The RNPL User's Guide

Robert Marsa
Matthew Choptuik
Center for Relativity
The University of Texas at Austin
Austin, TX, 78712-1081
marsa@hoffmann.ph.utexas.edu
matt@infeld.ph.utexas.edu

May 1995

Introduction

Writing a program to solve a system of partial differential equations takes a lot of work. It is not just a matter of implementing a clever solution scheme. A working program requires much more. It needs code for parameter fetching, initial data generation, input and output, memory management, and checkpointing as well as the actual routines for solving the equations.

`RNPL` (Rapid Numerical Prototyping Language) was written to help scientists solve equations quickly by automatically taking care of everything except the inner-most parts of the solution routines. In many cases, `RNPL` can generate the entire program–updates and all.

`RNPL` can be used in three basic ways: for producing complete programs, for producing skeleton programs, and for converting existing programs.

Chapter 1

Writing Complete Programs

1.1 3D Wave Equation

As a simple first example which shows off many of RNPL's features, let's consider the linear wave equation in three dimensions. This is an initial-value, boundary-value problem which can be stated as follows:

$$
\begin{aligned}
\partial_t^2 \phi &= \partial_x^2 \phi + \partial_y^2 \phi + \partial_z^2 \phi \\
\phi(xmin, y, z, t) &= 0 \\
\phi(xmax, y, z, t) &= 0 \\
\phi(x, ymin, z, t) &= 0 \\
\phi(x, ymax, z, t) &= 0 \\
\phi(x, y, zmin, t) &= 0 \\
\phi(x, y, zmax, t) &= 0 \\
\phi(x, y, z, 0) &= A e^{\frac{x-c_x}{\delta_x}^2} e^{\frac{y-c_y}{\delta_y}^2} e^{\frac{z-c_z}{\delta_z}^2} \\
where \quad xmin < x < xmax, ymin &< y < ymax, zmin < z < zmax
\end{aligned}
$$

Typically one would also specify ϕ', but we'll ignore this fact for now. To set this problem up with RNPL we must identify our requirements. We'll need one grid function ϕ. We'll need four difference operators for ∂_t^2, ∂_x^2, ∂_y^2, and ∂_z^2. We'll also need parameters for the initial data, namely c_x, c_y, c_z, A, δ_x, δ_y, and δ_z as well as parameters for the domain boundaries, $xmin$, $xmax$, $ymin$, $ymax$, $zmin$, and $zmax$.

We begin our RNPL program by specifying the parameters. While RNPL statements may appear in any order, it is good for users to keep things organized. Our parameter declarations look like this:

```
parameter float xmin := 0
parameter float xmax := 100
parameter float ymin := 0
parameter float ymax := 100
parameter float zmin := 0
parameter float zmax := 100
parameter float A    := 1.0
```

```
parameter float c_x  := 50.0
parameter float c_y  := 50
parameter float c_z  := 50
parameter float delta_x
parameter float delta_y
parameter float delta_z
```

Each parameter can be given a default value. There are other parameters the RNPL program will need, but they are automatically defined, so we'll discuss them later.

Next, we define the coordinate system. The declaration looks like this:

```
rect coordinates t,x,y,z
```

The name `rect` is the name of the coordinate system. We can now define a grid which uses these coordinates.

```
uniform rect grid g1 [1:Nx][1:Ny][1:Nz] {xmin:xmax}{ymin:ymax}{zmin:zmax}
```

We will define our grid function to have three time levels so we can use the standard leap-frog operators to solve the equation. The definition is:

```
float phi on g1 at -1,0,1
```

Now comes the operator definitions. We need four second derivatives, one for each coordinate.

```
operator D_LF(f,t,t) := (<1>f[0][0][0] - 2*<0>f[0][0][0] +
<-1>f[0][0][0])/(dt*dt)
operator D_LF(f,x,x) := (<0>f[1][0][0] - 2*<0>f[0][0][0] +
<0>f[-1][0][0])/(dx*dx)
operator D_LF(f,y,y) := (<0>f[0][1][0] - 2*<0>f[0][0][0] +
<0>f[0][-1][0])/(dy*dy)
operator D_LF(f,z,z) := (<0>f[0][0][1] - 2*<0>f[0][0][0] +
<0>f[0][0][-1])/(dz*dz)
```

Since we wish RNPL to produce the complete program, we must specify the partial-differential equations. This is done by defining the residual.

```
evaluate residual phi {
            [1:Nx][1:Ny][1:1] := <1>phi[0][0][0] = 0;
          [1:Nx][1:Ny][Nz:Nz] := <1>phi[0][0][0] = 0;
            [1:Nx][1:1][1:Nz] := <1>phi[0][0][0] = 0;
          [1:Nx][Ny:Ny][1:Nz] := <1>phi[0][0][0] = 0;
            [1:1][1:Ny][1:Nz] := <1>phi[0][0][0] = 0;
          [Nx:Nx][1:Ny][1:Nz] := <1>phi[0][0][0] = 0;
     [2:Nx-1][2:Ny-1][2:Nz-1] := D_LF(phi,t,t) = D_LF(phi,x,x) +
                                 D_LF(phi,y,y) + D_LF(phi,z,z)
                      }
```

The boundary conditions could also have been stated with a time derivative of phi, but this would have required another operator definition. The above method is the simplest.

To get RNPL to generate the initial data, we must provide an initialization for phi.

```
initialize phi {
  [1:Nx][1:Ny][1:Nz] := A*exp(-(x-c_x)^2/delta_x^2)*
                         exp(-(y-c_y)^2/delta_y^2)*
                         exp(-(z-c_z)^2/delta_z^2)
              }
```

We now instruct RNPL to solve the equation iteratively and to automatically generate the update routine.

```
looper iterative

auto update phi
```

Save the program to a file named wave3d_rnpl. The RNPL compiler can now produce C or FORTRAN code with one of the following commands.

```
rnpl -c wave3d_rnpl
rnpl -f77 wave3d_rnpl
```

We can produce a makefile for this program using a utility included with the RNPL distribution.

Before we can run the program, we need to create a parameter file. Here is an example:

```
parameters for wave3d
tag := "3d_"
lambda := .3
Nx0 := 32
Ny0 := 32
Nz0 := 32
xmin := 0
xmax := 10
ymin := 0
ymax := 10
zmin := 0
zmax := 10
ser := 0
fout := 1
iter := 10
epsiter := 1.0e-5
rmod := 1
A := 1.0
c_x := 5.0
c_y := 5
c_z := 5
delta_x := .8
delta_y := .8
delta_z := .8
in_file := "w3d_in.hdf"
out_file := "w3d_out.hdf"
level:=0
```

Along with the parameters we defined are some automatically defined parameters. See the Reference Manual for more information. Save this to a file called w3d_0.

The program can now be executed with the following command:

```
wave3d w3d_0
```

As stated earlier, the initial data should be specified with values for both ϕ and ϕ'. Since we've only specified ϕ, we have little control over the initial data. RNPL generates a second time level of data using an iterative procedure over which the user has no control. So how could we get proper initial data? One was is to write a separate program for generating initial data from user defined parameters. The only (current) way for RNPL to generate "good" initial data is to reduce the problem to 1st order form and specify data for the two derivatives of ϕ. This approach is taken in the next section.

1.2 1D Shifted Wave Equation

As a slightly more complicated example, let's consider the "shifted" wave equation in 1 dimension. We'll take the shift β to be a constant and the initial field configuration ϕ to be a left-moving Gaussian pulse. We'll fix the field values to 0 at the end points. This problem can be stated as follows:

$$
\begin{aligned}
\partial_t^2\phi(x,t) &= (1-\beta^2)\partial_x^2\phi(x,t) + 2\beta\partial_t\partial_x\phi(x,t) \\
\phi(xmin,t) &= 0 \\
\phi(xmax,t) &= 0 \\
\phi(x,0) &= Ae^{\frac{-(x-c)^2}{\Delta^2}} \\
\partial_t\phi(x,0) &= \frac{-2(x-c)}{\Delta^2}Ae^{\frac{-(x-c)^2}{\Delta^2}} \\
\beta(x) &= 0.5 \\
where \quad xmin &< x < xmax
\end{aligned}
$$

We can rewrite this equation in first order form by introducing the two auxilliary variables Φ and Π defined by:

$$
\begin{aligned}
\Phi &\equiv \partial_x\phi \\
\Pi &\equiv \partial_t\phi - \beta\partial_x\phi
\end{aligned}
$$

In terms of these variables, the problem becomes:

$$
\begin{aligned}
\partial_t\Phi(x,t) &= \partial_x(\beta\Phi + \Pi) \\
\partial_t\Pi(x,t) &= \partial_x(\beta\Pi + \Phi) \\
\Phi(x,0) &= \frac{-2(x-c)}{\Delta^2}Ae^{\frac{-(x-c)^2}{\Delta^2}} \\
\Pi(x,0) &= \Phi(x,0) \\
\partial_x\Phi(xmin,t) &= 0
\end{aligned}
$$

$$\partial_x \Phi(xmax, t) = 0$$
$$\Pi(xmin, t) = 0$$
$$\Pi(xmax, t) = 0$$

The boundary conditions are a bit tricky, but these seem to work. Below is an `RNPL` program to solve this problem. This program must be modified if FORTRAN output is desired since FORTRAN is not case sensitive. We use a 2 level Crank-Nicholson difference scheme.

```
# This program solves 1D 1st order shifted wave equation

parameter float xmin := 0
parameter float xmax
parameter float epsdis
parameter float c
parameter float A
parameter float delta

rec coordinates t,x
uniform rec grid g1 [0:Nx-1] {xmin:xmax}

float Phi on g1 at 0,1
float Pi on g1 at 0,1
float beta on g1 at 0,1
float phi on g1 at 0,1

operator D_FW1(f,x) := (<1>f[1] - <1>f[0])/dx
operator D_FW12(f,x) := (-3*<1>f[0] + 4*<1>f[1] - <1>f[2])/(2*dx)
operator D_BW12(f,x) := (3*<1>f[0] - 4*<1>f[-1] + <1>f[-2])/(2*dx)
operator D_CN(f,t) := (<1>f[0] - <0>f[0])/dt
operator D_CN(f,x) := (<1>f[1] - <1>f[-1] + <0>f[1] - <0>f[-1])/(4*dx)
operator D_CND(f,t) := (<1>f[0] - <0>f[0] +
             epsdis/16*(6*<0>f[0] + <0>f[-2] + <0>f[2] -4*(<0>f[-1] +
<0>f[1])))/dt
operator AVG(f,t) := (<1>f[0] + <0>f[0])/2
operator AVG(f,x) := (<0>f[0] + <0>f[-1])/2

evaluate residual Phi { [0:0] := D_FW12(Phi,x) = 0 ;
                        [1:1] := D_CN(Phi,t) = D_CN(beta*Phi + Pi,x);
                     [2:Nx-3] := D_CND(Phi,t) = D_CN(beta*Phi + Pi,x);
                  [Nx-2:Nx-2] := D_CN(Phi,t) = D_CN(beta*Phi + Pi,x);
                  [Nx-1:Nx-1] := D_BW12(Phi,x) = 0 }

residual Pi { [0:0] := <1>Pi[0] = 0 ;
              [1:1] := D_CN(Pi,t) = D_CN(beta*Pi + Phi,x);
           [2:Nx-3] := D_CND(Pi,t) = D_CN(beta*Pi + Phi,x);
        [Nx-2:Nx-2] := D_CN(Pi,t) = D_CN(beta*Pi + Phi,x);
        [Nx-1:Nx-1] := <1>Pi[0] = 0 }
```

```
residual beta { [0:Nx-1] := <1>beta[0] = <0>beta[0] }

residual phi { [0:0] := phi = 0 ;
            [1:Nx-1] := D_FW1(phi,x) = AVG(<1>Phi[0],x) }

initialize beta { [0:Nx-1]:= .5 }
initialize Phi { [0:Nx-1]:= -2*(x-c)/delta^2*A*exp(-(x-c)^2/delta^2) }
initialize Pi { [0:Nx-1]:= -2*(x-c)/delta^2*A*exp(-(x-c)^2/delta^2) }
initialize phi { [0:Nx-1]:= A*exp(-(x-c)^2/delta^2) }

looper iterative

auto update Phi, Pi
auto update beta, phi
```

A parameter file for the resulting programs is:

```
lambda := .5
Nx0 := 200
level := 0
xmin := 0
xmax := 50
ser := 0
fout := 1
iter := 400
epsiter := 1.0e-5
epsdis := .8
rmod := 10
A := 1.0
c := 25.0
delta := 4.0
in_file := "w_in0.hdf"
out_file := "w_out0.hdf"
```

Although rather simple, this program illustrates the use of multiple grid functions and implicit updates.

Chapter 2

Writing Skeleton Programs

Although RNPL can not automatically generate code to solve any problem, it can produce the bulk of the code, requiring the user only to write his own update routines.

2.1 Periodic Boundary Conditions

As a first example, let's consider the linear wave equation in 1 dimension with periodic boundary conditions. The reader familiar with *The RNPL Reference Manual* will know that RNPL doesn't currently handle periodic boundary conditions. So, we'll get RNPL to produce code and then we'll edit the update routine to provide the correct boundary conditions.

The problem is simple to define. We'll use a left-moving Gaussian pulse for initial data.

$$
\begin{aligned}
\partial_t^2\phi(x,t) &= \partial_x^2\phi(x,t) \\
\phi(x,0) &= Ae^{\frac{-(x-c)^2}{\Delta^2}} \\
\partial_t\phi(x,0) &= \frac{-2(x-c)}{\Delta^2}Ae^{\frac{-(x-c)^2}{\Delta^2}} \\
\phi(xmin,t) &= \phi(xmax,t) \\
where \quad & xmin < x < xmax
\end{aligned}
$$

Except for the boundary conditions, this problem is easy to set up using RNPL. We'll use the usual 2nd order leap-frog differencing. The result is:

```
# This program solves 1D 2nd order wave equation

parameter float xmin := 0
parameter float xmax := 100
parameter float A := 1.0
parameter float c
parameter float delta

rec coordinates t,x

uniform rec grid g1 [1:Nx] {xmin:xmax}
```

```
float phi on g1 at -1,0,1

operator D_LF(f,x,x) := (<0>f[1] - 2*<0>f[0] + <0>f[-1])/(dx*dx)
operator D_LF(f,t,t) := (<1>f[0] - 2*<0>f[0] + <-1>f[0])/(dt*dt)

evaluate residual phi { [1:1] := D_LF(phi,t,t) = D_LF(phi,x,x) ;
                     [2:Nx-1] := D_LF(phi,t,t) = D_LF(phi,x,x) ;
                      [Nx:Nx] := D_LF(phi,t,t) = D_LF(phi,x,x) }

initialize phi { [1:Nx] := A*exp(-(x-c)^2/delta^2) }

looper iterative

auto update phi
```

The residual for ϕ can't be left the way it is because the second x derivative would require points at 0 and Nx+1. We save this to a file called `wper_rnpl`. We can then run the compiler with `rnpl -f77 wper_rnpl`.

One of the files produced is called `updates.f`. This file contains:

```
!----------------------------------------------------------------------
!  This routine updates the following grid functions
!  phi
!----------------------------------------------------------------------
      subroutine update0(phi_np1,phi_n,phi_nm1,g1_shp,g1_bds,dx,dt)
        implicit none

        include 'globals.inc'

        integer g1_shp(1)
        integer g1_bds(2)
        real*8  phi_np1(g1_bds(1):g1_bds(2))
        real*8  phi_n(g1_bds(1):g1_bds(2))
        real*8  phi_nm1(g1_bds(1):g1_bds(2))
        real*8  dx
        real*8  dt
        integer i,j,k
        integer Nx

        Nx = Nx0 * 2**level + 1

          i=1
            phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
     &        dt*dt)-(phi_n(i+1)-2*phi_n(i)+phi_n(i-1))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
          do i=2, Nx-1
```

```
        phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
     &        dt*dt)-(phi_n(i+1)-2*phi_n(i)+phi_n(i-1))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
      end do
      i=Nx
        phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
     &        dt*dt)-(phi_n(i+1)-2*phi_n(i)+phi_n(i-1))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
    return
  end
```

This routine is easy to modify. We just change the reference to `phi_n(i-1)` in the first statement to `phi_n(Nx)`. We then change the reference to `phi_n(i+1)` in the last statement to `phi_n(1)`. The resulting code is:

```
!-----------------------------------------------------------------------
!  This routine updates the following grid functions
!  phi
!-----------------------------------------------------------------------
      subroutine update0(phi_np1,phi_n,phi_nm1,g1_shp,g1_bds,dx,dt)
        implicit none

        include 'globals.inc'

        integer g1_shp(1)
        integer g1_bds(2)
        real*8  phi_np1(g1_bds(1):g1_bds(2))
        real*8  phi_n(g1_bds(1):g1_bds(2))
        real*8  phi_nm1(g1_bds(1):g1_bds(2))
        real*8  dx
        real*8  dt
        integer i,j,k
        integer Nx

        Nx = Nx0 * 2**level + 1

          i=1
            phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
     &        dt*dt)-(phi_n(i+1)-2*phi_n(i)+phi_n(Nx))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
          do i=2, Nx-1
            phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
     &        dt*dt)-(phi_n(i+1)-2*phi_n(i)+phi_n(i-1))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
          end do
          i=Nx
            phi_np1(i)=phi_np1(i)-((phi_np1(i)-2*phi_n(i)+phi_nm1(i))/(
```

```
     &        dt*dt)-(phi_n(1)-2*phi_n(i)+phi_n(i-1))/(dx*dx))/(dt*dt/
     &        (dt*dt*dt*dt))
        return
      end
```

We then save the new file and make as usual. Here is an example parameter file:

```
parameters for wper
lambda := .8
Nx0 := 100
xmin := 0
xmax := 15
level := 0
ser := 0
fout := 1
iter := 1000
epsiter := 1.0e-4
rmod := 10
A := 1.0
c := 5.0
delta := 0.8
in_file := "wp_i0.hdf"
out_file := "wp_o0.hdf"
```

Unfortunately, due to the trouble with 2nd order initial data discussed in section 1.1, we end up with both a left-moving and a right-moving piece. But the periodic boundary conditions are still apparent as long as the Gaussian is not centered.

Chapter 3

Converting Existing Programs

RNPL can be used to convert existing programs. Such a converted program has access to all of RNPL's features including automatic check-pointing and interactive output control.

3.1 Reid's 3D Boson Stars

As a first example we'll look briefly at Reid Guenther's 3D boson star code. In order to convert a code we must know what grid functions it uses and what the update routine looks like. In the case of Reid's code we have a complex scalar field, a potential, and some other functions. The declarations are:

```
# This program solves 3D time dependent schrodinger equation
# coupled with a newtonian potential:
# phi_t = i/2*(phi_xx + phi_yy + phi_zz) - i*V*phi
# V_xx + V_yy + V_zz = phi*conjg(phi)

constant parameter float xmin := 0
constant parameter float xmax := 100
constant parameter float ymin := 0
constant parameter float ymax := 100
constant parameter float zmin := 0
constant parameter float zmax := 100
constant parameter int solve_ncycle0 := 10
parameter int solve_ncycle := 3
# these parameters are for determining nu - boundary layer function
constant parameter int order := 2
constant parameter float edgdst := 0
constant parameter float delta := 0
constant parameter float height := 0
constant parameter float epsilon := 1.0e-6

rect coordinates t,x,y,z

uniform rect grid g1
```

```
float phre on g1 at 0,1
float phim on g1 at 0,1
float v on g1 at -1,0,1 alias
float rho on g1
float nu on g1
float vnph on g1

looper standard
```

Parameters that are declared as `constant` will be passed to update routines in a global common block. Other parameters can be passed in the header. Since the update routine already exists, there is no reason to define operators or residuals. All that is left is the update declaration.

The update routine has the following header:

```
      subroutine evolve3d(phren, phimn, phrenp1, phimnp1,
     *                 vnm1, vn, vnph, vnp1,
     *                 rho, nu, wksp,
     *                 nwksp,
     *                 ds,
     *                 x, y, z,
     *                 dt,
     *                 solve_ncycle)
        integer   ds(3), solve_ncycle, nwksp
        real*8    phren(ds(1),ds(2),ds(3))
        real*8    phimn(ds(1),ds(2),ds(3))
        real*8    phrenp1(ds(1),ds(2),ds(3))
        real*8    phimnp1(ds(1),ds(2),ds(3))
        real*8    vnm1(ds(1),ds(2),ds(3))
        real*8    vn(ds(1),ds(2),ds(3))
        real*8    vnph(ds(1),ds(2),ds(3))
        real*8    vnp1(ds(1),ds(2),ds(3))
        real*8    rho(ds(1),ds(2),ds(3))
        real*8    nu(ds(1),ds(2),ds(3))
        real*8    x(ds(1)), y(ds(2)), z(ds(3))
        real*8    wksp(nwksp)
        real*8    dt
```

We must get `RNPL` to generate a header as close to this as possible. The first thing we do is take the code from the inside of `evolve3d` (everything after the header and before the return) and save it to a file which we'll call `evolve3d.inc`. The variable `wksp` is a temporary work array. `RNPL` names such arrays `work0`, `work1`, ...This means we must change the name of the work array in the code. This can be done with a global search and replace (for instance with `sed s/wksp/work0/g evolve3d.inc`. This will change `nwksp` to `nwork0` as it should. This is the only modification necessary to the existing code. Other codes may need other changes. For instance, the shape of the grid is called `ds` above but `g1_shp` by `RNPL`. If the shape is needed in the actual code, this name will have to be changed as well. The `RNPL` code to generate the header is:

```
evolve3d.inc evolve3d UPDATES phre, phim, v
   HEADER phre[phrenp1,phren], phim[phimnp1,phimn], v[vnp1,vn,vnm1], rho,
vnph,
          nu, AUTO work#0(5*nx*ny*nz), x, y, z, solve_ncycle, dt
```

The time levels that RNPL generates for phre will be phre_n and phre_np1 by default. However, in Reid's code these grid functions are called phren and phrenp1. Likewise for the other grid functions. This naming problem is resolved with the bracket notation above.

This completes the RNPL program. Save this to a file called bos0.rnpl.

When the RNPL compiler is run (rnpl -f77 bos0.rnpl), it pulls the contents of evolve3d.inc into the stub it produces and calls the routine evolve3d. The result is placed in a file called updates.f. The header in this file is:

```
!----------------------------------------------------------------------
!  This routine updates the following grid functions
!  phre phim v
!----------------------------------------------------------------------
      subroutine evolve3d(phrenp1,phren,phimnp1,phimn,vnp1,vn,vnm1,rho,
     &  nu,vnph,g1_shp,g1_bds,x,y,z,dt,solve_ncycle,work0,nwork0)
        implicit none

        include 'globals.inc'

        integer g1_shp(3)
        integer g1_bds(6)
        real*8  phre_np1(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  phre_n(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  phim_np1(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  phim_n(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  v_np1(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  v_n(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  v_nm1(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  rho(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  nu(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  vnph(g1_bds(1):g1_bds(2),g1_bds(3):g1_bds(4),
     &    g1_bds(5):g1_bds(6))
        real*8  x(*)
```

```
        real*8  y(*)
        real*8  z(*)
        real*8  dt
        integer solve_ncycle
        integer nwork0
        real*8  work0(nwork0)
```

When RNPL is run, it produces the following warnings:

```
warning: calc_resid: Update exists for phre but no residual.
warning: calc_resid: Update exists for phim but no residual.
warning: calc_resid: Update exists for v but no residual.
warning: calc_resid: No residual is being evaluated.
warning: calc_resid: Update exists for phre but no residual.
warning: calc_resid: Update exists for phim but no residual.
warning: calc_resid: Update exists for v but no residual.
warning: calc_resid: No residual is being evaluated.
```

These would be a problem if we were expecting RNPL to produce its own updates.

3.2 Initial Data

How do we get initial data? That depends on the structure of the original program. In this case, the evolution routine detects when it is being called the first time and produces its own initial data. When the evolver is run, it will give a warning that it can't read the initial data file and will expect the update to do its own initialization.

If, on the other hand, there is a separate initial data generator which produces a file, then we could simply insert some calls to write the data into an initial data HDF file. RNPL produces these calls in a file called `initfrag.f`. In this case the file contains:

```
!----------------------------------------------------------------------
!  Code fragment for creating initial data file
!----------------------------------------------------------------------
        call rnpl_id_set_file('idfile.hdf')
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('phre[0]',phre_n)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('phim[0]',phim_n)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('v[1]',v_n)
        call rnpl_id_write('v[0]',v_nm1)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('rho[0]',rho)
        call rnpl_id_set_rank(3)
```

```
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('nu[0]',nu)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(101,101,101)
        call rnpl_id_write('vnph[0]',vnph)
        call rnpl_id_end()
```

We just insert this into the initial data program and edit the shape information and the grid function names. Do NOT edit the names inside the single quotes, just the actual variable names. In our case we might edit the file to look like this:

```
!-----------------------------------------------------------------------
!  Code fragment for creating initial data file
!-----------------------------------------------------------------------
        call rnpl_id_set_file('bosid.hdf')
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('phre[0]',phren)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('phim[0]',phimn)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('v[1]',vn)
        call rnpl_id_write('v[0]',vnm1)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('rho[0]',rho)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('nu[0]',nu)
        call rnpl_id_set_rank(3)
        call rnpl_id_set_shape(nx,ny,nz)
        call rnpl_id_write('vnph[0]',vnph)
        call rnpl_id_end()
```

We can then name the initial data generator `bos0_init` and the evolver will automatically call it to generate the initial data. More information about these routines is provided in chapter 4 below.

Chapter 4

BBHUtil Library

The BBHUtil library contains a set of HDF reading and writing routines for both RNPL evolution and initial data files. It also contains routines for reading parameters from RNPL parameter files.

4.1 Evolution

4.1.1 FORTRAN Interface

The FORTRAN interface for evolution data files comprises eight routines, four for writing, two for reading, and two for closing. All routines return zero on error and one on success. The following program fragment provides declarations and sample calls for the writing routines.

```
integer  ret, gft_write, gft_write_brief
integer  gft_write_bbox, gft_write_full
integer  gft_close, gft_close_all
real*8   time, data(10,10,10)
integer  shape(3), rank, i, j, k
character*5 cnames(3)
real*8   bbox(6), coords(10,3)

time=1.2
rank=3
shape(1)=10
shape(2)=10
shape(3)=10
do i=1,shape(1)
  do j=1,shape(2)
    do k=1,shape(3)
      data(k,j,i)=i*j*k
    end do
  end do
end do
ret=gft_write('My function 1',time,shape,rank,data)
time=2.1
ret=gft_write_brief('My function 1',time,shape,rank,data)
bbox(1)=0.0
```

```
bbox(2)=10.0
bbox(3)=1.2
bbox(4)=103.5
bbox(5)=2.0
bbox(6)=58.1
ret=gft_write_bbox('My function 2',time,shape,rank,bbox,data)
cnames(1)='x'
cnames(2)='y'
cnames(3)='z'
do i=1,shape(1)
  coords(i,1)= 1 + .1*i
end do
do j=1,shape(2)
  coords(j,2)= 2 + .2*j
end do
do k=1,shape(3)
  coords(k,3)= 3 + .3*k
end do
ret=gft_write_full('My function 3',time,shape,cnames,rank,coords,data)
ret=gft_close('My function 2')
ret=gft_close_all()
```

The routines `gft_write` and `gft_write_brief` are identical.

The first parameter for each call is the grid function name. The file name is formed from the grid function name by removing blanks and appending `.hdf`. Thus, the first two calls would write to the file `Myfunction1.hdf`. Each grid function has its own evolution data file.

The time parameter contains the time at which the data is valid. Each time should only appear once in a file and the time levels should be written sequentially from earliest times to latest times. These conditions are not enforced.

The shape parameter is an integer array containing the sizes of each dimension of the data. The rank parameter is the number of dimensions of the data. The data parameter contains the values of the grid function at a single time.

The bbox parameter is a real*8 array of length 2*rank. It contains the coordinate bounding box. The bbox(1) is the minimum for coordinate 1, bbox(2) is the maximum for coordinate 1, bbox(3) is the minimum for coordinate 2, etc.

The cnames parameter is an array of strings containing the names of the coordinates. The names can be any length.

The coords parameter is an array of real*8 containing the coordinate values. The first shape(1) elements of coords contain the values of coordinate 1. The next shape(2) elements contain the values of coordinate 2, etc.

The routines `gft_write` and `gft_write_bbox` name the coordinates 'x', 'y', and 'z'. The routine `gft_write` sets the values of the ith coordinate to be the integers from 1 to shape(i).

The routine `gft_close` will close the named file if it is open. The routine `gft_close_all` will close all open HDF files. All files must be closed before the program exits, or data will be lost.

The two reading routines are similar to the writing routines. Sample calls follow:

```
integer  ret, gft_read_brief
integer  gft_read_full, gft_close_all
```

```
      real*8   time, data(10,10,10)
      integer  shape(3), rank, lev
      character*5 cnames(3)
      real*8   coords(10,3)

      lev=1
      ret=gft_read_brief('My function 1',lev,data)
      lev=2
      ret=gft_read_brief('My function 1',lev,data)
      lev=1
      rank=3
      ret=gft_read_full('My function 3',lev,shape,cnames,rank,time,coords,data)
      ret=gft_close_all()
```

The grid function name is treated the same way as for the writing routines. The lev parameter specifies which time level to read. This parameter starts at 1 and goes to the number of levels in the file. The rank is passed in to `gft_read_full`, but all other parameters are read from the file.

Both functions return zero on error and one on success.

4.1.2 C Interface

The C interface is identical to the FORTRAN interface with the addition of some extra routines. The C prototypes are:

```
#include <bbhutil.h>

int gft_close(const char *nm);
int gft_close_all(void);

int gft_set_d_type(int type)

int gft_write(const char *func_name, double time,
              int *shape, int rank, double *data);
int gft_write_brief(const char *func_name, double time,
                    int *shape, int rank, double *data);
int gft_write_bbox(const char *func_name, double time,
                   int *shape, int rank, double *coords, double *data);
int gft_write_full(const char *func_name, double time,
                   int *shape, char **cnames, int rank,
                   double *coords, double *data);

int gft_read_brief(const char *file_name, int level, double *data);
int gft_read_full(const char *file_name, int level,
                  int *shape, char **cnames,
                  int rank, char *time, double *coords,
                  double *data);
int gft_read_shape(const char *file_name, int *shape);
```

`gft_close` closes the HDF file whose name is passed as a parameter.

`gft_close_all` closes all open HDF files.

`gft_set_d_type` will set the grid function type which is written by the `gft_write*` calls. The default type is DFNT_FLOAT64. The only other valid type is DFNT_INT32.

`gft_write` and `gft_write_brief` will write the grid function given in data to the evolution HDF file whose name is formed from the func_name. Func_name is the name of the grid function. Time gives the time at which the grid function had the values given in data. The rank and shape provide the size of the grid function. The coordinates are named x,y,z and the coordinate values are integers running from 1 to shape.

`gft_write_bbox` performs the same function as `gft_write`. In addition, the coordinate values are taken from the bounding box coords. Coords is a one dimensional array the whose first two values provide the min and max for the first coordinate, the next two values provide the min and max for the second coordinate, etc. In general, the minimum for the nth coordinate is stored in position 2(n-1) and the maximum is in position 2(n-1) + 1.

`gft_write_full` works as the above except the coordinate names are given explicitly in cnames and the coordinate values are given explicitly in coords. The first shape[0] values of coords are the values of the first coordinate, the next shape[1] values are for the second coordinate, etc.

`gft_read_brief` returns a grid function from the file given by file_name. Level runs from 1 to the number of time levels stored in the file.

`gft_read_full` takes the file name, desired time level, and rank and returns complete information including the grid function, time, shape, coordinate names, and coordinate values. Storage for the returned values must be preallocated.

`gft_read_shape` returns the shape of the data sets in an RNPL evolution HDF file.

4.2 Initial Data

Initial data HDF files differ from evolution HDF files in several ways, most notably in that a single initial data file contains the complete information necessary for starting a calculation. This includes all grid functions and parameters.

4.2.1 FORTRAN Interface

The FORTRAN interface for the initial data routines comprises twelve routines, six for writing and six for reading. These routines allow the reading and writing of parameters as well as grid functions.

The following program fragment writes and then reads initial data for three grid functions.

```
integer gft_write_id_str_p, gft_write_id_float_p
integer gft_write_id_int_p, gft_write_idata
integer gft_write_1idata, gft_write_2idata
integer gft_read_id_str_p, gft_read_id_float_p
integer gft_read_id_int_p, gft_read_idata
integer gft_read_1idata, gft_read_2idata
integer ret, start
character*20 name(2)
real*8  time
integer rank, shape(1)
real*8  A_n(101), A_np1(101)
real*8  B_nm1(101), B_n(101), B_np1(101)
```

```
      real*8  C_nm1(101), C_n(101), C_np1(101), C_np2(101)

      name(1)='test name 1'
      name(2)='test name 2'
      ret=gft_write_id_str_p('sample','names',name,2)
      start=15
      ret=gft_write_id_int_p('sample','start',start,1)
      time=1.2
      ret=gft_write_id_float_p('sample','time',time,1)
c     set grid functions
      rank=1
      shape(1)=101
c     initial data for a two level function
      ret=gft_write_1idata('sample','A',shape,rank,A_n)
c     initial data for a three level function
      ret=gft_write_2idata('sample','B',shape,rank,B_nm1,B_n)
c     initial data for an n level function
      ret=gft_write_idata('sample','C[0]',shape,rank,C_nm1)
      ret=gft_write_idata('sample.hdf','C[1]',shape,rank,C_n)
      ret=gft_write_idata('sample','C[2]',shape,rank,C_np1)
      ret=gft_close('sample')
c     or ret=gft_close_all()
      rank=1
      shape(1)=101
      ret=gft_read_id_str_p('sample','names',name,2)
      ret=gft_read_id_int_p('sample','start',start,1)
      ret=gft_read_id_float_p('sample','time',time,1)
      ret=gft_read_1idata('sample','A',shape,rank,A_n)
      ret=gft_read_2idata('sample','B',shape,rank,B_nm1,B_n)
      ret=gft_read_idata('sample','C[0]',shape,rank,C_nm1)
      ret=gft_read_idata('sample','C[1]',shape,rank,C_n)
      ret=gft_read_idata('sample','C[2]',shape,rank,C_np1)
      ret=gft_close_all()
```

Since one file contains all the initial data, two names must be passed to each call, the file name and the item name. File names will have `.hdf` appended if it is not already there.

The parameter writing and reading functions take two additional parameters, an array of parameters and the array length.

The grid function writing and reading functions take shape and rank information and either write or read the grid function(s). The grid function name passed to the routines `gft_read_idata` and `gft_write_idata` must have [n] appended to it, where n is an integer. A value of 0 means the first (earliest) time level, then 1, 2, etc. Initial data need only be written for m-1 time levels, where m is the number of time levels carried for a particular grid function. All m levels can be written, but the RNPL generated program will only read the first m-1.

All routines return zero on failure and one on success.

4.2.2 C Interface

The C interface is identical to the FORTRAN interface. The C prototypes are given below.

```
#include <bbhutil.h>

int gft_read_id_str_p(const char *file_name, const char *param_name,
                      char **param, int nparam);
int gft_read_id_int_p(const char *file_name, const char *param_name,
                      int *param, int nparam);
int gft_read_id_float_p(const char *file_name, const char *param_name,
                        int *param, int nparam);
int gft_read_2idata(const char *file_name, const char *func_name,
                    int *shape, int rank,
                    double *datanm1, double *datan);
int gft_read_1idata(const char *file_name, const char *func_name,
                    int *shape, int rank, double *datan);
int gft_read_idata(const char *file_name, const char *func_name,
                   int *shape, int rank, double *data);

int gft_write_id_str_p(const char *file_name, const char *param_name,
                       char **param, int nparam);
int gft_write_id_int_p(const char *file_name, const char *param_name,
                       int *param, int nparam);
int gft_write_id_float_p(const char *file_name, const char *param_name,
                         double *param, int nparam);
int gft_write_2idata(const char *file_name, const char *func_name,
                     int *shape, int rank,
                     double *datanm1, double *datan);
int gft_write_1idata(const char *file_name, const char *func_name,
                     int *shape, int rank, double *datan);
int gft_write_idata(const char *file_name, const char *func_name,
                    int *shape, int rank, double *data);
```

4.3 Parameters

The following program fragments will read parameters from the sample file called `myparams` below.

```
This is an example parameter file.
array2 := [1.0 2.0 3.56 7.1 10.315]
This is an integer array called array1
array1 := [5 6 8]
string1 := "This is a sample string"
stringar := ["one" "two" "three" "four"]
p := 104
a := [1 2 3 4 5 6 7 8 9 10]
b := "test"
Here is some more text
```

4.3.1 FORTRAN Interface

The FORTRAN interface consists of three routines, one for each type of parameter. These routines can be called as shown below:

```
real*8 array2(5),p
integer arr1(3),len
character*16 b, sar(4)

call get_real_param('myparam','p',0,p,1)
call get_real_param('myparam','array2',0,array2,5)
arr1(1)=1.0
arr1(2)=3.0
arr1(3)=6.0
call get_int_param('myparam','array1',1,arr1,3)
len=10
call get_int_param('myparam','len',1,len,1)
b='my string'
call get_str_param('myparam','b',1,b,1)
call get_str_param('myparam','stringar',0,sar,4)
```

The first argument is the name of the file. The second argument is the name of the parameter. The third argument is a flag which is set to 1 if the parameter has a default value, and 0 if not. The fourth argument is the storage for the paramenter, and the last is the number of elements in the parameter.

4.3.2 C Interface

The C interface consists of a single routine whose prototype is in `librnpl.h`.

```
BBH_V get_param(const BBH_C *p_file, BBH_C *name, BBH_I def,
               BBH_C *type, BBH_I size, BBH_V *p);
```

This routine takes the file name, parameter name, default flag, parameter type, parameter size (in elements), and a pointer to the parameter. The following code fragment reads several parameters from a file called `myparams`.

```
long len,ar1[3];
double p,ar2[5];
char *s,**ar3;

ar3=(char **)malloc(4*sizeof(char *));
len=10;
get_param("myparams","len",1,"long",1,&len);
get_param("myparams","array1",0,"long",3,ar1);
get_param("myparams","p",0,"double",1,&p);
ar2[0]=ar2[1]=ar2[2]=ar2[3]=ar2[4]=1.0;
get_param("myparams","array2",1,"double",5,ar2);
get_param("myparams","string1",0,"string",1,&s);
get_param("myparams","stringar",0,"string",4,ar3);
```

Notice that the addresses of size 1 parameters must be passed. The parameter type is one of the three strings used above. The default flag is set to 1 if the parameter has a default and 0 if not. Storage for each string is allocated by the routine if the default flag is 0. Otherwise existing storage is freed and reallocated. Thus string parameters must be malloced and not taken from the stack.

4.4 Librnpl

The `RNPL` library contains routines used by the `RNPL` generated programs. These routines include all the routines found in the BBHUtil library as well as others. `RNPL` generated programs should link with `librnpl` only. Other programs that wish to use the HDF or parameter functions should link to `libbbhutil`.

Chapter 5

Utilities

Several utilites are included with the RNPL distribution. These are described in the following sections.

5.1 hdfinfo

`hdfinfo` is a program which gives information about HDF files. It takes one or two parameters. Either just a file name, or `-v` (for verbose) followed by a file name.

5.2 hdftovs

`hdftovs` is a program which reads 1D evolution HDF files and sends them to vs, a data visualization program written by Matthew Choptuik for SGI machines. It takes one arguement, a file name.