PHYS 517 Project Documentation: Spherical Symmetric Numerical Relativity

Siyang Ling

April 27, 2020

1 Overview of the `Code` Directory

Here we briefly describe all the items in the `Code` directory.

1.1 `C++` Files

The program for numerical simulation was written in `C++`. Here is a list of `C++` related files in the `Code` directory:

`main.cpp` The entry point of the program. Edit this file to change the discretization parameters, the time step parameters, or other parameter.

`linalg.h` Header for a simple linear algebra package.

`linalg.cpp` A simple linear algebra package. Provides only two functions to solve tridiagonal linear systems and a function to create a identity matrix.

`nr_sss.h` Header for functions used in solving the Einstein equation in spherical symmetric spacetime. Defines 2 types to store the non-linear terms and the gauge functions.

`nr_sss.cpp` Defines 2 functions to evaluate the Laplacian, 1 function to compute non-linear terms in the Einstein's equations, 1 function to compute the gauge functions, 1 function to update the particles, 1 function to update the light rays, and 1 function to evolve the metric.

`numvec.h` Header for simple numerical vector.

`numvec.cpp` Implements a simple numerical vector using `std::vector<double>`. Provides operators to add, subtract and multiply vectors. Also provide a `max_norm` function to evaluate the l^∞ norm of the vector.

`particle.h` Header for the particle/ray structure.

`particle.cpp` Defines functions to initialize particle/ray profiles.

`tridiag.h` Header for simple tridiagonal matrix.

`tridiag.cpp` Implements a simple tridiagonal matrix using `std::vector<std::array<double, 3>>`. Provides operators to add, subtract and apply matrices on `NumVec`s.

`utility.h` Header for utility function used in development.

`utility.cpp` Provides utility functions used for debug purpose.

1.2 `Python` Files

`Python` scripts was used to make plots from numerical results the `C++` program produced.

`Graph.py` Plots metric function A at different time slices.

`GraphAlpha.py` Plots time lapse α at different time slices.

`GraphParticle.py` Plots the world lines of tracer particles.

`GraphRay.py` Plots the world lines of tracer light rays. Also determines the event horizon.

`GraphTracer.py` Plots the fictitious Lagrangian matter tracers.

1.3 Miscellaneous Files & Directories

`Makefile` Makefile for the `C++` program.

`clear-solution` Deletes the outputs from directories `solution`, `alpha`, `particles`, `rays`, `mat_tracers`.

Directories `solution`, `alpha`, `particles`, `rays`, `mat_tracers` stores output (from `C++` program) for f, α, particle world line, light ray world line, and Lagrangian matter tracers, respectively.

2 Usage of `nr_sss.cpp` and `particle.cpp`

`nr_sss.cpp` and `particle.cpp` provide the main functionalities for solving the Einstein's equation with a particle simulation scheme, so they warrant some further explanation.

2.1 `particle.cpp`

Two types are defined in `particle.h`. The first type is `ParticleProfile`, which describes a collection of particles. It consists of 5 variables:

`M` An `int` storing the total number of particles.

`r` A vector storing the radial coordinates of particles.

`u_r` A vector storing the radial velocities of particles.

`u_phi` A vector storing the angular velocities of particles.

`m` A vector storing the rest masses of particles.

The function `uniform_static_ensemble(double M, double R, double m)` initializes a `ParticleProfile` with a `M` particles with mass `m`; all the velocities are set to 0, and the radial coordinates `r` are set to $(\frac{i+1}{M})^{1/3}R$(i is the particle index number), so the distribution of the particles is roughly uniform within a ball of radius `R`. One can create new functions to create `ParticleProfile` from other ensembles.

The other type is `RayProfile`, which represents outward-going light rays. It consists of only `M`, the number of light rays, and `r`, the radial coordinates of the light rays. Function `ray_ensemble(int M, double a, double b)` initializes a total of `M` light rays, with equal spacing within $[a, b]$.

2.2 `nr_sss.cpp`

All the functions in `nr_sss.cpp` has `N` and `L` as function arguments. `N` and `L` are discretization parameters: the radial grid $[0, L]$ is divided into $N-1$ equal intervals. In `nr_sss.h`, we also define a macro `R(i,h)`, which gives the discretized radial position $r_i = h(i-1/2)$ $(i = 1, ...N-1)$ with $h = L/(N-1)$. All `NumVec` and `TriDiag` (vectors and operators) representing the discretized spatial grid have dimension $N-1$.

The full Laplacian operator is implemented in `laplacian_tridiag(int N, double L)` and `add_laplacian_boundary(int N, double L, NumVec &f)`. `laplacian_tridiag` returns a `TriDiag` that represents the matrix for the discretized Laplacian $\frac{1}{r^2}\frac{\partial}{\partial r}\left(r^2\frac{\partial}{\partial r}\right)$. `add_laplacian_boundary` adds the inhomogenous Robin boundary term to `f` in-place; it doesn't return a new object. To evaluate the Laplacian for an `f` with inhomogeneous boundary conditions, apply the matrix returned by `laplacian_tridiag` to it and modify it using `add_laplacian_boundary`.

`compute_nonlinear_term` computes relevant non-linear terms in the Hamiltonian constraint equation from f and the particle profile, and returns them in a `NonLinearTerm` type packing both `F` and `DF`. Here, `F` is the non-linear term (a vector), and `DF` is the functional derivative of (a matrix).

`compute_gauge` computes the gauge functions from f and the particle profile, and returns them in a `Gauge` type packing both `alpha` and `beta`.

`update_particles` (or `update_rays`) takes in the metric function f and the gauge functions to update a `ParticleProfile` (or a `RayProfile`) in-place by time step `dt` using the geodesic equation.

`evolve_metric` takes in the gauge functions to update the metric function `f` by time step `dt` using the metric evolution equation.

3 General Usage Instructions

Most of the modification to the code, such as changing the discretization scheme, changing the time step, period for plotting, tracking of matter particles and light rays, can be done in `main.cpp`.

To run the program, run `make` to compile the `C++` project, then run `main` to start the simulation. During the simulation, the program will write the solutions to the various directories. Depending on the settings, the solution files could take from megabytes to hundred megabytes of space.

After the `C++` program terminates, you can run the standalone script `Grapy.py` to graph the solution for the metric, and `GraphAlpha.py` to graph the time lapse. `GraphRay.py` can be used to find the event horizon, stored in `t_plot` and `horizon_plot`; `GraphTracer.py` can be used to plot the Lagrangian matter tracers along with the event horizon. Note that you may need to change `M` manually in several files. (`M` is computed automatically in `Graph.py`)