Stereographic Coordinate Patches

R. Gómez, P. Papadopoulos and J. Winicour
Department of Physics and Astronomy, University of Pittsburgh,
Pittsburgh, PA 15260

Abstract

We present a method for dealing with the computational problems which arise in the use of spherical coordinates. The method employs two stereographic coordinate patches, with interpolations between the patches in the regions of overlap. It provides a new and effective computational tool for dealing with a wide variety of systems in which spherical coordinates are natural, such as the generation of radiation from an isolated source. We show how this formalism can be applied to the evolution of waves in three spatial dimensions.

Typeset using REVTEX

I. INTRODUCTION

The increase in computing power is making it feasible to tackle many formerly intractable 3-dimensional systems. The motivation for the present work is the problem of calculating the gravitational radiation from inspiral and merger of a binary black hole system. This is anticipated to be a prime source of gravitational radiation for future LIGO detectors. The numerical solution of the problem is the goal of the general relativity Grand Challenge. Spherical coordinates and spherical harmonics are standard analytic tools in the description of radiation [1]. The eth formalism [2,3] and the associated spin-weighted spherical harmonics [3,4] allow a simple and unified extension of these analytic techniques to vector and tensor fields, without the undue complexity of traditional vector and tensor harmonics. In computational work, spherical coordinates have mainly been used in axisymmetric systems, where the polar singularities may be regularized by standard tricks. In the absence of symmetry, these techniques do not easily generalize and they would be especially prohibitive to develop for tensor fields. Here we describe an alternative, more systematic approach based on stereographic coordinate patches.

Although our motivation stems from gravitational physics, these techniques are applicable to a wide class of 3-dimensional systems. As an example, consider a fluid flow described by the velocity vector field v. In a problem with spherical boundary conditions, it is useful to introduce spherical coordinates (r, θ, ϕ) and decompose v in terms of its components (v_r, v_θ, v_ϕ) with respect to a basis of unit vectors along the coordinate axes. For a fixed value of r, the component v_r is a scalar field on the sphere, whereas the components v_θ and v_ϕ combine to form a 2-dimensional vector field on the sphere. Similarly, a description of the shear of the flow leads to a tensor field on the sphere.

Most of the ideas underlying our work are straightforward. However, at each stage of development there are several options which at first sight may seem equally viable but entail vastly different levels of difficulty in implementation. In Sec. II, we discuss how traditional regularization methods for computational use of spherical coordinates in axisymmetric problems may be extended to problems without symmetry. This is for comparison with the computational ease of application of the stereographic coordinates approach developed in Sec. III. Although our presentation is on the algorithmic level, we present enough detail here and in the Appendices to direct a well defined transition from algorithm to code development. In Sec. V, we present an application of the formalism which demonstrate both its accuracy and usefulness.

II. REGULARIZATION OF POLAR COORDINATE SINGULARITIES

In order to illustrate the typical computational problems which arise in the use of spherical coordinates we begin with a discussion of scalar fields. In standard spherical coordinates, the 2-dimensional Laplacian operating upon a scalar field takes the form

$$D^2\Psi = \frac{\partial_\theta(\sin\theta\partial_\theta\Psi)}{\sin\theta} + \frac{\partial_\phi^2\Psi}{\sin^2\theta}. \tag{2.1}$$

It is clear that the term containing the ϕ-derivatives is singular at the poles where $\sin\theta = 0$. Unless the ϕ-derivatives vanish, as in the special case of axisymmetry, this implies that θ-

derivative term must also be singular. In order to obtain a regular finite difference expression, either the two terms must be combined and rewritten in manifestly nonsingular form or additional coordinate patches must be introduced to cover the poles.

First we consider regularization in terms of a single polar coordinate system. The basic strategy is that (2.1) is adequate for constructing a finite difference approximation to the Laplacian at grid points distinct from the poles. In order to construct a finite difference approximation at the poles, we use the 2-dimensional version of the divergence theorem to recast (2.1) in integral form. To be specific, at the north pole where $\theta = 0$,

$$\int_A D^2\Psi \sin\theta d\theta d\phi = \oint_C \sin\theta \partial_\theta \Psi d\phi, \tag{2.2}$$

where A is the disc lying between 0 and $\Delta\theta$ and C is the bounding circle. Then, by centering the area integral, we have

$$D^2\Psi(0) = \frac{1}{\pi\Delta\theta^2} \oint_C \sin\theta \partial_\theta \Psi d\phi + O(\Delta\theta^2). \tag{2.3}$$

Finally, by expanding Ψ about the pole in the line integral and noting that the only terms which survive the angular integration are harmonics with $m = 0$ that may be Taylor expanded in terms of $\cos\theta$, we obtain

$$D^2\Psi(0) = (1/\pi) \oint_C \partial_\theta^2 \Psi(0) d\phi + O(\Delta\theta^2), \tag{2.4}$$

where the integral represents averaging over directions ϕ.

This last equation may be readily discretized to give a second order accurate finite difference representation of $D^2\Psi(0)$. This can then be combined with the standard treatment of nonpolar points to give a second order approximation for the Laplacian over the entire sphere. However, this does not guarantee that the second order error will be smooth at the pole. This would require forcing Ψ to have the proper behavior near the poles to ensure that singular terms in (2.1) do not magnify the error near the poles. Otherwise, in practical applications, the second order accuracy would be lost after repeated calculations.

We now present a detailed implementation of (2.1) in terms of finite differences. Let the grid consist of the points on the sphere given by $(\theta_j, \phi_k) = (j\Delta\theta, k\Delta\phi)$, where $\Delta\theta = \pi/N_\theta$, $\Delta\phi = 2\pi/N_\phi$ and the indices span the range $j = 0, \ldots, N_\theta$ and $k = 0, \ldots, N_\phi$. Values of the field (and its derivatives) at grid points are denoted by subscripts, e.g. $\Psi_{j,k} = \Psi(\theta_j, \phi_k)$. We note that (2.1) can be expressed as

$$D^2\Psi = \partial_\theta^2 \Psi + \cot\theta \partial_\theta \Psi + \frac{\partial_\phi^2 \Psi}{\sin^2\theta}. \tag{2.5}$$

The terms in this equation can be immediately discretized using centered finite differences in the (θ_j, ϕ_k) grid. For instance, the second ϕ derivative can be approximated as

$$\partial_\phi^2 \Psi_{j,k} = (\Psi_{j,k+1} - 2\Psi_{j,k} + \Psi_{j,k-1}) \frac{1}{(\Delta\phi)^2} \tag{2.6}$$

to second order accuracy in $\Delta\phi$. Numerical analysis reveals, however, that the resulting finite difference approximation does not converge uniformly to second order in $\Delta\phi$ and $\Delta\theta$

for Y_{lm} with $m = 1$. Especially troublesome is the behavior of the error for the lower order spherical harmonics, $l = 1, 2$. Problems arise near the poles ($\theta = 0, \pi$) where $1/\sin^2\theta$ behaves as $1/(\Delta\theta)^2$. For grids with a comparable number of zones in the θ and ϕ direction, this introduces an $O(\Delta\phi)$ error in the last term of Eq.(2.5) near the poles for a Y_{11}. A similar problem arises with the next to last term of Eq.(2.5). The remedy is to difference the fields not with respect to the variables (θ, ϕ) but with respect to $(e^{i\theta}, e^{i\phi})$, respectively. We give here the expressions for the partial derivatives that enter (2.5):

$$\begin{aligned}
\partial_\theta \Psi_{j,k} &= (\Psi_{j+1,k} - \Psi_{j-1,k}) \frac{1}{2\sin(\Delta\theta)} \\
\partial_\theta^2 \Psi_{j,k} &= (\Psi_{j+1,k} - 2\Psi_{j,k} + \Psi_{j-1,k}) \frac{1}{4\sin^2(\Delta\theta/2)} \\
\partial_\phi^2 \Psi_{j,k} &= (\Psi_{j,k+1} - 2\Psi_{j,k} + \Psi_{j,k-1}) \frac{1}{4\sin^2(\Delta\phi/2)}.
\end{aligned} \tag{2.7}$$

From our earlier discussion leading to Eq. (2.4), at the poles the Laplacian has the finite difference approximation

$$\begin{aligned}
D^2 \Psi_{0,k} &= \frac{2\Delta\phi}{\pi(\Delta\theta)^2} \sum_{k=0}^{N_\phi} (\Psi_{1,k} - \Psi_{0,k}) \\
D^2 \Psi_{N_\theta,k} &= \frac{2\Delta\phi}{\pi(\Delta\theta)^2} \sum_{k=0}^{N_\phi} (\Psi_{N_\theta-1,k} - \Psi_{N_\theta,k}).
\end{aligned} \tag{2.8}$$

The resulting finite difference approximation of the Laplacian is exact (to machine precision) when applied to spherical harmonics with $l \leq 2$, and converges globally to second order accuracy for higher values of l. Even so, this approach would be cumbersome in a complicated problem involving vector and tensor fields. Furthermore, it can lead to a highly restrictive time step condition when used in the evolution of a hyperbolic system. (See the discussion in regard to Eq.(5.7)).

III. STEREOGRAPHIC COORDINATE PATCHES

Here we present an alternative finite difference treatment of angular derivatives on the sphere. It provides an efficient and accurate approach which, unlike the regularization method, generalizes in a natural way to derivatives of tensor fields. The approach is based upon the standard method of describing the global differentiability of functions by means of local coordinate patches, with smooth transition in their common overlap region.

Let θ and ϕ label the points on the sphere. Then the real and imaginary parts of the stereographic coordinate

$$\zeta_N = \left(\frac{1-\cos\theta}{1+\cos\theta}\right)^{\frac{1}{2}} e^{i\phi} \tag{3.1}$$

provide a smooth coordinatization of the sphere excluding the point $\theta = \pi$ at the south pole. Similarly $\zeta_S = 1/\zeta_N$ provides a smooth coordinatization except at the south pole.

A smooth scalar field Ψ on the sphere may be represented in terms of smooth functions Ψ_S of ζ_S on the lower hemisphere $|\zeta_S| \leq 1$ and as smooth functions Ψ_N of ζ_N on the upper hemisphere $|\zeta_N| \leq 1$. The continuity of Ψ is ensured by requiring $\Psi_S(\zeta_S) = \Psi_N(\zeta_N)$ at the equator but in order to ensure smoothness this equality must extend to a common overlap region about the equator. Our first task is to implement this in a finite difference treatment.

In order to construct the grid, let (S^1, S^2) and (N^1, N^2) be the real and imaginary parts of ζ_S and ζ_N, respectively. Let the grid consist of the points $S^i = s_i \Delta$ and $N^i = n_i \Delta$, where $\Delta = 1/M$, and s_i and n_i are each pairs of integers with $-M \leq s_i \leq M$ and $-M \leq n_i \leq M$. Thus the grid points at the intersection of the axes with the equator correspond to $s_i = (M, 0)$ and $s_i = (0, M)$; and $n_i = (M, 0)$ and $n_i = (0, M)$. Figure 1 shows the N grid over the unit sphere, discretized with $M = 16$. The grid points at the poles correspond to $s_i = (0, 0)$ and $n_i = (0, 0)$. As can be seen in Figure 1, the grid is nearly cartesian in a neighborhood of the pole, and the grid cells become smaller as we approach the equator.

The line element, or distance between neighboring points on the sphere is given by the quadratic form

$$ds^2 = q_{ab} dx^a dx^b \tag{3.2}$$

where the components q_{ab} of the metric tensor depend upon the choice of coordinates. In standard spherical coordinates this takes the familiar form

$$ds^2 = d\theta^2 + \sin^2\theta \, d\phi^2. \tag{3.3}$$

In stereographic coordinates the line element is

$$ds^2 = 4(1 + \zeta_S \bar{\zeta}_S)^{-2}[(dS^1)^2 + (dS^2)^2] \tag{3.4}$$

or

$$ds^2 = 4(1 + \zeta_N \bar{\zeta}_N)^{-2}[(dN^1)^2 + (dN^2)^2], \tag{3.5}$$

depending upon coordinate patch.

Our choice of grid domains is not unique. Other options can be made to economize the number of points in the overlap region between the S and N grids, but the choice of rectangular boundaries lends itself to a simple interpolation scheme at the edge of the grid.

Given a smooth function Ψ on the sphere, we now give instructions for representing it on the grid. Denote the values of $\Psi_S(\zeta_S)$ at points on the S-grid by $\Psi_{S s_1, s_2}$; and the values of $\Psi_N(\zeta_N)$ at points on the N-grid by $\Psi_{N n_1, n_2}$. In the overlap region, the grid representation must be consistent with the condition $\Psi_S(\zeta_S) = \Psi_N(1/\zeta_S)$. However, only in exceptional cases will a point that lies on the S-grid also lie on the N-grid. To deal with this, we introduce

$$\zeta_{SN n_1, n_2} = \frac{(n_1 - i n_2) M}{(n_1^2 + n_2^2)}, \tag{3.6}$$

which gives the value of ζ_S at a point which lies on the N-grid. Similarly, we define

$$\zeta_{NS s_1, s_2} = \frac{(s_1 - i s_2) M}{(s_1^2 + s_2^2)}. \tag{3.7}$$

In the same way, in the overlap region, Ψ_S and Ψ_N determine the values Ψ_{Ss_1,s_2} and Ψ_{NSs_1,s_2} at points which lie on the S-grid; and Ψ_{Nn_1,n_2} and Ψ_{SNn_1,n_2} at points on the N-grid. However, the grid values Ψ_{Ss_1,s_2} are weaker information than the smooth function Ψ_S and do not by themselves determine the value of Ψ at points on the N-grid. Similarly, the values Ψ_{Nn_1,n_2} do not by themselves determine Ψ at a point on the S-grid. In these cases interpolation is necessary to evaluate functions on one grid at points lying on the other grid.

In order to see how this applies to differentiation, consider the gradient of Ψ, with components $\partial_{S^i}\Psi_S(\zeta_S)$ in the S-patch; and $\partial_{N^i}\Psi_N(\zeta_N)$ in the N-patch. Then, at points which do not lie on the grid boundary, the partial derivatives can be approximated by centered finite differences in the standard way. For example, at the point $S^i = s_i\Delta$, the appropriate second order finite difference approximation is

$$\partial_{S^1}\Psi_S = \frac{\Psi_{Ss_1+1,s_2} - \Psi_{Ss_1-1,s_2}}{2\Delta}. \tag{3.8}$$

For a point on the grid boundary, e.g. the point $S^1 = (1, s_2\Delta)$, we use the same approximation except that the field at the virtual grid point $s_i = (M+1, s_2)$ is approximated by the value $\Psi_{NS(M+1),s_2}$ obtained from interpolating Ψ_N. In order to achieve an error of second order in the derivative the interpolation error must be of third order.

This provides second order accurate finite difference representations of the gradient of Ψ at all grid points and, in fact, two representations, corresponding to $\partial_{S^i}\Psi_S(\zeta_S)$ and $\partial_{N^i}\Psi_N(\zeta_N)$, at points which lie in the overlap between the two grids. The analytic expressions are related by the transformation

$$\partial_{\zeta_S}\Psi_S(\zeta_S) = \frac{\partial_{\zeta_N}}{\partial_{\zeta_S}}\partial_{\zeta_N}\Psi_N(\zeta_N). \tag{3.9}$$

As a check on the finite difference approximation, the finite difference equivalent of (3.9) must hold to second order accuracy. Equivalently, the gradient may be converted into a scalar field by taking basis components and the scalar field compared between patches.

Second derivatives of a scalar field with respect to the coordinates may also be approximated to second order by central differencing. The interior points in each grid patch are handled by the standard techniques and the points on the grid boundary are handled by introducing a virtual grid point, as above. The fourth order interpolation scheme given in Appendix IV ensures that the second derivative is approximated to second order accuracy at boundary points. The same interpolation may be used in the approximation of the first derivative, Eq. (3.8). Note that bilinear interpolation at the boundary would only give a first order accuracy for the first derivative and the error in the centered second derivative would not converge to zero at the boundary.

IV. INTERPOLATION BETWEEN STEREOGRAPHIC PATCHES

In order to obtain a second order accurate centered difference approximation to the second derivative at a grid boundary, the virtual grid point must be interpolated to fourth order accuracy. Here we present an interpolation scheme which provides the field value at the virtual point with this accuracy. The same interpolation scheme, of course, also provides second order accurate centered first derivatives at the boundary.

The requirement for $O(\Delta^4)$ error makes it difficult to use only information from the N patch to interpolate the value of the virtual point. Indeed an interpolation that uses quadratic Lagrange polynomials in two dimensions extends over nine points of the N grid and only gives $O(\Delta^3)$ accuracy. Additionally, we would like to utilize information on both grids to minimize discontinuities in the error between the two patches.

The following compact approach accommodates those considerations. Consider the point at the S grid boundary with grid coordinates $s_i = (M, s2)$. Centered derivatives in the S_1 direction require the field value at the virtual grid point $s_i = (M+1, s2)$. We interpolate the value at this point using a cubic Lagrange polynomial along the S_1 coordinate line. Clearly the points $s_i = (M-1, s_2)$ and $s_i = (M, s2)$ are the natural collocation points inside the S grid. However, there are in general no points of the N grid which lie along this line.

Expressed in terms of the coordinates of the S patch, the N_1 coordinate lines are circles of radius $1/2n_1$ displaced from the origin in the S_1 direction by the exact same amount. The same holds for the N_2 coordinate lines, i.e. they are circles of radius $1/2n_2$ displaced from the origin in the S_2 direction. Thus the natural collocation points outside the S grid are the crossing points of the S_1 coordinate line with the coordinate lines of the N grid. The value of the field at the crossing points would likewise be obtained to $O(\Delta^4)$ order using cubic Lagrangean interpolation along the lines of the N grid.

The virtual point $s_i = (M+1, s_2)$ should lie between the collocation points of the S and the N grids. This minimizes the error by ensuring interpolation rather than extrapolation. Also, care must be given to whether the N_1 or N_2 coordinate lines are used in the interpolation. The choice should be those lines which are maximally transverse to the S_1 coordinate line. This is important, since the crossing angle between the S and N coordinate lines varies by π within the length of the $S_1 = M$ edge. For instance, near the vertex $s_i = (M, M)$, the N_1 line becomes tangential to the S_1 line and so it would be grossly inaccurate to use it for interpolation in the S_2 direction.

A robust and simple scheme is to use the N_1 lines for interpolations up to the point $s_i = (M, M/2)$ and then switch to the N_2 lines until we reach the vertex. The same rule applies all over the boundary and as a consequence we have a split of the boundary interpolation algorithm into several blocks. Within each block the same interpolation procedure is applied.

In order to calculate the mixed partial derivative at the corners of the grid boundary, field values are also required at a virtual grid point displaced in the diagonal direction. For example, in order to obtain a centered approximation for the mixed derivative $\partial_{S^1}\partial_{S^2}\Psi_S$ at the S grid coordinate (M, M) it is necessary to interpolate the value at the virtual point $(M+1, M+1)$ using Ψ_N. In this case, symmetry makes it simple to locate the two N grid points on the diagonal which, combined with the S grid points $(M-1, M-1)$ and (M, M), allow fourth order interpolation in the diagonal direction.

Figure 2 shows the virtual points of the north patch, mapped on the grid for the south patch, again using $M = 16$.

V. THE 3-D WAVE EQUATION IN STEREOGRAPHIC COORDINATES

Consider the numerical solution of the 3-D wave equation in spherical coordinates. An algorithm for computing the solution of the wave equation in the characteristic initial value formulation is given in [5]. In retarded time coordinates the wave equation takes the form

$$r \Box \Phi = 2(r\Phi)_{,ur} - (r\Phi)_{,rr} - \frac{L^2}{r}\Phi = 0, \qquad (5.1)$$

which can be reexpressed as

$$2g_{,ur} - g_{,rr} + \frac{D^2 g}{r^2} = 0, \qquad (5.2)$$

where $g = r\Phi$ and $L^2 g = -D^2 g$ is the angular momentum operator. The numerical scheme for the solution of (5.2) is an explicit marching algorithm, starting from the inner boundary and propagating outwards via the integral identity

$$g_Q = g_P + g_S - g_R - \frac{1}{2}\int_A \frac{du\,dr}{r^2} D^2 g, \qquad (5.3)$$

where the points P, Q, R, S are the corners of a null parallelogram of area A. The discretization of (5.3) in the case of axisymmetry is described in detail in [5].

In terms of the complex coordinate ζ, the Laplacian operator on the sphere is given by

$$D^2\Psi = (1+\zeta\bar{\zeta})^2 \partial_\zeta \partial_{\bar{\zeta}} \Psi. \qquad (5.4)$$

A 2-nd order centered finite difference approximation at the point (s_1, s_2) in the S grid is

$$D^2\Psi_S = \frac{1}{4\Delta^2}(1+\frac{s_1^2+s_2^2}{M^2})^2(\Psi_{Ss_1+1,s_2} + \Psi_{Ss_1-1,s_2} + \Psi_{Ss_1,s_2+1} + \Psi_{Ss_1,s_2-1} - 4\Psi_{Ss_1,s_2}), \quad (5.5)$$

where use of a virtual grid point is implied if (s_1, s_2) lies on the grid boundary. Since D^2 is a scalar operator, (5.4) gives the same value in the overlap region when evaluated in either the S or N coordinates.

The convergence and stability of this marching algorithm is dictated by the Courant-Friedrichs-Lewy (CFL) which requires that the numerical domain of dependence includes the physical domain of dependence. This depends upon the details of the discretization and, in particular, upon the choice of angular coordinates. The dependence is especially sensitive in the case where the inner boundary is at $r = 0$, at the vertex of the outgoing light cones. For the case of a stereographic atlas on the sphere, the CFL condition requires that the timestep for numerical evolution be limited by

$$\Delta u \le k \Delta r \Delta s_1 \Delta s_2, \qquad (5.6)$$

where k is a number of order one, related to the details of the startup procedure near the vertex at $r = 0$. To good accuracy, our numerical investigations give the value $k = 1$ for stable evolution. This provides an exceptional advantage over the expected stability condition for the 3-D marching algorithm using regularization of the Laplacian in θ and ϕ coordinates, for which

$$\Delta u \le k \Delta r \Delta\theta^2 \Delta\phi^2. \qquad (5.7)$$

This restriction, which makes regularization impractical for 3-D evolution, arises from the CFL condition applied at points near the vertex at small polar angles.

A crucial test of the patching scheme focuses on the accuracy of the numerical evolution. The linearity of the problem allows a thorough check since exact solutions are known. A useful family of solutions of the wave equation (5.2) is provided by the multipole fields [5]

$$G_{lm}(u,r,\zeta,\bar{\zeta}) = \frac{r^{l+1}Y_{lm}(\zeta,\bar{\zeta})}{(u+1)^{l+1}(u+2r+1)^{l+1}}. \tag{5.8}$$

In stereographic coordinates, the standard spherical harmonics take the form [6]

$$Y_{lm}(\zeta,\bar{\zeta}) = \alpha_{lm}\sum_n \frac{(-1)^n\zeta^{l-n}\bar{\zeta}^{l-m-n}}{n!(l-m-n)!(l-n)!(m+n)!}, \tag{5.9}$$

where

$$\alpha_{lm} = (-1)^m\sqrt{\frac{2l+1}{4\pi}(l+m)!(l-m)!}\frac{l!}{(1+\zeta\bar{\zeta})^l}$$

and the summation index n takes values between $max(0,-m)$ and $min(l,l-m)$.

The characteristic scheme evolves a global solution to the wave equation and thus it is possible to check the global numerical error. Figure 3 illustrates the convergence of the l_2 norm. The initial data is an $l=2, m=2$ harmonic obtained from (5.8) by setting $u=0$. After a finite evolution time, the expected second order accuracy of the algorithm is confirmed.

ACKNOWLEDGMENTS

We are thankful for research support under NSF Grants PHY92-08349 and PHY93-18152/ASC93-18152 (ARPA supplemented) and for computer time made available by the Pittsburgh Supercomputing Center.

REFERENCES

[1] H. Bondi, M. van der Burg, and A. Metzner, Proc. R. Soc. London Ser. A 103 (1962).
[2] E. Newman and R. Penrose, J. Math. Phys. **7**, 863 (1966).
[3] R. Penrose and W. Rindler, *Spinors and Space-Time* (Cambridge University Press, Cambridge, 1984), Vol. 1.
[4] J. N. Goldberg *et al.*, J. Math. Phys. **8**, 2155 (1967).
[5] R. Gómez, R. Isaacson, and J. Winicour, J. Comp. Phys. **98**, 11 (1992).
[6] J. Stewart, *Advanced General Relativity* (Cambridge University Press, Cambridge, 1984).

FIGURES

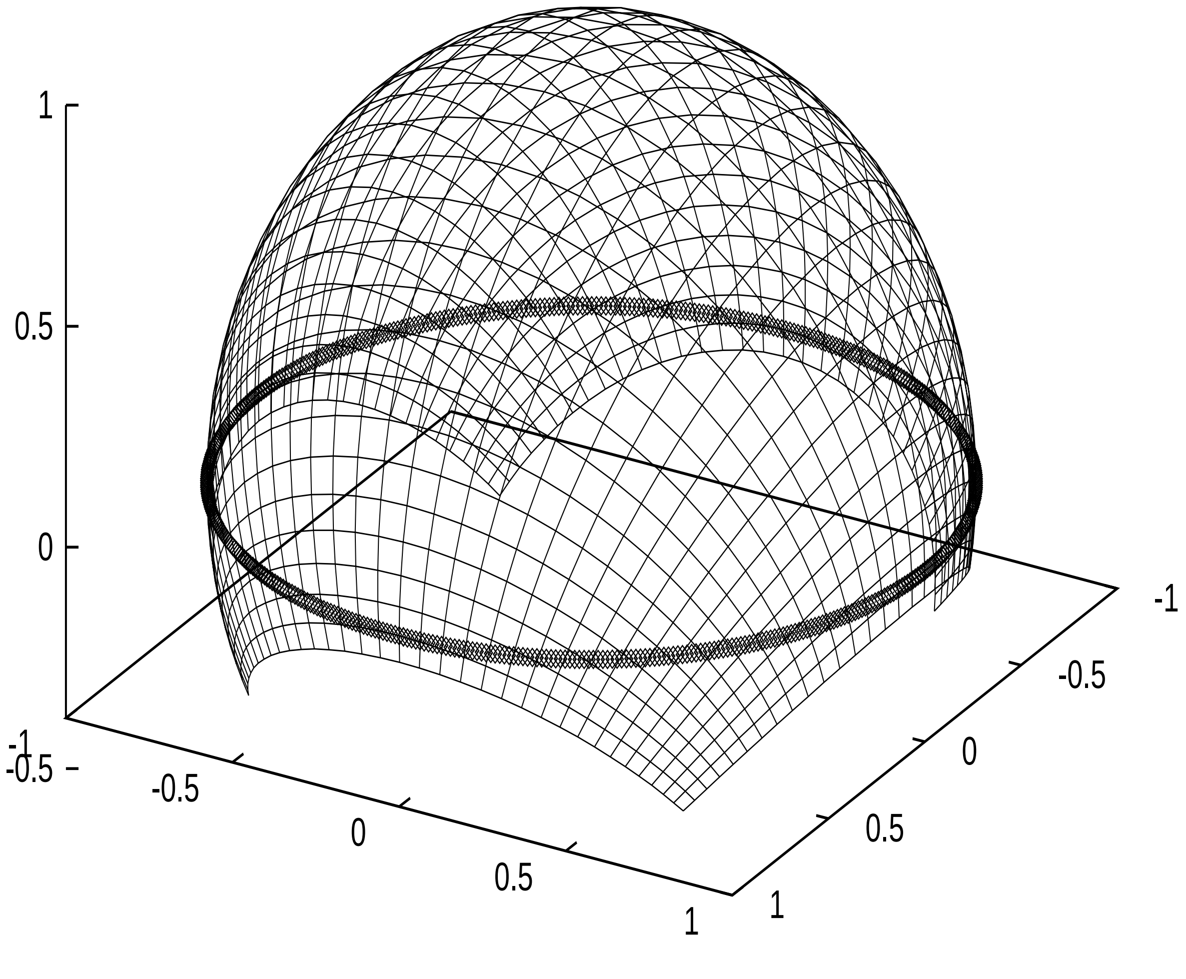

FIG. 1. The grid points that make up the north stereographic patch, shown here with a grid spacing of $\Delta = 1/16$. The thick circle represents the equator. Note that part of the N grid extends below the equator. The covering of the sphere is completed by a similar grid which defines the south stereographic patch, not shown.

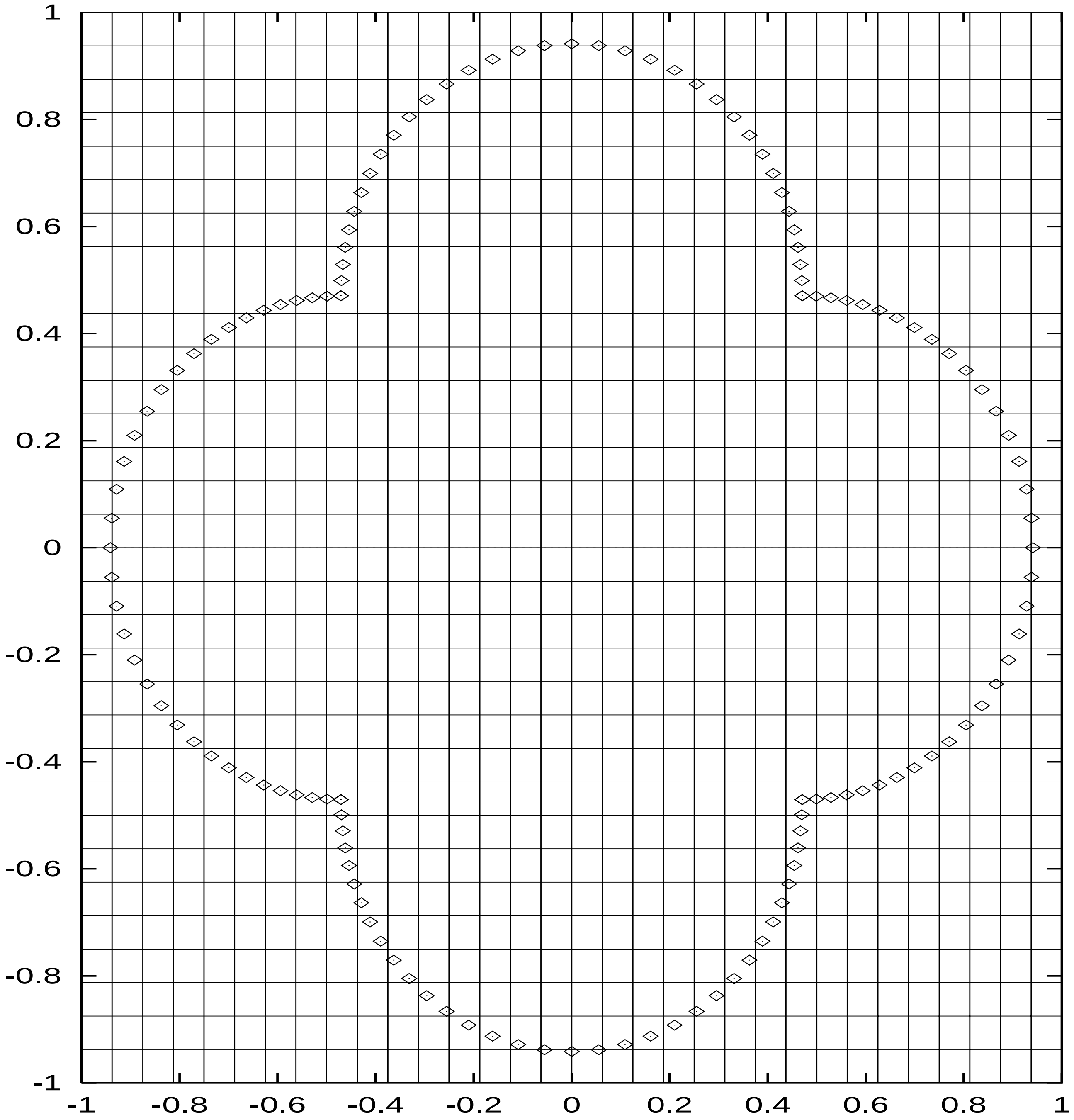

FIG. 2. The virtual grid points that extend the north stereographic patch, mapped on the south patch, are shown here for a grid with a spacing of $\Delta = 1/16$. The geomtry is the same for the virtual points of the south patch mapped on the north patch.

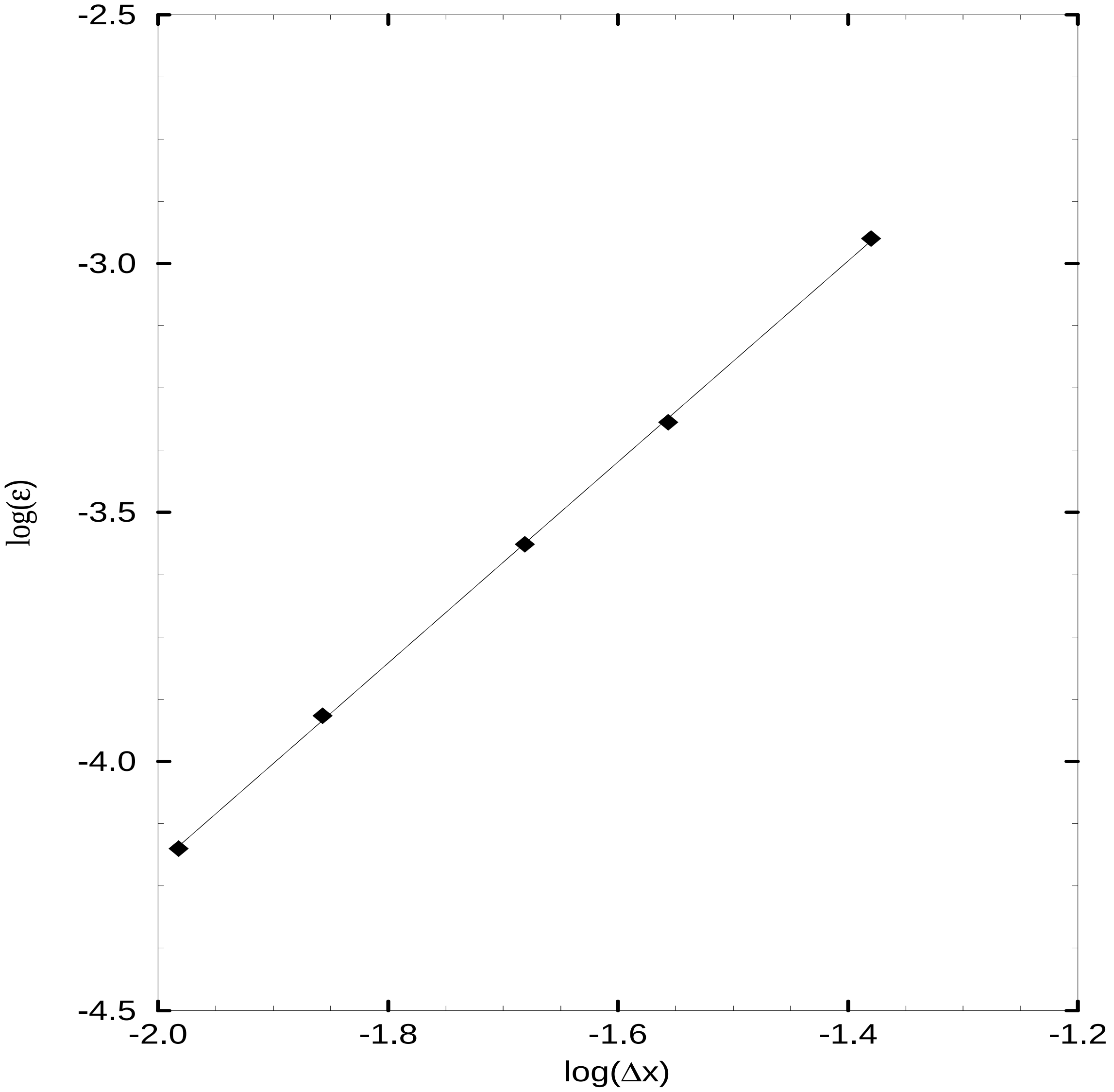

FIG. 3. The convergence of the wave equation algorithm, measured in terms of ϵ, the l_2 norm of the error. The exact solution is an $l = 2$, $m = 2$ multipole and the evolution time is $u = 0.5$. The convergence rate is 2.02. Δx is the radial step size in terms of the grid which compactifies infinity according to $x = r/(1 + r)$.

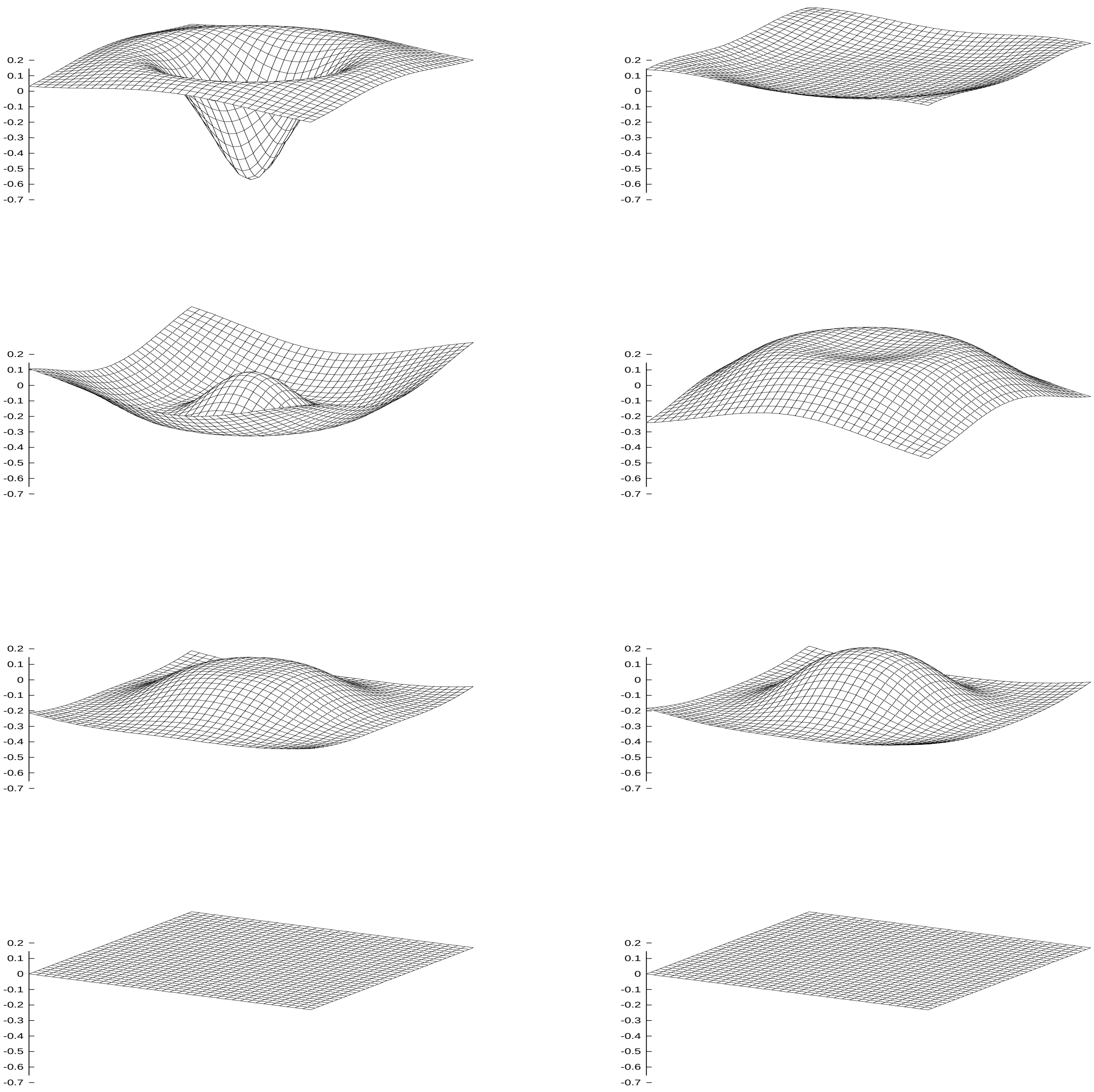

FIG. 4. A sequence of radiation patterns at infinity for initial data consisting of a localized pulse which is travelling towards the origin. The left column represents the upper (north) patch, the right column the lower (south) patch. The evolution proceeds from top to bottom. The first pair is at retarded time $u = 0.04$ and displays the initial radiation resulting from backscattering of the pulse off the centrifugal barrier created by its high angular momentum (l multipole values). This radiation shows up in the same hemisphere as the initial data. At the next two retarded times, $u = 0.16$ and $u = 0.32$, radiation appears in all directions as the pulse crosses the origin defining retarded time. Finally, at $u = 0.8$ the pulse has completely passed by the origin and the radiation ceases.

APPENDIX A: CODE LISTING

```
      program main

C     Numerical solution of the 3-D wave equation
C     using stereographic coordinate patches for the
C     calculation of the angular momentum operator.
C______________________________________________________________________
C
C     Begin: user - configurable parameters
C
C     Grid size

      integer          nd,     nx
      parameter       (nd = 8, nx = 32)

C     Safety margin factor, du = margin * dx * dd ** 2
C     The Courant condition might be violated if margin .gt. 1

      double precision margin
      parameter       (margin = 1.)

C     Starting and final evolution time

      double precision begintime,      endtime
      parameter       (begintime = 0., endtime = 1.)

C     Number of dumps to make - a value .lt. 0 indicates no output,
C     not even from the "norm" routine. Useful only to perform timing
C     comparisons.

      integer          dumps
      parameter       (dumps = 10)

C     Particular analytic solution to check against. The solution
C     is a pure Y_{ll,mm} spherical harmonic.

      integer          ll,      mm
      parameter       (ll = 2, mm = 2)

C     End: user - configurable parameters
C______________________________________________________________________

C     Time, grid spacings, iteration counters
```

```
      double precision time
      double precision dd, dx, du
      integer          i, it, nt, scoop

C     Grid arrays, field values, mask array

      double precision x(0:nx)
      double precision q(-(nd+1):(nd+1))
      double precision p(-(nd+1):(nd+1))
      double precision go(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      integer          mask(-(nd+1):(nd+1),-(nd+1):(nd+1))

C     Create grid functions, calculate timestep

      call setup(mask, x, q, p, nd, nx, dd, dx, du, ll, mm,
     &           margin, time, begintime, endtime, nt, dumps, scoop)

C     Load initial data on the (new) field array
C     Then interpolate and copy to (old) array

      call initial(gn, x, q, p, nd, nx, time, ll, mm)
      do i = 0, nx
        call interpolate(gn, q, p, mask, nd, nx, dd, i)
      end do
      call copy(go, gn, nd, nx)

C     Do some output of the initial data if requested

      if (dumps .gt. 0) then
        call monitor(gn, x, q, p, nd, nx, it, ll, mm, time)
      end if

C     Now do the numerical calculation

      do it = 1, nt

C       Advance time

        time = time + du

C       Do the radial march outwards

        i = 1
        call gstart(go, gn, x, q, p, nd, nx, dd, dx, du)
```

```
        call interpolate(gn, q, p, mask, nd, nx, dd, i)

        do i = 2, nx - 1
          call gmiddle(go, gn, x, q, p, nd, nx, dd, dx, du, i)
          call interpolate(gn, q, p, mask, nd, nx, dd, i)
        end do

        i = nx
        call gend(go, gn, x, q, p, nd, nx, dd, dx, du)
        call interpolate(gn, q, p, mask, nd, nx, dd, i)

C       Shift the field values and do some output

        call norm(gn, x, q, p, nd, nx, dd, dx, time, dumps, ll, mm)
        call copy(go, gn, nd, nx)
        if (dumps .gt. 0) then
          if ((mod(it,scoop) .eq. 0) .or. (it .eq. nt)) then
            call monitor(gn, x, q, p, nd, nx, it, ll, mm, time)
          end if
        end if

      end do

      end
      subroutine setup(mask, x, q, p, nd, nx, dd, dx, du, ll, mm,
     &                 margin, time, begintime, endtime, nt,
     &                 dumps, scoop)

      integer mask(-(nd+1):(nd+1),-(nd+1):(nd+1))
      double precision x(0:nx)
      double precision q(-(nd+1):(nd+1))
      double precision p(-(nd+1):(nd+1))
      integer nd, nx
      double precision dd, dx, du
      integer ll, mm
      double precision margin, time, begintime, endtime
      integer nt, dumps, scoop

      integer i

C     Fill-in grid coordinate functions
C     ND is the real grid dimension, the grid is enlarged by one,
C     so that it holds the interpolated values as well.

      dd = 1. / dble(nd)
```

```fortran
      dx = 1. / dble(nx)

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

      do i = -(nd+1), (nd+1)
         q(i)  = i * dd
         p(i)  = i * dd
      end do

C     Figure out the number of iterations (nt)
C     Calculate the timestep (du), and the sampling density (scoop).

      time = begintime
      du = margin * dx * dd ** 2

      nt = int((endtime - du) / du) + 1
      if (nt .lt. dumps) dumps = nt
      if (dumps .gt. 0) then
        scoop = int(nt / dumps)
      else
        scoop = 1
      end if

      if (dumps .gt. 0) then
        open (11,file='dat/run.log' ,status='new')
        write (11,*) 'Run Parameters'
        write (11,*) '------------------------------------'
        write (11,*) 'Grid size            : ', nx, ' x ', nd, ' x ', nd
        write (11,*) 'Initial time         : ', time
        write (11,*) 'Final time           : ', endtime
        write (11,*) 'Number of iterations : ', nt
        write (11,*) 'Number of data dumps : ', dumps
        write (11,*) ' '
        call flush(11)
      end if

C     Set mask array for interpolation between stereographic patches

      call setmask(mask, q, p, nd)

      return
      end
      subroutine initial(gn, x, q, p, nd, nx, time, ll, mm)
```

```
      double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision x(0:nx)
      double precision q(-(nd+1):(nd+1))
      double precision p(-(nd+1):(nd+1))
      double precision time
      integer          nd, nx, ll, mm

      integer          i, k, l
      double precision u, fg

C     Fill in initial data for all x's, both patches, all angles

      u = time

      do i = 0, nx
        do l = -nd, nd
          do k = -nd, nd
            gn(k,l,i,0) = fg(ll,mm,u,q(k),p(l),x(i),0)
            gn(k,l,i,1) = fg(ll,mm,u,q(k),p(l),x(i),1)
         end do
       end do
      end do

      return
      end

      double precision function fg(ll, mm, u, q, p, x, D)
      double precision u, q, p, x, y
      double precision lambda
      double precision cos1, ylm, cos2, phi
      integer ll,mm, D, l, m

C     Choose overall factor

      lambda = 1.
      l = ll
      m = mm

C     Calculate the cosine for this point and this patch

      y = cos1(q,p,D)
      phi = acos(cos2(q,p,D))
      if(abs(cos2(q,p,D)).gt.1) then
      print*, q, p, cos2(q,p,D)
      end if
```

```
      fg = lambda * 2**(l+1) *  x ** (l+1) * ylm(l, m, y, phi)
     &   / ((u + 1.) ** (l + 1) * (u * (1 - x) + x + 1.) ** (l+1))

      return
      end

      double precision function legendre(l,y)

C     Legendre Polynomials

      integer l, lc
      double precision y
      double precision pnext, tmp0, tmp1

C     l = 0 and 1 polynomials

      tmp0 = 1.
      tmp1 = y

      if(l.eq.0) then
         legendre = tmp0
      else if (l.eq.1) then
         legendre = tmp1
      else

C     Higher order recursive formula

         do lc = 2, l
            pnext = (y * (2*lc-1) * tmp1 - (lc-1) * tmp0) / lc
            tmp0 = tmp1
            tmp1 = pnext
         end do

 legendre = pnext

      endif
      return
      end

      double precision function ylm(l, m, x, phi)

C     Spherical Harmonics
C     Re(Ylm) (The real part)
```

```
      integer l, m
      double precision x, phi
      double precision pi, norm
      double precision plgndr, factrl

      pi = 3.14159265358979323846

      norm = sqrt((2. * l + 1.) / (4. * pi)
     &       * factrl(l - m) / factrl(l + m))

      ylm = norm * plgndr(l, m, x) * cos(m * phi)

      return
      end

      double precision function plgndr(l,m,x)

C     Associated Legendre Polynomials

      integer l,m
      double precision x
      integer i,ll
      double precision fact,pll,pmm,pmmp1,somx2

      if(m.lt.0.or.m.gt.l.or.abs(x).gt.1.) then
         stop 'bad arguments to plgndr'
      end if

      pmm=1.
      if(m.gt.0) then
somx2=sqrt((1.-x)*(1.+x))
fact=1.
do i=1,m
pmm=-pmm*fact*somx2
fact=fact+2.
enddo
      endif

      if(l.eq.m) then
         plgndr=pmm
      else
         pmmp1=x*(2*m+1)*pmm
         if(l.eq.m+1) then
            plgndr=pmmp1
 else
```

```
            do ll=m+2,l
pll=(x*(2*ll-1)*pmmp1-(ll+m-1)*pmm)/(ll-m)
pmm=pmmp1
pmmp1=pll
            enddo
plgndr=pll
endif
      endif
      return
      end

      double precision function factrl(n)

C     Factorial

      integer n
      double precision gammln
      integer j,ntop
      double precision a(33)
      save ntop,a
      data ntop,a(1)/0,1./

      if (n.lt.0) then
          stop 'negative factorial'
      else if (n.le.ntop) then
          factrl=a(n+1)
      else if (n.le.32) then
          do j=ntop+1,n
            a(j+1)=j*a(j)
          end do
          ntop=n
          factrl=a(n+1)
      else
      factrl = dexp(gammln(dble(n)+1.0D0))
      endif
      return
      end

      double precision function gammln(xx)
      double precision xx
      double precision half,one,fpf
      parameter (half=0.5d0, one=1.0d0, fpf=5.5d0)
      integer j
      double precision ser,stp,tmp,x,cof(6)
      save cof,stp
```

```
      data cof,stp/76.18009173d0,-86.50532033d0,24.01409822d0,
     & -1.231739516d0,.120858003d-2,-.536382d-5,2.50662827465d0/

      x = xx - one
      tmp = x + fpf
      tmp = (x+half) * log(tmp) - tmp
      ser = one

      do j=1,6
         x=x+one
         ser=ser+cof(j)/x
      end do
      gammln = tmp+log(stp*ser)

      return
      end

      double precision function p1(u,q,p,x,D)
      double precision u, q, p, x
      double precision lambda
      double precision cos2, sin1
      integer D

      lambda = 1.

      p1 = lambda * 4 * sin1(q,p,D) * cos2(q,p,D) * x**2 /
     &    ( (-x - u + u * x - 1)**2 * (u + 1)**2)


      return
      end

      double precision function sin1(q,p,D)
      double precision q, p
      integer D
      sin1 = 2. * sqrt(q*q + p*p) / (1. + q*q + p*p)
      if (sin1.gt.1) print*, 'Error in sin1'
      end

      double precision function sin2(q,p,D)
      double precision q, p
      integer D
      if(q.eq.0.and.p.eq.0) then
        sin2 = 0.
      else
```

```
          if(D.eq.0) then
               sin2 =   p / sqrt(q*q + p*p)
          else if(D.eq.1) then
               sin2 = - p / sqrt(q*q + p*p)
          end if
      end if
      if (sin2.gt.1) print*, 'Error in sin2'
      end

      double precision function cos1(q,p,D)
      double precision q, p
      integer D
      if(D.eq.0) then
         cos1 = (q*q + p*p -1.) / (1. + q*q + p*p)
      else if(D.eq.1) then
         cos1 = (1. - q*q - p*p) / (1. + q*q + p*p)
      end if
      if (cos1.gt.1) print*, 'Error in cos1'
      end

      double precision function cos2(q,p,D)
      double precision q, p
      integer D
      if(q.eq.0.and.p.eq.0) then
        cos2 = 0.
      else if(p.eq.0.and.q.gt.0) then
        cos2 = 1.
      else if(p.eq.0.and.q.lt.0) then
        cos2 = - 1.
      else
        cos2 = q / sqrt(q*q + p*p)
      end if
      if (cos2.gt.1) cos2 = 1.
      end
      subroutine copy(go, gn, nd, nx)
      double precision go(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      integer          nd, nx

      integer          i, k, l

C     Copy the "new" array into the "old" one. We could use a
C     "canned" routine from a library to do this chore.

      do i = 0, nx
```

```
    do l = -(nd+1), (nd+1)
      do k = -(nd+1), (nd+1)
         go(k,l,i,0) = gn(k,l,i,0)
         go(k,l,i,1) = gn(k,l,i,1)
      end do
    end do
  end do

  return
  end
  subroutine gstart(go, gn, x, q, p, nd, nx, dd, dx, du)
  double precision go(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
  double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
  double precision x(0:nx)
  double precision q(-(nd+1):(nd+1))
  double precision p(-(nd+1):(nd+1))
  integer          nd, nx
  double precision dd, dx, du

  integer          D, i, k, l
  double precision dold, dnew, d1, d2
  double precision xp, xq, xr, xs
  double precision rp, rq, rr, rs, rc
  double precision gp, gq, gr, gs
  double precision l2, l2im1, l2i, factor, area
  double precision gmean

C     Calculate the characteristic deviation

  i = 1
  dold = 0.5  * du * (1. - x(i-1)) ** 2
  dnew = 0.25 * du * (1. - x(i))   ** 2
  d1 = dold / dx
  d2 = dnew / dx

  xp = x(i-1)
  xq = x(i)   - dnew
  xr = x(i-1) + dold
  xs = x(i)   + dnew

  rp = xp / (1. - xp)
  rq = xq / (1. - xq)
  rr = xr / (1. - xr)
  rs = xs / (1. - xs)
```

```
      rc = 0.5 * (rp + rs)

C     area = \int_\Sigma du dr

      area = 0.5 * du * (rq - rp + rs - rr)

C     Calculate gn for both patches independently

      do D = 0, 1
        gmean = 0.
        do l = -nd, nd
          do k = -nd, nd

            factor = - 0.25 * (1 + q(k) ** 2 + p(l) ** 2) ** 2
            l2im1 = 0.
            l2i = factor * (1. / dd ** 2) *
     &             (go(k+1,l,i,D)
     &            + go(k-1,l,i,D) + go(k,l+1,i,D)
     &            + go(k,l-1,i,D) - 4. * go(k,l,i,D))
            l2 = 0.5 * (l2im1 + l2i)

C           Interpolate the g = r * f values

            gp = 0.

            gr = (1. - d1) * go(k,l,i-1,D) + d1 * go(k,l,i,D)

            gs = (1. - d2) * go(k,l,i,D)   + d2 * go(k,l,i+1,D)

            gq =  gp + gs - gr - 0.5 * area * l2 / rc ** 2

C           Extrapolate to obtain the new gamma grid value

            gn(k,l,i,D) = (gq - gn(k,l,i-1,D) * d2) / (1. - d2)
            gmean = gmean + gn(k,l,i,D)
          end do
        end do

        gmean = gmean / (2.* nd + 1.) ** 2

C       Average out the angular dependence

        do l = -nd, nd
          do k = -nd, nd
            gn(k,l,i,D) = gmean
```

```
         end do
       end do
     end do

     return
     end
     subroutine gmiddle(go, gn, x, q, p, nd, nx, dd, dx, du, i)
     double precision go(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
     double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
     double precision x(0:nx)
     double precision q(-(nd+1):(nd+1))
     double precision p(-(nd+1):(nd+1))
     integer          nd, nx
     double precision dd, dx, du
     integer          i

     integer          D, k, l
     double precision dold, dnew, d1, d2
     double precision xp, xq, xr, xs, xc
     double precision rp, rq, rr, rs
     double precision gp, gq, gr, gs
     double precision l2, l2im1, l2i, factor, integral

C    Calculate the characteristic deviation

       dold = 0.25 * du * (1. - x(i-1)) ** 2
       dnew = 0.25 * du * (1. - x(i))   ** 2

       d1 = dold / dx
       d2 = dnew / dx

       xp = x(i-1) - dold
       xq = x(i)   - dnew
       xr = x(i-1) + dold
       xs = x(i)   + dnew

       rp = xp / (1. - xp)
       rr = xr / (1. - xr)
       rq = xq / (1. - xq)
       rs = xs / (1. - xs)

       xc = 0.5 * (xp + xs)

C      integral = \int_\Sigma du dr 1/r^2
```

```
        integral = 2. * log((rq * rr) / (rp * rs))

C       Calculate gn for both patches independently

        do D = 0, 1
          do l = -nd, nd
            do k = -nd, nd

C             Calculate the Laplacian

              factor = - 0.25 * (1 + q(k) ** 2 + p(l) ** 2) ** 2

              l2im1 = factor * (1. / dd ** 2) *
     &                 (gn(k+1,l,i-1,D) + gn(k-1,l,i-1,D)
     &                + gn(k,l+1,i-1,D) + gn(k,l-1,i-1,D)
     &                - 4. * gn(k,l,i-1,D))

              l2i = factor * (1. / dd ** 2) *
     &               (go(k+1,l,i,D) + go(k-1,l,i,D)
     &              + go(k,l+1,i,D) + go(k,l-1,i,D)
     &              - 4. * go(k,l,i,D))

              l2 = 0.5 * (l2im1 / x(i-1) ** 2 +  l2i / x(i) ** 2)

C             Interpolate the g = r * f values

              gp = (1. - d1) * gn(k,l,i-1,D) + d1 * gn(k,l,i-2,D)

              gr = (1. - d1) * go(k,l,i-1,D) + d1 * go(k,l,i,D)

              gs = (1. - d2) * go(k,l,i,D)   + d2 * go(k,l,i+1,D)

C             Calculate the gq box value

              gq = gp + gs - gr - 0.5 * integral * l2 * xc ** 2

C             Extrapolate to obtain the new gamma grid value

              gn(k,l,i,D) = (gq - gn(k,l,i-1,D) * d2) / (1. - d2)

            end do
          end do
        end do

      return
```

```
      end
      subroutine gend(go, gn, x, q, p, nd, nx, dd, dx, du)
      double precision go(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision x(0:nx)
      double precision q(-(nd+1):(nd+1))
      double precision p(-(nd+1):(nd+1))
      integer          nd, nx
      double precision dd, dx, du

      integer          D, i, k, l
      double precision dold, dnew, d1, d2
      double precision xp, xq, xr, xs
      double precision rp,  rr
      double precision gp, gq, gr, gs
      double precision l2, l2im1, l2i, factor, integral

C     Calculate the characteristic deviation

      i = nx
      dold = 0.25 * du * (1. - x(i-1)) ** 2
      dnew = 0.
      d1 = dold / dx
      d2 = dnew / dx

      xp = x(i-1) - dold
      xq = 1.
      xr = x(i-1) + dold
      xs = 1.

      rp = xp / (1. - xp)
      rr = xr / (1. - xr)

C     integral = \int_\Sigma du dr 1/r^2

      integral = 2. * log(rr / rp)

C     Calculate gn for both patches independently

      do D = 0, 1
        do l = -nd, nd
          do k = -nd, nd

C           Calculate the Laplacian
```

```
          factor = - 0.25 * (1 + q(k) ** 2 + p(l) ** 2) ** 2

          l2im1 = factor * (1. / dd ** 2)
     &             * (gn(k+1,l,i-1,D) + gn(k-1,l,i-1,D)
     &              + gn(k,l+1,i-1,D) + gn(k,l-1,i-1,D)
     &              - 4. * gn(k,l,i-1,D))

          l2i = factor * (1. / dd ** 2)
     &           * (go(k+1,l,i,D) + go(k-1,l,i,D)
     &            + go(k,l+1,i,D) + go(k,l-1,i,D)
     &            - 4. * go(k,l,i,D))

          l2 = 0.5 * (l2im1 + l2i)

C         Interpolate the g = r * f values

          gp = (1. - d1) * gn(k,l,i-1,D) + d1 * gn(k,l,i-2,D)

          gr = (1. - d1) * go(k,l,i-1,D) + d1 * go(k,l,i,D)

          gs = go(k,l,i,D)

C         Calculate the gp box value

          gq = gp + gs - gr - 0.5 * integral * l2

C         Extrapolate to obtain the new gamma grid value

          gn(k,l,i,D) = (gq - gn(k,l,i-1,D) * d2) / (1. - d2)

        end do
      end do
    end do

    return
    end
    subroutine monitor(gn, x, q, p, nd, nx, it, ll, mm, time)
    double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
    double precision x(0:nx)
    double precision q(-(nd+1):(nd+1))
    double precision p(-(nd+1):(nd+1))
    integer          nd, nx, it, ll, mm
    double precision time

    integer          i, k, l
```

```
   double precision fg

   character*19 gefile, gnfile
   character*12 ctime

   open(60, file = 'dat/ge.gnu', status = 'unknown')
   open(61, file = 'dat/gn.gnu', status = 'unknown')

   write (ctime,'(D12.6E2)') time
   gefile = 'dat/ge_' // ctime
   gnfile = 'dat/gn_' // ctime
   open (30, file = gefile, status = 'unknown')
   open (31, file = gnfile, status = 'unknown')

   i = nx
   l = 0

   do i = 0, nx
     do k = -nd, nd
       write (30,10) fg(ll,mm,time,q(k),p(l),x(i),0) - gn(k,l,i,0)
       write (31,10) gn(k,l,i,0)
     end do
     write(30,20)
     write(31,20)
   end do

   close (30)
   close (31)

   write (60, '(a)') 'splot "' // gefile(5:19) // '"'
   write (61, '(a)') 'splot "' // gnfile(5:19) // '"'
   call flush (60)
   call flush (61)

10 format(3E23.12E3)
20 format('')

   return
   end
   subroutine interpolate(gn, q, p, mask, nd, nx, dd, i)
   double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
   double precision q(-(nd+1):(nd+1))
   double precision p(-(nd+1):(nd+1))
   integer          mask(-(nd+1):(nd+1),-(nd+1):(nd+1))
   integer          nd, nx, i
```

```
      double precision dd

      double precision z
      parameter       (z = 1.)
      integer          D, U, k, l
      integer          jg, ig, ig2, jg2
      integer          k1, k2, l1, l2
      double precision qf, pf, cp, cq, cp1, cp2, cq1, cq2
      double precision q1, q2, p1, p2, v1, v2, q1p, q2p, p1p, p2p
      double precision r1, r2, r3, r4, rx, qd1, qd2, pd1, pd2
      double precision q_s2n, p_s2n, cubic

C     Fourth order interpolation at the patch boundary
C     Upper is the grid we are looking at (rectangular)
C     Lower is the other patch, with circular arc coordinate lines

C     Do both grids

      do D = 0,1

        if(D.eq.0) U = 1
        if(D.eq.1) U = 0

        do k= -(nd+1), (nd+1)
          do l= -(nd+1), (nd+1)

C           Look for the boundary

            if(mask(k,l).ne.0) then

C             Find the coordinate of the fictitious point in the lower
C             grid.

              qf = q_s2n(q(k),p(l))
              pf = p_s2n(q(k),p(l))

C             Find the guard point index (see notes for formulae)
C             Also find the outer neighboor index.

              if(mask(k,l).eq.11.or.mask(k,l).eq.21) then
                ig = int(qf/dd)
                jg = int(pf/dd) + int((sign(z,pf) - 1)/2)
              else if(mask(k,l).eq.12.or.mask(k,l).eq.22) then
                ig = int(qf/dd) + int((sign(z,qf) - 1)/2)
                jg = int(pf/dd)
```

```
            else if(mask(k,l).eq.3) then
              ig = int(qf/dd)
              jg = int(pf/dd)
            end if
            ig2 = ig - int(sign(z,q(ig)))
            jg2 = jg - int(sign(z,p(jg)))

C           Find the upper grid points used in the cubic interpolation

            k1 = k - 1 * int(sign(z,q(k)))
            k2 = k - 2 * int(sign(z,q(k)))
            l1 = l - 1 * int(sign(z,p(l)))
            l2 = l - 2 * int(sign(z,p(l)))

C           Give symbolic names to the constant coordinate lines
C           involved

            cp   = p(l)
            cq   = q(k)
            cq1 = q(ig)
            cq2 = q(ig2)
            cp1 = p(jg)
            cp2 = p(jg2)

C           Calculate the intersection points depending on which
C           combination of crossing lines we use

            if(mask(k,l).eq.11) then
              q1  = (1. + sqrt(1. - 4. * cp**2 * cq1**2))/(2.*cq1)
              p1p = -2.*cp*cq1**2/(1.+ sqrt(1. - 4. * cp**2 *cq1**2))
              q2  = (1. + sqrt(1. - 4. * cp**2 * cq2**2))/(2.*cq2)
              p2p = -2.*cp*cq2**2/(1.+ sqrt(1. - 4. * cp**2 *cq2**2))
            else if(mask(k,l).eq.21) then
              p1  = sign(z,p(l)) * sqrt(cq/cq1 - cq**2)
              p1p = - (cq1/cq) * p1
              p2  = sign(z,p(l)) * sqrt(cq/cq2 - cq**2)
              p2p = - (cq2/cq) * p2
            else if(mask(k,l).eq.12) then
              q1  = sign(z,q(k)) * sqrt(-cp/cp1 - cp**2)
              q1p = - (cp1/cp) * q1
              q2  = sign(z,q(k)) * sqrt(-cp/cp2 - cp**2)
              q2p = - (cp2/cp) * q2
            else if(mask(k,l).eq.22) then
              p1  = (-1. - sqrt(1. - 4. * cq**2 * cp1**2))/(2.*cp1)
              q1p = 2.*cq*cp1**2/(1. + sqrt(1. - 4. * cq**2 *cp1**2))
```

```
              p2  = (-1. - sqrt(1. - 4. * cq**2 * cp2**2))/(2.*cp2)
              q2p = 2.*cq*cp2**2/(1. + sqrt(1. - 4. * cq**2 *cp2**2))
            end if

C           Do the lower interpolation depending on which direction
C           is used

            if(mask(k,l).eq.11.or.mask(k,l).eq.21) then

              v1 = cubic(p1p, p(jg-1), p(jg), p(jg+1), p(jg+2)
     &                  ,gn(ig,jg-1,i,U), gn(ig,jg,i,U)
     &                  ,gn(ig,jg+1,i,U), gn(ig,jg+2,i,U))

              v2 = cubic(p2p, p(jg-1), p(jg), p(jg+1), p(jg+2)
     &                  ,gn(ig2,jg-1,i,U), gn(ig2,jg,i,U)
     &                  ,gn(ig2,jg+1,i,U), gn(ig2,jg+2,i,U))

            else if(mask(k,l).eq.12.or.mask(k,l).eq.22) then

              v1 = cubic(q1p, q(ig-1), q(ig), q(ig+1), q(ig+2)
     &                  ,gn(ig-1,jg,i,U), gn(ig,jg,i,U)
     &                  ,gn(ig+1,jg,i,U), gn(ig+2,jg,i,U))

              v2 = cubic(q2p, q(ig-1), q(ig), q(ig+1), q(ig+2)
     &                  ,gn(ig-1,jg2,i,U), gn(ig,jg2,i,U)
     &                  ,gn(ig+1,jg2,i,U), gn(ig+2,jg2,i,U))

            else if(mask(k,l).eq.3) then

              v1 = gn(ig,jg,i,U)
              v2 = gn(ig2,jg2,i,U)

            end if

C           Do the upper interpolation depending on which direction
C           is used

            if(mask(k,l).eq.11.or.mask(k,l).eq.12) then

              gn(k,l,i,D) = cubic(q(k), q(k2), q(k1), q1, q2
     &                           ,gn(k2,l,i,D), gn(k1,l,i,D), v1, v2)

            else if(mask(k,l).eq.21.or.mask(k,l).eq.22) then

              gn(k,l,i,D) = cubic(p(l), p(l2), p(l1), p1, p2
```

```
     &                    ,gn(k,l2,i,D), gn(k,l1,i,D), v1, v2)

               else if(mask(k,l).eq.3) then

                 qd1 = q_s2n(q(ig),p(jg))
                 pd1 = p_s2n(q(ig),p(jg))
                 qd2 = q_s2n(q(ig2),p(jg2))
                 pd2 = p_s2n(q(ig2),p(jg2))

                 r1 = sqrt(q(k1)**2+p(l1)**2)
                 r2 = sqrt(q(k2)**2+p(l2)**2)
                 rx = sqrt(q(k)**2+p(l)**2)
                 r3 = sqrt(qd1**2+pd1**2)
                 r4 = sqrt(qd2**2+pd2**2)

                 gn(k,l,i,D) = cubic(rx,r1,r2,r3,r4
     &                             ,gn(k1,l1,i,D),gn(k2,l2,i,D),v1,v2)

               end if
             end if
           end do
         end do
       end do

       return
       end

       subroutine setmask(mask, q, p, nd)
       integer mask(-(nd+1):(nd+1),-(nd+1):(nd+1))
       double precision q(-(nd+1):(nd+1))
       double precision p(-(nd+1):(nd+1))
       integer nd

       integer i, j

C      Set mask = 0 for interior points
C      Set mask = 11 for x edge and interpolation along constant x'
C      Set mask = 12 for x edge and interpolation along constant y'
C      Set mask = 21 for y edge and interpolation along constant x'
C      Set mask = 22 for y edge and interpolation along constant y'
C      Set mask = 3 for vertices(?)

       do i = -(nd+1), (nd+1)
         do j = -(nd+1), (nd+1)
           if(abs(j).eq.(nd+1).or.abs(i).eq.(nd+1)) then
```

```
          if(abs(j)*abs(i).eq.((nd+1)*(nd+1))) then
            mask(i,j) = 3
          else
            if(abs(i).eq.(nd+1).and.abs(j).le.(nd+1)/2)  then
              mask(i,j) = 11
            else if(abs(i).eq.(nd+1).and.abs(j).gt.(nd+1)/2) then
              mask(i,j) = 12
            end if
            if(abs(j).eq.(nd+1).and.abs(i).le.(nd+1)/2) then
              mask(i,j) = 22
            else if (abs(j).eq.(nd+1).and.abs(i).gt.(nd+1)/2) then
              mask(i,j) = 21
            end if
          end if
        else
          mask(i,j) = 0
        end if
      end do
    end do

    return
    end

    double precision function q_s2n(qs,ps)
    double precision qs, ps
    q_s2n = qs / (qs**2 + ps**2)
    end

    double precision function p_s2n(qs,ps)
    double precision qs, ps
    p_s2n = - ps / (qs**2 + ps**2)
    end

    double precision function cubic(yp,yp1,yp2,yp3,yp4,f1,f2,f3,f4)
    double precision yp,yp1,yp2,yp3,yp4,f1,f2,f3,f4
    double precision c1, c2, c3, c4

    c1 = (yp  - yp2) * (yp  - yp3) * (yp  - yp4) / (
   &     (yp1 - yp2) * (yp1 - yp3) * (yp1 - yp4) )


    c2 = (yp  - yp1) * (yp  - yp3) * (yp  - yp4) / (
   &     (yp2 - yp1) * (yp2 - yp3) * (yp2 - yp4) )
```

```
      c3 = (yp  - yp1) * (yp  - yp2) * (yp  - yp4) / (
     &     (yp3 - yp1) * (yp3 - yp2) * (yp3 - yp4) )

      c4 = (yp  - yp1) * (yp  - yp2) * (yp  - yp3) / (
     &     (yp4 - yp1) * (yp4 - yp2) * (yp4 - yp3) )

      cubic = c1 * f1 + c2 * f2 + c3 * f3 + c4 * f4

      end
      subroutine norm(gn, x, q, p, nd, nx, dd, dx, time, dumps, ll, mm)
      double precision gn(-(nd+1):(nd+1),-(nd+1):(nd+1),0:nx,0:1)
      double precision x(0:nx)
      double precision q(-(nd+1):(nd+1))
      double precision p(-(nd+1):(nd+1))
      integer          nd, nx
      double precision dd, dx, time
      integer          dumps, ll, mm

      integer          i, k, l, D
      double precision fg ,ex, nu, diff, l1, l2, linf
      integer          mk, ml, mi

C     Calculate the global l1, l2 and linfinity norms.

      l1 = 0.
      l2 = 0.
      linf = 0.

      mk = 0
      ml = 0
      mi = 0

      D = 0
      do i = 0, nx
        do l = -nd, nd
          do k = -nd, nd
           ex = fg(ll,mm,time,q(k),p(l),x(i),D)
           nu = gn(k,l,i,D)
           diff = abs(ex - nu)
           l1 = l1 + diff
           l2 = l2 + diff * diff
           if (diff.ge.linf) then
               mk = k
               mi = i
               ml = l
```

```
          linf = diff
        end if
      end do
    end do
  end do

  l2 = sqrt(l2) / sqrt(dble(nx * nd * nd))
  l1 = l1 / (dble(nx * nd * nd))

  if (dumps .gt. 0) then
    open(14, file = 'dat/norml1.dat', status = 'unknown')
    open(15, file = 'dat/norml2.dat', status = 'unknown')
    open(16, file = 'dat/normli.dat', status = 'unknown')
    write (14,10) time, l1
    write (15,10) time, l2
    write (16,20) time, linf, mk, ml, mi
    call flush (14)
    call flush (15)
    call flush (16)
  end if

10 format(2E23.12E3)
20 format(2E23.12E3,3I5)

  return
  end
```