Code `OllinAxis-BiB`
User's manual

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

February, 2026

Contents

1 Introduction

This program solves the Einstein evolution equations in axial symmetry using a curvilinear version of the BSSN formulation for the 3+1 evolution equations, with different types of matter and different gauge conditions.

The main difference of this version of the code with respect to previous ones is the fact that it uses box-in-box mesh refinement (hence the BiB part of the name), and is parallelized with MPI. Many parts of this code are based on a previous version written by myself and José Manuel Torres. An even older version was written by Milton Ruiz.

Note: THIS MANUAL IS STILL INCOMPLETE! I haven't had the time to finish it. Still, it should give you a good idea of the basics. I will be adding a little bit more every now and again.

2 Downloading the code

If you are reading this it means you probably already downloaded the code. But if for some reason you need to download it here is how.

The easiest way to obtain the code is to download it from GitHub:

```
git clone https://github.com/malcubi/OllinAxis-BiB
```

Once you have the code, you can get updated versions by just doing "git pull" inside the code main directory.

3 Directory structure

The main directory for the code is `OllinAxis-BiB`. There are several sub-directories inside this main directory:

`CVS`	Contains information about the CVS root and server (see Sec. 20).
`doc`	Contains the tex and pdf files for this user's manual.

`exe`	This directory is created at compile time and contains the executable file. It also contains a copy of the parameter files found in directory `par` (see below).
`fakempi`	Contains fake MPI routines so that the compiler won't complain if MPI is not installed.
`gnuplot`	Contains a couple of simple gnuplot macros for visualization.
`objs`	This directory is created at compile time and contains all the object and module files.
`ollingraph`	Contains the visualization packages "ollingraph" and "ollingraph2D" for convenient "quick and dirty" visualization (see Section 21 below).
`par`	Contains examples of parameter files (see Section 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.

The directory `src` is itself divided into a series of sub-directories in order to better classify the different routines. These sub-directories are:

`auto`	Contains `FORTRAN` files that are automatically generated at compile time by the perl scripts. These files should not be edited manually!
`base`	Contains the routines that control the basic execution of the code, including the parameter and array declarations, the parameter parser, the output routines, the main evolution controllers, and generic routines for calculating derivatives, dissipation, etc. The code in fact starts execution at the routine `main.f90` contained in this directory.
`elliptic`	Contains routines for solving elliptic equations for initial data and/or maximal slicing for example.
`geometry`	Contains routines related to initial data, evolution and analysis of the spacetime geometric variables, including sources, gauge conditions, constraints, horizon finders, etc.

`matter`	Contains routines related to the initial data, evolution and analysis of the different matter models, including a generic routine for calculating the basic matter variables, and routines for evolving scalar fields, electric fields, fluids, etc.

The code is written in `FORTRAN 90` and is parallelized with `MPI` (Message Passing Interface). All subroutines are in separate files inside the directory `src` and its sub-directories.

3.1 Compiling

To compile just move inside the `OllinAxis-BiB` directory and type:

```
make
```

This will first run some perl scripts that create a series of automatically generated `FORTRAN` files that will be placed inside the directory `src/auto`. It will then compile all the `FORTRAN` routines that it can find inside any of the sub-directories of `src` (it will attempt to compile any file with the extension `.f90`).

The resulting object files and `FORTRAN` module files will be placed inside the sub-directory `objs`. The Makefile will then create a directory `exe` and will place in it the final executable file called `ollinaxis`. It will also copy to this directory all the sample parameter files inside the sub-directory `par`, and the visualization package `ollingraph`.

Notice that at this time the Makefile can use the compilers `g95`, `gfortran`, or the Intel compilers `ifc` and `ifort`, and it will automatically check if they are installed. If you have a different compiler then the Makefile will have to be modified (hopefully it won't be very difficult). The code will also attempt to find an `MPI` installation (it looks for the command `mpif90`), and if it does not find it it will use the fake routines inside the directory `fakempi`.

The Makefile has several other useful targets that can be listed by typing: `make help`.

3.2 Running

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

```
ollinaxis 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 to that directory.

For parallel runs using `MPI` one must use instead the command:

```
mpirun -np N ollinaxis name.par
```

where `N` should be an integer number that specifies the number of processors to be used.

4 Parameter files

At run time 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:

```
ollinaxis 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`, `outvars1D` and `outvars2D` that control which arrays get output and take lists of arrays as values, for example:

```
outvars0D = alpha,A,B,C,H
```

There is in fact one other parameter that can also take multiple values as input, it is the parameter `mattertype` that can accept several types of matter at once (see Section 9 below).

Parameters that do not appear in the parfile get the default values given in the file `src/base/param.f90`. Examples of parameter files can be found in the subdirectory `par`.

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`.

5 Output files

At run time, the codes creates an output directory 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`), `1D` files (with extensions `*.rl`, `*.zl` and `*.dl`), and `2D` files (with extension `*.2D`).

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 three different norms of the spatial arrays: maximum absolute value (`nm1`), root mean square (`nm2`), and total variation (`var`). The value of different variables at the origin is also output.

The `1D` files contain the complete arrays along the coordinate axis and diagonals at different times, while the `2D` files output the full arrays. Beware: If you do output very often these files can become quite large!

Since we can have several grid refinement levels, and grid boxes, the file names are appended with a number corresponding to the specific box and level (all grid levels have output). For example:

`alphab0l0.rl`:	Box 0, Level 0 (coarsest grid)
`alphab0l1.rl`:	Box 0, Level 1
...	

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?
`Noutput2D`	How often do we do 2D output?
`outvars0D`	Arrays that need 0D output (a list separated by commas).
`outvars1D`	Arrays that need 1D output (a list separated by commas).
`outvars2D`	Arrays that need 2D output (a list separated by commas).

6 Checkpoint

For very long runs, or in cases when we want to save the final state of the simulation so we can restart from that point, it is useful to have the ability to save the current state of the whole simulation at a given time. The code therefore is capable of doing a "checkpoint" every so often.

In order for the code to do checkpointing, we must add the following to the parameter file:

```
checkpoint = .true.
Ncheckpoint = N
```

where N is an integer number that specifies how often we want to do a checkpoint (it shouldn't be too often, checkpointing is slow and produces a lot of data). This will create a series of directories named `checkpoint_t=T` inside the output directory, with T a real number that indicates the time corresponding to the given checkpoint. Each of these directories will contain all the data needed to restart the simulation from that point.

There is also the possibility of only checkpointing the initial data. This is useful in cases when solving for the initial data is ver slow. To do this ass the following line to the paramert file (this is on by default):

```
checkpointinitial = .true.
```

In order to later restart a simulation from a given checkpoint file, one must first move the corresponding checkpoint directory to the main directory containing the executable and parameter files. One then deletes the line containing the parameter "idata" in the parameter file (or simply comments the line with #), and adds the lines:

```
idata = checkpoint
checkpointfile = checkpointfile
```

where `checkpointfile` should correspond to the full name of the checkpoint directory from which we want to restart. For example, adding the line:

```
checkpointfile = checkpoint_t=0.00000
```

causes the code to restart from $t = 0$ (notice that in that case the code will not calculate the initial data again, it will just take the data from the checkpoint file).

WARNING: When restarting from a checkpoint always remember to change the name of the output directory, as otherwise the code will just rewrite it and you will loose all your data until that point!

7 Numerical grid

7.1 Grid structure and staggering of the axis

The code works in cylindrical coordinates (r, z), where r represents distance to the axis of symmetry (not distance to the origin!), and z the height above or below the "equator". Grid functions are represented by $f_(i, j)$, with the index i representing the r position, and the index j the z position.

In order to avoid having divisions by zero in some terms, we stagger the symmetry axis. This means that there is no grid point at $r = 0$. Instead, the first grid point (grid point 0) is located at $r = -dr/2$ (with dr the grid spacing), the second one (grid point 1) at $r = +dr/2$, and so on. This also makes it easy to apply symmetry boundary conditions, since for an even grid function $f_{(i,j)}$ we can simply take $f_{0,j} = +f_{1,j}$, and for an odd function $f_{0,j} = -f_{1,j}$.

Grid points to the left of symmetry axis are known as "ghost points", and the code usually adds more than one in order to be able to use higher order differencing stencils. The positions

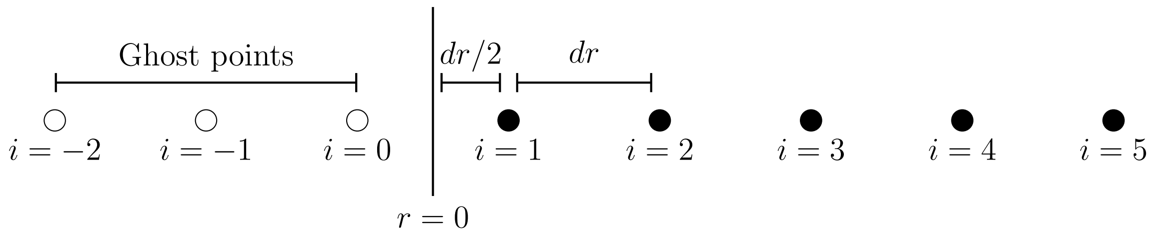

Figure 1: Basic grid structure close to the axis of symmetry $r = 0$, showing how the axis is staggered. We show the case of 3 ghost points to the left of the origin (for fourth order spatial differencing).

of the grid points along the r direction then correspond to $r_i = (i - 1/2) * dr$, where i runs from $(1 - g)$ to some maximum number `Nr`, and where g is the number of ghost points given by the parameter `ghost`. In fact, for second order spatial differencing the code takes `ghost=2`, while for fourth order differencing it takes `ghost=3`, see Figure 1 (the reason for this has to do with dissipation, which typically needs one extra ghost point that would normally be required for a given differencing order, see below). Notice that the parameter `ghost` is determined by the code at run time and should NOT be fixed in the parameter file.

The grid for the coordinate z, on the other hand, has two different structures depending on whether we have or not equatorial symmetry. If the logical parameter `eqsym` is set to `.false.`, that is if there is no equatorial symmetry, then the grid uses `Nz` points along the z direction starting from $-Nz/2$ to $+Nz/2$. Notice that of `Nz` is even then there will be no point at $z = 0$ (and the equator will be staggered9, while if `Nz` is odd there will be a point with $z = 0$.
On the other hand, if the logical parameter `eqsym` is set to `.true.`, then the equator is always staggered and symmetry conditions are applied in the z direction in the same way as it is done with the axis if symmetry for the coordinate r. Having equatorial symmetry then reduces the use of computational by half.

7.2 Box-in-box grid refinement

The also code uses “box-in-box” type grid refinement. That is, it has refinement levels with higher resolution set up at the beginning from the parameter file. The code allows one to set up to 4 different refinement regions called “boxes”, number 0 to 3. In each of these refinement boxed one can also specify how many refinement levels one needs. Box 0 level 0 always refers to the main coarse grid. While Box 0 levels 1,2,3,... refer to refinement levels on this main grid, always with half the resolution and the same number of grid points. So in essence, refinement levels of Box 0 always refine the region around the origin.

On the other hand, boxes 1,2,3 can be placed in different parts of the main grid. This can be useful if one need to refine a specific region of the grid know a priori which does not correspond to the origin.

The total number of refinement boxes is controlled by the integer parameter `Nb` which tells the code how many extra boxes are required. This parameter can take values from 0 to 3, with 0 (the default) referring to having only the main Box 0. The number of refinement levels on each box are given, respectively, by the parameters `Nl0,Nl1,Nl2,Nl3`.

For example, if we take `Nb=0` and `Nl0=2`, meaning only the main box with two extra refinement levels, the resulting grid structure is shown in Figure 2, where we have two extra levels of refinement around the origin $r = z = 0$ represented by the red and green colors.

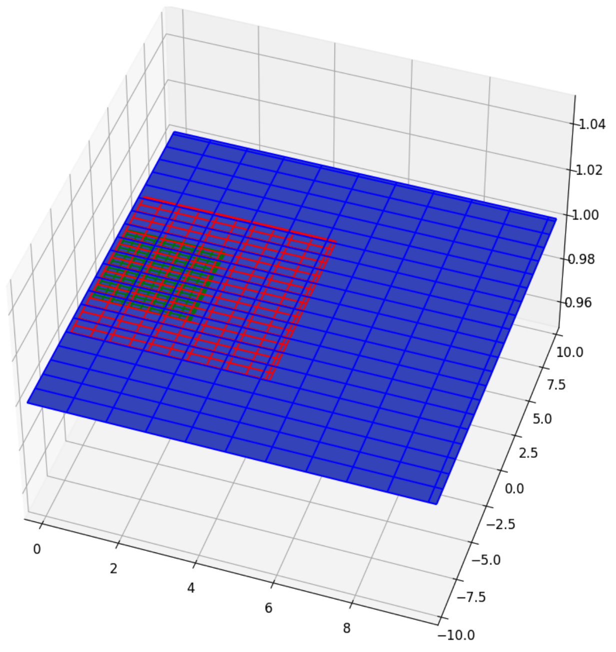

Figure 2: Schematic grid structure for the case with `Nb=0` and `Nl0=2`.

When we have more than one refinement box, `Nb=1,2,3`, the position and size of the extra boxes is controlled by the parameters `rbox#,zbox#,Nrbox#,Nzbox#`. For example, if we take: `Nb=3`, `Nl0=0`, `Nl1=Nl2=Nl3=1`, with `(rbox1=5,zbox1=0)`, `(rbox2=0,zbox2=5)`, `(rbox3=0,zbox3=-5)`, and `(Nrbox1=Nzbox1=50)`, `(Nrbox2=Nzbox2=50)`, `(Nrbox3=Nzbox3=50)`, then we are asking for three extra refinement boxes, centered on the points $(5, 0)$, $(0, 5)$ and $(0, -5)$, which sizes (50×50) grid point each. Notice that now box 0 has no refinement levels, while boxes 1,2,3 have one refinement level each. The resulting grid structure is represented in Figure 3, with the extra refinement boxes shown in different colors.

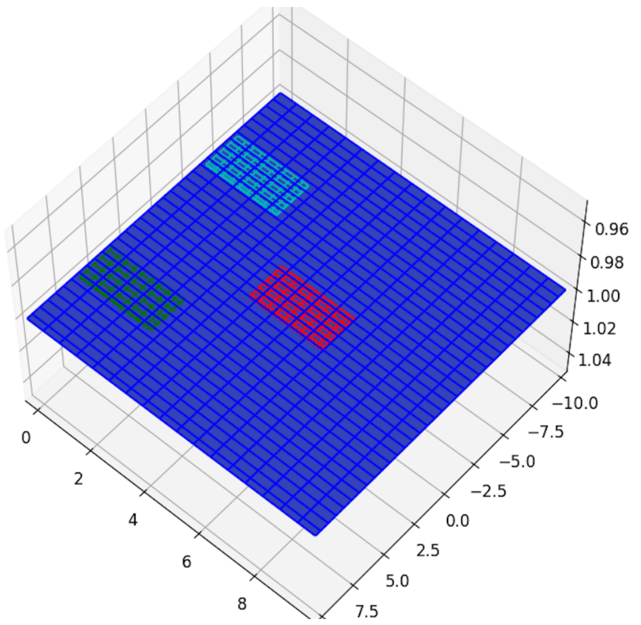

Figure 3: Schematic grid structure for the case with `Nb=3` and `Nl1=Nl2=Nl3=1`.

7.3 Domain decomposition paralellization

8 Spacetime and evolution equations

8.1 Spacetime metric

As already mentioned, the code uses cylindrical coordinates (r, z) in space. Notice that what the code calls r here represents distance to the symmetry axis and not distance to the origin. The distance to the origin in the code is in fact called $rr := (r^2 + z^2)^{1/2}$.

The spatial metric is written in the following way:

$$
\begin{aligned}
dl^2 = e^{4\phi(r,z,t)} \big[& A(r,z,t)dr^2 + B(r,z,t)dz^2 + r^2 H(r,z,t)d\varphi^2 \\
& + 2\, r \left(C(r,z,t)drdz + r^2 C_1(r,z,t)drd\varphi + rC_2(r,z,t)dzd\varphi \right) \big] \ ,
\end{aligned}
\tag{8.1}
$$

with ϕ the conformal factor. The powers of r in some of the terms come from the behaviour of the different metric components close to the axis. Factorizing those powers of r explicitly allows us to make sure that the functions (A, B, H, C, C_1, C_2) are well behaved at $r = 0$. Notice that the components of the extrinsic curvature K_{ij} are also decomposed in a similar way (see below).

We also define the function $\psi = e^{\phi}$ and use it instead of ϕ in many expressions. For the lapse function we use the array $\alpha(r, z, t)$, while the shift in principle has three components: $\beta^r(r, z, t)$, $\beta^z(r, z, t)$, $\beta^\varphi(r, z, t)$.

Notice that when there is no angular momentum the metric can be simplified since in that case we have $\beta^\varphi = 0$, and $C_1 = C_2 = 0$. The code controls this using the logical parameter `angmom`: if this parameter is true then those arrays are turned on, while if it is false they are off. By default the parameter is false, so that those arrays are turned off.

It is also useful to define an inverse to the conformal factor as:

$$
\chi := 1/\psi^n = \exp(-n\phi) \ , \tag{8.2}
$$

with n a power controlled by the parameter `chipower`. For black hole spacetimes the conformal factor ϕ is singular at the black hole position, while χ remains regular, so it is best to evolve χ instead of ϕ. This can be controlled with the logical parameter `chimethod` (which by default is set to `.true.`).

8.2 Evolution equations

For the evolution equations, the code uses a Baumgarte-Shapiro-Shibata-Nakamura (BSSN) formulation adapted to axial symmetry. The specific form of the evolution equations used here can be found in [2].

There are two BSSN variants or "flavors" controlled by the parameter `bssnflavor`, which can take the values `lagrangian` or `eulerian` (the default is `bssnflavor=lagrangian`). They refer to the way in which the shift terms are added to make determinant of the conformal metric constant along time-lines (lagrangian), or constant along the normal-lines (eulerian).

There is also a parameter that allows one to switch off the evolution of the spacetime, which is useful in case one wants to evolve some matter field in a fixed background spacetime. The parameter is called `spacetime` and can have the values `dynamic` or `static` (default is `spacetime=dynamic`).

The main evolution variables (arrays) are:

`alpha`	The lapse function α.
`beta_r`	Contravariant shift component β^r.
`beta_z`	Contravariant shift component β^z.
`beta_p`	Contravariant angular shift component β^φ.
`phi`	The conformal factor ϕ (see equation 8.1).
`psi`	The conformal factor $\psi = e^\phi$ (see equation 8.1).
`A`	Component (r, r) of conformal metric $\tilde{g}_{rr} = \mathtt{A}$.
`B`	Component (z, z) of conformal metric $\tilde{g}_{zz} = \mathtt{B}$.
`H`	Component (φ, φ) of conformal metric $\tilde{g}_{\varphi\varphi} = r^2\,\mathtt{H}$.
`C`	Component (r, z) of conformal metric $\tilde{g}_{rz} = r\,\mathtt{C}$.
`C1`	Component (r, φ) of conformal metric $\tilde{g}_{r\varphi} = r^2\,\mathtt{C1}$.
`C2`	Component (z, φ) of conformal metric $\tilde{g}_{z\varphi} = r\,\mathtt{C1}$.
`trK`	Trace of the extrinsic curvature $\mathtt{trK} := K^m{}_m$.
`KTA`	Component (r, r) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{rr} = \mathtt{KTA}$.
`KTB`	Component (z, z) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{rr} = \mathtt{KTA}$.
`KTH`	Component (φ, φ) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{\varphi\varphi} = r^2\,\mathtt{KTH}$.
`KTC`	Component (r, z) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{rz} = r\,\mathtt{KTC}$.
`KTC1`	Component (r, φ) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{r\varphi} = r^2\mathtt{KTC1}$.
`KTC2`	Component (r, φ) of conformal trace-free extrinsic curvature $\tilde{K}^{TF}_{z\varphi} = r\,\mathtt{KTC2}$.

`Delta_r` Contravariant component of auxiliary BSSN variable Δ^r.

`Delta_z` Contravariant component of auxiliary BSSN variable Δ^z.

`Delta_p` Contravariant component of auxiliary BSSN variable Δ^φ.

Notice that, even though the auxiliary BSSN variables Δ^i are initially defined in terms of metric derivatives, they are later evolved independently with their own evolution equations (modified using the momentum constraints), as required in the BSSN formulation.

The code also calculates a series of auxiliary geometric quantities defined as:

`psi` The conformal factor `psi` $:= \psi = e^\phi$. The code also calculates `psi2` $:= \psi^2$ and `psi4` $:= \psi^4$.

`chi` Another form of the conformal factor `chi` $= \chi = 1/\psi^n$. This function is in fact evolved instead of ϕ if one sets the logical parameter `chimethod=.true.` (by default it is true). This is useful for black hole evolutions as it improves the treatment of the central punctures.

9 Matter

When we write the Einstein equations in 3+1 form, the components of the stress-energy tensor that will be important are the energy density, momentum density and spatial stress tensor as measured by the so-called Eulerian observers, *i.e.* those that move along the normal direction to the spatial hypersurfaces:

$$\begin{aligned} \rho &= n^\mu n^\nu T_{\mu\nu} \, , && (9.1) \\ J_i &= -n^\mu P_i^\nu T_{\mu\nu} \, , && (9.2) \\ S_{ij} &= P_i^\mu P_i^\nu T_{\mu\nu} \, , && (9.3) \end{aligned}$$

with n^μ the unit normal vector to the spatial hypersurfaces of constant coordinate time, and P^μ_ν the projector operator onto these hypersurfaces.

The code defines as 2D arrays the energy density `rho`, the three components of the momentum density (index up) `J_r`, `J_z`, `J_p` (where the index "p" stands for the φ component), and the six components of the spatial stress tensor which is decomposed in the same way as the metric and extrinsic curvature: `S_A,S_B,S_H,S_C,S_C1,S_C2`. Additionally, the trace of the spatial stress tensor is also defined as `trS`. In the code these quantities are defined even for vacuum (they are set to zero), since they are required both in the evolution equations and for the calculation of the constraints.

The type of matter used by the code is controlled by the character-type parameter `mattertype`. At the moment it allows the following values:

`vacuum`	There is no matter (this is the default).
`scalar`	Real scalar field. The scalar field is assumed to have a potential whose form is controlled by the parameter `scalarpotential`, which can take the values (`none,phi2,phi4`), corresponding to a free scalar field (`none`), a massive scalar field with a potential of the form $V = m^2\Phi^2/2$ (`phi2`), where the mass m is give by the parameter `scalar_mass`, or a potential of the form $V = m^2\Phi^2/2 + \lambda\Phi^4/4$ (`phi4`), with the coefficient of the self-interaction term given by the parameter `scalar_lambda`.
`complex`	Complex scalar field. The complex scalar field is assumed to have a potential whose form is controlled by the parameter `complexpotential`, which can take the values (`none,phi2,phi4`), corresponding to a free scalar field (`none`), a massive scalar field with a potential of the form $V = m^2\|\Phi\|^2/2$ (`phi2`), where the mass m is give by the parameter `complex_mass`, or a potential of the form $V(\phi) = m^2\|\Phi\|^2/2 + \lambda\|\Phi\|^4/4$ (`phi4`), with the coefficient of the self-interaction term given by the parameter `complex_lambda`.

Any new type of matter model should be added to the list of allowed values for this parameter in the file `param.f90`. Notice that the code allows one to have several different types of matter at the same time, the corresponding stress-energy tensors are just added together. For example, if you want to have a real scalar field and a complex scalar field evolving together you can add the following line to the parameter file:

```
mattertype = scalar,complex
```

10 Constraints

10.1 Hamiltonian and momentum constraints

When required for output, the code calculates the Hamiltonian and the three components of the momentum constraints (index up) and saves them in the arrays `ham` and `mom_r`, `mom_z`, `mom_p` (read the comments in the routine `src/geometry/constraints.f90` to see how they are calculated).

Notice that for an exact solution both these constraints should be zero, so monitoring their behaviour, and particularly how they approach zero as we increase the resolution, gives us information about the accuracy of the code.

10.2 BSSN extra constraints

The BSSN formalism requires the introduction of the auxiliary variables Δ^i defined in terms of derivatives of the conformal metric components (see [2]). These are then promoted to independent variables, and their evolution equations are modified using the momentum constraints.

We therefore end up with a new constraints related to the definition of the Δ^i that are calculated for output in the arrays `CDelta_r`, `CDelta_z` and `CDelta_p` (read the comments in the routine `src/geometry/constraints.f90` to see how they are calculated).

11 Slicing conditions

The slicing condition is controlled by the parameter `slicing`, which at the moment can take one of the following values (default is `slicing=harmonic`):

`static` The lapse remains static at its initial value and does not evolve.

`harmonic` This is a harmonic-type slicing condition of the form:

$$\partial_t \alpha = -\alpha^2 f \, \mathrm{tr} K + \beta^m \partial_m \alpha \ , \qquad (11.1)$$

where f is a real positive number controlled by the parameter `gauge_f` (default is `gauge_f=1`). True harmonic slicing really corresponds to `gauge_f=1`, but here we allow any positive real number. This is a hyperbolic slicing condition corresponding to a speed of propagation of the gauge given by $\sqrt{f}$.

`1+log` This is a slicing condition of the 1+log family, which is very similar to the harmonic family above:

$$\partial_t \alpha = -\alpha f \, \mathrm{tr} K + \beta^m \partial_m \alpha \ , \tag{11.2}$$

where again f is a positive real number controlled by the same parameter `gauge_f`. Standard 1+log slicing corresponds to `gauge_f=2` (though as mentioned above the default value is 1).

`shockavoid` This corresponds to the family of shock avoiding slicing conditions, which is similar to the 1+log condition above but with $f(\alpha)$ a function given by (see comments in file `src/geometry/auxiliary_geometry.f90`):

$$f(\alpha) = 1 + k/\alpha^2 \ . \tag{11.3}$$

The code takes $k =$ `gauge_f` $- 1$.

`maximal` This is maximal slicing which corresponds to the condition that the trace of the extrinsic curvature remains equal to zero during evolution (which guarantees that the spatial volume elements remain constant), and which implies that the lapse must satisfy the following elliptic equation:

$$\nabla^2 \alpha = \alpha K_{ij} K^{ij} + 4\pi\alpha \left(\rho + \mathrm{tr} S \right) \ . \tag{11.4}$$

Notice that the choices `harmonic`, `1+log` and `shockavoid` are special cases of the Bona-Masso family of slicing conditions that has the general form:

$$\partial_t \alpha = -\alpha^2 f(\alpha) \mathrm{trK} + \beta^r \partial_r \alpha \ . \tag{11.5}$$

with $f(\alpha)$ an arbitrary (positive) function of the lapse.

The initial value of the lapse can also be controlled by the parameter `ilapse` which can take the values (default `ilapse=none`):

`none` The initial lapse it first set to 1, but can later be modified by initial data routines and will not be touched after this.

`one` The initial lapse is set to 1 again AFTER all initial data routines have been called.

`isotropic`	The initial lapse is set to the corresponding isotropic lapse for a single static black hole (useful for testing).
`psiminus2`	The initial lapse is set to $\alpha = 1/\psi^2$ after all initial data routines have been called, with ψ the conformal factor. This is useful for black hole evolutions where ψ becomes infinite at the origin, and results in a "'pre-collapsed" initial lapse that is already zero and smooth at $r = 0$.
`psiminus4`	Similar to the case above, but sets the initial lapse to $\alpha = 1/\psi^4$.
`maximal`	The maximal slicing condition is solved for the initial lapse, but the lapse can later be evolved using a different condition.

There is also the possibility of adding a Gaussian to the initial value of the lapse. This is useful for perturbing the initial data and causing interesting gauge dynamics. This is controlled by the parameters:

`lapsegauss`	Logical parameter that if set to `.true.` adds a Gaussian to the initial lapse (default is `.false.`).
`lapse_a0`	Amplitude of Gaussian.
`lapse_r0`	Center of Gaussian in r direction.
`lapse_sr0`	Width of Gaussian in r direction.
`lapse_z0`	Center of Gaussian in z direction.
`lapse_sz0`	Width of Gaussian in z direction.

12 Shift conditions

The shift condition is controlled by the parameter `shift`, which at the moment can take one of the following values (default is `shift=none`):

`none`	There is no shift, and all shift terms in the evolution equations are ignored. This is the default.

`zero` The shift is set to zero, but all the shift terms in the evolution equations are calculated. This is useful for testing.

`static` The shift is non-zero but it does not evolve in time. Again, useful for testing.

`Gammadriver1` Parabolic Gamma-driver shift. The shift condition is given by:

$$\partial_t \beta^i = \xi \partial_t \Delta^i \ . \tag{12.1}$$

`Gammadriver2` Standard second order hyperbolic Gamma-driver shift. The shift condition is given by:

$$\partial_t^2 \beta^i = \xi \partial_t \Delta^i - \eta \partial_t \beta^i \ . \tag{12.2}$$

In the code this is written in second order form as:

$$\begin{aligned} \partial_t \beta^i &= \mathcal{B}^i \ , && (12.3) \\ \partial_t \mathcal{B}^i &= \xi \partial_t \Delta^i - \eta \mathcal{B}^i \ . && (12.4) \end{aligned}$$

This is in fact the standard Gamma-driver used in most BSSN codes. Notice that in the code the components of $\mathcal{B}^i$ are in fact called (`dtbeta_r`, `dtbeta_z`, `dtbeta_p`).

The Gamma-driver shift conditions above relate the evolution of the shift to that of the BSSN auxiliary variables Δ^i. The values of the coefficients ξ and η are controlled with the real parameters `drivercsi` and `drivereta`. There is also the logical parameter `driveradv` which if set to `.true.` modifies the Gamma driver conditions by adding advection terms on the shift (by default it is `.false.`).

Just as in the case of the lapse, there is the possibility of adding a Gaussian to the initial value of the shift. This is useful for perturbing the initial data and causing interesting gauge dynamics.

13 Boundary conditions

The outer boundary conditions for the evolution are controlled by the character-type parameter `boundary`, which at the moment can take one of the following values (the default is `flat`):

`none`	No boundary condition is applied. The sources are applied all the way to the boundary using one-sided differences.
`static`	The sources are set to zero at the boundary. This is done only for evolving arrays that are not declared with the keyword `NOBOUND`.
`flat`	The sources at the boundary are copied from their value one grid point in. This is done only for evolving arrays that are not declared with the keyword `NOBOUND`.
`radiative`	Outgoing-wave boundary condition. This is somewhat more involved as it uses some information from eigenfields and eigenspeeds, and works quite well in practice.

Notice that all the boundary conditions are always applied at the level of the source terms, and are applied only for some of the evolving arrays and not all of them. In general, evolving arrays that are declared with the keyword `NOBOUND` have no need for a boundary condition since their source terms can be calculated all the way to the boundary.

Static and flat boundary conditions are applied in the automatically generated file `src/auto/simpleboundary.f90`. The radiative boundary conditions are explained with more detail below.

13.1 Radiative boundaries

14 Initial data

The type of initial data is controlled by the character-type parameter `idata`. If you add a new type of initial data it should be appended to the list of allowed values for this parameter in the file `src/base/param.f90`. You should also add a corresponding call to your initial data routine in the file `src/base/initial.f90`.

Simple types of vacuum initial data already there are:

`idata=minkowski`	The initial metric is simply set to Minkowski and the extrinsic curvature is set to zero. Notice that one can still have non-trivial gauge evolution if one adds a Gaussian perturbation to the initial lapse (see Section 11).

`idata=schwarzschild`	The initial metric is set to that corresponding to a Schwarzschild black hole in isotropic coordinates centered at the origin. The mass of the black hole controlled by the parameter `BH1mass` (default is 1). The initial extrinsic curvature is set to zero.
`idata=BrillLindquist`	This is initial data for two black holes in isotropic (conformally flat) coordinates. The masses of the two black holes are controlled by the parameters `BH1mass` and `BH2mass`, and their position on the z axis is controlled by the parameters `BH1Z` and `BH2Z`.

15 Horizon finder

16 Gravitational wave extraction

17 Numerical methods

17.1 Time integration

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

For the evolution of the field variables we can use one of two methods:

1. Standard fourth order Runge-Kutta.

2. 3-step iterative Crank-Nicholson scheme (ICN). The ICN scheme for N iterations is defined as follows (with $S(f)$ the source term of the evolution equation):

$$\begin{aligned} f_0 &= f(t) \ , && (17.1) \\ f_{i+1} &= f_0 + dt/2 \; S(f_i) \qquad \text{i from 1 to N-1} \ , && (17.2) \\ f_N &= f_0 + dt \; S(f_{N-1}) \ , && (17.3) \\ f_t + dt &= f_N && (17.4) \end{aligned}$$

   That is, we do $N-1$ half steps starting always from $f(t)$ but evaluating the source term using the results from the previous step. Finally, we do one last full step. Doing only 2 ($N=2$) steps is enough for second order accuracy, but we need to do at least 3 ($N=3$)

for stability. ICN is a rather robust method that can be applied to non-linear systems of equations, but it is somewhat dissipative.

The choice of the integration method is done through the parameter `integrator` which can take the values {`rk4,icn`}.

17.2 Spatial differencing

For spatial differentiation we can use either second or fourth order differencing (I have also added sixth and eight order, but this is usually not needed as the time integration is at most fourth order). Close to the outer boundaries we in fact reduce the differencing always to second order. The choice of the differencing order is controlled by the parameter `order` which can take the values {`two,four`}.

17.3 Dissipation

The code uses Kreiss-Oliger type dissipation to improve stability. Often one can turn this off and things will still work, but is seems to be important for scalar field evolutions and also to keep a black hole spacetimes stable for a long time. The dissipation of the geometric variables is controlled by the parameter `geodiss`, and the dissipation for the scalar field (both real and complex) is controlled by the parameter `scalardiss`. Both are set by default to a value of 0.01.

18 Elliptic solvers

For solving initial data, as well as some gauge conditions such as maximal slicing, we need to have elliptic equation solvers in the code At the moment there are only two very simple (and slow) elliptic solvers (and they have not been very heavily tested).

The choice of elliptic solver is controlled by the parameter `ELL_solver` which can take the following values:

`ELL_solver=none`	No elliptic solver is active.
`ELL_solver=wave`	Solver based on turning the elliptic equation into a wave-type hyperbolic equation and evolving this in fictitious time until we reach a steady state.

`ELL_solver=sor`	Simple elliptic SOR solver.

19 List of main code parameters

Here is a list of the main code parameters with their default values and ranges when applicable. For a list of all declared parameters see the file `src/base/param.f90`. But do notice that some of the "parameters" declared in `param.f90` should not be fixed in the parameter file since they are in fact defined at run time.

There are many more parameters that are not mentioned here, particularly related to matter models or initial data routines. Have a look at the file `src/base/param.f90` where they are defined, they are usually explained in the comments).

- Grid parameters:

`dr0 (real)`	Spatial interval along r direction for base (coarsest) grid (default 1.0).
`dz0 (real)`	Spatial interval along z direction for base (coarsest) grid (default 1.0).
`Nrtotal (integer)`	Total number of grid points along r direction for base grid (default 10).
`Nztotal (integer)`	Total number of grid points along r direction for base grid (default 10).
`Nb (integer)`	Number of refinement boxes other than the main one, as much as 3 (default 0). Notice that this does NOT mean the number of refinement levels, but rather the number of different regions to be refined.
`Nl0 (integer)`	Number of extra refinement levels on main box centered on the origin (default 0).
`Nl* (integer)`	Number of extra refinement levels on refinement box *, with *$= (1, 2, 3)$ (default 0).
`eqsym` (logical)	Do we have equatorial symmetry? (default `.false.`).
`ghost (integer)`	Number of ghost zones (default 0). Should NOT be fixed in the parameter file.

`interporder (character)`	Order of interpolation (default “bicubic”). Don’t change this unless you know what you are doing!
`boundtype (character)`	Type of outer boundary condition. Can take the following values: `none`, `static`, `flat`, `radiative` (default `radiative`).

- Time stepping parameters:

`dtfac (real)`	Currants parameter (default 0.5).
`dt0 (real)`	Time step for base (coarsest) grid (default 0.0). This should NOT be set in the parameter file, as it is calculated at run time based on the value of `dt0` and `dtfac`: `dt0 = dtfac*min(dr0,dz0)`.
`adjuststep (logical)`	Dynamically adjust the time step in order to satisfy the Courant stability condition (default .false.).
`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).
`Noutput2D (integer)`	Frequency of 2D 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`).
`outvars2D (character)`	Comma separated lists of variables that need 2D output (default `alpha`).
`commenttype (character)`	Type of comment lines used on the output files (range = {`xgraph`, `gnuplot`}, default `xgraph`).
`outparallel (character)`	Type of parallel output (range = {`singlefile`, `fileperproc`}, default `fileperproc`).

- Evolution parameters:

`spacetime (character)`	Can take the values `dynamic` or `background` to choose if we want a dynamic spacetime (evolve the Einstein field equations) or a static background spacetime (default `dynamic`).
`formulation (character)`	Defines the formulation of the 3+1 equations to use, and can take the values `bssn` or `z4c` (default `bssn`).
`bssnflavor (character)`	Controls the evolution of the volume element in BSSN, and can take the values `lagrangian` or `eulerian` (default `lagrangian`).
`integrator (character)`	Method for time integration. Can take the values `icn` for 3-step iterative Crank-Nicholson, or `rk4` for fourth order Runge-Kutta (default `icn`).
`order (character)`	Order of spatial finite-differencing. Can take the values `two` o `four` (default `two`).
`chimethod (logical)`	If true evolve $\chi = \exp(-n\phi)$ instead of ϕ. This is useful for black hole evolutions (default `.false.`).
`chipower (integer)`	Value of n in the definition $\chi = \exp(-n\phi)$ (default `2`).
`eta (real)`	Parameter that controls the multiple of the momentum constraints that is added to the evolution equation for the Δ^i in the BSSN formulation (default `2.0`).
`angmom (logical)`	Do we turn on angular momentum terms (default `.false.`).
`idata (character)`	Type of initial data (default "minkowski").
`slicing (character)`	Type of slicing condition (default "harmonic").
`geodiss (real)`	Kreiss-Oliger dissipation coefficient for geometric variables.

20 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.

20.1 Adding parameters

All parameters for the code should be declared in the file `src/base/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. 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: 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. If it is not there, any value is allowed (for example, directory names).

REMEMBER: All parameters must have a default value, as otherwise the code will not compile. The initial value should basically correspond to the code NOT doing anything weird. That is, initialize to Minkowski, static slicing, no shift, vacuum, and all special features turned off.

20.2 Adding arrays

Since this is a 2D code, it works with a series of 2D arrays that have all the dynamical and analysis variables. All arrays must be declared in the file `src/base/arrays.config`, using a very specific format.

21 Ollingraph

The code includes a simple 1D visualization package called "ollingraph". It is written in Python and uses Matplotlib to plot simple line plots and animations. At the moment it should work fine under python 3.11. It is supposed to have similar functionality to the old xgraph and ygraph packages, and expects the data files in the same format (see below).

The plots are quite simple, and meant for quick/dirty interactive visualization. These plots are not supposed to be used for figures intended for publication (use gnuplot or something similar for that). To use the package type:

```
ollingraph file1 file2 ...  filen
```

When plotting several data files at once, they are all assumed to be of the same type. One can also add a custom name for the plot using the option –title:

```
ollingraph --title="Plot name" file1 file2 ...  filen
```

If this option is not there the plot is just named after the name of the data file being plotted, assuming there is only one, or just says "Multiple data files" if there is more than one file.

Before using it make sure that `ollingraph` has execution permission:

```
chmod +x ollingraph
```

There is also a package for simple plots of 2D arrays called `ollingraph2D`. It is called in a similar way:

```
ollingraph2D file
```

But in this case the file to be plotted must have extension `.2D`.

The data files are expected to have the following format:

- 0D files (those ending in .tl):

    1. One comment line starting with `#` or `"` with the file name (this line could be missing and it should still work).

2. A series of lines with the data points, with the x and y values separated by blank spaces.

- 1D evolution-type files (those ending in .rl):

  1. Each time step begins with a comment line starting with `#` or `"` that contains the time in the format:

     ```
     #Time = (real number)
     ```

  2. A series of lines with two columns, with the x and y values separated by blank spaces.
  3. One or more blank lines to separate the next time step.

- 2D evolution-type files (those ending in .2D):
- Each time step begins with a comment line starting with `#` or `"` that contains the time in the format:

  ```
  #Time = (real number)
  ```

- A series of lines with three columns, with values separated by blank spaces, corresponding to the coordinates (r, z) and the value of the corresponding variable. The values are stored by first looping over the r coordinate with fixed value of z, and separating by blank spaces for differen values of z.

References

[1] "Regularization of spherical and axisymmetric evolution codes in numerical relativity", M. Ruiz, M. Alcubierre, D. Nuñez, Gen.Rel.Grav. **40**, 159-182 (2008); arXiv:0706.0923 [gr-qc].

[2] "Formulations of the 3+1 evolution equations in curvilinear coordinates", M. Alcubierre and M. D. Mendez, Gen.Rel.Grav. **43**, 2769 (2011); arXiv:1010.4013 [gr-qc].