Chapter 8. RNPL: The Language

RNPL (Rapid Numerical Prototyping Language) is a language for expressing time dependent systems of partial differential equations and the finite difference methods used to solve them. It was written specifically with the general relativistic evolution problem in mind, but it can also be used to solve a wide variety of differential systems. The language hides many of the details of a complete solver while still allowing enough freedom to express most finite difference techniques. It is based heavily on ideas developed by Matthew Choptuik throughout his work in numerical relativity.

RNPL was designed to provide the following capabilities:

- equation expression using a “natural” notation
- easy operator expression
- support for a wide range of difference techniques
- easy interfacing with existing programs
- automatic memory management
- check-pointing
- interactive output control
- parameter management
- easy adaptivization
- easy parallelization
- extensibility

Operators	Associativity
^ **	right
- + (unary)	right
* /	left
- +	left
> < >= <=	non
== !=	non
&&	left
||	left
=	non

Table 8.1. RNPL operators in order of precedence

- short development time
- easy changing of finite difference methods
- intelligent defaults

RNPL currently provides all these capabilities except adaptivization and parallelization, both of which will be added shortly. Once these features are added, any existing RNPL program can make use of them with a simple recompilation.

8.1. Program Structure

An RNPL program consists of a series of object declarations. RNPL is strongly typed, so all data objects which are referenced must be declared. There are some exceptions to this rule having to do with coordinates (see Section 9.1). Since RNPL is strictly declarative, there are no "executable" statements. Thus, there is no notion of order as in a traditional programming language. Declarations can occur in any order in the source file. The complete RNPL grammar is shown in Figure 8.1.

RNPL is case-sensitive in general, although case will be ignored if the backend language is case insensitive (see Chapter 9). RNPL statements are made up of tokens which fall into

RNPL Grammar

dec_list	→	
	→	*dec_list declaration*
declaration	→	*param_dec*
	→	*coord_dec*
	→	*grid_dec*
	→	*gfunc_dec*
	→	*attrib_dec*
	→	*d_operator*
	→	*residual*
	→	*initialization*
	→	*looper*
	→	*update*
param_dec	→	**param** *p_type name*
	→	**param** *p_type name becomes scalar*
	→	**param** *p_type name v_size*
	→	**param** *p_type name v_size becomes vector*
	→	**param ivec** *name*
	→	**param ivec** *name becomes ivec_list*
	→	**const param** *p_type name*
	→	**const param** *p_type name becomes scalar*
	→	**const param** *p_type name v_size*
	→	**const param** *p_type name v_size becomes vector*
	→	**sys param** *p_type name becomes scalar*
coord_dec	→	*name* **coordinates** *coord_list*
grid_dec	→	*g_type name* **grid** *name i_region c_region*
	→	*g_type name* **grid** *name*
	→	*g_type name* **obrack** *coord_list* **cbrack grid** *name i_region c_region*
	→	*g_type name* **obrack** *coord_list* **cbrack grid** *name*
gfunc_dec	→	*type name* **on** *name*
	→	*type name* **on** *name* **str**
	→	*type name* **on** *name* **at** *o_list*
	→	*type name* **on** *name* **at** *o_list* **alias**
	→	*type name* **on** *name* **at** *o_list* **str**
	→	*type name* **on** *name* **at** *o_list* **alias str**
attrib_dec	→	**attrib** *p_type name encoding*
	→	**attrib** *p_type name encoding becomes vector*
d_operator	→	**operator** *d_op becomes expr*
residual	→	**resid** *name* **obrace** *res_list opcolon* **cbrace**
	→	**resid** *time index name* **obrace** *res_list opcolon* **cbrace**
residual	→	**evaluate resid** *name* **obrace** *res_list opcolon* **cbrace**
	→	**evaluate resid** *time index name* **obrace** *res_list opcolon* **cbrace**
initialization	→	**initialize** *name* **obrace** *res_list opcolon* **cbrace**
looper	→	**looper** *name*
update	→	*name name* **update** *coord_list* **header** *ref_list*

	$\rightarrow$	**stub** *name* **update** *coord_list* **header** *ref_list*
	$\rightarrow$	**auto** *name* **update** *coord_list*
p_type	$\rightarrow$	**int**
	$\rightarrow$	**float**
	$\rightarrow$	**string**
name	$\rightarrow$	**iden**
scalar	$\rightarrow$	**inum**
	$\rightarrow$	**minus inum**
	$\rightarrow$	**num**
	$\rightarrow$	**minus num**
	$\rightarrow$	**str**
v_size	$\rightarrow$	**obrack inum cbrack**
becomes	$\rightarrow$	**assignop**
	$\rightarrow$	**equals**
vector	$\rightarrow$	**obrack** *scalar_list* **cbrack**
ivec_list	$\rightarrow$	*ivel* **minus** *ivel*
	$\rightarrow$	*ivel* **minus** *ivel* **divide inum**
	$\rightarrow$	*ivec_list* **comma** *ivel* **minus** *ivel*
	$\rightarrow$	*ivec_list* **comma** *ivel* **minus** *ivel* **divide inum**
coord_list	$\rightarrow$	*name*
	$\rightarrow$	*coord_list* **comma** *name*
g_type	$\rightarrow$	**uniform**
	$\rightarrow$	**nonuniform**
i_region	$\rightarrow$	**obrack** *expr* **colon** *expr* **cbrack**
	$\rightarrow$	**obrack** *expr* **colon** *expr* **cbrack** *i_region*
	$\rightarrow$	**obrack** *expr* **colon** *expr* **colon inum cbrack**
	$\rightarrow$	**obrack** *expr* **colon** *expr* **colon minus inum cbrack**
	$\rightarrow$	**obrack** *expr* **colon** *expr* **colon inum cbrack** *i_region*
	$\rightarrow$	**obrack** *expr* **colon** *expr* **colon minus inum cbrack** *i_region*
c_region	$\rightarrow$	**obrace** *name* **colon** *name* **cbrace**
	$\rightarrow$	**obrace** *name* **colon** *name* **cbrace** *c_region*
type	$\rightarrow$	**int**
	$\rightarrow$	**float**
o_list	$\rightarrow$	**inum**
	$\rightarrow$	**minus inum**
	$\rightarrow$	*o_list* **comma inum**
	$\rightarrow$	*o_list* **comma minus inum**
encoding	$\rightarrow$	**encodeone**
	$\rightarrow$	**encodeall**
d_op	$\rightarrow$	*name* **oparen** *expr* **comma** *coord_list* **cparen**
	$\rightarrow$	**expand** *name* **oparen** *expr* **comma** *coord_list* **cparen**

expr	→	*expr* **plus** *expr*
	→	*expr* **minus** *expr*
	→	*expr* **equals** *expr*
	→	*expr* **times** *expr*
	→	*expr* **divide** *expr*
	→	*expr* **caret** *expr*
	→	**plus** *expr*
	→	**minus** *expr*
	→	*expr* **equiv** *expr*
	→	*expr* **noteq** *expr*
	→	*expr* **less** *expr*
	→	*expr* **great** *expr*
	→	*expr* **lesseq** *expr*
	→	*expr* **greateq** *expr*
	→	*expr* **and** *expr*
	→	*expr* **or** *expr*
	→	**oparen** *expr* **cparen**
	→	*d_op*
	→	*func*
	→	*gfunc*
	→	*coord*
	→	*name*
	→	**num**
	→	**inum**
res_list	→	*i_region becomes expr*
	→	*res_list* **scolon** *i_region becomes expr*
	→	*i_region becomes ifstat*
	→	*res_list* **scolon** *i_region becomes ifstat*
opcolon	→	
	→	**scolon**
time	→	**time**
index	→	**obrack inum cbrack**
	→	**obrack minus inum cbrack**
	→	**obrack inum cbrack** *index*
	→	**obrack minus inum cbrack** *index*
ref_list	→	*reference*
	→	ref_list **comma** *reference*
scalar_list	→	*scalar*
	→	*scalar_list scalar*
ivel	→	**inum**
	→	**times**
func	→	*name* **oparen** *expr* **cparen**
gfunc	→	*time name mindex*
coord	→	*name indel*

ifstat	$\rightarrow$	**if** *expr* **then** *expr*
	$\rightarrow$	**if** *expr* **then** *expr* **else** *expr*
	$\rightarrow$	**if** *expr* **then** *expr* **else** *ifstat*
reference	$\rightarrow$	*name*
	$\rightarrow$	*name* **obrack** *coord_list* **cbrack**
	$\rightarrow$	**auto work pound inum oparen** *expr* **cparen**
	$\rightarrow$	**static work pound inum oparen** *expr* **cparen**
mindex	$\rightarrow$	*indel*
	$\rightarrow$	*mindex indel*
indel	$\rightarrow$	**obrack inum cbrack**
	$\rightarrow$	**obrack minus inum cbrack**
	$\rightarrow$	**obrace** *expr* **cbrace**

Figure 8.1. RNPL Grammar

four classes—reserved words, names, operators, and punctuation. These tokens are listed in Table 8.2 and the operators are listed again in Table 8.1. White space (space, tab, newline) is meaningless except as a token separator, so it can be used freely in a source file to provide clarity.

There are two kinds of comments in RNPL programs. The first kind is like a UNIX shell comment. It starts with a # at the beginning of a line and continues till the end of the line. The second kind is like a C++ comment. It starts with `//` and ends at the end of the line. The following example illustrates both kinds of comments.

```
# This is the first kind of comment
float A on grid1  // This is the second kind of comment
// So is this
```

8.1.1. Data Objects

There are five kinds of data objects available in an RNPL program: parameters, coordinates, grids, grid functions, and attributes. These objects are in turn made up of scalars, vectors, and index vectors. Scalars are made up of a single integer, float, or string, while vectors are one dimensional arrays of scalars. Index vectors are arrays of triples. The first element of the triple gives a starting index, the second gives an ending index, and the third gives a stride.

Name	**Value(s)**	**Category**
param	`parameter or PARAMETER`	reserved word
ivec	`ivec or IVEC`	reserved word
constant	`constant or CONSTANT`	reserved word
sys	`system or SYSTEM`	reserved word
coordinates	`coordinates or COORDINATES`	reserved word
grid	`grid or GRID`	reserved word
obrack	`[`	punctuation
cbrack	`]`	punctuation
on	`on or ON`	reserved word
str	`"any characters"`	name
at	`at or AT`	reserved word
alias	`alias or ALIAS`	reserved word
attrib	`attribute or ATTRIBUTE`	reserved word
operator	`operator or OPERATOR`	reserved word
resid	`residual or RESIDUAL`	reserved word
obrace	`{`	punctuation
cbrace	`}`	punctuation
evaluate	`evaluate or EVALUATE`	reserved word
initialize	`initialize or INITIALIZE`	reserved word
looper	`looper or LOOPER`	reserved word
update	`update or UPDATE or updates or UPDATES`	reserved word
header	`header or HEADER`	reserved word
stub	`stub or STUB`	reserved word
auto	`auto or AUTO`	reserved word
int	`int or INT`	reserved word
float	`float or FLOAT`	reserved word
string	`string or STRING`	reserved word
iden	`[a-zA-Z_][a-zA-Z_0-9]*`	name
inum	`positive integer`	name
minus	`-`	operator, punctuation
num	`positive real number`	name
assignop	`:=`	punctuation

equals	`=`	operator
comma	`,`	punctuation
divide	`/`	operator, punctuation
uniform	`uniform or UNIFORM`	reserved word
nonuniform	`nonuniform or NONUNIFORM`	reserved word
colon	`:`	punctuation
encodeone	`encodeone or ENCODEONE`	reserved word
encodeall	`encodeall or ENCODEALL`	reserved word
oparen	`(`	punctuation
cparen	`)`	punctuation
expand	`expand or EXPAND`	reserved word
plus	`+`	operator
times	`*`	operator, name
caret	`^ or **`	operator
equiv	`==`	operator
noteq	`!=`	operator
less	`<`	operator
great	`>`	operator
lesseq	`<=`	operator
greateq	`>=`	operator
and	`&&`	operator
or	`||`	operator
scolon	`;`	punctuation
time	`<inum> or <-inum>`	name
if	`if or IF`	reserved word
then	`then or THEN`	reserved word
else	`else or ELSE`	reserved word
work	`work or WORK`	reserved word
pound	`#`	punctuation
static	`static or STATIC`	reserved word

Table 8.2. RNPL Tokens

8.1.1.1. Parameters

Parameters are constants which are specified at run time. They can be scalars, vectors, or index vectors of any type. They are used to specify things like initial data, grid sizes, and output. Here are some example parameter declarations:

```
parameter int fred
constant parameter float jim := 5
parameter float george[10]
parameter float ted[3] := [2.0 1.7 11]
parameter string name := "file_name"
constant parameter string dft[2] := ["comment 1" "comment 2"]
parameter ivec output := *-10,11-50/10,51-*/15
```

As the examples show, a parameter declaration starts with the reserved word `parameter` or the pair of reserved words `constant parameter`. Then comes a type and a name. Parameters can be assigned default values. An RNPL generated program reads a user specified parameter file on startup. If a parameter has been declared without a default value, it must have a value in the file. The reserved word `constant` is meaningless except when the backend language is FORTRAN. There is also a special type of parameter known as the *system* parameter. System parameters only have meaning when the backend language is FORTRAN (see Section 9.2).

The size of a vector parameter must be included in the declaration (see the declarations for `george`, `ted`, and `dft` above). Scalar parameter declarations contain no size.

An ivec is an index vector. The only current use for index vectors is for controlling output. As shown in the example, the index vector can have a default value specified as a comma-separated list of triples. The third element of each triple (the stride) is optional. If it is not given, it is assumed to be one. The first and second elements can be integers or an asterisk. As the first element, an asterisk means *the first time step.* As the second element, an asterisk means *the last time step.* The example declaration above is interpreted to mean “output every time step from the first time step to the tenth time step, every tenth time step from the eleventh to the fiftieth time step, and every fifteenth time step from the fifty-first to the last time step."

8.1.1.2. Coordinates

AN RNPL program can use multiple coordinate systems. A single time coordinate can be used in multiple coordinate systems, but each spatial coordinate must be unique. Some example coordinate declarations are:

```
rect coordinates t,x,y,z
sph coordinates t,r,theta,phi
null coordinates u,v
```

The first coordinate in a list is assumed to be the time coordinate. The first time coordinate is used for *computational* time. The declaration begins with a user-chosen name, followed by `coordinates`, followed by a list of coordinate names.

8.1.1.3. Grids

Grids define the spatial regions over which the grid functions will be defined as well as their storage. A grid declaration can take one of several forms, the longest of which would be something like:

```
uniform rect[x,z] grid g1 [1:Nx][1:Nz] {xmin:xmax}{zmin:zmax}
```

Grids can be `uniform` or `nonuniform`, though only the former is currently defined. Next comes the name of the coordinate system followed by a list of coordinates on which the grid is defined. The above grid is two dimensional with coordinates x and z. After the coordinate system comes the reserved word `grid` followed by the grid name. Next comes the index region. In this example, the first index starts at 1 and goes to `Nx`, while the second starts at 1 and goes to `Nz`. `Nx` and `Nz` are names of grid sizes. The index regions can contain arbitrary expressions such as `[A*B+C-2:4*Nx-5/a]`, however, as discussed in Section 9.1.1, it is best to keep to forms like `[1:Nx]` and `[0:Nx-1]`. Finally, comes the coordinate region which gives the actual spatial ranges of the coordinates. In the example, we have xmin $\leq x \leq$ xmax and zmin $\leq z \leq$ zmax . Coordinate regions must be of the form `{name1:name2}`, where `name1` and `name2` have been declared as parameters.

Other forms of the grid declaration leave out one or more of the above parts. The minimum allowable declaration is:

```
uniform rect grid g2
```

This declaration (along with the example coordinate declaration in Section 8.1.1.2) declares `g2` to be a three dimensional grid with coordinates $x, y,$ and z. The index region will be `[0:Nx-1][0:Ny-1][0:Nz-1]` for C output and `[1:Nx][1:Ny][1:Nz]` for FORTRAN output. The coordinate region will be `{xmin:xmax}{ymin:ymax}{zmin:zmax}`.

8.1.1.4. Grid Functions

A grid function is a function defined on a grid at one or more times. Some examples of grid function declarations are:

```
float A on g1 at -1,0,1
int B on g2 at 0,1
float C on g1
float D on g2 at -1,0,1 alias
float E on g3 at 0,1 "Electric Field"
```

First comes the grid function type, either `float` or `int`. Next comes the name followed by the reserved word `on` and the grid name on which the function is defined. If the declaration stopped here (such as that for `C` above), we get a single time level. Adding the reserved word `at` followed by a list of offsets (positive or negative integers) gives a function defined on one time level for each offset. For instance the definition for `A` would give a three time level function defined at times $n-1$, n, and $n+1$. Next comes the optional reserved word `alias` which declares common storage for the first and last time levels. Following any of these declarations can be a string which is used as a *print name* for the grid function. There are not any real uses for the print name. It simply provides a more descriptive name by which the grid function will be identified during output and visualization (see Chapter 9 for information on these compiler features).

8.1.1.5. Attributes

An attribute is a flag array associated with the grid functions. For instance, an attribute may tell which grid functions are to be output and which are not. Attributes are defined in a similar manner to vector parameters, except the size is replaced by an encoding. The encoding is either `encodeone` or `encodeall`, with `encodeone` giving one value per grid function and `encodeall` giving one value per time level per grid function (see Section 8.1.1.4 for information about defining grid functions). For instance, if five grid functions are defined, three of which have three time levels each while the remaining two have two levels each, then an attribute marked as `encodeone` would have a length of five, while an attribute marked as `encodeall` would have a length of thirteen. In this case an output flag array could be defined as either

```
attribute int out_gf encodeone
```

or

```
attribute int out_gf encodeone := [0 0 1 1 0]
```

8.1.2. Difference Equations

The “heart” of any RNPL program is the definition of the system of equations to be solved and the methods to be used in solving it. This information is declared using derivative operators, residuals, initializations, and updates.

8.1.2.1. Derivative Operators

Derivative operators are operators which act on grid functions. They are used for turning differential equations into finite difference equations by substituting for continuum differential operators. Here is a declaration for a forward difference operator:

```
operator D_FW(f,r) := (<0>f[1] - <0>f[0])/dr
```

Whenever an operator is used in an expression (see Section 8.1.3), it is replaced by its definition. The name `f` is arbitrary. It simply shows where the expression goes in the definition. For instance, if `D_FW(3*A+B,r)` appeared in an expression, the `f`'s in the right hand side would be replaced by `3*A+B`. The notation `<0>f[1]` is interpreted as f^n_{i+1}. The `<>[]` is really an operator which acts on expressions as follows:

`<a>f[b]` $\rightarrow$ `f` if `f` is a number or parameter or time coordinate
`<a>f[b]` $\rightarrow f_{i+b}$ if `f` is a spatial coordinate
`<a>f[b]` $\rightarrow f^{n+a}_{i+b}$ if f is a grid function

Three dimensional forward difference operators would look like this:

```
operator D_FW(f,x) := (<0>f[1][0][0] - <0>f[0][0][0])/dx
operator D_FW(f,y) := (<0>f[0][1][0] - <0>f[0][0][0])/dy
operator D_FW(f,z) := (<0>f[0][0][1] - <0>f[0][0][0])/dz
```

Operator definitions can be nested as in:

```
operator D_FW(f,r) := (<0>f[1] - <0>f[0])/dr
operator D_BW(f,r) := (<0>f[0] - <0>f[-1])/dr
operator D_CN1(f,r,r) := D_BW(D_FW(<0>f[0],r),r)
operator D_CN2(f,r,r) := D_BW(D_FW(<1>f[0],r),r)
```

As you can predict, the definition of `D_CN1` will result in the usual centered second derivative, namely $\left(f^n_{i+1} - 2f^n_i + f^n_{i-1}\right)/dr^2$, while the definition of `D_CN2` will result in the same thing applied at the advanced time level, that is $\left(f^{n+1}_{i+1} - 2f^{n+1}_i + f^{n+1}_{i-1}\right)/dr^2$. The list of coordinate names after the `f` signifies with respect to which coordinate(s) the derivative is taken.

Although operators are defined like derivatives and act as derivatives under certain circumstances (see Section 8.1.3), they can be defined to perform other functions such as the spatial averaging operator defined below.

```
operator AVG(f,r) := (<0>f[1] + <0>f[0])/2
```

Because operators are internally treated as derivatives, even definitions such as this need the coordinate list.

8.1.2.2. Residuals

Residuals define the system of equations, typically by using derivative operators. Consider the following residual definition:

```
residual phi { [0:0]       := D_LF(phi,t) ;
               [1:Nx-2]    := D_LF(phi,t,t) - D_LF(phi,x,x) ;
               [Nx-1:Nx-1] := D_LF(phi,t) }
```

Assuming the proper definitions of the derivative operators, this residual encodes the linear wave equation on a string with the end points fixed. An equivalent declaration would be:

```
residual phi { [0:0]       := D_LF(phi,t) = 0 ;
               [1:Nx-2]    := D_LF(phi,t,t) = D_LF(phi,x,x) ;
               [Nx-1:Nx-1] := D_LF(phi,t) = 0 }
```

First comes the reserved word `residual` followed by the name of the grid function whose residual is being defined. Next comes a bracket-enclosed set of index regions and expressions. The index region shows over what range the expression is a valid description of the behavior of the system. The union of the index regions should equal the index region of the grid on which the function is defined, but this is not required. Note that each region-expression pair is separated by a semicolon. A semicolon can also follow the last expression but is not required.

An alternate form for an index region is `[expr1:expr2:stride]`, where `expr1` and `expr2` are the region bounds as above, and `stride` is an integer stride which can be positive or negative. In the first form, the stride is taken to be one.

The residual tells how to determine the advanced value of a grid function. Thus, the residual must contain `<a>f...` where `a` is the offset to the most advanced time level defined for grid function `f`.

Part of a residual for a three dimensional grid function would look like:

```
residual A { [1:Nx][1:1][1:1] := D_LF(A,x) + D_FW(B,y) ;
          [Nx:Nx][1:Ny][1:Nz] := <1>A[0][0][0] = 5.0*C }
```

The reserved word `residual` can be preceded by the reserved word `evaluate` which tells the compiler to produce code which will evaluate the residual.

In addition, the word `residual` can be followed by a global offset for example:

```
residual <1>[0] A { [1:Nx] := ... }
```

This offset is applied globally to each expression appearing in the residual.

Residuals also support the if-then-else construct. Consider the following declaration:

```
residual A { [1:Nx] := if(<0>C[0] == 1 && <0>C[-1] == 1) then
                         D_LF(A,t) = D_LF(A,x)
                       else if(C == 1) then
                         D_LF(A,t) = D_FW(A,x)
                       else D_LF(A,t) = 0 }
```

Here `C` is a characteristic function which tells which equation belongs in which region.

8.1.2.3. Initializations

An initialization defines the initial data for a grid function. Its form is identical to the residual declaration with `residual` replaced by `initialize`. However the expression is interpreted differently. Consider the following initialization declaration:

```
initialize phi { [1:Nr] := amp*exp(-((r-c)/delta)^2) }
```

Unlike the residual declaration, the expression in the initialization must not contain its grid function (though it can contain other grid functions). The retarded time level of the grid function is set to the expression. In the case above, `phi` will be set to a Gaussian. Initializations are evaluated in the order in which they are defined. Thus, if one grid function is used in the initialization for another, it must be initialized first.

8.1.2.4. Updates

Update declarations can take many forms. Here are some examples:

```
auto update phi,pi,beta
stub evolver updates A,B,C
  header A, B[Bnp1,Bn,Bnm1], C[C], x,y,z,dt,
         auto work#0(5*Nx*Ny*Nz),
         static work#1(3*Nx*Ny-.5*Nz)
myroutine.inc myupdate update A,B header A,B,dt
```

The first form defines an automatic update. The declaration above would cause the compiler to produce a routine to update the grid functions `phi`, `pi`, and `beta` if residuals have been declared for them. Otherwise, the compiler has no idea how to update the grid functions and will produce an error message. Grid functions are updated in the order they are listed in the update declaration.

The second declaration will cause the compiler to produce the header for a routine called `evolver` which is expected to update grid functions `A`, `B`, and `C`. The body of the routine is left blank, to be filled in by the user. Following the reserved word `header`, comes a list of things to appear in the calling sequence for the function. A grid function name such as `A` above will cause all the time levels of `A` to be passed to the function. If `A` has three time levels (1, 0, -1), then they will be named `A_np1`, `A_n`, and `A_nm1` by default. The user can provide his own names to override the defaults as in the case of `B`. If only one name is provided (as for `C`), the time levels will be passed in as the elements of a single vector, the first component of which will be the *advanced* time level.

Other things that can appear in the header list are coordinates (such as `x`, `y`, and `z` above) and coordinate differentials (such as `dt`). Parameters can also be included in the list. Work arrays are declared like the final two parameters. First comes `auto` or `static`. Static work arrays are declared at the start of the program and persist throughout. Auto work arrays are allocated before the call to the update routine and are destroyed afterwards. Next comes the word `work` followed by the # symbol and an integer. This integer is tacked onto the end of the word work to form the name of the array. Finally comes an expression for the size of the work array enclosed in parentheses.

The thrid declaration is much like the second, except the body of the update routine is taken from the file named `myroutine.inc`.

8.1.2.5. The Loop Driver

There is one remaining kind of declaration—the *loop driver*. This declaration defines the overall method used for solving the equations. It consists of the reserved word `looper` followed by a name. The currently defined loop drivers are `standard` and `iterative`.

Declaring the `standard` loop driver means that the update routines will be called once for each time step. This driver can be used for a fully explicit system or when a user-written, external update routine is called.

Declaring the `iterative` loop driver means that the update routines will be called from a loop which first makes an initial guess at the advanced values and then continues to call the update routines until the norm of the residual is below a certain threshold. This driver is useful for implicit schemes.

Future loop drivers will may include a full Newton iteration, and will definitely include various adaptive options.

8.1.3. Expressions

Expressions are made up of objects separated by operators (see Table 8.1 for a list of operators in order of precedence). Objects include numbers, identifiers, functions, derivative operators, coordinates, and grid functions.

Numbers can be integer or floating point. Floating point numbers can be written with exponents as in `1.0e-3`. Identifiers are strings beginning with a letter or an underscore and containing letters, digits, or underscores. Identifiers may be names of grid functions, coordinates, or parameters.

A function is a function name followed by a parentheses-enclosed expression, such as `cos(3*r)`. The only currently recognized function names are: exp, log, tan, sin, cos, sinh, cosh, tanh, and sqrt.

A derivative operator is a name followed by an opening parenthesis, an expression, a coordinate list, and a closing parenthesis. It may also begin with the reserved word `expand` which tells RNPL to symbolically expand the derivative before making the operator substitution. For instance, `expand D_(a*b + c,r)` would expand to `D_(a,r)*b + a*D_(b,r) + D_(c,r)`.

A coordinate is a coordinate name followed by a spatial offset. For example,

```
r[1]
x[-5]
y{a*b+5}
```

are valid coordinates. The bracket-enclosed offsets are interpreted as in Section 8.1.2.1. That is, `r[1]` becomes r_{i+1}. The object inside the brackets must be an integer. The brace-enclosed expression is interpreted as an absolute index. That is, `y{a*b+5}` becomes y_{a*b+5}. The object in the braces is an arbitrary expression. If a coordinate name appears in an expression without a following offset, it will still be recognized as a coordinate and will be given an offset of zero.

A grid function is much like a coordinate except the name is preceded by a temporal offset in angle brackets and is followed by one spatial offset for each dimension. The following are examples of grid functions.

```
<0>A[0][-1]
<-1>B[1]{b+2}[0]
```

Also like a coordinate, a grid function name can appear without offsets.

There are three "types" of expressions possible in RNPL. A type 1 expression is completely arbitrary. A type 2 expression is a type 1 expression that contains no logical operators. A type 3 expression is a type 2 expression that contains no derivative operators, grid functions, or coordinates. Table 8.3 shows which expression types can be used in which areas.

8.2. Examples

Since languages are best learned by example, I will present two which illustrate most of RNPL's features.

8.2.1. 3D Wave Equation

Consider the linear wave equation in three dimensions. This is an initial-value, boundary-value problem which can be stated as follows:

$$\partial_t^2\phi = \partial_x^2\phi + \partial_y^2\phi + \partial_z^2\phi$$

$$\phi(x_{\min}, y, z, t) = 0$$

$$\phi(x_{\max}, y, z, t) = 0$$

$$\phi(x, y_{\min}, z, t) = 0$$

$$\phi(x, y_{\max}, z, t) = 0$$

$$\phi(x, y, z_{\min}, t) = 0$$

$$\phi(x, y, z_{\max}, t) = 0$$

$$\phi(x, y, z, 0) = A\exp\left(\left(x - c_x\right)^2/\delta_x^2\right)\exp\left(\left(y - c_y\right)^2/\delta_y^2\right)\exp\left(\left(z - c_z\right)^2/\delta_z^2\right)$$

Location	Type	Example
if statement	1	if (expr) then
residual	2	residual A { [1:1] := expr }
initialization	2	initialize A { [1:1] := expr }
absolute index	2	x{ expr }
index region	3	residual A { [1: expr]

Table 8.3. RNPL Expression Types

where $x_{\min} \le x \le x_{\max}$, $y_{\min} \le y \le y_{\max}$, and $z_{\min} \le z \le z_{\max}$.

Typically one would also specify $\dot{\phi}$, but RNPL doesn't allow this (see Section 9.4 for a full discussion of the RNPL initial data problem). To set this problem up with RNPL we must identify our requirements. We need one grid function, ϕ. We need difference operators for ∂_t^2, ∂_x^2, ∂_y^2, and ∂_z^2. We 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, $x_{\min}$, $x_{\max}$, $y_{\min}$, $y_{\max}$, $z_{\min}$, and $z_{\max}$.

We begin this RNPL program by specifying the parameters. The 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
```

Default values are optional.

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

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

The name `rect` is arbitrary, but should be descriptive.

Since we need a grid function, we must define a grid.

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

As stated in Section 8.1.1.3, the index and spatial ranges will be automatically defined if we leave them out.

We 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 come 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 ϕ, 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 ϕ.

```
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
```

8.2.2. "Shifted" Wave Equation

As a slightly more complicated example, let's consider the "shifted" wave equation in one dimension with periodic boundary conditions. We'll take the shift β to be a constant and the initial field configuration ϕ to be a left-moving Gaussian pulse. This problem can be stated as follows:

$$\partial_t^2\phi(x,t) = (1-\beta^2)\,\partial_x^2\phi(x,t) - 2\,\beta\,\partial_t\partial_x\phi(x,t)$$

$$\phi(x_{\min},t) = \phi(x_{\max},t)$$

$$\phi(x,0) = A\exp\left(-(x-c)^2/\mathit{\Delta}^2\right)$$

$$\partial_t\phi(x,0) = \frac{-2(x-c)}{\mathit{\Delta}^2}A\exp\left(-(x-c)^2/\mathit{\Delta}^2\right)$$

$$\beta(x) = 0.5$$

where $x_{\min} \le x \le x_{\max}$.

We can rewrite this equation in first order form by introducing the two auxiliary variables $\mathit{\Phi}$ and $\mathit{\Pi}$ defined by:

$$\mathit{\Phi} \equiv \partial_x\phi\,,$$

$$\mathit{\Pi} \equiv \partial_t\phi - \beta\partial_x\phi\,.$$

In terms of these variables, the problem becomes:

$$\partial_t\mathit{\Phi}(x,t) = \partial_x(\beta\,\mathit{\Phi} + \mathit{\Pi})$$

$$\partial_t\mathit{\Pi}(x,t) = \partial_x(\beta\,\mathit{\Pi} + \mathit{\Phi})$$

$$\mathit{\Phi}(x,0) = \frac{-2(x-c)}{\mathit{\Delta}^2}A\exp\left(-(x-c)^2/\mathit{\Delta}^2\right)$$

$$\mathit{\Pi}(x,0) = \frac{-(x-c)}{\mathit{\Delta}^2}A\exp\left(-(x-c)^2/\mathit{\Delta}^2\right)$$

$$\Phi\left(x_{\min}, t\right) = \Phi\left(x_{\max}, t\right)$$

$$\Pi\left(x_{\min}, t\right) = \Phi\left(x_{\max}, t\right)$$

The RNPL program to solve this problem is shown below. It uses a two-level Crank-Nicholson difference scheme with numerical dissipation.

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

# with constant shift and periodic boundary conditions


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

float phi on g1 at 0,1


operator D_PER(f,x) := (<1>f[1] - <1>f{Nx-2} + <0>f[1] - <0>f{Nx-2})/(4*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 D_CNDP1(f,t) := (<1>f[0] - <0>f[0] +
  epsdis/16*(6*<0>f[0] + <0>f{Nx-3} + <0>f[2] -4*(<0>f{Nx-2} + <0>f[1])))/dt
operator D_CNDP2(f,t) := (<1>f[0] - <0>f[0] +
  epsdis/16*(6*<0>f[0] + <0>f{Nx-2} + <0>f[2] -4*(<0>f[-1] + <0>f[1])))/dt
operator D_CNDP3(f,t) := (<1>f[0] - <0>f[0] +
  epsdis/16*(6*<0>f[0] + <0>f[-2] + <0>f{1} -4*(<0>f[-1] + <0>f[1])))/dt
operator AVG(f,t) := (<1>f[0] + <0>f[0])/2

evaluate residual Phi { [0:0] := D_CNDP1(Phi,t) = D_PER(beta*Phi + Pi,x);
                        [1:1] := D_CNDP2(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_CNDP3(Phi,t) = D_CN(beta*Phi + Pi,x);
                  [Nx-1:Nx-1] := <1>Phi[0] = <1>Phi{0} }

residual Pi { [0:0] := D_CNDP1(Pi,t) = D_PER(beta*Pi + Phi,x);
              [1:1] := D_CNDP2(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_CNDP3(Pi,t) = D_CN(beta*Pi + Phi,x);
        [Nx-1:Nx-1] := <1>Pi[0] = <1>Pi{0} }

residual phi { [0:Nx-1] := D_CN(phi,t) = AVG(Pi + beta*Phi,t) }

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]:= -(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, phi
```

Notice that the periodic boundary conditions are enforced in the operators and the residuals by using an absolute index. Without this construct, such a boundary condition is impossible to implement except for a fixed size grid.

Chapter 9. RNPL: The Compiler

I have written a compiler for RNPL. This compiler is a complete implementation of the language as defined in Chapter 8. As new constructs and features are added to the language, the compiler will be updated to include support for them.

Other authors [23] have argued that compilers for numerical languages are best written in a language such as Maple or Mathematica because of their in-built symbolic manipulation capability. It is my opinion that compilers (including ones for math-oriented languages such as RNPL) should be written in a traditional programming language such as C in combination with compiler writing tools such as lex and yacc (readers interested in using lex and yacc should consult [16]).

I first planned to implement the RNPL compiler in Maple. However, it soon became clear that such a language is not suited to the task. On the one hand, RNPL does require some symbolic manipulation (simple algebra and differentiation), a task which Maple can perform well. On the other hand, compilers require complex data structures and excellent text handling, both tasks which are better suited to a general purpose programming language. I decided to write the compiler in C, using lex and yacc for the parser. I wrote my own symbolic manipulation routines. As it turned out, the time spent implementing the symbolic capabilities was **much** less than the time spent on code generation, thus justifying my decision.

It can certainly be argued that C++ would be better suited to the task than C, however I decided to use C due to its wider availability on the target platforms. A future version of the compiler may use C++.

9.1. Compiler Assumptions

As discussed in Chapter 8, RNPL has no “executable” statements. Because of this, it is up to the compiler to decide how to translate the source into an executable program (or portion thereof). In this way, the user is shielded as much as possible from the details of programming and can concentrate on the equations. In order to successfully generate a program, the compiler must make certain assumptions which place some constraints on the form of the source. Although some of these assumptions seem to violate the strong typing rules of the language, in actuality, they simply move the responsibility for some declarations from the user to the compiler. That is, all data objects are still declared, just some are declared automatically. If a name is multiply defined, the first definition will be kept and subsequent ones will be discarded. If the objects with the same name are of the same type, for instance two parameters named X, then no error will be reported. However, if X is first declared to be a parameter and later redeclared to be a grid function, the compiler will report an error.

9.1.1. Coordinates and Differentials

Coordinate differentials are automatically defined by the compiler. They are given the same names as the coordinates except they begin with a lower case d. Thus, if t and x are coordinates, dt and dx will be coordinate differentials. At run time the coordinate differentials are assigned values based on the grids. For instance, if an RNPL program contains the following definitions:

```
rectangular coordinates t,x,y


uniform rectangular grid G [1:Nx][0:Ny-1]
                         {xmin:xmax}{ylow:yhigh}
```

then the spatial coordinate differentials will be assigned values using

$$dx = \frac{xmax - xmin}{Nx - 1}$$

and

$$dy = \frac{yhigh - ylow}{Ny - 1},$$

while the time coordinate differential will be assigned by

$$dt = \lambda\sqrt{\frac{dx^2 + dy^2}{2}},$$

where λ is the Courant factor (see below).

When there is more than one grid defined, the compiler uses the first grid which uses a given coordinate to define that coordinate's differential. If along with the above definitions, an RNPL program contains

```
uniform rectangular[x] grid g [1:N] {min:max}
```

then dx will still be defined as above. The user must construct his own grid spacing variable for grid g. On the other hand, if the definition for g came before the definition for G, dx would be assigned from g by $dx = (\text{min} - \text{max})/(N - 1)$, while dy would still be defined from G, since that is the first grid which uses y.

Along with the coordinate differentials, the grid sizes are defined automatically. These are named like the differentials except with an initial upper case N instead of the d. Also like the differentials, the grid sizes are not assigned values by the user. Their values are calculated from the *grid base* parameters (see below). Although the user can use any expression he likes to define the index regions, using the grid sizes is best since it allows for easy convergence testing by changing only the *level* parameter (see below).

9.1.2. "Special" Parameters

There are several parameters which are needed by every RNPL program. Instead of forcing the user to always define these, the compiler takes care of them. These parameters are listed in Table 9.1. Most of their uses are obvious. The others will be explained here or in the following sections.

The *level* parameter and the grid bases can be used together for easy convergence testing. Assume we have defined the coordinates t, x, y, and z. Then at run time, the grid sizes will be set as follows:

Name	**Type**	**Default**	**Use**
start_t	float	0	start time
s_step	int	0	starting iteration
iter	int	100	number of iterations
epsiter	float	1.0e-5	iteration threshold
fout	int	0	file output (0 no, 1 yes)
ser	int	0	fs output (0 no, 1 yes)
lambda	float	.5	Courant factor
output	ivec	*-*/1	output control
in_file	string	none	name of file from which initial data will be read
out_file	string	none	name of file to which final state will be written
level	int	0	refinement level
tag	string	""	prepend symbol for grid function names
N*c*0	int	0	base number of grid points for the coordinate named *c* (there is one for each spatial coordinate)

Table 9.1. Special Parameters

$$Nx = Nx0 \cdot 2^{level} + 1$$

$$Ny = Ny0 \cdot 2^{level} + 1$$

$$Nz = Nz0 \cdot 2^{level} + 1.$$

That is, if *Nx0* is 100 and *level* is 0, then *Nx* will be set to 101. If *level* is 1, then *Nx* will be set to 201. If any grid base is an odd number, then the above procedure will not give proper lengths for a convergence series. In this case, the grid size will be set to the grid base and a warning message will be printed. This way, a user can do a single run with an arbitrary grid size.

If a grid has been declared without using the grid sizes, then its size will remain unaffected by changes to *level*.

9.2. Language Modifications

Feedback from use by myself and others has resulted in several modifications of the initial language design which make it more effective. These modifications are discussed below.

One of the initial design goals for RNPL was to allow users to easily convert existing programs to RNPL so they could take advantage of the built-in features. Initially, this was going to be done by having the user directly edit the RNPL-generated code. This proved too burdensome, so the update declaration was modified to allow automatic header generation and code inclusion.

In order to have the compiler output FORTRAN 77, I had to add *constant* parameters and *system* parameters. It was also necessary to either make RNPL completely case-insensitive, or case-insensitive when producing FORTRAN output. I chose the latter option.

Constant parameters are needed because of FORTRAN's lack of support for globals. They behave exactly like regular parameters except they can not be passed to update routines or residual evaluators in the argument list. They are placed in a separate common block from the other parameters and globals and are declared in every function.

System parameters are needed because of FORTRAN's lack of support for dynamically allocated memory. There is currently only one system parameter defined—`memsiz`. This parameter has the default value of 2000000. It is the size of the FORTRAN program's heap in doubles. It can be changed with a declaration like:

```
system parameter int memsiz := 8000000
```

A few syntactic changes were made to RNPL at user's requests. These were support for all capital reserved words, the # in the work array declaration, and the optional semicolon to end a residual block.

9.3. Equation Solver

The primary job of an RNPL program is to solve a system of evolution equations. The equations are specified through the residuals and it is up to the compiler to decide how to solve them. The solution is generated by symbolically solving each grid function's residual for its advanced value.

This method is exactly what is required for solving explicit schemes. Using the iterative loop driver, one can use this method to solve implicit schemes as well. Such an iterative method may in fact be sufficient for all implicit schemes, though some early attempts to use this method on the fully-averaged implicit scheme failed to converge.

In the future, I will add support for a global Newton iteration which should solve any implicit scheme. Although such a scheme is no more difficult to support than the current scheme (from the compiler's point of view), it requires a banded matrix solver. Due to the large number of architectures that RNPL supports (these include serial, vector, and massively parallel machines), I decided to wait on the matrix routines until it was clear that they were needed.

9.4. Initial Data Generation

The weakest part of the compiler is its initial data support. It only allows analytic initial data to be specified. However, the initial data problem is very different from the evolution problem that RNPL was designed to solve. If not known analytically, initial data is usually solved for from elliptic equations. Techniques to solve elliptic equations are very different from techniques used to solve hyperbolic equations.

There are also problems with initial data that are inherent to finite differencing. For instance, assume we are attempting to solve the one dimensional linear wave equation in flat space. This equation is $\partial_t^2\phi = \partial_x^2\phi$. The initial state of the system is specified by giving $\phi\left(x,0\right)$ and $\dot{\phi}\left(x,0\right)$.

In RNPL, we can only give $\phi\left(x,0\right)$. If we wish to use a three-level scheme to solve this equation, then we need two time levels of initial data. RNPL will find the second time level through an iterative procedure using the evolution equations. This will result in time-symmetric initial data which contains both an ingoing and an outgoing piece. If this is not what we want, we must modify the compiler-generated initial data generator to produce our own data.

However, assume we wish to use a two-level scheme. In this case we can still only give $\phi(x,0)$. However, that is all the data we can possibly give. This is a problem with using the two-level scheme. It does not capture both degrees of freedom available in the physical problem.

Now assume we decide to rewrite the problem in first order form. The equations of motion are $\partial_t \Phi = \partial_x \Pi$ and $\partial_t \Pi = \partial_x \Phi$, with $\Phi \equiv \partial_x \phi$ and $\Pi \equiv \partial_t \phi$. Now using a two level scheme we can specify one time level for each function. This allows us to completely specify the initial configuration.

A three-level scheme for this problem will require four levels of initial data, two for each function. RNPL will only allow us to specify two levels and will solve for the other two using an iterative procedure. In this case, unlike the second order problem above, this is the correct way of generating the extra time levels since the physical problem only contains two degrees of freedom.

These examples show that RNPL will correctly handle the specification of analytic initial data for systems written in first order form and using two or three-level finite difference schemes for solution. A simple modification to RNPL would allow the user to specify multiple time levels of initial data for each grid function. This would allow RNPL to correctly handle initial data for systems written in higher order form.

9.5. Parameter Files

At run time, RNPL generated programs look for a parameter file. This is an ASCII file consisting of arbitrary text. The RNPL program will scan this file for lines of the form `name := value`, where `name` is the name of a parameter, and `value` is the value of the parameter given in the same form as default values in RNPL source. If the program finds such a line for a parameter, it will set that parameter to the value in the file. If not, it will use the default value for the parameter. If the parameter has no default, the program will exit with an error message.

Here is an example of a parameter file:

```
This is a parameter file
tag := "a_"
level := 1
Nx0 := 100
xmin := -10
This is a comment
in_file := "init_data.hdf"
out_file := "dump.hdf"
```

9.6. Output Control

RNPL generated programs allow the user to dynamically control output during execution. At any time, the user can interrupt the program and turn output on or off for any grid function, as well as control the frequency of output.

By default, output is controlled by three parameters and an attribute. The parameter *fout* can be set to zero or one. If it is set to one, then output is sent to HDF files. If the parameter *ser* is set to one, then output is sent to Matthew Choptuik's *vs* graphics server. Since *vs* only handles one dimensional functions, only one dimensional grid functions will be sent from the program.

The parameter *output* is an index vector. It controls when the program generates output. For example, if the interesting dynamics occur after iteration 100, output could be set like:

```
output := *-100/50,101-150/2,151-*/50
```

This would output only once every 50 time steps during the first part of the calculation, every other time step during the dynamic part, and then every 50 time steps till the end. The initial * is set to `s_step` and the final * is set to `iter`.

There is one automatically defined attribute. It controls which grid functions are output. It is called `out_gf` and has encoding `encodeone`. Any of its elements can be set to one to enable output for that grid function or to zero to disable output.

9.7. Check Pointing

Check pointing provides a way for calculations to be stopped during execution and then restarted at a later time without losing any information. All RNPL generated programs automatically dump state upon completion. If a calculation needs to be stopped during execution, it can be interrupted and told to dump state. The state is saved in the file whose name is given by the parameter `out_file`. The state file is identical in format to an initial data file. Thus, to restart, the user simply has to set the parameter `in_file` to the name of the state file and rerun the program. The parameters `start_t` and `s_step` are read from the state file, so their values in the parameter file do not need to be changed.