

GPUs and Einstein's Equations (Midyear Report)

Timothy Dewey
tdewey@math.umd.edu
Advisor: Dr. Manuel Tiglio
tiglio@umd.edu
Physics Department, CSCAMM*
University of Maryland

December 16, 2010

Abstract

This document outlines the project progress for the AMSC 663/664 course series. The project is to write code to solve Einstein's equations using graphics processing units. We provide a brief introduction to the project and then provide an outline of the proposed implementation and validation. Furthermore, a summary of the progress on the project is provided.

1 Problem Introduction

The aim of this project is to write code to solve Einstein's equations using spectral methods. This implementation should be optimized in order to give meaningful performance data that can be used to make a determination about the utility of graphics processing units (GPUs) for solving large scale problems in numerical relativity. We believe that we can make significant time and power improvements by solving Einstein's equations on GPUs instead of CPUs. Einstein's equations are a good candidate for GPUs because they are computationally intensive and highly parallelizable. Such applications are where GPUs often show significant performance advantages over CPUs (see [1]). The first step, however, and the focus of the Fall 2010 semester, is to prototype the code in MATLAB.

2 The Problem

Einstein's equations are a system of hyperbolic PDE's which are first order in space and time. In order to solve these equations, we will use a pseudo-spectral collocation method (we will just

*Center for Scientific Computation and Mathematical Modeling

say spectral method from now on). The code will be implemented first in MATLAB, then in C, and finally in Compute Unified Device Architecture (CUDA) for GPUs (see [2]).

The first step (completed during the Fall 2010 semester), is to write MATLAB code to solve (numerically) the Schwarzschild equations for a spherically symmetric black hole. These equations are coupled hyperbolic equations that are first order in space and time. Since we have a spherically symmetric black hole, the coordinates are one dimensional in space (r is the radius) instead of three dimensional in space. There are six main variables $g_{rr}, g_T, K_{rr}, K_T, f_{rrr}, f_{rT}$, that describe a spherically symmetric metric on a Lorentzian manifold. The equations being solved are listed below (see [3] for more detail). Also, the MATLAB function that performs the right hand side computations is found in Appendix A.

$$\dot{g}_{rr} = \beta g'_{rr} + 2g_{rr}\beta' - 2\tilde{\alpha}g_{rr}^{1/2}g_T K_{rr},$$

$$\dot{g}_T = \beta g'_T + 2/r\beta g_T - 2\tilde{\alpha}g_{rr}^{1/2}g_T K_T,$$

$$\begin{aligned} \dot{K}_{rr} = & \beta K'_{rr} - 1/g_{rr}^{1/2}g_T\tilde{\alpha}f'_{rrr} + 2K_{rr}\beta' - 8/g_{rr}^{1/2}\tilde{\alpha}f_{rrr}f_{rT} - 6g_{rr}^{1/2}/g_T\tilde{\alpha}f_{rT}^2 - \\ & 1/g_{rr}^{1/2}g_T f_{rrr}\tilde{\alpha}' + 2/g_{rr}^{1/2}g_T/r f_{rrr}\tilde{\alpha} - g_{rr}^{1/2}g_T\tilde{\alpha}'' + 4g_{rr}^{1/2}g_T/r\tilde{\alpha}' + \\ & 2g_{rr}^{1/2}\tilde{\alpha}K_{rr}K_T - 1/g_{rr}^{1/2}g_T\tilde{\alpha}K_{rr}^2 - 6g_{rr}^{1/2}g_T\tilde{\alpha}/r^2 + 2/g_{rr}^{3/2}g_T\tilde{\alpha}f_{rrr}^2, \end{aligned}$$

$$\begin{aligned} \dot{K}_T = & \beta K'_T - 1/g_{rr}^{1/2}\tilde{\alpha}g_T f'_{rT} + 2/r\beta K_T - 1/g_{rr}^{1/2}g_T\tilde{\alpha}'f_{rT} - 2/g_{rr}^{1/2}\tilde{\alpha}f_{rT}^2 + \\ & g_{rr}^{1/2}g_T\tilde{\alpha}/r^2 + 1/g_{rr}^{1/2}\tilde{\alpha}K_T K_{rr}g_T, \end{aligned}$$

$$\begin{aligned} \dot{f}_{rrr} = & -\tilde{\alpha}g_{rr}^{1/2}g_T K'_{rr} + \beta f'_{rrr} - \tilde{\alpha}/g_{rr}^{1/2}K_{rr}f_{rrr}g_T - 10\tilde{\alpha}g_{rr}^{1/2}K_{rr}f_{rT} + \\ & 12g_{rr}^{3/2}/g_T f_{rT}\tilde{\alpha}K_T + 3\beta' f_{rrr} - 4g_{rr}^{3/2}K_T\tilde{\alpha}' + 8g_{rr}^{3/2}/r K_T\tilde{\alpha} + \\ & g_{rr}\beta'' - 4g_{rr}^{1/2}\tilde{\alpha}K_T f_{rrr} - g_{rr}^{1/2}g_T K_{rr}\tilde{\alpha}' + 2g_{rr}^{1/2}g_T/r K_{rr}\tilde{\alpha}, \end{aligned}$$

$$\begin{aligned} \dot{f}_{rT} = & -\tilde{\alpha}g_{rr}^{1/2}g_T K'_T + \beta f'_{rT} + 2/r\beta f_{rT} + \beta' f_{rT} - g_{rr}^{1/2}g_T K_T\tilde{\alpha}' - \\ & \tilde{\alpha}/g_{rr}^{1/2}K_T f_{rrr}g_T + 2\tilde{\alpha}g_{rr}^{1/2}K_T f_{rT}. \end{aligned}$$

Note that $\tilde{\alpha} = 1$, $\tilde{\alpha}' = 0$, $\beta = \sqrt{2/r}$, $\beta' = -1/\sqrt{2r^3}$, and $\beta'' = 3/\sqrt{8r^5}$. The initial values of the main variables are computed using the analytic solution described in a later section. Also, the spatial derivatives of the main variables, $g'_{rr}, g'_T, K'_{rr}, K'_T, f'_{rrr}$, and f'_{rT} are approximated by performing a matrix vector multiplication that will be described in the following section.

Now that the MATLAB code is complete, we have a set of functions that can be used easily to check more complex C/CUDA code. For next semester, the plan is to write code in C and CUDA that will solve the equations above numerically. If time permits, the code can be extended to handle the 2-d and 3-d cases. The same equations can be solved in the 2-d and 3-d case to verify the computations.

2.1 Spectral Methods

Spectral methods enable us to solve Einstein’s equations with a high degree of accuracy, but with a small amount of memory. The second fact is important because GPUs have very limited (fast) memory. Now we will discuss the details of spectral methods that are relevant to this project.

First, we want to be able to approximate the derivative of a smooth function (i.e. a function with several derivatives that exist). Using spectral methods, the error in this approximation can converge to 0 faster than any power law convergence. In order to realize this “spectral convergence”, we use the roots of orthogonal polynomials (e.g., Legendre or Chebyshev) as collocation points. Specifically, we use Chebyshev polynomials with the Gauss-Lobatto points (see [4]).

The Chebyshev collocation points are $x_i = \cos(\pi \cdot i/N)$, for $i \in \{0, \dots, N\}$. These points determine a differentiation matrix \mathbf{D} . The matrix \mathbf{D} is $(N+1) \times (N+1)$, and it is analogous to the idea of implementing a finite difference derivative approximation using banded sparse matrices. With spectral methods the differentiation matrix is dense. The definition of the entries of the matrix \mathbf{D} as found in [4] are as follows:

$$\mathbf{D}_{00} = \frac{2N^2 + 1}{6}, \quad (1)$$

$$\mathbf{D}_{NN} = -\frac{2N^2 + 1}{6}, \quad (2)$$

$$\mathbf{D}_{jj} = \frac{-x_j}{2(1 - x_j^2)}, \quad j = \{1, \dots, N-1\}, \quad (3)$$

$$\mathbf{D}_{ij} = \frac{c_i (-1)^{i+j}}{c_j (x_i - x_j)}, \quad i \neq j, \quad i, j = \{0, \dots, N\}, \quad (4)$$

where $c_i = 2$ for $i = 0$ or N and $c_i = 1$ otherwise. MATLAB code that implements the above equations is found in Appendix A.

Define a column vector of the collocation points $\mathbf{x} = \{x_0, \dots, x_N\}^T$. Now we can approximate the derivative of a function $u(x)$ at the collocation points by simply performing the matrix vector multiplication $\mathbf{D}u(\mathbf{x}) \approx u'(\mathbf{x})$. This computation is fast and requires little memory for a small number of collocation points $N+1$. Typically, we have $N \leq 40$; this fact comes from using a multiple-domain approach for large scale numerical relativity problems. As the problem is broken into multiple domains, the minimum distance between collocation points gets smaller and it scales as $O(N^2)$. This means for large N this minimum distance becomes small and, in turn, the CFL limit becomes small. In order to keep a reasonable size for the time step, we want to keep N small (say $N \leq 40$). Furthermore, because we have spectral convergence, the error in our approximations will typically approach machine precision with $N < 40$.

Technically, \mathbf{x} is a vector of values from the interval $[-1, 1]$; recall the Chebyshev collocation points are $x_i = \cos(\pi \cdot i/N)$, for $i \in \{0, \dots, N\}$. However, we want our collocation points to represent values from some other interval of the real line. The solution for this is to scale and shift the interval $[-1, 1]$ to fit the interval of the spatial domain of interest. For example, we might choose the interval $[1.9, 11.9]$ as our spatial domain. The units on the radius are in terms of the mass of the black hole. We let the mass M of the black hole be 1, which places the event

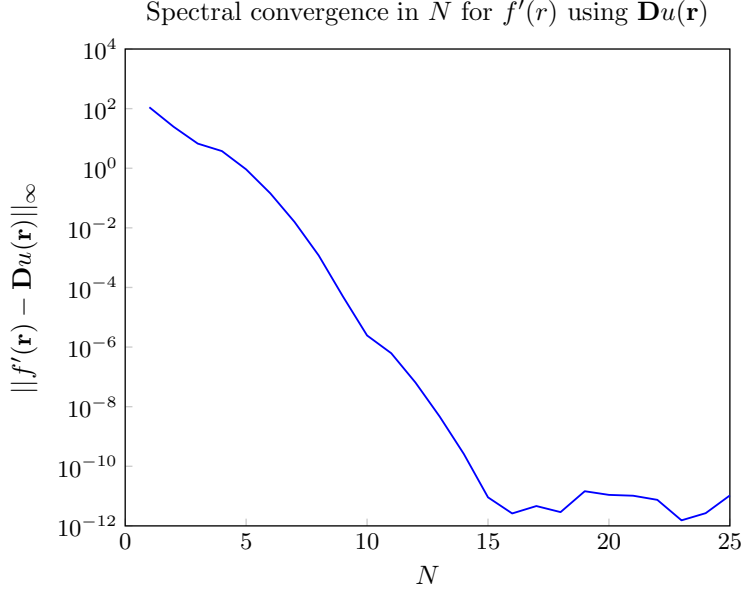


Figure 1: Error in the approximation of the derivative of $f(r) = 5e^r \cdot \sin(\frac{r}{2})$.

horizon of the black hole at $r = 2$. The choice of 1.9 as in the inner boundary is not arbitrary. The value 1.9 lies within the event horizon of the black hole which lies at $r = 2M$. The choice of 11.9 is essentially arbitrary; it simply lies outside the black hole. Because the inner boundary is inside the black hole, only the outer boundary requires enforcement of boundary conditions. This will be discussed later.

Now, we shift and scale the interval $[-1, 1]$ and get the new collocation points $r_i = x_i \cdot ((r_{max} - r_{min})/2) + ((r_{max} + r_{min})/2)$, where $r_{min} = 1.9$ and $r_{max} = 11.9$. Let $\mathbf{r} = \{r_0, \dots, r_N\}^T$. To compute the approximation of a derivative at the points \mathbf{r} , we use:

$$u'(\mathbf{r}) \approx \mathbf{D}u(\mathbf{r}) \cdot \frac{dx}{dr} = \mathbf{D}u(\mathbf{r}) \cdot 2/(r_{max} - r_{min}).$$

In Figure 1, we show spectral convergence for the approximation of $f'(r)$ where $f(r) = 5e^r \cdot \sin(\frac{r}{2})$, and $r \in [1, 4]$.

2.2 Runge-Kutta and Boundary Conditions

Next we want to be able to evolve Einstein's equations in time. To do this, we use a 4th order Runge-Kutta method as described in [5]. The Runge-Kutta method, as outlined in Algorithm

1, is the method we use for time stepping. The MATLAB code we use for the Runge-Kutta algorithm is found in Appendix A.

Algorithm 1 Runge-Kutta

Let $du/dt = f(t, u)$.
1: Set $t_0 = 0$, $u_0 = u(t_0)$, $h = \Delta t$.
2: **for** $n = 0$ to num_steps
3: $t_{n+1} = t_n + h$
4: $k_1 = f(t_n, u_n)$
5: $k_2 = f(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hk_1)$
6: $k_3 = f(t_n + \frac{1}{2}h, u_n + \frac{1}{2}hk_2)$
7: $k_4 = f(t_n + h, u_n + hk_3)$
8: $u_{n+1} = u_n + \frac{1}{6}h(k_1 + 2k_2 + 2k_3 + k_4)$

Each time we update our approximate solution u , we must enforce the boundary conditions. As we mentioned before, the enforcement of boundary conditions is only required at the outer boundary ($r = 11.9$). We enforce the boundary conditions on the incoming characteristic variables (i.e., characteristic variables with positive speed). The characteristic variables v are as follows:

$$v_1 = g_{rr}, \quad (5)$$

$$v_2 = g_T, \quad (6)$$

$$v_3 = K_{rr} - g_{rr}^{-1/2} f_{rrr}, \quad (7)$$

$$v_4 = K_T - g_{rr}^{-1/2} f_{rT}, \quad (8)$$

$$v_5 = K_{rr} + g_{rr}^{-1/2} f_{rrr}, \quad (9)$$

$$v_6 = K_T + g_{rr}^{-1/2} f_{rT}. \quad (10)$$

We set the time derivatives \dot{v}_i to zero for $i = \{1, \dots, 4\}$ (these are the characteristic variables with positive speed), then we want to solve for time derivative of the main variables and assign this adjusted value as the new value of the time derivative at the outer boundary.

$$\dot{g}_{rr} \leftarrow \dot{v}_1 = 0, \quad (11)$$

$$\dot{g}_T \leftarrow \dot{v}_2 = 0, \quad (12)$$

$$\dot{K}_{rr} \leftarrow (\dot{v}_3 + \dot{v}_5)/2 = (\dot{K}_{rr} - \dot{g}_{rr}^{-1/2} \dot{f}_{rrr} + \dot{K}_{rr} + \dot{g}_{rr}^{-1/2} \dot{f}_{rrr})/2, \quad (13)$$

$$\dot{f}_{rrr} \leftarrow \frac{\sqrt{g_{rr}}}{2}(\dot{v}_5 - \dot{v}_3) = \frac{\sqrt{g_{rr}}}{2}(\dot{K}_{rr} + \dot{g}_{rr}^{-1/2} \dot{f}_{rrr} - (\dot{K}_{rr} - \dot{g}_{rr}^{-1/2} \dot{f}_{rrr})), \quad (14)$$

$$\dot{K}_T \leftarrow (\dot{v}_4 + \dot{v}_6)/2 = (\dot{K}_T - \dot{g}_{rr}^{-1/2} \dot{f}_{rT} + \dot{K}_T + \dot{g}_{rr}^{-1/2} \dot{f}_{rT})/2, \quad (15)$$

$$\dot{f}_{rT} \leftarrow \frac{\sqrt{g_{rr}}}{2}(\dot{v}_6 - \dot{v}_4) = \frac{\sqrt{g_{rr}}}{2}(\dot{K}_T + \dot{g}_{rr}^{-1/2} \dot{f}_{rT} - (\dot{K}_T - \dot{g}_{rr}^{-1/2} \dot{f}_{rT})). \quad (16)$$

Notice that in equations (14) and (16), we have $\sqrt{g_{rr}}$ multiplied by $\dot{g}_{rr}^{-1/2}$. The value $\sqrt{g_{rr}}$ should use the analytic value of g_{rr} as provided in the initial conditions. The outer

boundary conditions are enforced after each approximation of the time derivative (i.e., lines 4-7 in Algorithm 1).

3 Validation

The final step is to validate the approximate solution that is computed numerically by the PDE solver. In order to validate the implementation, we rely on the Schwarzschild solution. This is an analytic solution to Einstein's equations for a spherically symmetric black hole (see [6], Section II.D.2). The equations we are using are in the Painleve-Gullstrand coordinate system. In this system, the Schwarzschild solution given in [6] is as follows:

$$g_{rr} = 1, \tag{17}$$

$$g_T = 1, \tag{18}$$

$$\tilde{\alpha} = 1, \tag{19}$$

$$\beta^r = \sqrt{\frac{2M}{r}}, \tag{20}$$

$$K_{rr} = -\sqrt{\frac{M}{2r^3}}, \tag{21}$$

$$K_T = -\sqrt{\frac{2M}{r^3}}, \tag{22}$$

$$f_{rrr} = \frac{4}{r}, \tag{23}$$

$$f_{rT} = \frac{1}{r}. \tag{24}$$

Notice that the event horizon is at $r = 2M$ as we mentioned before. Again, for our solutions we assume $M = 1$, so the horizon is located at $r = 2$.

This analytic solution is the known solution to which we compare our computed solution at each stage of the computation. Furthermore, there will be certain functions in the implementation that can be validated independently. For infinitely differentiable functions (i.e., functions in C^∞), we can show that the vector $\mathbf{D}u(\mathbf{x})$ approaches $u'(\mathbf{x})$ with exponential convergence in N (up to machine precision). When plotting $\|\mathbf{D}u(\mathbf{x}) - u'(\mathbf{x})\|_\infty$ versus N , we should get a roughly log-linear plot. This is what we see in Figure 1. This is an important intermediate verification step.

In order to test the validity of our final code, we use the analytic solution to Einstein's equations found in [3], and compare the approximate solution after each step of the Runge-Kutta. In this special case of the spherically symmetric black hole, the solution should be in a steady state. That is, the time derivative of the solution should be 0 in each component of the solution. This gives another check on our code. When evaluating the right hand side (RHS) of Einstein's equations, we should have approximately zero (up to machine precision, or the precision provided by the degree N Chebyshev approximation, whichever has less precision). The RHS for each equation is the time derivative of one of the components of the solution.

The arguments to the PDE solver include boundary points in space (e.g., $[1.9, 11.9]$), the initial and final times (e.g., $[0, 10]$), the size of the time step Δt (e.g., $\Delta t = 0.001$), and the

degree of the Chebyshev polynomials N (e.g, $N = 30$). The parameters Δt and N determine the error of the approximation. As we increase N and decrease Δt the error approaches 0 (up to machine precision). Furthermore, as we increase N , we see spectral convergence; the error rapidly decreases to machine precision as N increases. This spectral convergence is how we validate our code against the known solution.

In Figure 2, we see the error in the solution computed using various values for N with $\Delta t = 0.001$. For the analytic solution u and approximate solution \hat{u} , Figure 2 has the value $\|u(\mathbf{r}) - \hat{u}(\mathbf{r})\|_2$ on the vertical axis. Recall, $\mathbf{r} = \{r_0, \dots, r_N\}^T$ for $r_i = x_i \cdot ((r_{max} - r_{min})/2) + ((r_{max} + r_{min})/2)$, where $x_i = \cos(\pi \cdot i/N)$, for $i \in \{0, \dots, N\}$, $r_{min} = 1.9$, and $r_{max} = 11.9$.

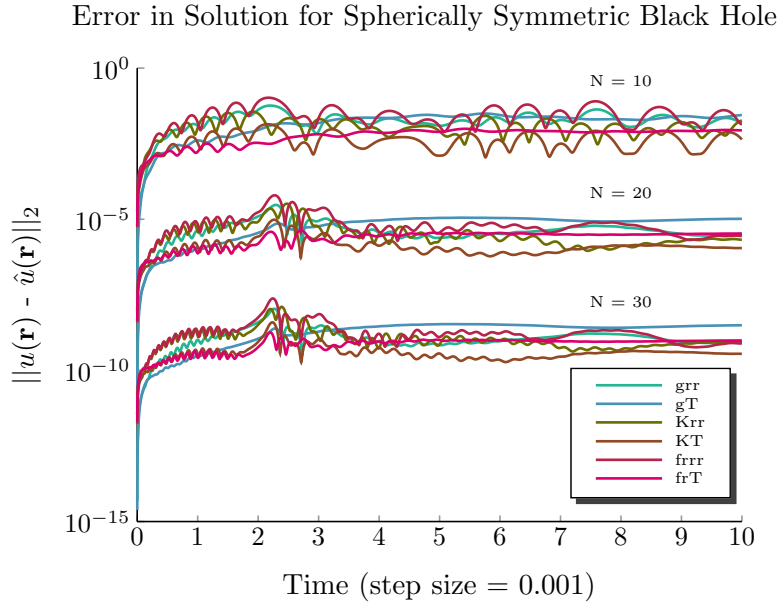


Figure 2: Error in the components of the numerical solution \hat{u} for multiple values of N .

Notice that for the three values of N shown in Figure 2, the errors differ by a few orders of magnitude. We stopped at $N = 30$ in the figure, but for $N = 40$ the error improves as one would expect from the figure. This figure demonstrates the spectral convergence, and in turn, validates our code. Furthermore, another important observation to be made from the figure is that the error stays bounded as a function of time. In fact, this figure is only the first 10 time units in a plot that continues up to 200 time units. In the plot of the full 200 time units, the error reaches a near steady state. In Figure 2, from time 4 to time 10, the error has already begun to settle down. There is not a proof that the error will not diverge, but after 200,000 steps of the algorithm and seeing little change in the error for the last 190,000 steps does instill confidence that the algorithm is stable given sufficiently large N and sufficiently small Δt (i.e., Δt less than the CFL limit).

4 Project Schedule/Milestones

The major milestones of the project are the MATLAB, C, and CUDA versions of the spectral method solver for Einstein's equations. The MATLAB implementation is complete and verified as demonstrated in Figure 2. Also, a basic plan for the C code is in Appendix B. The next steps are to implement and verify the PDE solver in C, write a plan for the CUDA implementation, implement and verify the PDE solver in CUDA, optimize the code, and write a report of the results.

Below is the projected timeline from the project proposal.

Timeline

Fall 2010 (AMSC 663)

- ✓ December 1, 1-d MATLAB code verified on test problem.
- ✓ December 1, Written plan for C code.

Spring 2011 (AMSC 664)

- February 10, 1-d C code verified on test data.
- February 15, Written plan for CUDA.
- March 15, 1-d CUDA code verified on test data.
- April 15, Optimized 1-d CUDA code.
- May 1, Complete writeup of results.
- May, Time permitting, 2-d and/or 3-d version.

References

- [1] NVIDIA. <http://www.nvidia.com/object/why-choose-tesla.html>. 2010.
- [2] NVIDIA CUDA C Programming Guide. Version 3.2. November 9, 2010.
http://developer.download.nvidia.com/compute/cuda/3.2_prod/toolkit/docs/CUDA_C_Programming_Guide.pdf
- [3] Gioel Calabrese, Luis Lehner, and Manuel Tiglio. "Constraint-preserving boundary conditions in numerical relativity." *Phys. Rev. D* 65, 104031. May 10, 2002.
- [4] L. Trefethen. "Spectral Methods in MATLAB." SIAM, 2000.
- [5] "Runge-Kutta methods." http://en.wikipedia.org/wiki/Runge-Kutta_methods.
- [6] Lawrence E. Kidder, Mark A. Scheel, Saul A. Teukolsky, Eric D. Carlson, and Gregory B. Cook. "Black hole evolution by spectral methods." *Phys. Rev. D* 62, 084032. September 26, 2000.

Appendix A

Differentiation Matrix Code

```
function [D,x] = cheby(N)
% cheby  Chebyshev Differentiation Matrix and Grid
%
% Syntax:
%   [D,x] = cheby(N)
%
% Summary:
%   This function computes the Chebyshev Differentiation Matrix of degree N.
%   The output matrix D is a N+1 x N+1 square differentiation matrix, and
%   the Chebyshev grid x has is a vector of length N+1. This function gives
%   the same output (up to rounding error) as cheb.m from [1].
%
% Input:
%   N = Degree of the polynomials used to differentiate.
%
% Output:
%   D = The differentiation matrix.
%
%   x = The Chebyshev grid points.
%
% References:
%   [1] Lloyd N. Trefethen. "Spectral Methods in Matlab." SIAM, 2000.
%   http://www.comlab.ox.ac.uk/oucl/work/nick.trefethen.
%
% Author:
%   Timothy D. Dewey (September 20, 2010)

% Define the grid points x.
x = cos(pi*(0:N)/N);

D = zeros(N+1);
c = [2; ones(N-1,1); 2];
for i = 1:N+1
    for j = [1:i-1,i+1:N+1]
        D(i,j) = c(i) * (-1)^(i-1+j-1) / (c(j) * (x(i) - x(j)));
    end
    % Use the opposite of the sum of off diagonals for better stability as
    % suggested in [1].
    D(i,i) = -sum(D(i,:));
end
```

RHS Code

```

function u_dot = rhs_rk4sp4(u, r, D)
% rhs_rk4sp4 Right hand side of Einstein's equations (Vectorized).
%
% Syntax:
%   u_dot = rhs_rk4sp4(u, r, D)
%
% Summary:
%   Compute the right-hand side of Einstein's equations using for a
%   spherically symmetric black hole using Painleve-Gullstrand coordinates.
%
% Input:
%   u = [grr, gT, Krr, KT, frrr, frT] is an (N+1) x 6 matrix where (N+1) is
%   the number of collocation points used.
%
%   r = the collocation points in the scaled interval.
%
%   D = the differentiation matrix.
%
% Output:
%   u_dot = [grr_dot, gT_dot, Krr_dot, KT_dot, frrr_dot, frT_dot] is an
%   N x 6 matrix where N is the number of collocation points used.
%
% Example:
%
%   rmin = 1.9;
%   rmax = 11.9;
%   t0 = 0;
%   t1 = 1.5;
%   num_steps = 1000;
%
%   norm_err = zeros(6, num_steps + 1);
%   N = 20;
%   [D, x] = cheby(N);
%   r = shiftInterval(x, rmin, rmax);
%   [grr, gT, Krr, KT, frrr, frT] = schwarzschild(r);
%   u0 = [grr, gT, Krr, KT, frrr, frT];
%   [y,t] = rk4sp4(@rhs_rk4sp4, [t0 t1], u0, r, D, num_steps);
%   err = bsxfun(@minus, y, u0);
%   for i = 1:6
%       for j = 1:num_steps + 1
%           norm_err(i,j) = norm(err(:,i,j),2);
%       end
%   end

```

```

%     end
%
%     figure, semilogy(t, norm_err)
%
%
% References:
% [1] Lawrence E. Kidder, Mark A. Scheel, Saul A. Teukolsky, Eric D.
%     Carlson, and Gregory B. Cook. "Black hole evolution by spectral
%     methods." Phys. Rev. D 62, 084032. September 26, 2000.
%
% [2] Gioel Calabrese, Luis Lehner, and Manuel Tiglio. "Constraint-
%     preserving boundary conditions in numerical relativity." Phys. Rev. D
%     65, 104031. May 10, 2002.
%
% Author:
% Tim Dewey, (November, 2010)

% Break out u components.
% u = ([grr, gT, Krr, KT, frrr, frT]);
grr = u(:,1);
gT = u(:,2);
Krr = u(:,3);
KT = u(:,4);
frrr = u(:,5);
frT = u(:,6);

alphaT = ones(size(u(:,1)));
dalphaT = zeros(size(u(:,1)));
d2alphaT = zeros(size(u(:,1)));

beta = ones(size(u(:,1))) .* sqrt(2./r);
dbeta = ones(size(u(:,1))) .* -1./sqrt(2.*r.^3);
d2beta = ones(size(u(:,1))) .* 3./sqrt(8.*r.^5);

% Derivative in space. d./dx .* dx./dr f -> d./dr f
dr = 2/(max(r) - min(r));

dgrr = D * grr * dr;
dgT = D * gT * dr;
dKrr = D * Krr * dr;
dKT = D * KT * dr;
dfrrr = D * frrr * dr;
dfrT = D * frT * dr;

```

```

grr_dot = beta .* dgrr + 2 .* grr .* dbeta - ...
          2 .* alphaT .* grr.^(0.5) .* gT .* Krr;

gT_dot = beta .* dgT + 2./r .* beta .* gT - ...
          2 .* alphaT .* grr.^(0.5) .* gT .* KT;

Krr_dot = beta .* dKrr - 1./grr.^(0.5) .* gT .* alphaT .* dfrrr + ...
           2 .* Krr .* dbeta - 8./grr.^(0.5) .* alphaT .* frrr .* frT - ...
           6 .* grr.^(0.5)./gT .* alphaT .* frT.^2 - ...
           1./grr.^(0.5) .* gT .* frrr .* dalphaT + ...
           2./grr.^(0.5) .* gT./r .* frrr .* alphaT - ...
           grr.^(0.5) .* gT .* d2alphaT + 4.*grr.^(0.5) .* gT./r .* dalphaT + ...
           2 .* grr.^(0.5) .* alphaT .* Krr .* KT - 1./grr.^(0.5) .* gT .* ...
           alphaT .* Krr.^2 - 6.*grr.^(0.5) .* gT .* alphaT./r.^2 + ...
           2./grr.^(1.5) .* gT .* alphaT .* frrr.^2;

KT_dot = beta .* dKT - 1./grr.^(0.5) .* alphaT .* gT .* dfrT + ...
           2./r .* beta .* KT - 1./grr.^(0.5) .* gT .* dalphaT .* frT - ...
           2./grr.^(0.5) .* alphaT .* frT.^2 + grr.^(0.5) .* gT .* ...
           alphaT./r.^2 + 1./grr.^(0.5) .* alphaT .* KT .* Krr .* gT;

frrr_dot = -alphaT .* grr.^(0.5) .* gT .* dKrr + ...
            beta .* dfrrr - alphaT./grr.^(0.5) .* Krr .* frrr .* gT - ...
            10 .* alphaT .* grr.^(0.5) .* Krr .* frT + ...
            12 .* grr.^(1.5)./gT .* frT .* alphaT .* KT + ...
            3 .* dbeta .* frrr - 4 .* grr.^(1.5) .* KT .* dalphaT + ...
            8 .* grr.^(1.5)./r .* KT .* alphaT + grr .* d2beta - ...
            4 .* grr.^(0.5) .* alphaT .* KT .* frrr - ...
            grr.^(0.5) .* gT .* Krr .* dalphaT + 2 .* grr.^(0.5) .* ...
            gT./r .* Krr .* alphaT;

frT_dot = -alphaT .* grr.^(0.5) .* gT .* dKT + ...
            beta .* dfrT + 2./r .* beta .* frT + dbeta .* frT - ...
            grr.^(0.5) .* gT .* KT .* dalphaT - alphaT./grr.^(0.5) .* ...
            KT .* frrr .* gT + 2 .* alphaT .* grr.^(0.5) .* KT .* frT;

u_dot = [grr_dot, gT_dot, Krr_dot, KT_dot, frrr_dot, frT_dot];

```

Runge-Kutta Code

```
function [y, times] = rk4sp4(fun, tspan, y0, xi, D, num_steps)
% rk4sp4 4th Order Runge-Kutta Method (Spectral).
%
% Syntax:
% [y, times] = rk4sp4(fun, tspan, y0, N)
% [y, times] = rk4sp4(fun, tspan, y0, xi, D)
% [y, times] = rk4sp4(fun, tspan, y0, xi, D, num_steps)
%
% Summary:
% ODE solver using 4th order Runge-Kutta method with a fixed step size.
%
% Input:
% fun = Function handle that returns the value of  $y' = f(t, y(t, x))$ .
%
% tspan = [T_0 T_final] (i.e., the range of times to solve for  $y(t, x)$ ).
%
% y0 = Initial values at  $y(t = 0, x = xi)$ .
%
% N = the degree of the approximation (N + 1 collocation points).
% OR
% xi = the collocation points (can be from a scaled interval).
%
% D = the differentiation matrix.
%
% num_steps = The number of time steps that the solver will take. Default
% will be the number of steps that make the step size  $h = 0.01$  (or as
% close as it can be to 0.01).
%
% Output:
% y = The values of the function  $y(t)$ .
%
% times = The points  $t$  where  $y(t)$  was evaluated.
%
% Example:
%
%     rmin = 3;
%     rmax = 5;
%     t0 = 0;
%     t1 = 1.5;
%     num_steps = 750;
%
%     norm_err = zeros(6, num_steps + 1);
```

```

% N = 20;
% [D, x] = cheby(N);
% r = shiftInterval(x, rmin, rmax);
% [grr, gT, Krr, KT, frrr, frT] = schwarzschild(r);
% u0 = [grr, gT, Krr, KT, frrr, frT];
% [y,t] = rk4sp4(@rhs_rk4sp4, [t0 t1], u0, r, D, num_steps);
% err = bsxfun(@minus, y, u0);
% for i = 1:6
%     for j = 1:num_steps + 1
%         norm_err(i,j) = norm(err(:,i,j),2);
%     end
% end
% figure, semilogy(t, norm_err)
%
%
% References:
% [1] "Runge-Kutta methods." http://en.wikipedia.org/wiki/Runge-Kutta\_methods.
%
%
% Author:
% Timothy D. Dewey (November, 2010)

if nargin == 4
    [D, xi] = cheby(xi);
end

if nargin < 6 % h ~ 0.01
    num_steps = (tspan(2) - tspan(1)) * 100;
end

times = linspace(tspan(1), tspan(2), num_steps+1);
h = times(2) - times(1);

yn = y0;
% Column oriented.
y = zeros(numel(y0),length(times));
y(:,1) = yn(:);

cnt = 1;
for t = times(2:end);

    cnt = cnt + 1;

    k1 = fun(yn, xi, D);
    k1(1,:) = boundary(k1(1,:), yn(1,:), y0(1,:));

```

```

    yn = yn + h/2 * k1;

    k2 = fun(yn, xi, D);
    k2(1,:) = boundary(k2(1,:), yn(1,:), y0(1,:));
    yn = yn + h/2 * k2;

    k3 = fun(yn, xi, D);
    k3(1,:) = boundary(k3(1,:), yn(1,:), y0(1,:));
    yn = yn + h * k3;

    k4 = fun(yn, xi, D);
    k4(1,:) = boundary(k4(1,:), yn(1,:), y0(1,:));
    yn = yn + h/6 * (k1 + 2*k2 + 2*k3 + k4);

    y(:,cnt) = yn(:);

end

% Reshape the output.
new_size = length(times);
sizey0 = size(y0);
nonsingsize = sizey0(sizey0 > 1);

if isempty(nonsingsize)
    new_size = [new_size, 1];
else
    for i = length(nonsingsize):-1:1
        new_size = [nonsingsize(i), new_size]; %#ok<AGROW>
    end
end

y = reshape(y, new_size);

```

Appendix B: C Code Outline

Outline of main() and command line usage

Outline of main():

```
void main()
{
    parseArgs();
    initializeVariables();
    collocationPoints(r_i, r0, r1, N);
    analytic(u0, r_i, N);
    rk(u, u0, dt, t0, t1, D, r_i, N);
}
```

Input Arguments:

```
-t, --time    t0 t1  Time range (double t0, double t1, set default to
                    (0, 100)
-e, --delta   dt     Time step size delta t (double dt, set default to
                    0.001)
-r, --radius  r0 r1  Radius range (double r0, double r1, set default to
                    (1.9, 11.9))
-g, --degree  N      Degree (double N, set default to 30)
```

Command line usage example:

```
einsteinSolver --time 0 200 --radius 1.9 20 --delta 0.0001 --degree 40
```

Einstein Functions in “einstein.h”

```
// Set initial conditions in u0.
void analytic(double u0**, double* r_i, uint32_t N);

// Compute the approximation to the time derivative u_dot.
void rhs(double** u_dot, double** u, double** D, double* r_i, uint32_t N);

// Compute the adjustment to for the outer boundary.
void boundary(double** u_dot, double** u, double** u_analytic);
```

PDE Functions in “pde.h”

This file should contain PDE functions. Spectral code and Runge-Kutta code.

Solution structure:

```
struct soln
{
    uint32_t steps;
    uint32_t N;
```



```

    uint32_t num_comp; // number of components
    double*** solution; // (steps x (N+1) x num_comp)
};

void initSolution(soln* S, uint32_t steps, uint32_t N, uint32_t num_comp);
void freeSolution(soln* S);

```

Chebyshev differentiation matrix:

```

// Compute Chebyshev collocation points and differentiation matrix.
void cheby(
    double** D,      // Sets D to (N+1) x (N+1) differentiation matrix
    double* x_i,     // Sets x_i to collocation pts in [-1, 1]
    double* r_i,     // Sets r_i to collocation pts in [r0, r1]
    double r0,
    double r1,
    uint32_t N
);

```

Call Runge-Kutta (1-D in space and time):

```

void rk(
    soln* u,
    double** u0      // (N+1) x num_comp, loop over collocation points
    double dt,       // size of the time step
    double t0,
    double t1,
    double** D,      // (N+1) x (N+1) Differentiation Matrix
    double* r_i,     // (N+1) Collocation points
    uint32_t N,
);

```