Code `OllinToy`. User's manual

Miguel Alcubierre
Instituto de Ciencias Nucleares, UNAM
malcubi@nucleares.unam.mx

April, 2005

Contents

1 Introduction

The program `OllinToy` solves the Einstein evolution equations in the case of 1+1 dimensions and in vacuum. It is well known that in this case there are no true dynamics and all the evolution is pure gauge. Still, this is very useful to test gauge conditions.

The purpose of the code is therefore to be a platform for both studying the effect of some simple gauge conditions, and learning the basics of different numerical methods. The code will then continue to grow as we have more ideas to try out, but hopefully it will remain manageable, readable and well documented. If you plan on editing the code to add some new functionality, have a look at Sec. 18.

If you find that something is missing from this documentation (or is not clear) please let me know. Some sections ARE missing and I know about it, for example I still need to explain how the code tracks characteristics, and how the harmonic shift works.

2 Directory structure

The main directory for the code is `OllinToy`. There are several subdirectories inside this main directory:

`CVS`	Contains information about the CVS root and server (see Sec. 18).
`doc`	Contains the documentation.
`exe`	Created at compile time and contains the executable file.
`objs`	Created at compile time and contains all the object files.
`par`	Contains examples of parameter files (see Sec. 4 below.)
`prl`	Contains perl scripts used at compile time to create the subroutines that manage parameters and arrays.
`src`	Contains the source files for all the code routines.
`testdata`	Contains parameter files and data for testsuites.

3 Compiling and running the code

The code is written in `FORTRAN 90`. All subroutines are in separate files inside the subdirectory `src`.

3.1 Compiling

To compile just move inside the `OllinToy` directory and type:

```
gmake
```

This will create a series of object files for each routine inside the subdirectory `objs`. The makefile will then create a directory `exe` and will place in it the final executable file called `ollintoy`. It will also copy to this directory all the parameter files inside the subdirectory `par`.

Notice that at this point the code assumes that if you are working on a Linux platform you are using the free Intel F90 compiler, and that this compiler is called `ifc`. If you are working on a Mac, the code assumes that you have the IBM compiler and it is called `f90`. If you have a different compiler then the Makefile will have to be modified (hopefully it won't be very difficult). And if you are working on Windows you are on your own (and why on Earth are you working on Windows and pretending to use a real code anyway?).

The object files and the executable file can be deleted by typing in the main directory:

```
gmake clean
```

If one wants to delete only the object files one needs to type:

```
gmake cleanobjs
```

And if one wants to delete the subdirectories `exe` and `objs` with ALL their contents one can type:

```
gmake veryclean
```

Be VERY careful with this last option, as you will end up deleting all your data files if you left them inside the directory `exe`.

3.2 Running

To run the code move into the directory `exe` and type:

```
./ollintoy name.par
```

Where `name.par` is the name of your parameter file (more on parameter files below). The code will then read data from the parameter file silently and hopefully start producing some output to the screen. The code will also create an output directory and will write the data files on that directory.

4 Parameter files

At runtime, the code reads the parameter values from a parameter file (parfile) with a name of the form `name.par` that must be specified in the command line after the executable:

```
./ollintoy name.par
```

The data in this parameter file can be given in any order, using the format:

```
parameter = value
```

Comments (anything after a `#`) and blank lines are ignored. Only one parameter is allowed per line, and only one value is allowed per parameter, with the exception of the parameters `outvars0D` and `outvars1D` that control which arrays get output and take lists of arrays as values, for example:

```
outvars0D = alpha,A,B
```

Parameters that do not appear in the parfile get the default values given in the file `param.f90`. Example parameter files can be found in the subdirectory `par`. A list of all currently available parameters can be found in Sec. 16.

IMPORTANT: Even though `FORTRAN` does not distinguish between upper and lower case in variable names, the names of parameters are handled as strings by the parameter parser, so lower and upper case are in fact different. The name of parameters in the parameter file should then be identical to the way in which they appear in the file `param.f90`.

Example parameter files can be found in the subdirectory `par`.

5 Output files

At runtime, the code creates an output subdirectory whose name should be given in the parameter file. It then produces a series of output files with the data from the run. There are so called "0D" files (with extension `*.tl`) and "1D" files (with extension `*.xl`).

The `0D` files refer to scalar quantities obtained from the spatial arrays as functions of time. These scalar quantities include the maximum (`max`), the minimum (`min`) and two different norms of the spatial arrays: maximum absolute value (`nm1`) and root mean square (`nm2`). The code also outputs the total variation (`var`) of a given variable, defined as the sum of the absolute values of the differences between adjacent grid points.

The `1D` files contain the complete spatial arrays at different times. Even though the code is only one-dimensional, these files can become quite big if you are not careful, so beware.

All files are written in ASCII, and using a format adapted to XGRAPH (but other graphic packages should be able to read them).

Output is controlled by the following parameters:

`directory`	Name of directory for output.
`Ninfo`	How often do we output information to screen?
`Noutput0D`	How often do we do 0D output?
`Noutput1D`	How often do we do 1D output?
`outvars0D`	Arrays that need 0D output (a list separated by commas).
`outvars1D`	Arrays that need 1D output (a list separated by commas).

6 Spacetime and evolution equations

The spacetime metric used in the code has the following form:

$$ds^2 = \left(-\alpha + g\beta^2\right)\, dt^2 + 2\, g\beta\, dx\, dt + g\, dx^2 \ , \tag{6.1}$$

where α is the lapse function, $\beta \equiv \beta^x$ the contravariant shift and g the spatial metric.

As slicing condition the code uses the Bona-Masso family of generalized harmonic slicing conditions [6]:

$$\partial_t \alpha - \beta\, \partial_x \alpha = -\alpha^2 f(\alpha) K \ , \tag{6.2}$$

with $K = K^x_x$ the trace of the extrinsic curvature and $f(\alpha)$ a positive but otherwise arbitrary function of α.

The evolution equations used in the code are the ADM [5, 10] equations in 1+1 dimensions, which can be written in first order form as

$$\begin{aligned} \partial_t \alpha &= -\alpha^2 f K + \alpha\beta D_\alpha \ , & (6.3)\\ \partial_t g &= -2\alpha g K + g\beta D_g + 2g\, \partial_x \beta \ , & (6.4) \end{aligned}$$

and

$$\begin{aligned} \partial_t D_\alpha + \partial_x \left(\alpha f K - \beta D_\alpha\right) &= 0 \ , & (6.5)\\ \partial_t D_g + \partial_x \left(2\alpha K - \beta D_g - 2\, \partial_x \beta\right) &= 0 \ , & (6.6)\\ \partial_t K - \beta\, \partial_x K + \partial_x \left(\alpha D_\alpha / g\right) &= \alpha \left(K^2 - D_\alpha D_g / 2g\right) \ , & (6.7) \end{aligned}$$

where we have defined $D_\alpha := \partial_x \ln \alpha$ and $D_g := \partial_x \ln g$.

The code in fact does not use K itself, but rather $\tilde{K} := g^{1/2} K$, as with that rescaling of the extrinsic curvature the evolution equations for the first order variables $\vec{v} := (D_\alpha, D_g, \tilde{K})$ can be written as a conservative system of the form

$$\partial_t \vec{v} + \partial_x \left(\mathbf{M}\, \vec{v}\right) = 2 \left(\partial_x^2 \beta\right) \hat{e}_g \ , \tag{6.8}$$

where $\hat{e}_g = (0, 1, 0)$. Notice that there is a source term in the equation for D_g, but this source term is a total derivative, so the system is still formally flux conservative (one can always add this source to the flux). The characteristic matrix $\mathbf{M}$ is given by:

$$\mathbf{M} = \left(\begin{array}{ccc} -\beta & 0 & \alpha f / g^{1/2} \\ 0 & -\beta & 2\alpha / g^{1/2} \\ \alpha / g^{1/2} & 0 & -\beta \end{array} \right) \ . \tag{6.9}$$

The last matrix has the following eigenvalues

$$\lambda_0 = -\beta \; , \qquad \lambda_\pm = -\beta \pm \alpha \, (f/g)^{1/2} \; , \tag{6.10}$$

with corresponding eigenvectors

$$\vec{\xi}_0 = (0,1,0) \; , \qquad \vec{\xi}_\pm = \left(f, 2, \pm f^{1/2} \right) \; . \tag{6.11}$$

Since the eigenvalues are real for $f > 0$ and the eigenvectors are linearly independent, the system (6.8) is strongly hyperbolic.

The code does not evolve the above equations directly, but rather evolves the evolution equations for the "eigenfields" which are defined as

$$\omega_0 = D_\alpha / f - D_g / 2 \; , \qquad \omega_\pm = \tilde{K} \pm D_\alpha / f^{1/2} \; . \tag{6.12}$$

These equations can be easily inverted to give

$$\begin{aligned} \tilde{K} &= \frac{(\omega_+ + \omega_-)}{2} \; , & (6.13) \\ D_\alpha &= \frac{f^{1/2} \; (\omega_+ - \omega_-)}{2} \; , & (6.14) \\ D_g &= \frac{(\omega_+ - \omega_-)}{f^{1/2}} - 2\omega_0 \; . & (6.15) \end{aligned}$$

With the eigenfunctions defined as above, their evolution equations also turn out to be conservative and have the simple form:

$$\partial_t \vec{\omega} + \partial_x \left(\mathbf{\Lambda} \, \vec{\omega} \right) = -2 \left(\partial_x^2 \beta \right) \hat{e}_{w_0} \; , \tag{6.16}$$

with $\mathbf{\Lambda} = \text{diag} \left\{ \lambda_0, \lambda_+, \lambda_- \right\}$, $\vec{\omega} = (\omega_0, \omega_+, \omega_-)$ and $\hat{e}_{w_0} = (1,0,0)$. These are the actual equations coded.

The evolution equations for the original variables can later be easily recovered from ($f' := df/d\alpha$):

$$\begin{aligned} \partial_t \tilde{K} &= \frac{1}{2} \; (\partial_t \omega_+ + \partial_t \omega_-) \; , & (6.17) \\ \partial_t D_\alpha &= \frac{f^{1/2}}{2} \; (\partial_t \omega_+ - \partial_t \omega_-) + \frac{f'}{2f} \, D_\alpha \, \partial_t \alpha \; , & (6.18) \\ \partial_t D_g &= \frac{1}{f^{1/2}} \; (\partial_t \omega_+ - \partial_t \omega_-) - 2 \, \partial_t \omega_0 - \frac{f'}{f^2} \, D_\alpha \, \partial_t \alpha \; . & (6.19) \end{aligned}$$

7 Slicing condition

As mentioned above, the slicing condition used in the code is the Bona-Masso condition

$$\partial_t \alpha - \beta \, \partial_x \alpha = -\alpha^2 f(\alpha) K \; , \tag{7.1}$$

with $f > 0$.

Two parameters control the choice of the function $f(\alpha)$: "slicing" and "gauge_f". The parameter "gauge_f" is a real constant, and the parameter "slicing" is a character type parameter that can have one of the following values:

- `slicing = harmonic`. Harmonic type slicing:

$$f = \mathrm{gauge\_f} \; .$$

- `slicing = 1+log`. 1+log type slicing:

$$f = \mathrm{gauge\_f}/\alpha \; .$$

- `slicing = shockavoid`. Shock avoiding family:

$$f = 1 + \mathrm{gauge\_f}/\alpha^2 \; .$$

- `slicing = shock0`. Zero order shock avoiding family:

$$f = \mathrm{gauge\_f}^2 / \left[(2 - \mathrm{gauge\_f}) + 2\alpha \, (\mathrm{gauge\_f} - 1) \right] \; .$$

- `slicing = shock1`. First order shock avoiding family:

$$f = \mathrm{gauge\_f} / \left\{ (4 - 3\mathrm{gauge\_f}) + \alpha \, (1 - \mathrm{gauge\_f}) \left[\alpha \, (4 - \mathrm{gauge_f}) - 8 \right] \right\} \; .$$

The last three cases avoid gauge shocks to different degrees. To see exactly what a gauge shock is and what the different shock avoiding families do see [1, 2, 3].

Notice also that in the case when we chose `slicing=harmonic` (that is, constant f), and we also choose initial data corresponding to waves traveling only in one direction (see Sec. 11), then one can in fact predict the time when a gauge shock is expected. In this case the code outputs the predicted shock formation time to the screen immediately after computing the initial data.

8 Shift condition

The shift conditions supported by the code are controlled by the parameter `shift`. The allowed values are:

- `shift = none`. This means that the shift arrays do not even have storage, and the shift terms are not calculated. This is the default.
- `shift = zero`. In this case the shift exists, but it is set equal to zero. The arrays have storage and the shift terms are calculated, but they should all vanish.
- `shift = constant`. In this case the shift is simply a constant given by the value of the real parameter `beta0`.
- `shift = static`. In this case the shift is left equal to its initial profile, which might be non-trivial.
- `shift = harmonic`. Generalized harmonic shift. Not yet fully implemented. A detailed explanation will come later.

9 Boundary conditions

The different boundary conditions in the code are controlled by the character type parameter `boundtype`. Notice that the metric variables $\{\alpha, g\}$ do not need a boundary condition, as they evolve only through source terms and can then be updated all the way to the boundaries. Boundary conditions are only needed for the variables that have non-trivial fluxes.

The different possibilities are:

- `boundtype = flat`. In this case the boundary value is copied from one point in. This is a very simple boundary condition, but it is not very good. I keep it only because it is so simple.
- `boundtype = periodic`. Periodic boundary conditions. They are implemented by making the value on the last point on one side equal to the value on the second point on the other side for all variables.
- `boundtype = outgoing`. This is simple, just don't touch outgoing modes and set the sources for incoming modes equal to zero. Notice that this is only so simple when one is evolving the eigenfields directly.

- `boundtype = reflectplus`. Here we just set the incoming field at a given boundary equal to the outgoing field.
- `boundtype = reflectminus`. Save as above, but with a minus sign.

10 Constraints

The code monitors the constraints associated with the definition of derivative quantities:

$$
\begin{aligned}
C_\alpha &:= D_\alpha - \partial_x \ln \alpha \ , && (10.1)\\
C_g &:= D_g - \partial_x \ln g \ , && (10.2)\\
C_\sigma &:= D_\sigma - \partial_x \sigma \ . && (10.3)
\end{aligned}
$$

The values of the constraint variables can be output if desired.

11 Initial data

Initial data is controlled through the character type parameter `initialdata`. The different types of initial data are:

- `initialdata = metricgauss`

  In this case we take a trivial initial slice and an initial lapse of 1, but we use non-trivial spatial coordinates. The spatial metric is obtained from a simple transformation from Minkowski coordinates:

  $$x_M = x + f(x) \tag{11.1}$$

  The metric then takes the form:

  $$g = (1 + df/dx)^2 \tag{11.2}$$

  The function $f(x)$ is taken to be a simple gaussian controlled by the parameters:

`metricgauss_a0` (real)	Amplitude of gaussian.
`metricgauss_x0` (real)	Center of gaussian.
`metricgauss_s0` (real)	Width of gaussian.

For this initial data we only expect evolution caused by the presence of a non-zero shift.

- `initialdata = lapsegauss`.

  This case corresponds to an initially trivial slice in Minkowski coordinates with a gaussian initial lapse of the form

$$\alpha = 1 + A \exp\left[- \left(x - x0 \right)^2 / \sigma^2 \right] \; . \tag{11.3}$$

  This gaussian is controlled by the parameters:

`lapsegauss_a0` (real)	Amplitude of gaussian.
`lapsegauss_x0` (real)	Center of gaussian.
`lapsegauss_s0` (real)	Width of gaussian.

  This initial data inevitably has contributions from waves traveling in both directions.

- `initialdata = shiftgauss`.

  This case corresponds to an initially trivial slice in Minkowski coordinates with a gaussian initial shift of the form

$$\beta = 1 + A \exp\left[- \left(x - x0 \right)^2 / \sigma^2 \right] \; . \tag{11.4}$$

  This will act as a source term for the evolution of the metric, but the lapse and extrinsic curvature will remain trivial. The gaussian is controlled by the parameters:

`shiftgauss_a0` (real)	Amplitude of gaussian.
`shiftgauss_x0` (real)	Center of gaussian.
`shiftgauss_s0` (real)	Width of gaussian.

- `initialdata = pureright`.

  This correspond to a pulse traveling only to the right. Notice that this implies that we can no longer start with a trivial slice. Also, this option can only be used for $f(\alpha) =$ constant.

  The idea here is to take a non-trivial given in Minkowski coordinates (t_M, x_M) as

$$t_M = h(x_M) \; , \tag{11.5}$$

with h a profile function that decays rapidly. It is then not difficult to show that if we use $x = x_M$ as our spatial coordinate, the spatial metric an extrinsic curvature are

$$\begin{aligned} g &= 1 - h'^2 \quad \Rightarrow \quad D_g = -2h'h''/g \; , & (11.6) \\ K &= -h''/g^{3/2} \quad \Rightarrow \quad \tilde{K} = -h''/g \; . & (11.7) \end{aligned}$$

Assume now that we want to have only waves traveling to the right, this means that we want $\omega_- = 0$, which implies:

$$D_\alpha = \sqrt{f}\, \tilde{K} = -\sqrt{f}\, h''/g \; . \qquad (11.8)$$

The above equation gives us a differential equation for α. In the particular case when f is a constant the equation can be easily integrated to find

$$\alpha = \left(\frac{1 - h'}{1 + h'} \right)^{\sqrt{f}/2} \; . \qquad (11.9)$$

See [3] for more details. Notice that in this case the code takes the function $h(x)$ to be a gaussian controlled by the same parameters:

`pulsegauss_a0` (real)	Amplitude of gaussian.
`pulsegauss_x0` (real)	Center of gaussian.
`pulsegauss_s0` (real)	Width of gaussian.

- `initialdata = pureleft`.

  This is just the same as the last case, but for pulses traveling to the left. The construction of the initial data is the same except for a change in the sign of D_α, and a corresponding change in α.

12 Tracking observers

The code tracks the Minkowski coordinates of the coordinate observers if the logical parameter `trackobs` is set to `.true.` in the parfile:

```
trackobs = .true.
```

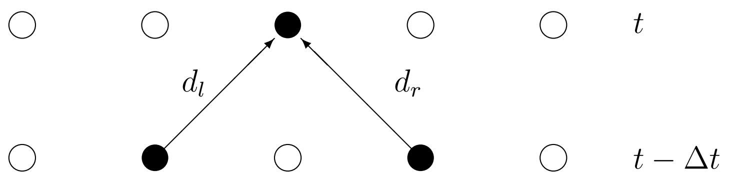

Figure 1: Grid points used to find Minkowski coordinates of observers.

This is done through the arrays `xm` and `tm` which contain the Minkowski position in both space and time of these observers. The Minkowski position of the observers is initialized in a way that is appropriate to the type of initial data chosen (see Section 11 for a description of how the code chooses initial data).

In order to update the position of the observers, once the geometric variables have been updated the code calculates the proper distances d_l and d_r from the grid point under consideration to the grid points immediately to the left and right in the previous time level (see Figure 1). This can be easily done as we know the full spacetime metric at all grid points and we can simply interpolate.

Once these distances are known, and assuming we know the positions in Minkowski spacetime of all grid points in the old time level, the Minkowski coordinates of the point in the new time level can be obtained by solving the following system of equations for $(t_M(t,x), x_M(t,x))$:

$$d_l^2 = [x_M(t-\Delta t, x-\Delta x) - x_M(t,x)]^2 - [t_M(t-\Delta t, x-\Delta x) - t_M(t,x)]^2 \ , \quad (12.1)$$
$$d_r^2 = [x_M(t-\Delta t, x+\Delta x) - x_M(t,x)]^2 - [t_M(t-\Delta t, x+\Delta x) - t_M(t,x)]^2 \ . \quad (12.2)$$

This system of equations is non-linear, and to solve it the code uses Newton-Raphson iterations. Notice that since the equations are quadratic there are two possible solutions, one in the past and one in the future. In order to obtain the solution in the future we simply take the following initial guess for the Newton-Raphson iterations:

$$t_M(t,x) = t_M(t-\Delta t, x) + \Delta t \ , \qquad x_M(t,x) = x_M(t-\Delta t, x) \ . \quad (12.3)$$

13 Tracking characteristics

The code tracks a characteristic scalar if the logical parameter `trackchar` is set to `.true.` in the parfile:

```
trackchar = .true.
```

How is this done? ...

14 Numerical grid

The finite differencing grid is a simple uniform 1D grid. The parameters associated with the grid are:

`dx`	Grid spacing.
`Nx`	Number of grid points.

The code takes half the grid points to be on the negative side of the origin an half on the positive side. If "Nx" is even there will be a point on the origin, while if it is odd we stagger the origin.

For the time stepping we have the following parameters:

`dtfac`	Courant parameter (dtfac=dt/dx).
`Nt`	Total number of time steps.

Notice that the time step `dt` is NOT a parameter, but is rather a derived quantity computed inside the code as: `dt = dtfac` $\times$ `dx`.

15 Numerical methods

For the time integration the code uses a method of lines, where the time integration and spatial differentiation are considered independent of each other.

The time integrator is controlled by the character type parameter `integrator`:

- `integrator = icn`. This is 3-step iterative-Crank-Nicholson. For a reference of this integration method see Appendix A of [4] and also [9].
- `integrator = rk4`. This is standard fourth order Runge-Kutta, see [8].

The spatial differentiation is controlled by the character type parameter `method`:

- `method = upwind`. Upwinded differences (first order).
- `method = center`. Standard centered differences (second order).
- `method = minmod`. Minmod type slope limiter (second order).
- `method = vanleer`. Van Leer type slope limiter (second order).

Notice that the sources are always calculated for the eigenfields first, the sources for the primary fields (D_α, D_g, $\tilde{K}$) are later reconstructed from these. The upwind and slope limiter methods are therefore well defined. For more details on these methods see [7].

16 List of all currently available parameters

Here is a list of all currently available parameters with their default values and ranges when applicable.

- Grid parameters:

`dx (real(8))`	Spatial interval (default 1.0).
`Nx (integer)`	Total number of grid points (default 10).

- Time stepping parameters:

`dtfac (real(8))`	Courant parameter dt/dx (default 0.5).
`Nt (integer)`	Total number of time steps (default 10).

- Output parameters:

`directory (character)`	Name of output directory (default `output`).
`Ninfo (integer)`	Frequency of output to screen (default 1).
`Noutput0D (integer)`	Frequency of 0D output (default 1).

`Noutput1D (integer)`	Frequency of 1D output (default 1).
`outvars0D (character)`	Comma separated lists of variables that need 0D output (default `alpha`).
`outvars1D (character)`	Comma separated lists of variables that need 1D output (default `alpha`).
`commentype (character)`	Type of comment lines used on the output files (range = {`xgraph`, `gnuplot`}, default `xgraph`).
`outtype (character)`	Type of output (range = {`standard`, `test`}, default `standard`). The `test` type output is only used for the testsuites and is designed to minimize as much as possible the round-off errors on output.

- Slicing parameters:

`slicing (character)`	Type of slicing condition (range = {`harmonic`, `1+log`, `shockavoid`, `shock0`, `shock1`}, default `harmonic`).
`gauge_f (real(8))`	Coefficient used in slicing condition (default 1).

- Shift parameters:

`shift (character)`	Type of shift condition (range = {`none`, `zero`, `constant`, `static`, `harmonic`}, default `none`).
`beta0 (real)`	Constant value of shift for the case `shift = constant`.
`gauge_h (real)`	Gauge coefficient for harmonic shift.

- Initial data parameters:

`initialdata (character)`	Type of initial data (range = {`metricgauss`, `lapsegauss`, `shiftgauss`, `pureright`, `pureleft`}, default `lapsegauss`).
`lapsegauss_a0 (real(8))`	Amplitude of gaussian for lapsegauss (default 0).
`lapsegauss_x0 (real(8))`	Center of gaussian for lapsegauss (default 0).
`lapsegauss_s0 (real(8))`	Width of gaussian for lapsegauss (default 1).
`shiftgauss_a0 (real(8))`	Amplitude of gaussian for shiftgauss (default 0).
`shiftgauss_x0 (real(8))`	Center of gaussian for shiftgauss (default 0).

`shiftgauss_s0 (real(8))`	Width of gaussian for shiftgauss (default 1).
`metricgauss_a0 (real(8))`	Amplitude of gaussian for metricgauss (default 0).
`metricgauss_x0 (real(8))`	Center of gaussian for metricgauss (default 0).
`metricgauss_s0 (real(8))`	Width of gaussian for metricgauss (default 1).
`pulsegauss_a0 (real(8))`	Amplitude of gaussian for pureright and pureleft (default 0).
`pulsegauss_x0 (real(8))`	Center of gaussian for pureright and pureleft (default 0).
`pulsegauss_s0 (real(8))`	Width of gaussian for pureright and pureleft (default 1).

- Evolution parameters:

`integrator (character)`	Time integration scheme (range = {`icn`, `rk4`}, default `icn`).
`method (character)`	Spatial differencing method (range = {`upwind`, `center`, `minmod`, `vanleer`}, default `center`).

- Boundary parameters:

`boundtype (character)`	Type of boundary condition (range = {`flat`, `periodic`, `outgoing`, `reflectplus`, `reflectminus`}, default `outgoing`).

- Parameters to track observers:

`trackobs (logical)`	Do we track observers? (default `.false.`).

- Parameters to track characteristics:

`trackchar (logical)`	Do we track characteristics? (default `.false.`).

17 List of all currently available arrays

Here is a list of all currently available arrays. All arrays extend from 0 to Nx (they have $Nx + 1$ points). Notice that half the grid points are on the negative side of the origin: If Nx is even, we will have a point on the origin, while if it is odd we stagger the origin.

- Arrays related to coordinates:

`x`	Spatial coordinate of grid points.
`xm`	Spatial Minkowski coordinate of observers.
`xm_p`	Spatial Minkowski coordinate of observers at past time level.
`tm`	Temporal Minkowski coordinate of observers.
`tm_p`	Temporal Minkowski coordinate of observers at past time level.

- Arrays related to slicing:

`alpha`	Lapse function.
`alpha_p`	Lapse function at past time level.
`salpha`	Source term for lapse.
`alpha_a`	Auxiliary array for lapse.
`Dalpha`	Logarithmic derivative of lapse: $d \ln \alpha / dx$.
`Dalpha_p`	Logarithmic derivative of lapse at past time level.
`sDalpha`	Source term for logarithmic derivative of lapse.
`Dalpha_a`	Auxiliary array for logarithmic derivative of lapse.
`f`	Bona-Masso gauge function: $f(\alpha)$.
`fp`	Derivative of Bona-Masso gauge function: $df/d\alpha$.

- Arrays related to the shift:

`beta`	Shift.
`beta_p`	Shift at past time level.
`Dbeta`	Derivative of shift ($D_\beta = d\beta/dx$).
`sigma`	Rescaled shift ($\beta = \alpha\sigma$).

`sigma_p`	Rescaled shift at past time level.
`ssigma`	Source term for rescaled shift.
`sigma_a`	Auxiliary array for rescaled shift.
`Dsigma`	Derivative of rescaled shift ($D_\sigma = d\sigma/dx$).
`Dsigma_p`	Derivative of rescaled shift at past time level.
`sDsigma`	Source term for derivative of rescaled shift.
`Dsigma_a`	Auxiliary array for derivative of rescaled shift.
`h`	Shift gauge function $h(\alpha)$.
`hp`	Derivative of shift gauge function $h_p = dh/d\alpha$.

- Arrays related to spatial metric:

`g`	Spatial metric function: $g := g_{xx}$.
`g_p`	Spatial metric at past time level.
`sg`	Source term for spatial metric.
`g_a`	Auxiliary array for spatial metric.
`Dg`	Logarithmic derivative of spatial metric: $d \ln g/dx$.
`Dg_p`	Logarithmic derivative of metric at past time level.
`sDg`	Source term for logarithmic derivative of metric.
`Dg_a`	Auxiliary array for logarithmic derivative of metric.

- Arrays related to extrinsic curvature:

`trK`	Trace of extrinsic curvature: $\mathrm{tr}K := K_{xx}/g_{xx}$ (analysis).
`Ktilde`	Densitized extrinsic curvature: $\tilde{K} := \sqrt{g}\, \mathrm{tr}K$.
`Ktilde_p`	Densitized extrinsic curvature at past time level.
`sKtilde`	Source term for densitized extrinsic curvature.
`Ktilde_a`	Auxiliary array for densitized extrinsic curvature.

- Arrays related to eigenfields:

`lambda0`	Time-line eigenspeed: $\lambda_0 := -\beta$.

`lambdafp`	Plus slicing eigenspeed: $\lambda^f_+ := -\beta + \alpha\sqrt{f/g}$.
`lambdafm`	Minus slicing eigenspeed: $\lambda^f_- := -\beta - \alpha\sqrt{f/g}$.
`lambdahp`	Plus shift eigenspeed: $\lambda^h_+ := -\beta + \alpha\sqrt{h/g}$.
`lambdahm`	Minus shift eigenspeed: $\lambda^h_- := -\beta - \alpha\sqrt{h/g}$.
`W0`	Time-line eigenfield: $\omega_0 := D_\alpha/f - D_g/2$.
`sW0`	Source term for time-line eigenfield.
`Wfp`	Right propagating slicing eigenfield: $\omega^f_+ := \tilde{K} + D_\alpha/\sqrt{f}$.
`sWfp`	Source term for right propagating slicing eigenfield.
`Wfm`	Left propagating slicing eigenfield: $\omega^f_- := \tilde{K} - D_\alpha/\sqrt{f}$.
`sWfm`	Source term for left propagating slicing eigenfield.
`Whp`	Right propagating shift eigenfield: $\omega^h_+ := (1/\sqrt{gh}+\sigma)\tilde{K} + D_g/(2\sqrt{g}) - D_\sigma/\sqrt{h}$.
`sWhp`	Source term for right propagating shift eigenfield.
`Whm`	Left propagating shift eigenfield: $\omega^h_- := (1/\sqrt{gh}-\sigma)\tilde{K} - D_g/(2\sqrt{g}) - D_\sigma/\sqrt{h}$.
`sWhm`	Source term for left propagating shift eigenfield.

- Arrays related to characteristics:

`char0`	Time line characteristic scalar.
`char0_p`	Time line characteristic scalar at past time level.
`schar0`	Time line for right propagating characteristic scalar.
`charp`	Right propagating characteristic scalar.
`charp_p`	Right propagating characteristic scalar at past time level.
`scharp`	Source term for right propagating characteristic scalar.
`charm`	Left propagating characteristic scalar.
`charm_p`	Left propagating characteristic scalar at past time level.
`scharm`	Source term for left propagating characteristic scalar.
`Dchar0`	Inverse derivative of time line scalar: $(d\,\mathrm{charp}/dx)^{-1}$.

`Dcharp`	Inverse derivative of right propagating scalar: $(d\,\mathrm{charp}/dx)^{-1}$.
`Dcharm`	Inverse derivative of left propagating scalar: $(d\,\mathrm{charm}/dx)^{-1}$.

- Arrays related to constraints:

`Calpha`	Constraint: $C_\alpha := D_\alpha - d\ln\alpha/dx$.
`Cg`	Constraint: $C_g := D_g - d\ln g/dx$.
`Csigma`	Constraint: $C_\sigma := D_\sigma - d\sigma/dx$.

18 Editing the code

If you are planning to edit the code and add your own new parameters, arrays or routines, here are some details you should know:

18.1 Adding parameters

All parameters for the code should be declared in the file `src/param.f90`, using a very specific format:

```
type ::  name = value
```

Here `type` can be any of the standard `FORTRAN` variable types (`logical`, `integer`, `real`, `character`), `name` is the name of the parameter, and `value` is a default initial value that makes some sense for the code. All parameters must have a default value, otherwise the code will not compile. Only one parameter can be declared per line, since this file will be read at compilation time by a perl script that expects that structure.

The following variations are permitted when declaring a parameter: Character type parameters that are allowed to receive multiple values at the same time (separated by commas) should be declared as:

```
character :: name = value     ! multiple
```

Also, a range can be defined for character type parameters as:

```
character :: name = value     ! range = (value1,value2,...,valueN)
```

The range is not compulsory (but it is a good idea), if it is not there then any value is allowed.

18.2 Adding arrays

Since this is a 1D code, it works with a series of 1D arrays that have all the dynamical and analysis variables. All arrays must be declared in the file `src/arrays.f90`, using a very specific format:

```
real(8), allocatable, dimension (:) :: arrayname
```

Only one array can be declared per line, since this file will be read at compilation time by a perl script that expects this structure.

One can also add control information at the end of a declaration as a comment. For example, an array can be declared for "analysis" in the following form:

```
real(8), allocatable, dimension (:) :: arrayname     ! analysis
```

This is then interpreted as an array that will only need memory allocated if one wants output for it.

In the same way, one can add a conditional statement that will control when memory is allocated. For example:

```
real(8), allocatable, dimension (:) ::  varname   ! if (expression)
```

Here "expression" must be a correctly formulated Fortran logical expression involving the value of some parameter the code knows about.

18.3 Adding routines

Routines should be added directly in the subdirectory `src` with the extension `.f90`. There is no need to modify the Makefile, as it will automatically compile and link all files it finds with that extension.

Since all the parameters and arrays are declared in the F90 module files `param.f90` and `arrays.f90`, if you want your routine to see them you need to add the following two lines immediately after the name of the routine (before declaring any variables):

```
use param
use arrays
```

18.4 Testsuites

The code has a series of testsuites already prepared in the subdirectory `testdata`. Whenever the code is edited it is a good idea to check that the tests still pass, which would mean that nothing has been broken by the new routines. To run the testsuites simply go to the main directory and type:

```
gmake test
```

This will run all existing tests sequentially. In the future it might be possible to choose just some of the tests, but this is currently not available. There is a chance that the tests might fail when different compilers are used simply because of round-off errors, so always see that the differences reported are really significant (this can be minimized by adding the following line to the parameter file for the test:

```
outtype = test
```

Whenever you add new parameters and options, it might be a good idea to add a new test to the code. This is easy to do, just add a sample parameter file and the result of the run for that parameter file to the directory `testdata`. It is important to make sure that the output from the run is inside a directory that has the same name as the parameter file (without the extension `.par`), otherwise the testsuite won't find it. Once this is done the Makefile will run the new test whenever the command `gmake test` is typed.

18.5 CVS

CVS stands for "Concurrent Versions System", and chances are you got this code through CVS so you already know what it is. If you don't, CVS is a system for keeping well organized versions of a code developed by several people, by having the "official version" on a central repository and keeping track of all new changes. It is amazing how rapidly one looses track of what is the "most up-to-date version" of a multi-author code (or any multi-author files for that matter) without using something like CVS.

If you add new routines to the code that you think might be useful for other users, it would be nice to add them to the CVS repository. Just ask me if you need access to the repository (malcubi@nucleares.unam.mx).

References

[1] M. Alcubierre. The appearance of coordinate shocks in hyperbolic formulations of general relativity. *Phys. Rev. D*, 55:5981–5991, 1997. gr-qc/9609015.

[2] M. Alcubierre. Hyperbolic slicings of spacetime: Singularity avoidance and gauge shocks. *Class. Quantum Grav.*, 20:607–624, 2003. gr-qc/0210050.

[3] M. Alcubierre. Are gauge shocks really shocks? 2005. gr-qc/0503030.

[4] M. Alcubierre, B. Brügmann, T. Dramlitsch, J.A. Font, P. Papadopoulos, E. Seidel, N. Stergioulas, and R. Takahashi. Towards a stable numerical evolution of strongly gravitating systems in general relativity: The conformal treatments. *Phys. Rev. D*, 62:044034, 2000. gr-qc/0003071.

[5] R. Arnowitt, S. Deser, and C. W. Misner. The dynamics of general relativity. In L. Witten, editor, *Gravitation: An Introduction to Current Research*, pages 227–265. John Wiley, New York, 1962.

[6] C. Bona, J. Massó, E. Seidel, and J. Stela. New formalism for numerical relativity. *Phys. Rev. Lett.*, 75:600–603, 1995. gr-qc/9412071.

[7] R. J. Leveque. *Numerical Methods for Conservation Laws.* Birkhauser Verlag, Basel, 1992.

[8] W. H. Press, B. P. Flannery, S. A. Teukolsky, and W. T. Vetterling. *Numerical Recipes.* Cambridge University Press, Cambridge, England, 1986.

[9] S. Teukolsky. On the stability of the iterated Crank-Nicholson method in mumerical relativity. *Phys. Rev. D*, 61:087501, 2000. gr-qc/9909026.

[10] J. York. Kinematics and dynamics of general relativity. In L. Smarr, editor, *Sources of Gravitational Radiation.* Cambridge University Press, Cambridge, England, 1979.