An Introduction to the Einstein Toolkit

Miguel Zilhão[1]**, Frank Löffler**[2]

[1] Center for Computational Relativity and Gravitation,
Rochester Institute of Technology, 74-2068, Rochester, NY 14623, USA
[2] Center for Computation & Technology,
Louisiana State University, Baton Rouge, LA, USA

E-mail: `mzilhao@astro.rit.edu`, `knarf@cct.lsu.edu`

Abstract. We give an introduction to the Einstein Toolkit, a mature, open-source computational infrastructure for numerical relativity based on the Cactus Framework, for the target group of new users. This toolkit is composed of several different modules, is developed by researchers from different institutions throughout the world and is in active continuous development. Documentation for the toolkit and its several modules is often scattered across different locations, a difficulty new users may at times have to struggle with. Scientific papers exist describing the toolkit and its methods in detail, but they might be overwhelming at first. With these lecture notes we hope to provide an initial overview for new users. We cover how to obtain, compile and run the toolkit, and give an overview of some of the tools and modules provided with it.

1. Introduction

Dating back to Hahn and Lindquist's first attempts at numerically evolving Einstein's field equations for a binary black hole spacetime [1], numerical relativity is now an essential tool to study systems with strong and dynamical gravitational fields. Originally motivated essentially from astrophysics and gravitational wave astronomy, it was soon realized that numerical relativity could be useful to a much wider spectrum, with additional motivations coming even from fields other than gravity itself. See for instance [2] for an overview of some of these topics.

Developing efficient computational codes to probe this highly non-linear regime is a very non-trivial and time-consuming process, however. In fact, only in 2005 were the first simulations of stable, long-term evolutions of the inspiral and merger of two black holes [3–5] successfully accomplished. It is therefore a great advantage if well-tested, easy to use computational infrastructures are available that allow researchers to focus less on the computational aspect of such tasks, allowing for more time being spent on the actual physics.

To illustrate the helpfulness of having such an infrastructure, we start by noting that a typical numerical relativity code will have to, among others, incorporate a mesh refinement algorithm, use an efficient parallelization scheme, efficiently deal with large input and output and also use somewhat complex tools for analysis.

In these notes we introduce the Einstein Toolkit [6–8], an open source, community-driven, freely accessible infrastructure for numerical relativity. The toolkit uses the BSSN (Baumgarte, Shapiro, Shibata, Nakamura) evolution system for spacetime evolution [9, 10] with a finite-volume general-relativistic hydrodynamics solver. Within this document, however, we have chosen to only describe components and tools that we believe are of most interest for a new user, necessarily leaving out large parts of the toolkit that are of interest for advanced users. For a more detailed description the interested reader is directed to [7], and for an overview of the MHD capabilities to [8].

This document is structured as follows. We start with a high-level overview of the most important modules for new users in section 2, before we describe basic steps to use the Einstein Toolkit in section 3. Some details about the structure of most modules are given in section 4, which is not only interesting for users to understand how to configure simulations, but also for developers that are interested in extending the toolkit. Finally, some examples show-case Einstein Toolkit usage in section 5, having in mind that these examples can be used within tutorials or workshops. In particular, this means that these are not tailored for the best possible physical results, but instead for short run-time and low system requirements. Finally, we conclude in section 6.

2. Structural Overview

The Einstein Toolkit (ET) consists of quite a large number of components, of which most currently use the `Cactus` Computational Toolkit [11, 12] framework, which provides the

basic modular infrastructure for numerical simulations. A number of tools are provided surrounding this framework environment, from simulation management to data analysis and visualization. In the following we briefly introduce `Cactus`, as well as some of the arrangements provided with the toolkit. Some of the more important tools are introduced later in section 3, as part of a description of typical Einstein Toolkit usage.

2.1. Cactus

The `Cactus` framework [11–13] is a general framework for the development of portable, modular applications, wherein programs are split into components (called *thorns*) with clearly defined dependencies and interactions. Thorns are typically developed independently and should be interchangeable with others with same functionality. They usually do not directly interact with each other; rather each of them interacts with the `Cactus` framework *flesh* which provides the "glue" between different thorns. Thorns can be written in different languages, with `C`, `C++`, `Fortran`, `OpenCL` and `CUDA` being supported currently.

`Cactus`, originally developed for numerical relativity, is a direct descendant of many years of code development in Ed Seidel's group of researchers at NCSA; its version 1.0 was released in 1995. Over the years it has been generalized for use by scientists in other domains.

At this point, introducing some `Cactus`-related terminology is beneficial, making text within this publication compatible to existing documentation within the Einstein Toolkit.

`Cactus` simulations require an executable to be compiled, which is done typically by the users themselves. This setup allows for quick adaptation to changes in the local environment, e.g., updated libraries, but also for a simple way to tailor the source code to the users' needs, most of the time through usage of own modules. This executable has one mandatory argument: a *parameter file*. This is a simple text file, containing key-value pairs of parameter names and the desired settings within the simulation. An executable from all thorns within the Einstein Toolkit can be used to model a variety of physical scenarios. The parameter file is used to choose which of these should be realized.

An often used term within `Cactus` is a so-called *grid function*. This is a (discretization of a) variable which is defined on every point of a given grid. Examples are the rest-mass density of a fluid which is defined in all cells within a given domain, or the components of the metric tensor in general relativity. In parallel simulations, a grid function is typically split across processes, allowing storage of large grid functions and parallel work on each part.

2.2. Carpet

`Cactus` separates physics code from infrastructure code, i.e., a typical physics thorn will not contain any memory management, parallelization, Input/Output (IO) or mesh

refinement code. Most of these tasks are bundled in one special thorn, the so-called *driver thorn*, or driver.

Two drivers are provided with the ET, `PUGH` and `Carpet`. `PUGH` implements a uniform Cartesian grid, while `Carpet` [14–16] provides more than unigrid: Berger-Oliger style [17] adaptive mesh refinement (AMR). Its capabilities include

- splitting grid functions and arrays among the MPI processes,
- setting up a mesh refinement grid hierarchy,
- communicating grid function information between MPI processes,
- communicating between refinement levels by prolongation and restriction,
- modifying grid hierarchy (regridding) when requested,
- performing parallel IO.

Together with the `Llama` code [18], Carpet can provide multiblock infrastructure, providing different patch systems that cover the simulation domain by a set of overlapping patches. `Llama` is publicly available, but, as of writing, not yet part of the Einstein Toolkit.

For the sake of simplicity, and because of the almost exclusive usage of `Carpet` within numerical relativity using the Einstein Toolkit, we will in the following concentrate exclusively on examples using the AMR mesh refinement driver, without multiple patches.

2.3. Arrangements

Arrangements are collections of *thorns*, usually signaling a common task, or a common origin. This grouping is solely done for the benefit of a better overview for the user. Being part of one or another arrangement does not have any special meaning for a `Cactus` thorn beyond that.

Let us first give an overview on some of the basic `Cactus` and ET arrangements, before we present some details on a few of the more important thorns.

2.3.1. Core Cactus arrangements The main core `Cactus` arrangements include the following.

Arrangements

CactusBase

Provides infrastructure thorns for boundary conditions, setting up the coordinates, Input and Output, symmetries and time.

CactusNumerical

Provides numerical infrastructure thorns for time integration, artificial dissipation, symmetry boundary conditions, setting up spherical surfaces, interpolation, Method of Lines (MoL) implementation (see section 4.2.5 for more details), and many others.

CactusUtils

Provides some utility thorns: Formaline (see section 2.4.3 for details), nan-checking, termination triggering and timer reports.

ExternalLibraries

Provides external libraries—LAPACK, GSL, HDF5, FFTW, Lorene (for initial data), MPI, among others—<mark>that are automatically configured and compiled if they are not found on the user's machine</mark>.

2.3.2. EinsteinBase `EinsteinBase` thorns <mark>define and register basic variables within numerical relativity</mark>. Thorns that make use of or modify, for example, the ADM [19, 20] variables, or the stress-energy tensor, should <mark>inherit</mark> these thorns instead of defining their own. See Figure 1 for a sketch showcasing typical relations between these thorns and other "user" thorns. <mark>Almost all thorns relating to the physics of numerical relativity inherit from some, if not all of these thorns</mark>. It is vital even to a new user to have at least an overview of their structure. Main thorns include:

C++

ADMBase

Defines groups of grid functions for <mark>basic spacetime variables</mark>, based on the $3 + 1$ ADM construction [19, 20], which makes it the natural choice of a common foundation for exchanging data between modules using different formalisms. Concretely, `ADMBase` defines:

- 3-metric γ_{ij} (`gxx, gxy, gxz, gyy, gyz, gzz`),
- extrinsic curvature K_{ij} (`kxx, kxy, kxz, kyy, kyz, kzz`),
- lapse α (`alp`), shift β^i (`betax, betay, betaz`),
- time derivative of lapse $\partial_t \alpha$ (`dtalp`) and time derivative of shift $\partial_t \beta^i$ (`dtbetax, dtbetay, dtbetaz`).

`ADMBase` also defines basic parameters to choose the initial data, evolution method and other details common to a number of other thorns, e.g., schedule groups (specific points in time within a simulation) for other thorns to schedule their routines modifying the `ADMBase` variables. We stress that, by inheriting these variables from `ADMBase`, different thorns (typically written by different people) are then able to interact smoothly.
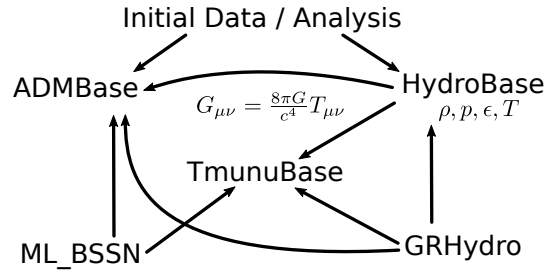
HydroBase

Defines basic variables and grid functions for hydrodynamics evolutions, e.g.:

- rest mass density ρ (`rho`),
- pressure P (`press`),
- specific internal energy ϵ (`eps`),
- contravariant fluid three velocity v^i (`vel[3]`),
- Lorentz factor W (`w_lorentz`),
- electron fraction Y_e (`Y_e`),
- temperature T (`temperature`),
- specific entropy per particle s (`entropy`),
- contravariant magnetic field vector B^i (`Bvec[3]`).

`HydroBase` also sets up scheduling groups for recommended interaction with the fluid variables.

Figure 1. Simplified overview over the central thorns of `EinsteinBase`: `ADMBase`, `HydroBase`, and `TMunuBase`, together with some example "user" thorns, and their dependencies. In this case "user" thorns are `ML_BSSN` for curvature evolution, `GRHydro` for magneto-hydrodynamics evolution and initial data and analysis thorns, all not directly depending on the evolution thorns, nor on each other.

TmunuBase

Defines grid functions for <mark>stress-energy tensor</mark> ("right-hand-side" of Einstein's equations):

- time component T_{00} (`eTtt`),
- mixed components T_{0i} (`eTtx, eTty, eTtz`),
- spatial components T_{ij} (`eTxx, eTxy, eTxz, eTyy, eTyz, eTzz`).

`TmunuBase` also sets up scheduling groups for other thorns to schedule routines that adds to the stress-energy tensor.

2.3.3. EinsteinInitial Arrangement `EinsteinInitial` contains several initial data thorns. The most widely used of these include the following:

TwoPunctures

Very efficient pseudo-spectral code that computes puncture-type binary black hole initial data [21].

TOVSolver

Initial data for a single TOV star.

2.3.4. EinsteinEvolve / McLachlan These arrangements contain several thorns related with the numerical evolution of Einstein's field equations:

McLachlan

`McLachlan` [22–24] is ET's spacetime evolution code. It uses a accurate finite differencing scheme up to eighth order with corresponding Kreiss-Oliger dissipation terms, to discretize spacetime variables in the BSSN form[9, 10, 25]. It is designed to inter-operate through the `ADMBase` and `TmunuBase` interface. All `McLachlan` thorns are auto-generated from tensor equations via `Kranc` [26–28], a `Mathematica`

application which converts a high-level continuum description of a PDE into a highly optimized module for `Cactus`.‡

GRHydro

General relativistic magneto-hydrodynamics (GRMHD) matter evolution [8, 29, 30], also designed to inter-operate through the `ADMBase` and `TmunuBase` interface.

PunctureTracker

Takes care of black hole tracking, e.g., to allow the automatic steering of mesh refinement grids according to black hole positions.

NewRad

Implements radiative, Sommerfeld-type, boundary conditions on specified grid functions [31].

2.3.5. EinsteinAnalysis This arrangement includes several thorns useful for analysis:

AHFinderDirect

This thorn finds black hole apparent horizons§ given the 3-metric and extrinsic curvature [32, 33]. `AHFinderDirect` is very fast and accurate, but requires an initial guess for the apparent horizon position.

WeylScal4

Calculates the Weyl scalar Ψ_4, often used within the context of the Newman-Penrose formalism [34]. Ψ_4, in the appropriate frame, can be shown to encode the outgoing gravitational radiation of an asymptotically flat system, making this quantity a very useful one for gravitational wave analysis. `WeylScal4` is auto-generated from tensor equations via `Kranc`.

Multipole

Decomposes arbitrary grid functions into spin-weighted spherical harmonics. Often used in combination with `WeylScal4`.

ADMAnalysis

Calculates several quantities from the ADM variables. In particular it can compute, if requested,

- the trace of the extrinsic curvature (`trK`);
- the determinant of the metric (`detg`);
- the components of the metric and extrinsic curvature in spherical coordinates (`grr`, `grq`, `grp`, `gqq`, `gqp`, `gpp`, `Krr`, `Krq`, `Krp`, `Kqq`, `Kqp`, `Kpp`);
- the components of the Ricci tensor (`Ricci11`, `Ricci12`, `Ricci13`, `Ricci22`, `Ricci23`, `Ricci33`) and the Ricci scalar (`Ricci`).

EHFinder

Finds event horizons in numerical spacetimes by integrating a null surface

‡ We note that `Mathematica` is not needed to compile the thorn.
§ More generically, `AHFinderDirect` looks for a closed 2-surface with S^2 topology having any desired constant expansion.

backwards in time. `EHFinder` is a post-processing analysis thorn. It is therefore necessary to evolve the initial data forward in time, beforehand, while outputting enough 3D data. The 3D data is then read back in, in reverse order. `EHFinder` can follow the event horizon during the merger of two (or more) black holes into one black hole. For more information, we refer to [35] and to the thorn's documentation.

`ADMConstraints / ML_ADMConstraints`

`ADMConstraints` calculates the ADM constraints violation (`ham`, `hamnormalized`, `momx`, `momy`, `momz`) from the ADM variables. Note that `McLachlan` contains thorn `ML_ADMConstraints` which calculates the same quantities, but directly using potentially higher-order derivatives, and is, in general, preferred over ADMConstraints.

2.4. Tools

Having briefly described some of the main arrangements provided with ET, let us now turn our attention to some tools that are provided as well.

2.4.1. SimFactory *SimFactory* (for Simulation Factory), to be discussed in more detail in section 4.3, is a tool that incorporates a set of abstractions for tasks which are necessary to successfully use the `Cactus` framework. These abstraction layers include:

- accessing remote systems and synchronizing source code trees;
- configuring and building on different systems semi-automatically;
- providing maintained list of supercomputer configurations;
- managing simulations (follow "best practices" and avoid human errors).

2.4.2. GetComponents As we have seen, the ET is composed of several different independent components (`Cactus`, `Carpet`, and several thorns). Typically, these different parts are hosted at different repositories, with different version control systems, and are maintained by different groups. `GetComponents` [36, 37] provides a unified way of downloading all of these components in a user-friendly way, hiding this complexity from the user.

`GetComponents` is a script with one mandatory argument, a so-called thornlist. This is a simple text file describing the locations and methods of how to retrieve components. The ET provides one such thornlist, describing these details for all its components (see section 3.2). This thornlist can be either given as URL, or as local file name. The most interesting optional parameter to `GetComponents` for new users is probably `--parallel`, to enable parallel checkouts, reducing the time to retrieve the whole toolkit considerably at times. Help and a list of other options can be obtained via the `--help` option.

2.4.3. Formaline Results obtained via computation have to be repeatable. However, ensuring this in practice can be a tedious task. One tool which can greatly help here is `Formaline`. All a user has to do is to compile and active this thorn. `Formaline` then:

- Ensures that simulations are and remain repeatable, remembering exactly how they were performed.

- Takes snapshots of source code, system configuration; stores it in executable and/or git repository.

- Tags all output files.

It should be noted that the usage of `Formaline` is in no way a guarantee that simulations are, in fact, repeatable exactly. Almost no simulation can be repeated bit-identical due to the parallel nature of some algorithms and their inherent "random" numerical errors. However, `Formaline` automates tasks a user *can* do in such a convenient way that most users actually *do* them.

3. Using the ET

3.1. Requirements

`Cactus` and the Einstein Toolkit should run on all major variants of Unix. Main requirements are the following: `C`, `C++` and `Fortran 90` compilers; an MPI implementation (e.g., OpenMPI, MPICH; will be provided by the ET if not found); HDF5 (will also be provided if not found); `Perl` and `Python`. Obtaining the ET directly from its repositories also requires `Subversion` and `git` tools. All of these requirements are usually conveniently available via the respective distribution repositories, or are commonly available on supercomputers.

In the following we will be assuming that all such tools are available.

3.2. Downloading and Running

This tutorial will be based on the latest stable ET release, which at the time of writing is "Ørsted" (released on November 8th, 2012). We expect a new release of the ET to be published between submission, and publication of this article. Exchanging the string `ET_2012_11` by `ET_2013_05` in the following should be all that is needed to use the new release.

To download the ET, one must first obtain the `GetComponents` script and point it to the appropriate *thornlist*:

```
curl -O https://raw.github.com/gridaphobe/CRL/master/GetComponents
perl GetComponents https://svn.einsteintoolkit.org/manifest/branches/
   ET_2012_11/einsteintoolkit.th
```

A *thornlist* is a text file containing the list of thorns to be downloaded together with the path and version controlling system used for each thorn. Such a file will typically have the following structure:

```
!CRL_VERSION = 1.0

!DEFINE ROOT = Cactus
!DEFINE ARR  = $ROOT/arrangements
```

```
!DEFINE COMPONENTLIST_TARGET = $ROOT/thornlists/

!DEFINE ET_RELEASE = ET_2012_11

# Cactus Flesh
!TARGET   = $ROOT
!TYPE     = svn
!AUTH_URL = https://svn.cactuscode.org/flesh/branches/$ET_RELEASE
!URL      = http://svn.cactuscode.org/flesh/branches/$ET_RELEASE
!CHECKOUT = Cactus
!NAME     = .

# Cactus thorns
!TARGET   = $ARR
!TYPE     = svn
!AUTH_URL = https://svn.cactuscode.org/arrangements/$1/$2/branches/
    $ET_RELEASE
!URL      = http://svn.cactuscode.org/arrangements/$1/$2/branches/
    $ET_RELEASE
!CHECKOUT =

CactusBase/Boundary
CactusBase/CartGrid3D
CactusBase/CoordBase
CactusBase/Fortran
CactusBase/InitBase
CactusBase/IOBasic
CactusBase/IOUtil
CactusBase/Time

(...)
```

The user is also free to download the list first to a file and then add its own private thorns. The `GetComponents` script accepts a thornlist file as an argument. To download all the thorns specified in the `einsteintoolkit.th` file (in this example using the `--parallel` option to retrieve repositories in parallel to speed-up the process):

```
perl GetComponents --parallel einsteintoolkit.th
```

Once `GetComponents` finishes downloading one should have a folder called `Cactus` with the following structure:

```
Cactus/
├── arrangements/
├── bin/
├── doc/
├── lib/
├── manifest/
├── par/
├── repos/
├── simfactory/
```

```
src/
thornlists/
utils/
CONTRIBUTORS
COPYRIGHT
Makefile
```

We are now ready for the configuring and compiling stage. The first step here is choosing the config (`*.cfg`) file for the machine. A few examples (which include the configuration files for Fedora, Debian, Ubuntu and OS X) are provided under `./Cactus/simfactory/mdb/optionlists/`. The second step is having the required thornlist for the configuration intended to be built. Different configurations, compiled with different thornlists, are free to co-exist. Note that, typically, one will *not* be compiling *all* the thorns provided with the ET. Compilation is time-consuming, and different configurations also take a significant amount of disk space. One therefore typically builds a thornlist that is as small as possible, including only the required thorns. Care should be taken, though, as there are often non-trivial dependencies between thorns. If one thorn which is required by another thorn is not mentioned in the thornlist, compilation will abort (with the corresponding error message).

The traditional way of compiling `Cactus` (i.e., without SimFactory—see below) is as follows. First, the configuration is created

```
cd Cactus
make ET-config options=<machine config file> THORNLIST=<thornlist>
```

This creates a configuration called "ET", but any other name could be chosen here. Once the configuration is done, the compilation process is simply

```
make -j <number of processes> ET
```

If everything is compiled correctly, the executable `cactus_ET` will be created under `./exe/`. These steps need to be repeated for every different configuration (typically, with different thornlists) built.

We should now be ready for running. For this, a parameter file is needed, specifying which thorns to use within the simulation (not all compiled thorns need to be active), and which specific model parameters have been chosen. A few examples are provided under `./Cactus/par/`, including the parameter file for an inspiraling collision of black holes in vacuum (`qc0-mclachlan.par`) as well as the parameter file for a static TOV star (`static_tov.par`). The typical procedure for running, as with other MPI executables, is

```
mpirun -np <num procs> ./exe/cactus_ET <parameter file>
```

3.3. Analyzing the output

Output is typically controlled by the following thorns:

CarpetIOASCII

> writes 0, 1, 2 or 3D output from the specified variables onto a text file (`.asc`)

CarpetIOScalar

> performs scalar reductions (maximum, minimum, norm, ...) of the specified variables and writes onto a text file (`.asc`);

CarpetIOHDF5

> writes 0, 1, 2 or 3D output from the specified grid functions onto an HDF5 file; also handles check pointing (`.h5`).

All of these thorns include, among others, the following parameters: `out?D_vars`, `every?D_vars`, where `?=0,1,2,3` for 0-, 1-, 2- or 3-dimensional output respectively. `out?D_vars` is a list of grid functions to output at every `every?D_vars` iteration. 3-dimensional output is by default produced over the entire grid; 2-dimensional output is given at specified 2D-planes (the default being the `xy`, `xz` and `yz` planes containing the origin); 1-dimensional output is given along lines (default being the `x`, `y` and `z` axis); 0-dimensional output is given at specified points (default being the coordinate origin).

For visualizing 1-dimensional ASCII output, standard tools like `matplotlib`, `gnuplot` and `xmgrace` are often used; for 2- and 3-dimensional HDF5 output, `VisIt` and `DV` are popular (freely available) options.

4. Anatomy of a Cactus thorn

The knowledge about the structure of `Cactus` thorns is strictly speaking not necessary for new users, as information about their interface can be obtained via built documentation. However, a short overview is given here to illustrate how first steps towards development within the toolkit look like.

Any given `Cactus` thorn will have the following directory structure (under `./Cactus/arrangements/`)

```
ArrangementName/ThornName/
├── COPYRIGHT
├── README
├── test/
├── doc/
├── par/
├── configuration.ccl
├── interface.ccl
├── schedule.ccl
├── param.ccl
└── src/
    └── make.code.defn
```

COPYRIGHT, README, `doc/`, `test/` and `par/` are optional; `doc/` will be typically used to store some documentation related with the thorn, and `par/` can have example

parameter files for the thorn. Under `src/`, the file called `make.code.defn` is needed with the list of files to be compiled.

4.1. Cactus configuration files

Knowledge of these files is strictly only necessary for developers, as information within is automatically included in build documentation. However, we include a short description of their contents here because even for a new user it can be useful at times to have at least a general overview of the interface of `Cactus` thorns with the flesh.

Cactus thorns use four files to specify their interface with the `Cactus` flesh (three compulsory and one optional). They can be found in the top level thorn directory, using the `*.ccl` file name extension (`Cactus` Configuration Language, see [37] for extensive details).

The file `configuration.ccl` (optional) lists inter-thorn build dependencies. `interface.ccl` is used to define thorn-wide variables, grid functions, and shared functions. It consists of a header block detailing the thorn's relationship with other thorns, a block stating which include files are used from other thorns, and which include files are provided by this thorn, blocks with aliased functions provided or used by this thorn and a series of blocks listing the thorn's global variables. Note that functions can be called by a different name within a given thorn. `schedule.ccl` takes care of all the function scheduling and controls the global storage of grid functions. And finally, `param.ccl` defines all parameters (which, in contrast to regular variables can be set at start-time) and sets their default values.

4.2. Thorn Examples

In the following, we present a few examples of thorns within the Einstein Toolkit, with the aim of deepening the understanding of `Cactus` thorns obtained in previous sections.

4.2.1. ADMBase This thorn, already briefly covered in section 2.3.2, is fundamental for most (if not all) numerical relativity simulations, providing the core infrastructure for thorns implementing general relativity on a 3D grid in the 3+1 formalism. `ADMBase` provides the basic variables (3-metric, extrinsic curvature, lapse function and shift vector) for the 3+1 formalism. These grid functions are then inherited by every thorn using them. This allows a well-defined interaction between thorns implementing fundamentally different functionality, e.g., thorns providing initial data, evolution methods and analysis routines for the 3+1 formalism.

The ET uses this thorn, ensuring that different analysis and initial data thorns are able to communicate with each other and evolution thorns, independently of the variables used for the evolution (typically, *not* the ADM variables). Generically, an initial data thorn will initialize the `ADMBase` variables to the initial data describing the desired physical configuration. The evolution thorn will then import the `ADMBase` variables, convert them into its evolution variables (the BSSN variables, for example)

and perform the evolution using these variables. <mark>At every time step, the `ADMBase` variables are reconstructed from the evolved variables so that they always contain the current solution</mark>. Analysis thorns such as those for horizon-finding or wave extraction then use the `ADMBase` variables as well. This effectively decouples initial-data solvers, evolution methods and analysis routines, making them inter-changeable and compatible with each other.

4.2.2. HydroBase Similarly to `ADMBase`, `HydroBase` defines and stores the primitive variables common among hydrodynamic simulations, needed parameters and schedule groups for the main functions of a hydrodynamics code. `HydroBase` does not contain the actual source code of typical routines of hydrodynamics codes; it merely provides a common setup in which hydrodynamics codes can schedule their routines.

This way, different modules of hydrodynamics codes (such as initial data solvers or analysis modules) working only with entities defined in `HydroBase` can be used interchangeably. Yet another advantage is that the output generated by different hydrodynamics codes within `Cactus` would be the same, including variable names and unit conventions, thus improving the ability to compare results from different codes.

`HydroBase` also sets up scheduling blocks organizing main functions which modules of a hydrodynamics code may need. These scheduling blocks are optional, but when used may simplify existing codes and make them more inter-operable. Currently the scheduling blocks are:

- initializing the primitive variables;
- converting primitive variables to conservative variables;
- calculating the right hand side in the method of lines (MoL);
- setting and updating an excision mask;
- applying boundary conditions.

This way, initializing the primitive variables, recovering the conservative variables, and basic atmosphere handling can be implemented in different thorns while allowing a central access point for analysis thorns.

4.2.3. SphericalSurface `SphericalSurface` defines two-dimensional surfaces with spherical topology. <mark>The thorn itself only acts as a repository for other thorns to set and retrieve such surfaces, making it a pure infrastructure thorn</mark>. Different thorns can then update a given spherical surface, while others then read that information without having to know about the first thorn.

Within the ET, uses of spherical surfaces include the following: storing apparent horizon information (used by `AHFinderDirect`); tracking black hole location (by thorn `CarpetTracker`), information which is then used to determine where to perform mesh refinement.

The thorn provides, for each surface, a two-dimensional grid array `sf_radius` and grid scalars `sf_origin_x`, `sf_origin_y`, and `sf_origin_z`. The number of surfaces is determined by the parameter `nsurfaces`, which has to be set in the parameter file.

4.2.4. Multipole `Multipole` is a thorn that decomposes any `Cactus` grid function into spherical harmonics on coordinate spheres of given radii. One application is to use it in combination with the `WeylScal4` analysis thorn to produce the mode decomposition of the Weyl scalars, often used in numerical relativity.

Specifically, a grid function $u(t, r, \theta, \varphi)$ is expanded in spin-weight s spherical harmonics:

$$u(t, r, \theta, \varphi) = \sum_{l=0}^{\infty} \sum_{m=-l}^{l} C^{lm}(t, r) {}_sY_{lm}(\theta, \varphi),$$

where the coefficients $C^{lm}(t, r)$ are given by

$$C^{lm}(t, r) = \int {}_sY_{lm}^* u(t, r, \theta, \varphi) r^2 d\Omega.$$

The thorn computes $C^{lm}(t, r)$ for the requested grid functions at given coordinate radii r_i.

To use this thorn, one specifies the grid functions to decompose, as well as the number and radii of the extraction spheres, as in the following example parameter file:

```
ActiveThorns = "AEILocalInterp Multipole"
Multipole::nradii       =   3
Multipole::radius[0]    =  10
Multipole::radius[1]    =  20
Multipole::radius[2]    =  30
Multipole::variables    = "MyThorn::u"
```

The default parameters will compute all $l = 2$ modes assuming a spin-weight $s = 0$ on every iteration of the simulation.‖ The coefficients C^{lm} will be output in files with names of the form `mp_<var>_l<lmode>_m<mmode>_r<rad>.asc`:

```
mp_u_l2_m2_r10.00.asc
mp_u_l2_m-1_r20.00.asc
```

Note that, unlike most thorns, and since the spherical harmonic coefficients are defined on coordinate spheres (and *not* on the whole grid), `Multipole` takes care of the output by itself. There is thus no need to give `Multipole` grid functions to `CarpetIOASCII` or `CarpetIOHDF5` in order to have output.

4.2.5. MoL The Method of Lines (MoL) is a technique for numerically integrating time dependent partial differential equations (PDEs). The method discretizes only the spatial derivatives of the PDE, thereby converting it into a large system of ordinary differential

‖ Note that for gravitational wave extraction, one will typically be interested in a spin-weight $s = -2$ decomposition.

equations (ODEs). This system can then be integrated using well-known methods (such as Runge-Kutta and Crank-Nicolson). Consider the following PDEs system

$$\partial_t \mathbf{q} + \mathbf{A}^i(\mathbf{q})\partial_i \mathbf{B}(\mathbf{q}) = \mathbf{s}(\mathbf{q}),$$

which we re-write in the form

$$\partial_t \mathbf{q} = \mathbf{L}(\mathbf{q}). \tag{1}$$

Assuming that the right-hand side is discretized, this is nothing but a system of ODEs, which, as we mentioned, can be integrated using known ODEs integrators.

The `MoL` thorn provides a simple interface that implements this technique inside `Cactus`. We will here provide only some basic information about the thorn; for more information we refer to its documentation.

The idea behind the `MoL` thorn is that the user should not be concerned with writing the integration procedure itself. Instead, in the user's evolution thorn, grid functions that are to be evolved are *flagged* as *evolved* variables, and their right-hand side (cf. equation (1)) is given. The `MoL` thorn is then called where appropriate and takes care of the evolution procedure itself, using the method chosen by the user (currently supported include Runge-Kutta of different orders, Iterative Crank-Nicolson (with and without averaging)).

For `MoL`, grid functions need to be split into four categories: variables for which we have a time evolution equation are *evolved*; variables which the thorn sets but does not evolve are *constrained*; any other variable which the thorn reads during evolution is a *save and restore* variable; other variables can be ignored. Bear in mind that, generically, one can have different variables being evolved by different thorns. A grid function can thus be an *evolved* variable from the point of view of one thorn and a *save and restore* from the point of view of another—each thorn should register the grid function as they see it. An example would be the metric components in a general relativistic hydrodynamics evolution: the hydrodynamics thorn will see the metric as a *save and restore* variable whereas the spacetime evolution thorn will see the metric as *evolved.* The `MoL` thorn takes care automatically of the right treatment of each variable.

In section 5.3 we will present a simple example of how to use `MoL`, to see how this procedure works in practice when evolving the wave equation.

4.3. Using SimFactory

The *Simulation Factory* [38, 39] is designed to manage several tasks necessary to set up and execute successful numerical simulations using `Cactus`. Consider the following, very typical, scenario: an ET user performs all his code development locally on a laptop, where a few simple low resolution runs can be performed. To test more time consuming cases the user may use a local workstation, which implies syncing the code and parameter files, compiling and running the code on the workstation, and then analyzing the output. For higher resolution cases, the user may have to resort to computer clusters, and the

whole process of syncing, compiling, running and analyzing will be repeated. It is easy to see that this process can be very error prone.

SimFactory provides an easy abstraction layer to this process, allowing the user to do all his code development locally, and then syncing and compiling to all other machines he may use with simple commands. A record is also kept of where and what simulations have been run.

To use SimFactory, we must first configure it. For machines known to SimFactory this step consists essentially in configuring the user name and email address (for job submission status emails). If a given machine configuration is not already stored, SimFactory also needs an optionlist to use for the ET build.

```
cd Cactus
./simfactory/bin/sim setup --optionlist=<machine .cfg file>
```

The ET will then be built using

```
./simfactory/bin/sim build [<configurationname>] --thornlist=<
    thornlist>
```

The configuration name can be omitted, in which case it will default to "sim". Different configurations, with different thornlists and configuration files, can co-exist.

To run, for example, the `static_tov.par` parameter file and follow the output, one would then execute

```
./simfactory/bin/sim submit static_tov --configuration <
    configurationname> --parfile=par/static_tov.par --procs=1 --
    walltime=8:0:0
./simfactory/bin/sim show-output --follow static_tov
```

In this example we created a simulation called "static_tov", only one processor was requested and the simulation will abort after the specified walltime of 8 hours. As for building, if `--configuration <configuration>` is omitted, the default "sim" configuration is used.

This specific parameter file sets up a static TOV star (a model of a single neutron star) with a mass of 1.4 solar masses, and integrates the combined relativistic fluid dynamics and spacetime evolution equations in time. The spacetime is evolved using the BSSN 3+1 formulation of Einstein's equations and the fluid is evolved using a high resolution shock capturing method.

To check the status of the simulation, use

```
./simfactory/bin/sim list-simulations
```

If you accepted the default values at the setup stage, simulations will run in the `$HOME/simulations` folder (which needs to be created beforehand).

4.3.1. Configuring additional machines Let us now see how to add an additional machine (say, your local workstation) to SimFactory. The first step is adding the configuration `<machine name>.ini` file to `./simfactory/mdb/machines/`. Such a file can be easily adapted from the provided `generic.ini`:

```
[generic]

# Machine description
nickname        = generic
name            = Generic Machine
location        = somewhere
description     = Whatever
status          = personal


# Access to this machine
hostname        = generic.some.where
aliaspattern    = ^generic\.some\.where$


# Source tree management
sourcebasedir   = /home/@USER@
optionlist      = generic.cfg
submitscript    = generic.sub
runscript       = generic.run
make            = make -j2
basedir         = /home/@USER@/simulations
ppn             = 1   # or more
max-num-threads = 1   # or more
num-threads     = 1   # or more
nodes           = 1
submit          = exec @SCRIPTFILE@ < /dev/null > /dev/null 2> /dev/
   null & echo $!
getstatus       = ps @JOB_ID@
stop            = kill @JOB_ID@
submitpattern   = (.*)
statuspattern   = "^ *@JOB_ID@ "
queuedpattern   = $^
runningpattern  = ^
holdingpattern  = $^
exechost        = echo localhost
exechostpattern = (.*)
stdout          = cat @SIMULATION_NAME@.out
stderr          = cat @SIMULATION_NAME@.err
stdout-follow   = tail -n 100 -f @SIMULATION_NAME@.out
   @SIMULATION_NAME@.err
```

For the most straightforward cases, it should be enough changing the Machine description, hostname, aliaspattern and optionlist. Examples for optionlists can be found in `./simfactory/mdb/optionlists/`, and if none of these can be used directly, a new list can be created and placed within that directory.

Once this is done, the newly added machine should appear in the list of known machines:

```
./simfactory/bin/sim list-machines
```

and its configuration

```
./simfactory/bin/sim print-mdb <machine>
```

We should now be ready to synchronize our files

```
./simfactory/bin/sim sync <machine>
```

Once that is done, we repeat the building process from before, but with the `--remote <machine>` argument:

```
./simfactory/bin/sim --remote <machine> build [<configuration name>]
   --thornlist=<thornlist>
```

All SimFactory commands accept the `--remote` option, taking a machine name as argument. This allows one to build, submit and manage simulations on several machines without ever having to directly login.

For more information, check [39].

5. Examples

So far, we described core parts of the Einstein Toolkit as generally as possible to give a consistent overview of its capabilities and components. Quite often however, simple, additional examples can help to provide deeper understanding. This is especially true in a workshop-like setting where teaching time is usually scarce. Therefore, we present in this section three very hands-on examples of how to use the Einstein Toolkit.

The first two examples, an evolution of a black hole binary, and the evolution of a stable star, only present how users would set parameters to obtain the respective simulations, using existing code and without the need to change it. In contrast to these two, the third and last example show-cases an implementation of the wave-equation, in particular using `MoL`.

5.1. Binary black hole coalescence

The ET provides, out-of-the-box, all the tools needed to evolve, with minimal input from the user, a vacuum binary black hole coalescence. We will here not describe all the modules involved in such a simulation; we will merely briefly go over the provided parameter file adapted for such an evolution and note the parameters that a user would first change for slightly different physical configurations.

Having obtained the ET as explained in the previous sections, a few example parameter files are provided within the directory `./Cactus/par/`. In particular, `./Cactus/par/arXiv:1111.3344/bbh/BBHLowRes.par` (and corresponding higher resolution versions) uses the thorn `TwoPunctures` [21] to set up initial data for two black holes located at the x-axis with (opposite) linear momentum along the y-axis. This configuration is then evolved using the `ML_BSSN` (`McLachlan` BSSN) thorn. `AHFinderDirect` searches for the black hole's apparent horizons at the designated intervals and `WeylScal4` extracts gravitational wave information. We should note that modern initial data using the `TwoPunctures` technique uses a few more tricks to obtain more realistic initial data (e.g. by reducing the initial eccentricity), but the simpler

procedure presented here is sufficient to capture the main ideas of such an evolution, and is therefore preferred in a workshop or tutorial setting.

A streamlined, easier to follow, version of BBHLowRes.par is provided at http://blackholes.ist.utl.pt/nrhep2/?page=material, under the name inspiral_d06_lres.par. Let us now explore some of the parameters within this file. We begin by noting the following group:

```
ActiveThorns = "ReflectionSymmetry RotatingSymmetry180"
ReflectionSymmetry::reflection_z   = yes
CoordBase::domainsize = "minmax"
CoordBase::xmin =     0.00
CoordBase::ymin = -120.00
CoordBase::zmin =     0.00
CoordBase::xmax = +120.00
CoordBase::ymax = +120.00
CoordBase::zmax = +120.00
CoordBase::dx   =     2.00
CoordBase::dy   =     2.00
CoordBase::dz   =     2.00
```

This sets up the overall size of the numerical grid, which spans $[[x_{\min}, x_{\max}], [y_{\min}, y_{\max}], [z_{\min}, z_{\max}]]$ = $[[0, 120], [-120, 120], [0, 120]]$. The problem has a mirror-symmetry in the plane of the inspiral, which is used here to reduce the computational cost by a factor of 2 by not evolving the domain with $z < 0$. In addition, the π-symmetry in the problem can be used to only evolve points with $x > 0$ (populating the missing part by rotating the existing domain for 180 degrees along the z-axis). This again reduces the computational cost by about a factor of 2, and is achieved by activating thorn RotatingSymmetry180. The grid-spacing given here corresponds to the coarsest grid. The grid-spacing of the inner regions (using AMR) will then depend on how many refinement levels are chosen. Each level refines the region it is covering by a factor of two. A user wanting to experiment with different grid sizes and/or different resolutions would need to change these parameters accordingly.

The group

```
CarpetRegrid2::num_centres = 2

CarpetRegrid2::num_levels_1         =   7
CarpetRegrid2::position_x_1         =   3.0
CarpetRegrid2::radius_1[ 1]         =  64.0
CarpetRegrid2::radius_1[ 2]         =  16.0
CarpetRegrid2::radius_1[ 3]         =   8.0
CarpetRegrid2::radius_1[ 4]         =   4.0
CarpetRegrid2::radius_1[ 5]         =   2.0
CarpetRegrid2::radius_1[ 6]         =   1.0
CarpetRegrid2::movement_threshold_1 =   0.16

CarpetRegrid2::num_levels_2         =   7
CarpetRegrid2::position_x_2         =  -3.0
CarpetRegrid2::radius_2[ 1]         =  64.0
```
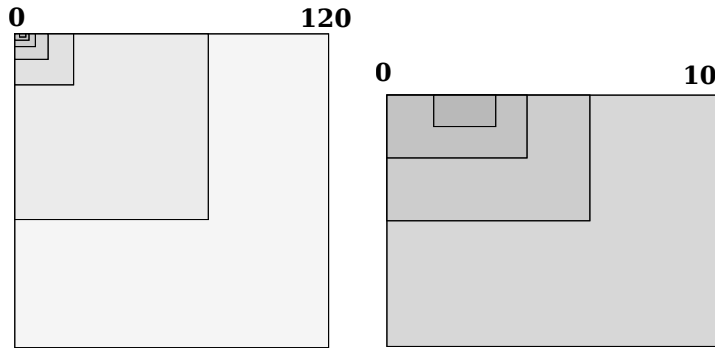
Figure 2. Sketch of the used grid setup. Note that only one of the finest region is shown because of the used symmetries. Also note that some of the overhead points are not shown, i.e. the ghost zones at the edge of refinement levels. On the right we show the same, but zoomed-in (with the outer-most three levels not shown).

```
CarpetRegrid2::radius_2[ 2]          =   16.0
CarpetRegrid2::radius_2[ 3]          =    8.0
CarpetRegrid2::radius_2[ 4]          =    4.0
CarpetRegrid2::radius_2[ 5]          =    2.0
CarpetRegrid2::radius_2[ 6]          =    1.0
CarpetRegrid2::movement_threshold_2  =    0.16
```

defines the number and radii of the inner refined regions, centered around each of the black holes. Should a user wish to change the initial position of the black holes, it is important to also modify the `CarpetRegrid2::position_x_1`, `CarpetRegrid2::position_x_2` settings accordingly.

In Figure 2 we show the used grid setup in the xy plane, including all nested grids. Note that only one of the finest region is shown because of the used symmetries.

The initial physical configuration is handled by the initial data thorn, which in this case is `TwoPunctures`. The group

```
TwoPunctures::par_b           =    3.001
TwoPunctures::par_m_plus      =    0.47656
TwoPunctures::par_m_minus     =    0.47656
TwoPunctures::par_P_plus [1]  =   +0.13808
TwoPunctures::par_P_minus[1]  =   -0.13808
```

defines the initial positions of the black holes to be at $(x, y, z) = (\pm 3.001, 0, 0)$ (by default, `TwoPunctures` places the black holes along the x-axis—use the parameter `swap_xz` to place them along the z-axis), sets the black holes' "bare mass" parameter and its Bowen-York linear momentum parameter. This would be the first place to change should a different physical configuration be needed.

When changing the initial position of the black holes, besides changing the position of the inner refined regions, it is also important to correspondingly change the initial guess for the apparent horizon finder:

```
AHFinderDirect::origin_x                           [1] = +3.0
AHFinderDirect::initial_guess__coord_sphere__x_center[1] = +3.0
```
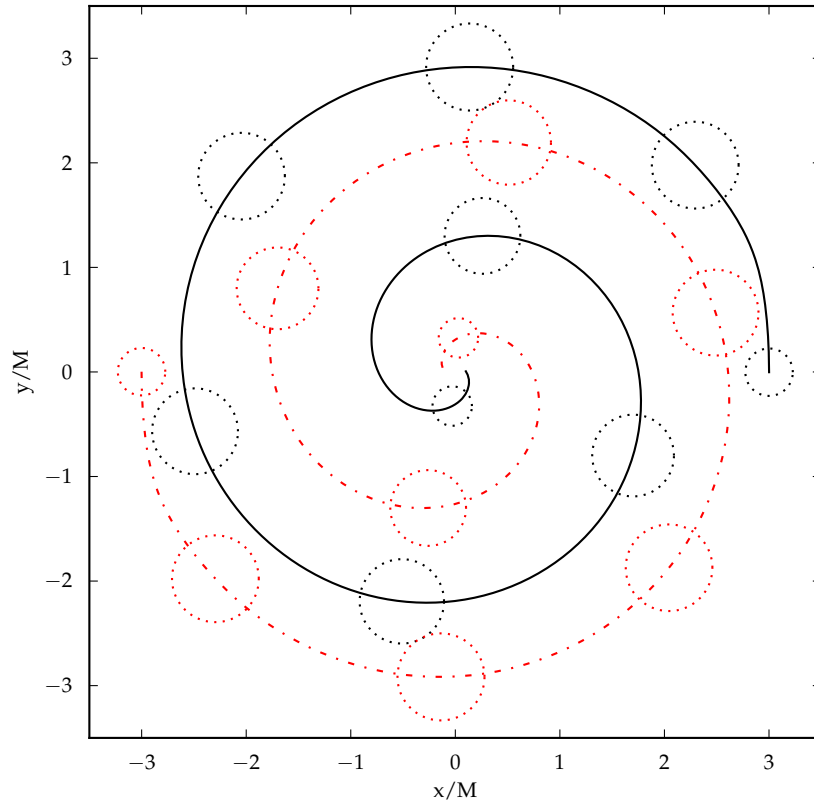
Figure 3. Puncture locations and mean radius of the corresponding apparent horizons (plotted every $15M$) during the binary inspiral.

```
AHFinderDirect::origin_x                                [2] = -3.0
AHFinderDirect::initial_guess__coord_sphere__x_center[2] = -3.0
```

In Figure 3 we plot the evolution of the mean apparent horizon radius of each black hole during the evolution of this setup. The script and data files used to produce the plot are available at `http://blackholes.ist.utl.pt/nrhep2/?page=material`. An easy way to produce a simpler version of the figure, also with `matplotlib`, is the following:

```
ipython --pylab
  In [1]: xbh1, ybh1 = np.loadtxt("BH_diagnostics.ah1.gp", usecols
     =(2,3), unpack=True)
  In [2]: xbh2, ybh2 = np.loadtxt("BH_diagnostics.ah2.gp", usecols
     =(2,3), unpack=True)
  In [3]: axis("equal")
  In [4]: plot(xbh1, ybh1, linestyle="-", color="black")
  In [5]: plot(xbh2, ybh2, linestyle="-.", color="red")
```

5.2. Simple TOV star

The `TOVSolver` routine in the Einstein Toolkit solves the standard TOV¶ equations [40, 41] for the pressure, enclosed gravitational mass M_e, and gravitational potential $\Phi = \log \alpha$ in the interior of a spherically symmetric star in hydro-static equilibrium. Details of the procedure, including equations, can be found in [7]. We will use these initial data to show linear oscillations of an otherwise stable TOV star.

The system setup by `TOVSolver` is evolved using the BSSN evolution system implemented in `McLachlan` and the hydrodynamics evolution system implemented in `GRHydro`.

For the simple evolution presented here, we set up a stable TOV star described by a polytropic equation of state with $K = 100$ and $\Gamma = 2$, and an initial central density of $\rho_c = 1.28 \times 10^{-3}$. This model can be taken to represent a non-rotating NS with a mass of $M = 1.4 M_\odot$, and an EOS which mimics a cold neutron star reasonably well, at least to the extend the EOS of such an object is currently known.

For reasonably accurate results of an evolution of such a system, runs are typically performed with fixed mesh refinement, using about 5 levels of refinement on a quadrant grid (symmetries provided by `ReflectionSymmetry` and `RotatingSymmetry180`). The outer boundaries are placed at a few hundred M (more then ten times the stellar radius), and refined boxes are centered around the star at the origin, each doubling the resolution. Typical resolutions on the finest grid covering the entire star are 0.500M to 0.125M.

Evolutions of this size take much too long for a fast-paced tutorial where participants are expected to run these on their own. Computational resources might also be an issue if these resolutions would be tried. For such a tutorial setting, though, a much lower resolution and a smaller global grid usually suffices, lowering the computational requirements and time substantially. It should, of course, be noted that results from such ultra-low resolution runs are not to be trusted for scientific work, but that is not the aim here. Increasing the parameters for resolution and grid size is an easy task, best left to the interested participant, as follow-up after the workshop. Thus, the evolution presented here only covers a global domain of 120M, just about ten times the stellar radius. Only four levels of mesh refinement are used, with a fine resolution of 1M. Note that this means that there are only a few tens of points across the star diameter— probably one of the lowest resolutions possible using the methods used here that still produces somewhat sensible results.

In Figure 4 we show the evolution of the central density of the star over an evolution time of 750M (3.6ms). The initial spike is due to the perturbation of the solution resulting from the mapping onto the evolution grid. After this (about 1ms), regular star oscillations can be seen, damped mostly by adiabatic heating.

Figure 4 represents a paper-quality plot of the central density. During a workshop, quality requirements are much less stringent and interactivity is much more important. One easy way to plot the same data much quicker would be using `matplotlib`:

¶ Tolman-Oppenheimer-Volkoff
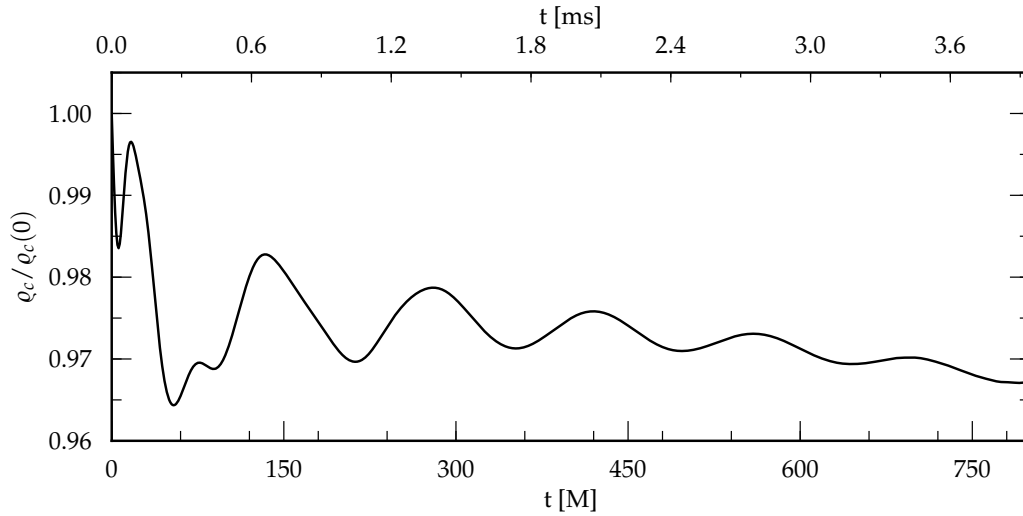
Figure 4. Evolution of the central density for the TOV star. Clearly visible is an initial spike, produced by the mapping of the one-dimensional equilibrium solution onto the three-dimensional evolution grid. Visible following this are damped oscillations of the central density of the star.

```
ipython --pylab
  In [1]: Fx, Fy = loadtxt("hydrobase::rho.maximum.asc", comments="#",
      usecols=(1,2), unpack=True)
  In [2]: plot(Fx, Fy/Fy[0])
```

As mentioned before, these results are in general to be considered quite poor. However, the goal was not accuracy here. This becomes clear when considering the computational requirements: about 400MB of memory and about 2.5 core-hours (2.5 hours on one current compute core). This code can take advantage of both multi-code machines, as well as multi-node clusters (usually a combination of these), reducing the overall physical runtime substantially.

5.3. WaveMoL

Let us explore a quite small thorn: `CactusExamples/WaveMoL`. While not a part of the Einstein Toolkit, this thorn illustrates the basic of usage of the previously introduced `MoL` (Method of Lines) `Cactus` thorn, which is used by `McLachlan` and `GRHydro`.

The `MoL` thorn, already introduced in section 4.2.5, provides a convenient way to implement the Method of Lines technique inside `Cactus`. The advantage of using this thorn is that one does not need to worry about writing the time integration procedure itself; instead, one simply *flags* the variables that will be evolved and provide their "right-hand-sides". `MoL` will then take care of the evolution procedure using one of the implemented methods, e.g., second, third of fourth order Runge-Kutta, or Iterative Crank-Nicolson (to be chosen by the user at start-time).

For our example, let us then consider the wave equation

$$\partial_t^2 \phi = \partial_{x^i}^2 \phi^i$$

To illustrate usage of the `MoL` thorn, we rewrite the equations in first order form, as was done in section 4.2.5:

$$\partial_t \Phi = \partial_{x^i} \Pi^i, \tag{2}$$

$$\partial_t \Pi^j = \partial_{x^j} \Phi, \tag{3}$$

$$\partial_t \phi = \Phi, \tag{4}$$

$$\partial_{x^j} \phi = \Pi^j. \tag{5}$$

The first three equations (five separate PDEs) will be evolved. The final equation is used to set the initial data and can be thought of as a constraint. This determines how each of these variables has to be registered with `MoL`.

To obtain `WaveMoL`, save the following text to a file (called, e.g., `WaveMol.th`)

```
!CRL_VERSION = 1.0
!DEFINE ROOT = Cactus
!DEFINE ARR  = $ROOT/arrangements
!DEFINE COMPONENTLIST_TARGET = $ROOT/thornlists/
!DEFINE CACTUS_RELEASE = Cactus_4.1.0
!TARGET   = $ARR
!TYPE     = svn
!AUTH_URL = https://svn.cactuscode.org/arrangements/$1/$2/branches/
   $CACTUS_RELEASE
!URL      = http://svn.cactuscode.org/arrangements/$1/$2/branches/
   $CACTUS_RELEASE
!CHECKOUT =
CactusExamples/IDWaveMoL
CactusExamples/WaveMoL
```

and then use `GetComponents` as previously:

```
./GetComponents WaveMol.th
```

You should now have (under `./Cactus/arrangements/CactusExamples/`) the following directory structure

```
WaveMoL/
 ├── configuration.ccl
 ├── interface.ccl
 ├── schedule.ccl
 ├── param.ccl
 └── src/
      ├── InitSymBound.c
      ├── Startup.c
      ├── WaveMoLRegister.c
      ├── WaveMoL.c
      └── make.code.defn
```

The `.ccl` files were already covered in section 4, so we will here look into the C source code files.

We start with the `WaveMoL_RegisterVars` routine, under `WaveMoLRegister.c`:

```
void WaveMoL_RegisterVars(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS;
  DECLARE_CCTK_PARAMETERS;
  CCTK_INT ierr = 0, group, rhs, var;

  group = CCTK_GroupIndex("wavemol::scalarevolvemol_scalar");
  rhs   = CCTK_GroupIndex("wavemol::scalarrhsmol_scalar");
  ierr += MoLRegisterEvolvedGroup(group, rhs);

  group = CCTK_GroupIndex("wavemol::scalarevolvemol_vector");
  rhs   = CCTK_GroupIndex("wavemol::scalarrhsmol_vector");
  ierr += MoLRegisterEvolvedGroup(group, rhs);

  var   = CCTK_VarIndex("wavemol::energy");
  ierr += MoLRegisterConstrained(var);
}
```

Note here how the variables `wavemol::scalarevolvemol_scalar` and `wavemol::scalarevolvemol_vector` are being flagged as *evolved* for `MoL`, with right-hand side `wavemol::scalarrhsmol_scalar` and `wavemol::scalarrhsmol_vector` respectively. As we explained in section 4.2.5, this allows the `MoL` thorn to evolve these grid functions from one time step to the next and there is no need for the user to explicitly code any time-integration routine.

For completeness, let us briefly look under the `WaveMoL.c` file, where we find the routine `WaveMoL_CalcRHS`:

```
void  WaveMoL_CalcRHS(CCTK_ARGUMENTS) {
  DECLARE_CCTK_ARGUMENTS;
  int i,j,k, index, istart, jstart, kstart, iend, jend, kend;
  CCTK_REAL dx,dy,dz, hdxi, hdyi, hdzi;

  /* Set up shorthands */
  dx = CCTK_DELTA_SPACE(0); dy = CCTK_DELTA_SPACE(1); dz =
     CCTK_DELTA_SPACE(2);
  hdxi = 0.5 / dx; hdyi = 0.5 / dy; hdzi = 0.5 / dz;
  istart = 1; jstart = 1; kstart = 1;
  iend = cctk_lsh[0]-1; jend = cctk_lsh[1]-1; kend = cctk_lsh[2]-1;

  /* Calculate the right hand sides. */
  for (k=0; k<cctk_lsh[2]; k++) {
    for (j=0; j<cctk_lsh[1]; j++) {
      for (i=0; i<cctk_lsh[0]; i++) {
        index = CCTK_GFINDEX3D(cctkGH,i,j,k);
        phirhs[index] = phit[index];
        phitrhs[index] = 0;
        phixrhs[index] = 0;
        phiyrhs[index] = 0;
```

```
          phizrhs[index] = 0;
      }
    }
  }
  for (k=kstart; k<kend; k++) {
    for (j=jstart; j<jend; j++) {
      for (i=istart; i<iend; i++) {
        index = CCTK_GFINDEX3D(cctkGH,i,j,k);
        phitrhs[index] =
            (phix[CCTK_GFINDEX3D(cctkGH, i+1, j, k)] -
             phix[CCTK_GFINDEX3D(cctkGH, i-1, j, k)]) *hdxi
          + (phiy[CCTK_GFINDEX3D(cctkGH, i, j+1, k)] -
             phiy[CCTK_GFINDEX3D(cctkGH, i, j-1, k)]) *hdyi
          + (phiz[CCTK_GFINDEX3D(cctkGH, i, j, k+1)] -
             phiz[CCTK_GFINDEX3D(cctkGH, i, j, k-1)]) *hdzi;
        phixrhs[index] = (phit[CCTK_GFINDEX3D(cctkGH, i+1, j, k)] -
                          phit[CCTK_GFINDEX3D(cctkGH, i-1, j, k)]) *
                             hdxi;
        phiyrhs[index] = (phit[CCTK_GFINDEX3D(cctkGH, i, j+1, k)] -
                          phit[CCTK_GFINDEX3D(cctkGH, i, j-1, k)]) *
                             hdyi;
        phizrhs[index] = (phit[CCTK_GFINDEX3D(cctkGH, i, j, k+1)] -
                          phit[CCTK_GFINDEX3D(cctkGH, i, j, k-1)]) *
                             hdzi;
      }
    }
  }
}
```

This routine computes the right-hand sides from equations (2–5) using first order finite-differencing. These, as seen above, are then given to the `MoL` thorn for the time-evolution.

Analogous, albeit more complicated procedures, are used within the `McLachlan` and `GRHydro` thorns for evolving Einstein's field equations and the dynamics of matter within.

6. Final remarks

In these notes we hope to have provided a useful first guide for new users of the Einstein Toolkit, an open-source computational infrastructure for numerical relativity and relativistic astrophysics. Its capabilities include:

- evolutions of vacuum spacetimes (e.g., binary black hole coalescence) through the BSSN general relativity spacetime evolution equations with standard "moving puncture" Gamma-driver and 1+log gauge conditions;
- general-relativistic magneto-hydrodynamics evolutions, allowing, for example, simulations of magnetized isolated and binary neutron stars and collapsing stellar cores;

- various initial data solvers or data importers for configurations of general relativistic objects;

- tools for apparent and event horizon finding, black hole excision, gravitational wave extraction and spherical harmonics decomposition;

- multidimensional Input/Output using HDF5, ASCII, Images;

- Method of Lines time integration;

Tools and capabilities we have not explored include `Kranc` [26, 28]—a Mathematica application that converts continuum descriptions of PDEs into a `Cactus` thorn; and `Llama` [18]—a 3-dimensional multiblock infrastructure with adaptive mesh-refinement providing different patch systems that cover the simulation domain by a set of overlapping patches.

We emphasize that we have here barely scratched the surface of the current state of the toolkit and that large parts of it have intentionally been left out. The goal of this work is not to provide an overview of all capabilities of the ET, but to provide insight into the usage of the toolkit using a few simple examples. Advanced users are then referred to more literature covering parts of the toolkit, e.g. [7, 8], and are encouraged to seek direct contact to other active users of the toolkit, e.g., using the ET mailing list `users@einsteintoolkit.org`.

References

[1] Hahn S G and Lindquist R W 1964 *Annals of Physics* **29** 304–331

[2] Cardoso V, Gualtieri L, Herdeiro C, Sperhake U, Chesler P M *et al.* 2012 *Class.Quant.Grav.* **29** 244001

[3] Pretorius F 2005 *Phys. Rev. Lett.* **95** 121101

[4] Baker J G, Centrella J, Choi D I, Koppitz M and van Meter J 2006 *Phys. Rev. Lett.* **96** 111102

[5] Campanelli M, Lousto C O, Marronetti P and Zlochower Y 2006 *Phys. Rev. Lett.* **96** 111101

[6] Einstein Toolkit: Open software for relativistic astrophysics URL `http://einsteintoolkit.org/`

[7] Löffler F, Faber J, Bentivegna E, Bode T, Diener P, Haas R, Hinder I, Mundim B C, Ott C D, Schnetter E, Allen G, Campanelli M and Laguna P 2012 *Class. Quantum Grav.* **29** 115001

[8] Moesta P, Mundim B C, Faber J A, Haas R, Noble S C, Bode T, Löffler F, Ott C D, Reisswig C and Schnetter E 2013 (*Preprint* `1304.5544`)

[9] Baumgarte T W and Shapiro S L 1999 *Phys. Rev. D* **59** 024007

[10] Shibata M and Nakamura T 1995 *Phys. Rev. D* **52** 5428–5444

[11] Cactus Computational Toolkit URL `http://www.cactuscode.org/`

[12] Goodale T, Allen G, Lanfermann G, Massó J, Radke T, Seidel E and Shalf J 2003 The Cactus framework and toolkit: Design and applications *Vector and Parallel Processing – VECPAR'2002, 5th International Conference, Lecture Notes in Computer Science* (Berlin: Springer) URL `http://edoc.mpg.de/3341`

[13] Allen G, Goodale T, Lanfermann G, Radke T, Rideout D and Thornburg J 2011 *Cactus Users' Guide* URL `http://www.cactuscode.org/Guides/Stable/UsersGuide/UsersGuideStable.pdf`

[14] Carpet: Adaptive Mesh Refinement for the Cactus Framework URL `http://www.carpetcode.org/`

[15] Schnetter E, Hawley S H and Hawke I 2004 *Class. Quantum Grav.* **21** 1465–1488

[16] Schnetter E, Diener P, Dorband E N and Tiglio M 2006 *Class. Quantum Grav.* **23** S553–S578

[17] Berger M J and Oliger J 1984 *J. Comput. Phys.* **53** 484

[18] Pollney D, Reisswig C, Schnetter E, Dorband N and Diener P 2011 *Phys. Rev. D* **83** 044045

[19] Arnowitt R L, Deser S and Misner C W 2008 *General Relativity and Gravitation* **40** 1997–2027

[20] York Jr J W 1979 Kinematics and dynamics of general relativity *Sources of Gravitational Radiation* ed L L Smarr (Cambridge and New York: Cambridge University Press) pp 83–126

[21] Ansorg M, Brügmann B and Tichy W 2004 *Phys. Rev. D* **70** 064011

[22] Brown J D, Diener P, Sarbach O, Schnetter E and Tiglio M 2009 *Phys. Rev. D* **79** 044023

[23] Reisswig C, Ott C D, Sperhake U and Schnetter E 2011 *Phys. Rev. D* **83** 064008

[24] McLachlan, a public BSSN code URL `http://www.cct.lsu.edu/~eschnett/McLachlan/`

[25] Alcubierre M, Brügmann B, Dramlitsch T, Font J A, Papadopoulos P, Seidel E, Stergioulas N and Takahashi R 2000 *Phys. Rev. D* **62** 044034

[26] Husa S, Hinder I and Lechner C 2006 *Comput. Phys. Commun.* **174** 983–1004

[27] Lechner C, Alic D and Husa S 2004 *Analele Universitatii de Vest din Timisoara, Seria Matematica-Informatica* **42**

[28] Kranc: Kranc assembles numerical code URL `http://kranccode.org/`

[29] Baiotti L, Hawke I, Montero P J, Löffler F, Rezzolla L, Stergioulas N, Font J A and Seidel E 2005 *Phys. Rev. D* **71** 024035

[30] Hawke I, Löffler F and Nerozzi A 2005 *Phys. Rev. D* **71** 104006

[31] Alcubierre M, Brügmann B, Diener P, Koppitz M, Pollney D, Seidel E and Takahashi R 2003 *Phys. Rev. D* **67** 084023

[32] Thornburg J 2004 *Class. Quantum Grav.* **21** 743–766

[33] Thornburg J 1996 *Phys. Rev. D* **54** 4899–4918

[34] Newman E and Penrose R 1962 *J.Math.Phys.* **3** 566–578

[35] Diener P 2003 *Class. Quantum Grav.* **20** 4901–4918

[36] Seidel E L, Allen G, Brandt S R, Löffler F and Schnetter E 2010 Simplifying Complex Software Assembly: The Component Retrieval Language and Implementation *Proceedings of the 2010 TeraGrid Conference* (*Preprint* `arXiv:1009.1342[cs.PL]`)

[37] Allen G, Goodale T, Löffler F, Rideout D, Schnetter E and Seidel E L 2010 Component Specification in the Cactus Framework: The Cactus Configuration Language *Grid2010: Proceedings of the 11th IEEE/ACM International Conference on Grid Computing* (*Preprint* `arXiv:1009.1341[cs.DC]`)

[38] Thomas M and Schnetter E 2010 Simulation factory: Taming application configuration and workflow on high-end resources *Grid Computing (GRID), 2010 11th IEEE/ACM International Conference on* pp 369 –378 (*Preprint* `arXiv:1008.4571[cs.DC]`)

[39] SimFactory: Herding numerical simulations URL `http://simfactory.org/`

[40] Tolman R C 1939 *Phys. Rev.* **55** 364–373

[41] Oppenheimer J R and Volkoff G M 1939 *Phys. Rev.* **55** 374–381

[42] Hunter J D 2007 *Comput. Sci. Eng.* **9** 90–95