, G_{,ux} &= [\xi(1-\omega) - (1-\omega)]G^n_{k\,l\,m} \\
A\Delta u\Delta x\, G_{,xx} &= [\xi a f\omega + af]G^n_{k\,l\,m} \\
B\Delta u\Delta x\, G_{,yy} &= [\xi bg\omega + bg]G^n_{k\,l\,m} \\
C\Delta u\Delta x\, G_{,zz} &= [\xi ch\omega + ch]G^n_{k\,l\,m}
\end{aligned} \qquad (7)
$$

where $f = e^{ikx} - 2 + e^{-ikx}$, $g = e^{ily} - 2 + e^{-ily}$ and $g = e^{imz} - 2 + e^{-imz}$, with $(f, g, h) \in \mathcal{R}$ and $\omega = e^{-ikx}$, with $\omega \in \mathcal{Z}$, $|\,\omega| = 1$. The amplification factor ξ is given by

$$
\xi = -\omega\frac{1 - (R+1)\bar{\omega}}{1 - (R+1)\omega} \qquad (8)
$$

where $R = af + bg + ch \in \mathcal{R}$, which clearly indicates $|\xi| = 1$. This result depends strongly on the slanted averages used in the discretization of the spatial derivatives (see Eq. (6)). It is not clear that any other explicit discretization would have produced a stable (and unimodular) algorithm. This approach will be taken again when source terms are added to the equation. As with all applications of the Von Neuman stability analysis, the results we obtain are rigurously valid only in the limit of constant coefficients and with the additional assumption of periodic boundary conditions. Experience indicates, however, that when an algorithm satisfies the Von Neuman stability criteria, it is numerically stable even when these conditions are relaxed. The analysis we just outlined can be carried out even if in Eqn. (5) A, B and C are smoothly varying functions of the coordinates (u, x, y, z) and the first derivatives $G_{,x}$, $G_{,y}$ and $G_{,z}$, i.e. for a quasi-linear second order equation.

The algorithm presented is fully explicit for interior points, and it leads naturally to a marching algorithm: given the value of G at time $u = (n+1)\Delta u$, for all points at $x = i\Delta x$, it can be used to advance the solution to the set of points at $x = (i+1)\Delta x$. This procedure can be started near the inner boundary, given the values of G at $x = \frac{1}{2} + \Delta x$, continuing the outward sweep until the points just before the outer boundary have been reached. It can not be applied to the points at $x = \frac{1}{2} + \Delta x$ since the algorithm becomes implicit at the inner boundary, hence a modified form must be used there. It is necessary to use a modified form of the algorithm at the outer boundary as well.

Our approach differs from the ADE method of Roberts and Weiss for the inviscid transport equation described in [6]. In their case, they considered a Cauchy problem, and the technique was to march in one direction, reach one of the boundaries of the grid, and then reverse the slant of the angled derivatives and the direction of the sweep in the next time iteration.

This is not possible in the CIVP formulation, since the boundary conditions are specified only at the inner boundary. The outer boundary must, in a sense, be considered as a free boundary. It is one striking feature of the CIVP that the asymptotic form of equations, when written in terms of compactified coordinates, provides the correct boundary behaviour and it does not require the introduction of artificial outgoing wave conditions.

Note also that we foliate the domain D with characteristic surfaces, and the physical speed of propagation of the signal is infinite on this surfaces, i.e. any signal which reaches the inner boundary registers instantly at the outer boundary. This behaviour is modeled correctly by a marching algorithm of the type described. This is another difference with the ADE method mentioned, since theirs was a Cauchy problem and the data was given on surfaces at constant time t, and therefore the infinite speed of propagation that their algorithm introduced was a numerical artifact.

5 Current State of Development

The code described in Sec. 6 implements the approach outlined earlier for Eq. 2. We have not included the source terms in the RHS of the equation at this point, neither have we built in the machinery to specify the boundary conditions (at present the boundary condition is simply $G(r = 1) = 0$). It would be trivial to specify data at $r = 1$. The code was built mainly as a proof of concept, and little effort has been spent to this point in optimization. Some trivial changes are possible, e.g. reordering the indices of the 3-D arrays so that the innermost loops stride is along the first array index.

Timing of the code has been minimal too, as it is still in a state of flux. To give some idea of the CPU requirements, a run on a grid of 128 radial points times 64x64 angular points from $u = 0$ to $u = 1$ takes about 20 minutes on a Sparc2-type workstation. For gravitational wave extraction, it is likely that we will need to use grids of the order of $(256)^3$ to $(1024)^3$, while the complexity of the calculations involved in the RHS of the equation will probably increase the time requirements by a factor of ten, judging from what we have seen when comparing the case of the axysimmetric SWE to the axysimmetric vacuum Einstein equations [7]. We implemented the code in standard Fortran-77 since it was what we have readily available.

For completeness, the code produces output in the form of 2-D slices of the evolved function $G(u, x, y, z)$, such that with the appropiate choice of input parameters (explained in the main routine) one can look at the radiation field G at null infinity. The output is formatted so that a poor-man's version of a movie can be obtained by looking at the sequence of graphs generated with the public domain package gnuplot [8]. A suitable command file is generated automatically. This has been useful at the development state, but more sofisticated

graphics will be required to explore 3-D datasets.

As far as documentation, at present it consists only of this writeup and the references here. The code could be better commented, but work on that will have to wait until some issues of coding practice have been settled, i.e. use of commons for argument passing, etc. Comments and criticisms are most welcome.

References

[1] R. Isaacson, J. Welling, and J. Winicour, J. Math. Phys. **24**, 1824 (1983).

[2] R. Isaacson, J. Welling, and J. Winicour, J. Math. Phys. **26**, 2859 (1985).

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

[4] R. Gómez and J. Winicour, Phys. Rev. D **45**, 2776 (1992).

[5] A. Abrahams and C. R. Evans, Phys. Rev. D **37**, 318 (1988).

[6] P. J. Roache, *Computational Fluid Dynamics* (Hermosa Publishers, Albuquerque, 1985), Chap. III, p. 98.

[7] R. Gómez, J. Winicour, and P. Papadopoulos, in preparation.

[8] Anonymous FTP to ftp.dartmouth.edu [129.170.16.4], directory pub/gnuplot.

6 Code Listing

```
*     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 = 32, nz = 32)
      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

      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)

      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)

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

 end do

 close (unit = 11)

 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(nx/2,j,k) = 0.
         end do
      end do
```

```
      return
      end
      subroutine wslice(g, x, y, z, nx, ny, nz, islice, isliceno, u)
      integer nx, ny, nz, islice, isliceno
      real x(0:nx), y(0:ny), z(0:nz)
      real g(0:nx,0:ny,0:nz)
      real u

      integer i, j, k
      character filename*14

      filename(1:2) = 'u='
      write (filename(3:14),'(e12.6)') u
      open (unit = 10, file = filename, status = 'unknown')

      if (islice .eq. 1) then
         i = isliceno
         do k = 0, nz
            write (10,120) (y(j), z(k), g(i,j,k), j = 0, ny)
            write (10,130)
         end do
      else if (islice .eq. 2) then
         j = isliceno
         do k = 0, nz
            write (10,120) (x(i), z(k), g(i,j,k), i = 0, nx)
            write (10,130)
         end do
      else if (islice .eq. 3) then
         k = isliceno
         do j = 0, ny
            write (10,120) (x(i), y(j), g(i,j,k), i = 0, nx)
            write (10,130)
         end do
      end if

      close (unit = 10)

      write (unit = 11, 150) 'splot "' // filename // '"'

120   format(3(5x,e12.5))
130   format('')
150   format(a)
```

```
      return
      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+1,k)
     &             - 2. * g(i,j,k)
     &                 + g(i,j-1,k)
     &                  ) / (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 (*,'(a$)') 'u initial: '
 read (*,*) ubegin
 write (*,'(a$)') 'u final: '
 read (*,*) ufinal
 write (*,'(a,i4)') 'estimated iters: ', int(ufinal / du)
 write (*,'(a$)') 'iterskip: '
 read (*,*) iterskip
 write (*,'(a$)') 'slicing choice [1=x, 2=y, 3=z]: '
 read (*,*) islice
 write (*,'(a$)') 'slice #: '
 read (*,*) isliceno

 return
 end
```