SOLVING SCALAR FIELDS IN ADS SPACE-TIME USING PARALLEL COMPUTING AND GPUS

by

Philipp Gregoryanz

A thesis submitted to the Faculty of Graduate Studies in partial fulfillment of the requirements for the Masters of the Science degree.

Department of Applied Computer Science
Master of Science in Applied Computer Science
The University of Winnipeg
Winnipeg, Manitoba, Canada
December, 2024

Contents

Abstract

Computers have been around a long time and used for many resources like gaming, emailing and science. Concerning the sciences, computers are excellent at repetitive tasks and crunching numbers without (much) error. In data science there are two methods of programming - serial and parallel. Both have the same goal and endpoint however depending on the problem, one might be more efficient in solving the problem. My thesis is on using parallel computing with GPUs to numerically solve a system of non-linear first order differential equations that describe scalar fields in AdS space-time. Since this has been done by another using CPUs, it will be important to compare that the simple case matches that of the previous paper. However once we have confirmed that our results match the previously found results, we can use graphic processing units to solve the problem to higher order and more accuracy.

Chapter 1

Introduction

In physics and many other sciences, data science, data collection and data calculation is a necessary step to solve a problem. Before the advent of computers many difficult and lengthy calculations were done by people, which took lots of time and were prone to many errors. Some of the mechanical devices that were created were the abacus, the mechanical calculator and the analytical engine. Once computers were introduced these calculations were solved with incredible speeds and to great accuracy. This is because computers are exceptional at preforming repetitive tasks at great speed with (almost) no error. Hence many sciences used computers to make numerical calculations and shuffle through big data sets which would normally take very long for a human to do. In order to make a computer do calculations one needs to define a problem that needs solving and decide on which technique or type of computing to use.

There are two types of computing in computer science: sequential computing and parallel computing. Every computer requires a processor to operate. There exists many processors, but the two main ones are the central processing unit (CPU) and the graphics processing unit (GPU). Both of these processors can handle sequential and parallel computing. In fact a lot of parallel computing is done with multiple CPUs. However, CPUs excel at sequential computing while GPUs excel at parallel computing. Interestingly enough, the GPU was originally designed to accelerate image processing, graphics, and intense calculations [1].

Many problems in physics are too difficult to be solved analytically by hand and thus require a computer to numerically solve these problems. Since computers work in discrete intervals one needs to discretize the equations before programming them into a computer. Once run, one will (usually) get at least one list, sometimes many, of numbers that then can be analyzed in many ways including plotting them graphically.

1.1 The problem

General relativity tells us that space-time can be curved, responding dynamically to mass/energy, in many ways. Maximally symmetric space-times, that is space-times that maximize the result of the spherical symmetry, are important space-times which can be described according to the sign of the curvature term, called the Ricci scalar. Taking the trace of the Einstein's equation one can see that the sign of the Ricci scalar is the negative of the cosmological constant. These types are:

1. Negative curvature - Anti-de Sitter space (AdS) (negative cosmological constant)
2. Zero curvature - flat space (zero cosmological constant)
3. Positive curvature - de Sitter space (dS) (positive cosmological constant)

However not all space-times are stable. Some collapse under the gravitational pull of matter in the space-time into a black hole.[1] Whether the AdS space-time can collapse into a black hole if small amounts of matter is added is of interest because the Anti-de Sitter/Conformal field theory (AdS/CFT) correspondence [2] says that gravity in AdS is equivalent to a type of particle physics theory, which is conformal field theory. This CFT idea is very similar to the one describing the strong nuclear force — which is very difficult to solve. While the AdS space-time does not describe our real universe, it is an important solution in mathematical studies in general relativity as well as the AdS/CFT correspondence (see [3] for a review of AdS geometry and the AdS/CFT correspondence). There is a large literature [4, 5, 6, 7, 8, 9, 10] about the stability of AdS space-time if given some small perturbation. In this thesis our focus will be in Anti-de Sitter space-time (particularly in four dimensions [three space and one time]) and whether or not small perturbations in this AdS space-time causes it to collapse into a black hole. In this case a perturbation is when one adds a small amount of a matter field into the system in question (the system is deviated by an outside influence in a small amount).

Since the problem deals with black hole formation, it is useful to note that the physics of black hole formation in the three maximally symmetric space-times introduced above, are manifestly different. For example, in flat space, upon introducing a perturbation, non-interacting matter may collapse into a black hole under its own self-gravity, but if it fails then the matter will simply disperse. However in AdS space massless waves, acting as the perturbation, can travel an infinite distance in a finite time and bounce back. Thus if the formation of a black hole does not happen in

[1] A black hole is a region of space-time where the gravitational pull is so immense that not even light (which travels at the maximum possible speed consistent with known physics) can escape the black hole once it enters its event horizon.

the first attempt, the matter will disperse, and upon hitting the edge of the space-time will bounce back and possibly form a black hole again. Given the right initial conditions, this will keep on happening until a black hole forms. The question becomes which initial conditions will lead to the formation of a black hole.

To understand how light can travel an infinite distance in finite time in AdS, consider an AdS space-time. One can describe this space-time and its properties by the line element ds^2 which gives the square of the infinitesimal space-time interval, analogous to the infinitesimal Pythagorean distance in curvilinear co-ordinates. For the problem of black hole formation, we consider a self-gravitating massless scalar field embedded in a spherically symmetric, asymptotically AdS space-time. This space-time has n spatial dimensions, one time dimension, and the line element,

$$ds^2 = \left[-dt^2 + dx^2 + \ell^2 \sin^2\left(\frac{x}{\ell}\right) d\Omega_{n-1}^2\right] \sec^2\left(\frac{x}{\ell}\right), \tag{1.1}$$

where $d\Omega_{n-1}^2$ is the line element on S^{n-1}, which is the spherical surface of an n-dimensional sphere, ℓ is a curvature parameter of the AdS space-time with dimensions of length, x is the radial co-ordinate, $x/\ell \in [0, \pi/2]$, and t is the time co-ordinate, $t/\ell \in [0, \infty)$ [3]. We will be working in natural units with units for which $\ell = 1$ for ease of calculations. Natural units are units in which some fundamental constants are set to one. In our case we also set $c = 1$ which means that time and space have the same units. This makes it easier to write equations without worrying about dimensional analysis.

Note that for massless particles (e.g. light rays) $ds^2 = 0$. Also we will consider a radially directed light ray, thus, $d\Omega_{n-1}^2 = 0$. Hence equation 1.1 becomes,

$$0 = \left[-dt^2 + dx^2\right] \sec^2(x) \tag{1.2}$$

which, when simplified and rearranged, gives $dt = dx$. Now integrating over $x \in [0, \pi/2]$ gives

$$t = \int_0^{\frac{\pi}{2}} dx = \frac{\pi}{2}. \tag{1.3}$$

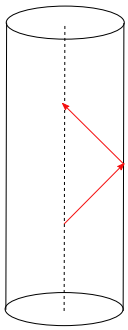


Figure 1.1: AdS space-time showing the universe and the wave in the field (in red) bouncing from the edge. The circular cross sections are infinite spatial slices. Note that time is along vertical axis. This diagram shows how the light rays see the space-time. Image provided by Dr. Andrew Frey, used by permission

Figure 1.1 illustrates an AdS space-time where the cross-sectional disk of the cylinder is infinite in spatial extent, due to the $\sec(x)$ term in the line element, the path along x from 0 to $\pi/2$ is infinite, since $\int_0^{\pi/2} \sec(x)dx$ is divergent, but the path of a light ray (diagonal red line) depicts that it travels the space in a finite time.

Thus the total time it takes for a massless particle (e.g. light ray) to travel to the edge of this AdS space-time and back is, $t = 2\left(\frac{\pi}{2}\right) = \pi$, which is finite and hence this allows oscillations. After each bounce the self-gravity of the wave focuses the energy, since energy is conserved in each bounce. Thus it becomes more dense as the wave approaches the origin - this allows a black hole to form after multiple oscillations.

Our problem consists of finding the time evolution of a scalar field in AdS space-time which is influenced under its own self gravity. This will be done by numerically solving a set of ordinary differential equations using parallel computing on GPUs.

1.2 Sequential versus parallel computing

Given a calculation that needs to be done on a computer, one will always have input(s), an algorithm that does the calculation and output(s). One in general can have multiple inputs and outputs, which will depend on the algorithm used. An algorithm is a sequence of finite steps that attempt so solve a particular problem. This problem can be in general anything as long as it has inputs,

outputs and has a finite number of steps. In general different algorithms which attempt to solve the same problem will have different steps in the calculation but each algorithm will have the same input(s) and output(s).

Every computer has at least one processor which is an integrated circuit whose task is to preform particular calculations and operations that drives a computer. Every computer has a main processor called the central processing unit (CPU). The CPU has multiple cores. Each core acts as the 'brain' of the CPU which processes one instruction per cycle. The CPU can offload different tasks to each core. However a key feature of the CPU is that it excels at doing tasks sequentially (that is the next instruction won't be processed until the previous one is completed). This ends up taking time and depending on the problem/calculation at hand, this might be unfeasible. Enter the graphical processing unit (GPU). The primary purpose of a GPU is to handle graphics (in say a video game) such that the game does not experience any lag. However a key feature of GPUs is that they have thousands of cores and thus excels at doing (various) tasks in parallel (that is multiple instructions are carried out simultaneously). This can in fact speed up the calculations and therefore the programs run time down to feasible times. It should be noted that both CPUs and GPUs can handle sequential and parallel tasks (one can especially use multiple CPUs for parallel processing), however CPUs are great for sequential computing while GPUs are great for parallel computing. Whether one uses sequential computing, parallel computing, CPUs or GPUs is partly independent of the problem. Though due to speed some problems might not be feasible with sequential computing because they are too intensive.

CUDA is a parallel computing platform and programming model created by NVIDIA that makes using a GPU for general purpose computing simple [11]. CUDA can be used with any (Turing complete) programming language to optimize GPUs and uses parallel programming. To work with GPUs is somewhat different than to work with CPUs when it comes down to programming. While both have keywords, loops, control structure, etc, there are significant differences. GPUs have three predefined types of functions: the global function, the device function and the host function. The global function is a void function which can be accessed through the keyword __global__ and runs on the GPU, while the device function is a non-void function which can be accessed through the keyword __device__ and runs on the GPU, which can only be called from a global function. The GPU kernel must be started by a global function call. Lastly, if one needs to write serial code, one can use the host function which can be accessed through the keyword __host__ and runs on the CPU.

However before looking at these one must understand threads. In computer science a thread of execution is the smallest sequence of programmed instructions that can be executed independently

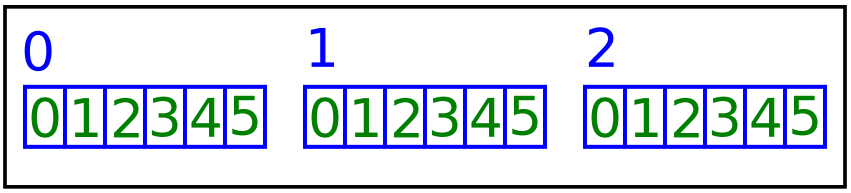


Figure 1.2: 1D grid containing 1D blocks. Blocks are labeled in blue, threads are labeled in green. Image provided by Dr. Andrew Frey, used by permission

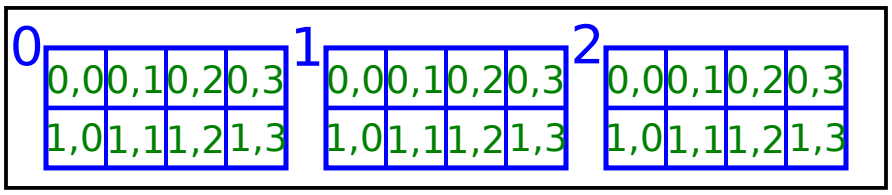


Figure 1.3: 1D grid containing 2D blocks. Blocks are labeled in blue, threads are labeled in green. Image provided by Dr. Andrew Frey, used by permission

on the GPU kernel [12]. They are similar to CPU threads but there are many more of them in GPUs. Multiple threads can be executed simultaneously thereby accessing the parallel part of parallel programming. If one has multiple threads however, these threads need to be synchronized before being executed inside the kernel. This is achieved via the command,

```
__syncthreads();
```

This command ensures that all the threads of a block will execute their next phase only after all the threads have finished their previous phase. This way the resulting calculation will be correct for when there are tasks (code) that must be done in a particular order (even if concurrently), when parallel programming is used. Hence threads need to be synchronized at specific times.

To count and identify these threads one uses gridDim, blockIdx, blockDim and threadIdx. BlockDim contains the dimension of the block and thus gives the number of threads in a block in a particular direction. GridDim contains the dimensions of the grid, that is the number of blocks in a grid in a particular direction. BlockIdx gives the block ID in the corresponding axis and threadIdx gives the thread ID in the corresponding axis. To access the full thread ID in a particular dimension (which is an integer) one uses:

```
int i = blockIdx.[x,y,z]*blockDim.[x,y,z]+threadIdx.[x,y,z]
```

where a particular direction (x, y, or z) is used. Thus we have a grid of 1, 2 or 3 dimensional blocks, each of which has multiple threads shown in figures 1.2 and 1.3.

Another important CUDA memory concept is that of shared memory. A shared memory vari-

able is available to all the threads in the same block with the same values. However, because blocks are physical hardware units, the shared memory is different from block to block. That means declaring a shared memory variable effectively means that one is creating a variable with the same name but different values on each block. The kernel has to know how much memory to allocate, thus another variable needs to be used in the kernel call.

The temp array is declared as a shared array using the keyword __shared__.

```
extern __shared__ double temp[]
```

The compiler assigns a certain amount of memory to the variable, which determines the number of elements in the array. This is done by introducing a third variable in the kernel call. The extern keyword indicates that the memory assigned is determined externally to the function by this third variable.

After writing the calculational parallel part of the code, to call it one needs to call the CUDA kernel along with its parameters. The CUDA kernel is a function that gets executed on the GPU. The parallel portion of the code is executed by starting a process (kernel) on the GPU with a global function call. This call must state how many blocks will be in the grid (per dimension) as well as the number of threads in each block (per dimension). This call will be executed in parallel by using multiple threads (one thread per each time). Every CUDA kernel uses the keyword __global__ and is accessed via the call

```
function_name<<<number_of_blocks, number_of_threads_per_block,
shared_memory_size>>>(parameters);
```

The memories of the CPU and the GPU are physically different and thus normally one has to copy variables back and forth from the GPU and the CPU. However starting with CUDA 6, one can use unified memory which creates a pool of managed memory which is shared between the CPU and the GPU. It is accessible to both the CPU and the GPU and thus bridges the gap between the two processors. Thus instead of copying variables to the GPU, doing parallel calculations on it then copying back to the CPU, one can bypass the copying via unified memory and use the same variable on both the CPU and GPU simultaneously! Unified memory allows for single allocation by using only a single pointer. This makes it simpler to access and manage data. To allocate memory on both processors one uses the call:

```
cudaMallocManaged(&variable, size_of_variable)
```

along with,

```
cudaDeviceSynchronize();
```

which synchronizes the kernel calls and ensures that all parallel threads have completed in the kernel before moving to the next line in the host code. We declare the variable first without the size. The cudaMallocManaged command then tells the size of the variable.

With this basic introduction to parallel programming with CUDA GPUs, one can write both sequential and parallel code as needed to simplify the cost of running the program.

Chapter 2

The Physics Problem

2.1 Main equations

Fields[1] are fundamental in the universe and permeate space-time. Recall from chapter 1 that it takes light a finite amount of time to travel though the AdS space-time. This includes the field excitations (waves and particles) and so if a gravitational collapse does happen, it will happen either quickly or after the wave has traversed the space-time an integer number of times. In AdS space-time, if gravity is ignored, a wave in the field will oscillate. If gravity is included, the perturbing field will have a small self-interaction. In figure 1.1 we have a massless scalar field, traveling in a closed AdS space-time. The wave (depicted in red) travels to the edge and bounces back. With a gravitational self-interaction, the wave focuses each time it returns to the center and thus may form a black hole.

Since the field introduced is a scalar field we can describe the field by a scalar function which depends on both space and time, $\phi(x,t)$, where x is the radius co-ordinate and t is the time co-ordinate. This massless scalar field propagates waves that obey the Klein-Gordon equation. In a space-time with metric given by equation 1.1 in four space-time dimensions, the Klein-Gordon equation is[2]

$$-\frac{\partial^2\phi(x,t)}{\partial t^2}+\frac{\partial^2\phi(x,t)}{\partial x^2}+\frac{2}{\sin x\cos x}\frac{\partial\phi(x,t)}{\partial x}=0. \tag{2.1}$$

By the method of separation of variables, one can write the solution $\phi(x,t)$ as

[1] A field is a physical quantity that associates a value to each point in the space-time manifold. A field is continuous and can be described by a scalar, a vector, a spinor or a tensor quantity.

[2] See appendix B.2 for the derivation and the equations that follow.

$\phi(x,t) = \sum_i A_i \cos(\omega_i t + B_i) e_i(x)$ where $e_i(x)$ are the eigenfunctions of the operator

$$\hat{L} = -(\tan^{1-n} x)\frac{\partial}{\partial x}\left(\tan^{n-1} x \frac{\partial}{\partial x}\right) \tag{2.2}$$

with eigenvalues ω_i^2, where A_i and B_i are constants and n is the number of spatial dimensions. The eigenfunctions are given by

$$e_i(x) = \kappa_i \cos^{\lambda_\pm}(x) P_i^{n/2-1,\pm n}(\cos(2x)), \tag{2.3}$$

in terms of the Jacobi polynomials $P_i^{j,k}$ where

$$\kappa_i = \sqrt{\frac{2(2i+\lambda_\pm) i! \Gamma(i+\lambda_\pm)}{\Gamma(i+n/2)\Gamma(i \pm n/2 + 1)}} \tag{2.4}$$

and $\omega_i = \lambda_\pm + 2i$ and $\lambda_\pm = \frac{n}{2} \pm \frac{n}{2}$ [3]. Since we want the normalizable solution we have to use the λ_+ since the λ_- option is non-normalizable. These eigenfunctions represent oscillating standing waves.

Once we consider the gravitational effects of the scalar field, it changes the metric. Assuming spherical symmetry, the line element becomes

$$ds^2 = \sec^2(x)[-A(x,t)e^{-2\delta(x,t)}dt^2 + A(x,t)^{-1}dx^2 + \sin^2(x)d\Omega_{n-1}^2], \tag{2.5}$$

where $A(x,t)$ and $\delta(x,t)$ are integrals of derivatives of $\phi(x,t)$ [7]. The Klein-Gordon equation with this line element is a partial differential equation which is very difficult to solve. The standard approach to solve this partial differential equation is to use numerical algorithms of a discretized version by parallel processing at a high resolution (many grid points). This includes numerical integration for $A(x,t)$ and $\delta(x,t)$.

We will consider a perturbative approach where the field amplitude is small and allow A_i and B_i to vary with time. Therefore we have $\phi(x,t) = \sum_i A_i(t) \cos(\omega_i t + B_i(t)) e_i(x)$. [13] showed that the Klein-Gordon equation at the first non-trivial order in the amplitude can be written as a system of coupled ordinary differential equations for $A_i(t)$ and $B_i(t)$.

When expanded to third order in powers of $A_i(t)$, the equations of motion for this scalar field are given by a system of two first order non-linear coupled ordinary differential equations via

$$\frac{dA_l}{dt} = - \sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} \frac{S_{ijkl}}{2\omega_l} A_i A_j A_k \sin(B_l + B_k - B_i - B_j) \tag{2.6}$$

for A_l and

$$\frac{dB_l}{dt} = - \sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} \frac{S_{ijkl}}{2\omega_l A_l} A_i A_j A_k \cos(B_l + B_k - B_i - B_j) - \frac{T_l}{2\omega_l} A_l^2 - \sum_{\substack{i \\ i\neq l}} \frac{R_{il}}{2\omega_l} A_i^2 \tag{2.7}$$

for B_l [13].

These two equations are the perturbation theory version of the Klein-Gordon equation for the scalar field $\phi(x,t)$ with self-interacting gravity. The functions $A_i(t)$ are the amplitudes of the scalar field while the functions $B_i(t)$ are the phases of the scalar field. If the gravitational self-interaction of the field is ignored, $A_i(t)$ and $B_i(t)$ are constants. When the gravitational self-interaction of the field is included, then $A_i(t)$ and $B_i(t)$ become time dependent functions whose derivatives follow equations 2.6 and 2.7. That is, a very weak (low amplitude) massless wave with self-gravitation in AdS space-time is described by equations 2.6 and 2.7. Equations 2.6 and 2.7 have a symmetry in which if $A \to \lambda A$ and $t \to t\lambda^{-2}$ then the equations remain invariant, for a scalar λ. This means that an overall normalization constant is irrelevant.

In equations 2.6 and 2.7 we have three coefficient arrays, T_l, R_{il} and S_{ijkl} which are one, two and four indexed arrays respectively. The S, R and T coefficients are given by integrals of products of the eigenfunctions $e_i(x)$[14] [15]. The equations depend on the space-time dimension because the coefficients depend on the space-time dimension. The S coefficient has a particular property that the coefficient is zero unless $i+j = k+l$ and $(i,j) \neq (k,l)$. Also it will be useful, later on, to use the fact that $S_{ijkl} = S_{ijlk}$, and $S_{ijkl} = S_{klij}$ which is shown by[14]. From these two properties it can be seen that $S_{ijkl} = S_{jikl}$. We use the coefficients calculated by [16] for AdS_4 (and AdS_5) up to index 399[3]. Our aim is to solve for the functions $A_i(t)$ and $B_i(t)$ which will allow us to find the scalar field $\phi(x,t)$ in this space-time. Once $\phi(x,t)$ is found one can get many properties of this space-time such as how the scalar field affects the metric and how long it takes for a black hole to

[3]We thank the author of [16] for providing these coefficients.

form.

We can also look at the energy of this system and ask about its conservation. Define the 'energy per mode' as $E_i = \omega_i^2 |A_i(t)|^2$. The total energy will then be $E_T = \sum_i E_i$. This total energy is conserved by the perturbative equations [13] [15]. Furthermore, the number of exitations at time t is defined as $N_i = \omega_i |A_i(t)|^2$, and, like the total energy, the total excitation number, $N_T = \sum_i N_i$ is also conserved. This can be seen by taking the time derivative of E_T and N_T and using equations 2.6 and 2.7:

$$\begin{aligned}
\frac{dE_T}{dt} &= \sum_l \omega_l^2 2A_l(t)\frac{dA_l(t)}{dt} = -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} \frac{2S_{ijkl}\omega_l^2}{2\omega_l} A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl}\omega_l A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl}\left(\frac{\omega_l + \omega_k}{2}\right) A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl}\left(\frac{\omega_i + \omega_j + \omega_k + \omega_l}{4}\right) A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= \sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl}\left(\frac{\omega_i + \omega_j + \omega_k + \omega_l}{4}\right) A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \qquad (2.8)
\end{aligned}$$

The replacement in line 3 is due to $l \leftrightarrow k$ symmetry of all the other factors in line 2. Since $\omega_i + \omega_j = \omega_k + \omega_l$ (via the condition $i+j = k+l$). we can replace $(\omega_l + \omega_k)/2$ to $(\omega_l + \omega_k + \omega_i + \omega_j)/4$. The final line swaps (i, j) with (k, l). The sine factor is odd and all the other factors are even making the entire sum odd. Thus $\frac{dE_T}{dt} = 0$ and so E_T is conserved.

Similarly the total excitation number should be conserved. The derivation for the conservation of the total excitation number is similar to the conservation for the total energy. In particular,

$$
\begin{aligned}
\frac{dN_T}{dt} &= \sum_i \omega_l 2A_l(t)\frac{dA_l(t)}{dt} = -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} \frac{2S_{ijkl}\omega_l}{2\omega_l} A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl} A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \\
&= -\sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{klij} A_i A_j A_k A_l \sin(B_i + B_j - B_l - B_k) \\
&= \sum_{\substack{i,j,k,l \\ i+j=k+l \\ \{i,j\}\neq\{k,l\}}} S_{ijkl} A_i A_j A_k A_l \sin(B_l + B_k - B_i - B_j) \qquad (2.9)
\end{aligned}
$$

But these last two summations are opposite sign since $S_{ijkl} = S_{ljki}$ and since the sine function is odd, thus $\frac{dN_T}{dt} = 0$ and so N_T is conserved. Note that to show these two quantities are conserved, the fact that sine is odd was used as well as rearranging dummy indices (in particular $i \leftrightarrow j, k \leftrightarrow l$ in 2.9 in the second to last step of the derivation. The last line of 2.9 used the symmetry properties.

2.2 Previous solutions

The problem of black hole formation in AdS space-time is not new. Many previously worked on this problem including [7] who presented numerical evidence of the instability to black hole formation for low amplitudes. The authors of [7] concluded that AdS space-time became unstable under arbitrarily small perturbations. They conjectured that this instability is triggered by a resonant mode mixing which gives rise to diffusion of energy from low frequencies to high frequencies. Figure 2.1 shows the time that a black hole forms as a function of wave amplitude. It shows that large amplitude waves form black holes nearly immediately, but gravitational collapse occurs after crossing the space-time more and more times with decreasing amplitude. However [17] found that some initial conditions do not form black holes at small amplitudes. This is depicted in 2.2 where it can be seen that at small amplitudes (ϵ), the time it takes to form a black hole, t_H suddenly increases rapidly (and is assumed to diverge).

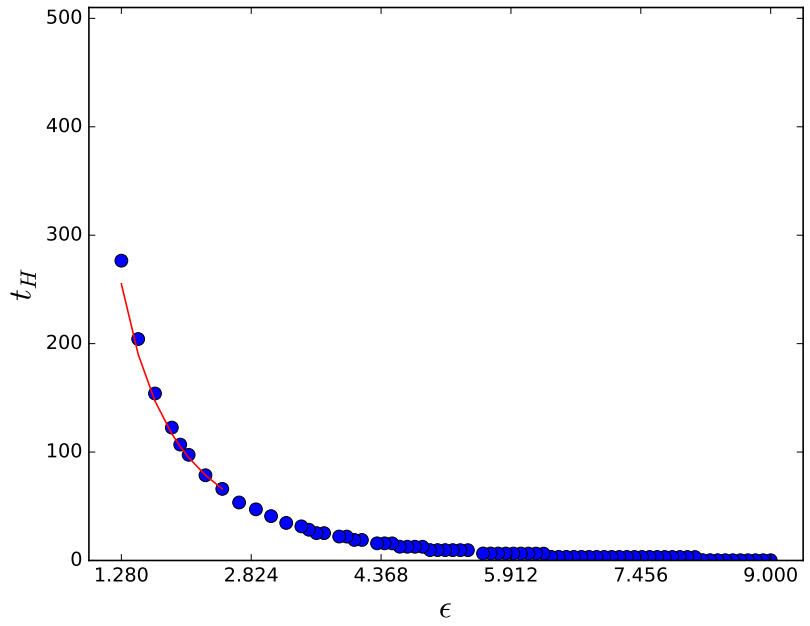


Figure 2.1: Fit for AdS scalar black hole collapse [7], where the vertical axis is the times for black hole formation and the horizontal axis is the amplitudes of the perturbation. This figure was released under the CC BY 4.0 license

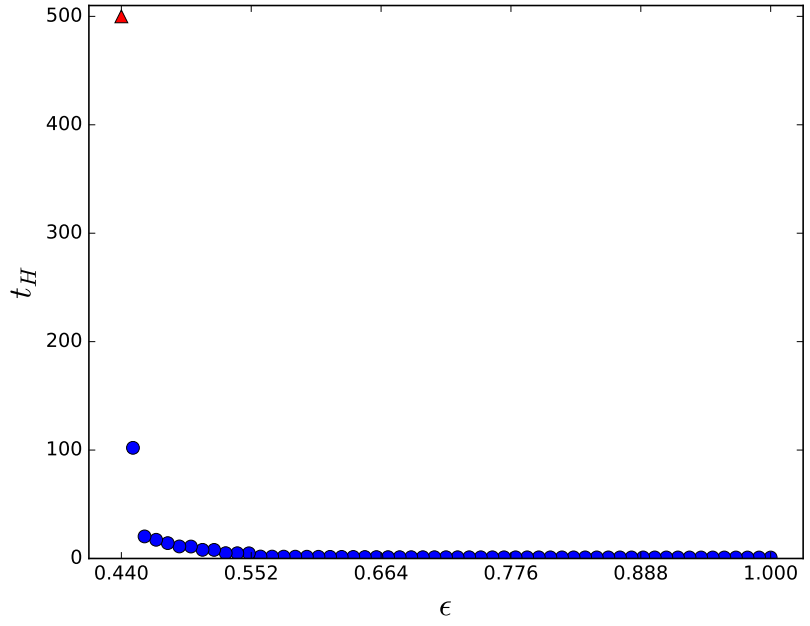


Figure 2.2: Fit for AdS scalar black hole collapse [17], where the vertical axis is the times for black hole formation and the horizontal axis is the amplitudes of the perturbation. This figure was released under the CC BY 4.0 license

These works were done using real massless scalar fields, which was then extended to complex masseless scalar field by [5]. [5] extended the work of [7] which showed an instability of pure AdS to formation of black holes due to complex scalar fields, in which they tried to reproduce the results. In this study, they demonstrated that for complex scalar fields black holes in AdS space-time form even when the reflecting boundary is placed at a finite distance. This indicates that the sharpening is a property of gravity and not the AdS space-time itself. Sharpening here simply means that the waves get more focused, so more energy is found in a smaller region. Furthermore massive scalars were analyzed by [18] and [6]. [18] showed that all the known stable initial data for massless scalar fields are dominated by single scalar eigenmodes. This means that initial data with equal energies in two modes collapse on time scales of order of of the inverse square of the field amplitude.

[13] was the first to develop the perturbative theory of instability of AdS space-times. Many authors then developed this perturbative theory including, [14],[15],[19],[20], [21] and [22]. There has been some controversy in the literature about initial conditions where $A_i \neq 0$ for only $i = 0, 1$ which equal energy in those two modes, known as "equal-energy two-mode initial data". [13] and [23] argued that the energy in the space-time would not form a black hole based on perturbation theory for equal-energy two-mode initial data. [21] however argued that a black hole would form in the perturbative case in a full numerical simulation which [18] agreed. Its interesting to note that certain types of initial data appear unstable at low amplitudes which lead to some disagreement [13, 23, 21].

These full partial and ordinary differential equations have been studied previously. Here we will use CUDA parallel programming to allow mass solution of many initial conditions. Once the solution is found, a fit in python will be done to analyze the results. The fit will take the form of $A_l = \alpha l^{-\beta} e^{-\gamma l}$ where α, β, γ are the fit parameters and l is the index. We expect the coefficients to decay exponentially for smooth solution, but to become algebraic if the system develops a singularity. γ measures the distance from the real axis to the closest singularity in the complex plane. If γ goes to zero, the system develops a singularity and coincides with the black hole formation in general relativity.

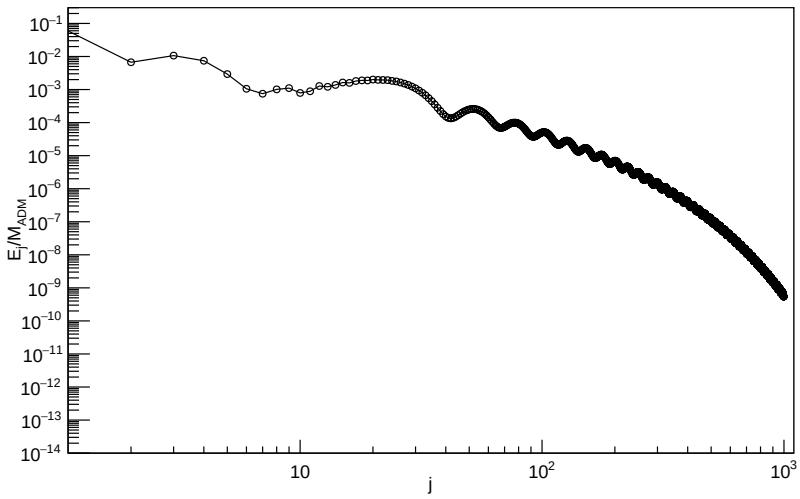


Figure 2.3: How the energy spectrum E_i becomes close to a power law right before black hole formation[6]. This figure was released under the CC BY 4.0 license

Figure 2.3 shows how the energy spectrum E_i becomes close to a power law right before black hole formation. When $\gamma \rightarrow 0$ A_i starts to follow a power law, so does E_i. Since the high l eigenfunctions have more structure ("wiggles") at smaller distances, this concentrates energy at small length scales, which means higher density that can form a black hole.

The problem of numerically solving equations 2.6 and 2.7 were previously done using parallelism on the CPU. [16] used the Dormand-Prince algorithm to carry out the time evolution. The Dormand-Prince method is a variation of the Runge-Kutta method with the difference being in the number of intermediate steps used in going from one time step to the next.

Chapter 3

Methods

3.1 Introduction

As can be seen from the previous chapter, the problem of black hole formation in AdS space-time boils down to solving a system of first-order, coupled, non-linear ordinary differential equations (2.6 and 2.7). Since these equations are non-linear we must resort to numerical methods in an attempt to solve them. There exist many algorithms for numerically solving differential equations, and the choice of one algorithm over another comes down to a choice of accuracy and personal preference. The fourth-order Runge-Kutta (RK4) algorithm and variants presented here are used here with GPU parallel computing. That is, the sums on the right-hand side of the differential equations are done parallely with GPUs, while the RK4 was done on the CPU. These algorithms generate values of $A_i(t)$ and $B_i(t)$ at each of a series of discrete time steps. These then can be plotted and analyzed as required. The evolution of the time-dependent coefficients, $A_i(t)$ and $B_i(t)$ are given by equations 2.6 and 2.7.

As stated earlier we will be working in AdS_4 space-time (4-D) and AdS_5 space-time (5-D). We are given the governing equations for the time-dependent functions which makes up the scalar field, $\phi(x,t)$, that is $A_i(t)$, $B_i(t)$. Here the discrete index i runs from $i = 0$ to $i = \infty$, however since computers are finite and work discretely we need to go up to a (large but finite) value of i (cutoff limit), which we label as n. Here we will start off with $n = 100$ where $0 \leq i < n$. This is a proof of the concept.

The governing equations for the time-dependent coefficients depend on three coefficient arrays (T_l, R_{il} and S_{ijkl}) and the eigenvalues, ω_l. We use these coefficients as calculated by [16] and thank the author for providing them for us.

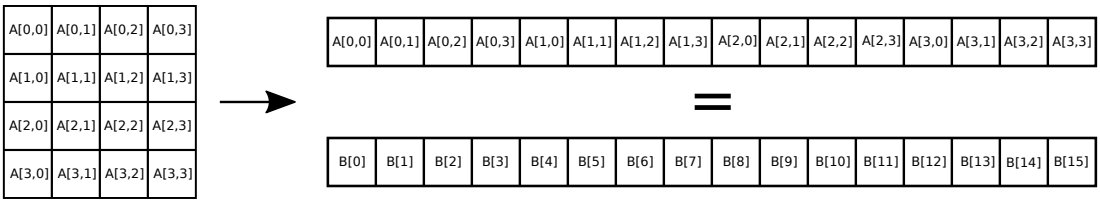


Figure 3.1: 2D array flattened to 1D. Image provided by: Dr. Andrew Frey and used with permission.

3.2 Flattened arrays

3.2.1 Regular flattened arrays

For CPU computing, one can store multi-dimensional arrays as just that, multi-dimensional arrays. When using GPUs, one can theoretically work with multi-dimensional arrays. However this way gets tedious when it comes to allocating memory. One can use pointers pointing to pointers but this technique is difficult when working with a high multi-dimensional arrays. The work around solution to this is to use flattened arrays as opposed to multi-dimensional arrays. Furthermore, variables are passed to global functions as pointers, which is simpler with flattened arrays. Flattening is possible for multi-dimensional arrays of any dimension; however, we will only use this for our two dimensional R_{ij} array (see below for a discussion of the S_{ijkl} array). Consider a two dimensional matrix $A[i][j]$ where $0 \leq i < n$ and $0 \leq j < m$. The process of flattening a multi-dimensional array reduces the original two dimensional array $A[i][j]$ to a one dimensional array $B[k]$ where $k = im + j$. The new array $B[k]$ will have a larger size for the one dimension than either dimension of A, namely $0 \leq k < nm$. This is convenient because one only needs to access one index as opposed to two. Thus the array was 'flattened' as shown in figure 3.1.

3.2.2 The S array

If there were no restrictions on the allowed index values then the S array would have

$$\sum_{i=0}^{n-1}\sum_{j=0}^{n-1}\sum_{k=0}^{n-1}\sum_{l=0}^{n-1} 1 = n^4 \tag{3.1}$$

elements. The reason we do not flatten the four dimensional S_{ijkl} array into a one dimensional array of size n^4 or treat is as a regular array is because many of the S values are zero and thus not stored. This allows us to avoid using excess memory to store trivial values. The non-zero values have indices that satisfy the conditions

1. $i + j = k + l$

2. $i \neq k$, $j \neq l$

3. $i \neq l$, $j \neq k$

For the S array we first find at which indices, (i, j, k, l) the value S_{ijkl} would give a non-zero value. This is important because if one tried to access a index whose respective value is not stored, an error would occur. Thus it was required to first determine which position, given by indices, i, j, k, l gave an allowed position (that is, at what unique set of i, j, k, l indices, which will be the 'position' index, will the S array have a corresponding non-trivial value. From the conditions said above we know that $0 \leq i, j, k, l < n$ and $k = i + j - l$ where n is the maximum index value. For a specific value of l we can think of S_{ijkl} as an $n \times n$ matrix with column and row denoted by i and j indices. We can calculate the total number of allowed values of j for a given i and l (i.e. the number of allowed elements in the i^{th} row of a matrix for a given l) and find

$$n - l + i - 1. \tag{3.2}$$

or

$$n + l - i - 1. \tag{3.3}$$

depending on whether i is smaller or larger than l respectively. Summing over i, we find the total number of allowed (i, j) pairs for a given l to be

$$\frac{n^2}{2} + n\left(l - \frac{3}{2}\right) - l^2 - l + 1. \tag{3.4}$$

For explicit calculations see appendix B.1.

The unique integer position of the elements for $i, j, k = i + j - l, l$ is the count of the elements before the j^{th} element of the i^{th} row of matrix l.

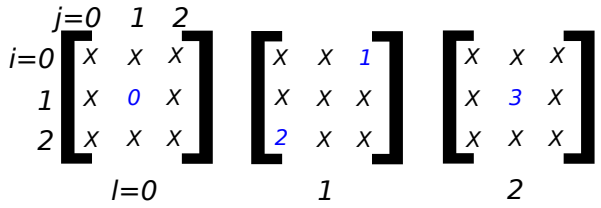


Figure 3.2: The position value for each valid i, j, l combination (with $k = i + j - l$). For example for $n = 3$ the allowed values are shown here. The black x's represents an element that is not allowed, while the blue numbers give the positions. Image provided by: Dr. Andrew Frey and used with permission.

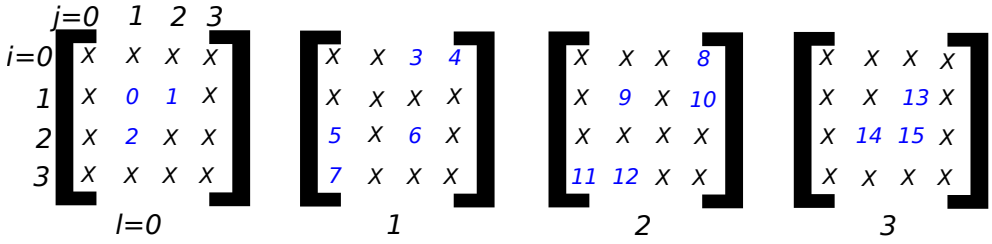


Figure 3.3: The position value for each valid i, j, l combination (with $k = i + j - l$). For example for $n = 4$ the allowed values are shown here. The black x's represents an element that is not allowed, while the blue numbers give the positions. Image provided by: Dr. Andrew Frey and used with permission.

For example, if we wish to find the position of $i = 3, j = 1, l = 2, k = 2$ the position is the total number of (i, j) pairs for $l = 0$ and $l = 1$ (the sizes of the two matrices) plus the total number of j values for $l = 2$ and $i = 0, 1, 2$ (the sizes of the three rows where the count is 0 for $i = 2 = l$) plus one (the number of allowed values of j in the $l = 2, i = 3$ row before $j = 1$). This is depicted in figure 3.3. As usual the first allowed element ($l = 0$, $i = j = 1$, $k = 2$) has position index of 0. The total number allowed values of the S array for $n = 3$, for example, is four as depicted in figure 3.2. The formulas for the allowed positions in the S array are,

$$\frac{1}{6}(3i^2 - 6il + 6in - 3i + 6j - 2l^3 + 3l^2n + 3ln^2 - 12ln + 2l), \tag{3.5}$$

if $i < l$ and $j < l$,

$$\frac{1}{6}(3i^2 - 6il + 6in - 3i + 6j - 2l^3 + 3l^2n + 3ln^2 - 12ln + 2l - 6). \tag{3.6}$$

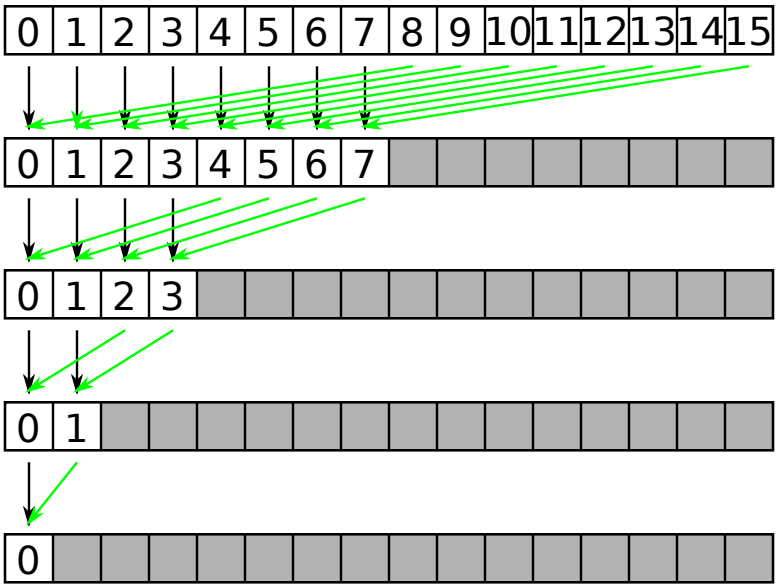


Figure 3.4: Parallel Reduction. Image provided by: Dr. Andrew Frey.

if $i < l$ and $j > l$,

$$\frac{1}{6}(-3i^2 + 6il + 6in - 3i + 6j - 2l^3 + 3l^2n + 3ln^2 - 6l^2 - 12ln + 2l - 6n + 6), \tag{3.7}$$

if $i > l$ and $j < l$, or

$$\frac{1}{6}(-3i^2 + 6il + 6in - 3i + 6j - 2l^3 + 3l^2n - 6l^2 + 3ln^2 - 12ln + 2l - 6n) \tag{3.8}$$

if $i > l$ and $j > l$.

3.3 Parallel reduction

To evaluate 2.6 and 2.7 we need to evaluate sums. We carry out sums using parallel reduction. Parallel reduction is an algorithm to sum the elements of a one-dimensional array in a few parallelizable steps.

The way parallel reduction works is by using half the number of threads of the elements in the array. Every thread calculates the sum of its own element (the element that the thread is pointing to) and another element. Then the result is stored in the original element's location. The number

of threads is then halved and the process repeats until only one element remains. This element will contain the sum of the original array. The black arrows show the values that aren't changed while the green arrows show the elements that are added to them as seen in figure 3.4. In this figure it is shown how in each step there is a simultaneous addition of two elements in the array, which gets repeated as the rows go down. At the end of the reduction sum, the total sum will be in the first element of the array.

A simple example is given in appendix A.2 under the function `Array_sum_GPU(double* A,double* result)`. In this example we are given an array A and are supposed to sum the elements of this array using the parallel reduction technique described above. The algorithm, which can be viewed in appendix A.2 is quite simple. We only have one block. The size of the shared memory is the number of threads per block times the size of a double.

```
extern __shared__ double temp[];

for(int i = threadIdx.x; i < sizeA; i += blockDim.x){
temp[threadIdx.x] += A[i];
}
__syncthreads();
```

Then we need to sync the threads to make sure the calculation will be evaluated correctly. After which we loop though this temp array with the index, size, going from the amount of threads per block down to zero, each time halving the index and adding the result to the resulting array.

```
for(int size = ThreadPerBlock/2; size > 0; size /=2){
if(threadIdx.x < size){
temp[threadIdx.x] += temp[threadIdx.x+size];
__syncthreads();
}
}
```

In this loop we check first if the thread index is inside the array length and add the corresponding value to the original value. This keeps repeating, each time halving the size variable in the loop until one is left with only the first entry in the array. After this a final call of `__syncthreads()` is used and we access the first entry of the resulting array. This first entry will contain the sum of the array which is accessed via:

```
*result = temp[0];
```

Note that after each addition one needs to sync the threads via the command

```
__syncthreads();
```

else the end sum will be incorrect.

The parallel reduction sum discussed above is for a single sum. In our model we have double sums. While the overall method is similar there are a few changes between the single and double parallel reduction sums.

In a normal reduction sum over the index $i < n$, we need to first sum up elements in a temp array of size blockDim.x. That is why we do a for loop over i =threadIdx.x increasing by blockDim.x for $i < n$, as shown above. However we require a double parallel reduction sum over the indices $i < n$ and $j < n$. However since CUDA works best with single arrays, to reduce the 2-D array, first we need to flatten it to one dimension. This flattened array will have the size of n^2.

```
for(int i = 0; i < n; i++){
for(int j = 0; j < n; j++){
B[i*n+j]=A[i][j];
}
}
```

i.e. the 2-D array A was flattened to the 1-D array B. Through this flattening process, all the data of A gets copied over to B. This is done through the fact that the size of B (the flattened array) is now greater in a single dimension than before.

We can now do the reduction sum on this new flattened B array.

```
    extern __shared__ double temp[];

  temp[blockDim.y*threadIdx.x+threadIdx.y] = 0.0;
  __syncthreads();

  for(int size = blockDim.x/2; size > 0; size /= 2){
      if(threadIdx.x < size){
        temp[blockDim.y*threadIdx.x+threadIdx.y] +=
        temp[blockDim.y*(threadIdx.x+size)+threadIdx.y];
      }
      __syncthreads();
}
```

The sum above is over the x dimension of the block, which effectively sums over each column of the array, putting the sum on the zeroth index of each column. Then the sum below (over

threadIdx.y) sums the columns together.

```
for(int size = blockDim.y/2; size > 0; size /=2){
    if(threadIdx.y < size){
        temp[threadIdx.y] += temp[threadIdx.y + size];
    }
    __syncthreads();
}
}

if(threadIdx.x == 0 && threadIdx.y == 0){
    *result = temp[0];
}
}
```

Here we only have one block. However in our case we need to many sums so we need to call on one block per sum. Therefore the result variable will now be an array of the total number of blocks, with an extra index.

3.4 Runge-Kutta Methods

The Runge-Kutta methods are a family of algorithms used for solving first order differential equations with an initial condition. The differential equations must be first order for these algorithms to work. If one has an n^{th} order differential equation, one must first convert it into a system of first order differential equations before applying these algorithms.

In order to solve differential equations numerically, one has to first discretize the differential equations into finite time steps. This can be done in many ways, each of which gives a different algorithm.

In general the Runge-Kutta p method has p steps in it. The most famous and widely used is the RK4 method. Some others that exist are the Euler method (RK1), the Runge-Kutta 2 method (RK2) and the Runge-Kutta 3 method (RK3). Higher step methods also exist; however, once passed the fourth step, the accuracy of the algorithms decrease.

As mentioned, the most widely used method is the RK4 method, since it has a fine balance between accuracy and time cost. The algorithm is as follows. Given a first order ordinary differential equation $\frac{dy(t)}{dt} = f(t, y(t))$ with an initial condition (t_0, y_0), the RK4 method is a four step method

defined as

1: **Given:** t_i, y_i, step size h
2: **Compute:**
3: $k_1 = f(t_i, y_i)$
4: $k_2 = f\left(t_i + \frac{h}{2}, y_i + \frac{hk_1}{2}\right)$
5: $k_3 = f\left(t_i + \frac{h}{2}, y_i + \frac{hk_2}{2}\right)$
6: $k_4 = f(t_i + h, y_i + hk_3)$
7: $y_{i+1} = y_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
8: $t_{i+1} = t_i + h$
9: **return** y_{i+1}, t_{i+1}

where i is the iteration number and h is the step-size. To derive this algorithm we Taylor expand $y(t_{i+1})$ to fourth order and for the derivatives of y with respect to t we use the original ordinary first order differential equation with implicit differentiation. Next we Taylor expand $y_{i+1} = y_i + w_1k_1 + w_2k_2 + w_3k_3 + w_4k_4$ about k where the k_i's are given as above. Finally we equate the two expansions by matching the coefficients [24]. For an explicit derivation of the RK4 please see B.3.

The steps 3.4 is the Runge-Kutta 4 algorithm for numerically solving first order ordinary differential equations with step-size h. The smaller the step-size, the more accurate the algorithm is; however, the trade-off is that it will take longer to run the algorithm. If one makes the step size too small, there can be numerical instability. Also a single step of the approximation of $y(t)$ is of order $\mathcal{O}(h^5)$ and the total error is $\mathcal{O}(h^4)$.

The steps 3.4 assumed that at each step one has a single number, however in our case we have an array of numbers at each time step (i.e. $A[l]$ and $B[l]$). The algorithm outlined above still works; however, we have to make a separate function that the RK4 function called at each intermediate step. This function takes in an array and an array of step-sizes and adds them together element-wise as outlined in the RK4 algorithm. The result, instead of being a number that is the value at the next time-step, is an array of numbers that are the multiple values, one for each i, at the next time-step. When there are multiple variables (functions of time), one evaluates each stage of 3.4 for all the variables in sequence.

Modifiying the steps 3.4 to a system of two first order differential equations can be easily extended as follows. For example, a system of two first order differential equations,

$$\frac{dx}{dt} = f(t, x(t), y(t))$$

$$\frac{dy}{dt} = g(t, x(t), y(t))$$

with, now, two initial conditions, (t_0, x_0) and (t_0, y_0), the RK4 method here is defined as

1: **Given:** t_i, x_i, y_i, step size h
2: **Compute:**
3: $k_1 = f(t_i, x_i, y_i)$
4: $l_1 = g(t_i, x_i, y_i)$
5: $k_2 = f\left(t_i + \frac{h}{2}, x_i + \frac{hk_1}{2}, y_i + \frac{hl_1}{2}\right)$
6: $l_2 = g\left(t_i + \frac{h}{2}, x_i + \frac{hk_1}{2}, y_i + \frac{hl_1}{2}\right)$
7: $k_3 = f\left(t_i + \frac{h}{2}, x_i + \frac{hk_2}{2}, y_i + \frac{hl_2}{2}\right)$
8: $l_3 = g\left(t_i + \frac{h}{2}, x_i + \frac{hk_2}{2}, y_i + \frac{hl_2}{2}\right)$
9: $k_4 = f(t_i + h, x_i + hk_3, y_i + hl_3)$
10: $l_4 = g(t_i + h, x_i + hk_3, y_i + hl_3)$
11: $x_{i+1} = x_i + \frac{h}{6}(k_1 + 2k_2 + 2k_3 + k_4)$
12: $y_{i+1} = y_i + \frac{h}{6}(l_1 + 2l_2 + 2l_3 + l_4)$
13: $t_{i+1} = t_i + h$
14: **return** $x_{i+1}, y_{i+1}, t_{i+1}$

where i is the iteration number. We have two functions of arrays (i.e. differential equations for each variable $A[i]$ and $B[i]$. We can evaluate all the differential equations for $A[i]$ in parallel and then the same for $B[i]$.

Chapter 4

Implementation

4.1 Hardware

To implement this parallel code the software CUDA was used on the GPU. CUDA is a parallel computing platform and programming model developed by NVIDIA released in 2007. CUDA, which stands for Compute Unified Device Architecture, was made for general computing in GPUs. The CUDA programs were compiled and run on the Graham computer at the Digital Research Alliance of Canada, which had CUDA since June 2017 [25]. The GPU that was used was NVIDIA v:100 Volta GPU, which is a general purpose NVIDIA GPU card. We used CUDA version 12.2.

4.2 Makefile

Some CPU functions that we require are more stable using a C++ compiler rather than a CUDA compiler. To get the benefits of the C++ compiler while still using CUDA code, one needs to use a makefile in order to use both compilers.

Normally, to compile a program that has only one file, one can simply compile that file inside the terminal. However if the project is split into multiple files, compiling a single file will result in errors since the composite files depend on one another. To successfully compile the program one must compile and link the files in a particular way.

This is where a makefile comes in. A makefile is a text file that contains instructions for how to compile and link (or build) a set of related source code files. It defines a set of tasks to be executed. In other words, it defines the dependencies between files and specifies the commands to build the final executable. A program (often called a make program) reads the makefile and

invokes a compiler, linker, and possibly other programs to make an executable file. To compile the program one simply opens the directory that the files are contained and runs the makefile with the command `make`.

This command tells the processor to read the makefile and execute the instructions to compile and link the files. If there are no errors, an executable file is created which is run using the command `./<name of executable>`.

To view the full makefile please see appendix A.7.

4.3 Reading Coefficients from File

As said, the main equations governing the time-dependent variables $A_i(t)$ and $B_i(t)$ are given by equations 2.6 and 2.7 which include coefficients labeled T_i, R_{il} and S_{ijkl}. These coefficients are defined as integrals over products of the eigenmodes 2.3 [14, 15]; we use the coefficients as calculated numerically by [16]. The coefficents are stored in compressed binary files; for the S coefficients, the coefficients that are zero because they violate the conditions listed in section 3.2.2 are not stored in the binary file. This reduced the file size a great amount, which was efficient.

To be able to read the coefficients in the binary file it was necessary to first understand how the data was stored. Simply put, the T_i is a one indexed array and the data file stored them as the index i value, which is an integer followed by the corresponding T_i value, which is a double, and then moves onto the next index and value, repeating until the end of the file. The R_{il} is a two indexed array and the data file stored them as the index i value, which is an integer, then the index l value, which is an integer, then the corresponding R_{il} value, which is a double, and then moves onto the next two indices and value, repeating until the end of the file. Lastly the S_{ijkl} is a four indexed array and the data file stored them as the index i value, which is an integer then the index j value, which is an integer, then the index k value, which is an integer, then the index l value, which is an integer, then the corresponding S_{ijkl} value, which is a double, and finally moves onto the next four indices and value, repeating until the end of the file. To be able to read in this data file, we had to use the commands fopen(), fread() and fclose(). This is accomplished serially in the C++ file of the Makefile. See Appendix A.1 for code. For example, to read in the T array we use the following code,

```
void read_in_coefficients_T4(int n, double* Tval){
    // T is a one indexed array where the index is read in first
```

```
    FILE* Coefficients = fopen("AdS4_T_j400.bin", "rb");

    int i;

    double dummy;
    size_t err;

    while(true){
        err = fread(&i,sizeof(int),1,Coefficients);
        if(err != 1 and feof(Coefficients)){
            break;
        }

        if(i < n){
            err = fread(&(Tval[i]),sizeof(double),1,Coefficients);
        } else {
           err = fread(&(dummy),sizeof(double),1,Coefficients);
        }
        if(err != 1 && feof(Coefficients)){
            break;
        }
    }
    fclose(Coefficients);
}
```

We use similar code for the R and S coefficients.

The if statements involving the 'err' term exit the loop at the end of the file. Furthermore the statement `if(i < n)` checks if the index is less than the maximum value considered, and if so, saves the value to the T array, otherwise it saves the value to a 'dummy' index, which we do not keep.

Due to CUDA using pointers, we had to flatten the two indexed R array into one index. For the S array since the zero coefficients were not stored, we had to first determine at which position in the S contained non trivial values. This was done in section 3.2.2 where a function was written that took in i, j, l and n and returned the corresponding position. This the the special flattened S array from section 3.2.2. To view how the position index was done, see A.6.

4.4 Runge-Kutta 4 Implementation

The Runge-Kutta 4 algorithm requires the evaluation of the right hand side of the differential equations 2.6 and 2.7. The `__global__` functions made are called so as to evaluate the sums on the right hand side of the differential equations for each variable $A[l]$ and $B[l]$. The actual Runge-Kutta 4 algorithm is done on the CPU which repeatedly calls the `__global__reduced_sum_A` and `__global__ reduced_sum_B`. These global functions were inside the wrapper functions `Array_sum_A_GPU` and `Array_sum_B_GPU`.

In this snippet of the RK4 function, we call `Array_sum_A_GPU` and `Array_sum_B_GPU` with the dimension of the grid set to the array size and the dimension of the block set to 256. Moreover, the array sums uses 16×16 blocks and the `add_array` function use 1D blocks with `blockDim.x` equal to 512.

```
void RK_4(int array_size_x, double h, double* A_GPU, double* B_GPU,
double* A_GPU_new, double* B_GPU_new, double* divAS, double* divBS,
 double* divBR, double* divBT, double* Tvals, double* Rvals,
double* Svals, double* k1, double* k2, double* k3, double* k4,
 double* l1, double* l2, double* l3, double* l4){
    Array_sum_A_GPU(array_size_x,A_GPU,B_GPU,divAS,Tvals,Rvals,Svals,k1);
    cudaDeviceSynchronize();
    Array_sum_B_GPU(array_size_x,A_GPU,B_GPU,divBS,divBR,divBT,Tvals,
Rvals,Svals,l1);
    cudaDeviceSynchronize();

    Add_Array(array_size_x,1.0,A_GPU,0.5,k1,A_GPU_new);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU,0.5,l1,B_GPU_new);
    cudaDeviceSynchronize();
```

We pass in the `Avals` and `Bvals` arrays as well as the three coefficients, T_i, R_{il} and S_{ijkl}. Since `Array_sum_A_GPU` and `Array_sum_B_GPU` are global CUDA functions the last variable called in those functions are the four steps of the RK4 algorithm (k_1, k_2, k_3, k_4 for the A equation and l_1, l_2, l_3, l_4 for the B equation).

We had to create functions that took care of the division in the two global GPU functions. These had to be done seperately since if we combined them in the main function, the divisions would not work properly and would give an incorrect result. This is due to how threads access

memory and performance issues. Division is more computationally expensive than multiplication and many CUDA devices take longer time to preform division. Due to this the thread that does the division can encounter memory contention or conflicts. By writing a separate function for the division, it is ensured that the threads are properly synchronized.

After each time we call `Array_sum_A_GPU` and `Array_sum_B_GPU` together, we would then call the add array function which is defined as

```
__global__ void add_array(int size, double mult_factor1, double* array_1,
double mult_factor2, double* array_2, double* result){

int k = threadIdx.x;
if(k < size){
result[k] = mult_factor1*array_1[k] + mult_factor2*array_2[k];
}
}
```

Here we add the array of the steps in the RK4 obtained to the `Avals` and `Bvals` arrays, which then are returned and used in the next iteration. Since different steps had different coefficients, an extra input variable in the add array function was made (scale) which varied depending on the algorithm step. In the add array global function the grid dimension used was 1 and the dimension of the block used was 512. Lastly, after calling the A and B sums each four times (the four RK4 steps) and calling the add array function respectively, the add array function was called eight more times. Four for `Avals` where we combined the k_i's and the $A[l]$ to form the next iterative $A[l]$, and four for the `Bvals` where we combined the l_i's and the $B[l]$ to form the next iterative $B[l]$ according to the last step of the RK4 algorithm. This RK4 CPU function was called in `main` where it was looped through each time from $t = 0$ to $t = t_{\text{end}}$, updating the A and B arrays for each time step. Lastly, when printing out the data both the t loop and the i loop were utilized where inside the nested loops the arrays tvals, Avals and Bvals were printed at each i for each time step. To make sure the calculations are correct a call to `cudaDeviceSynchronize()` was used in the `for loops` after each call to the RK4 function. This call avoids timing mismatches between the CPU requesting the result and the GPU finishing the computation. `CudaDeviceSynchronize()` makes the host (the CPU) wait until the device (the GPU) have finished executing all the threads it has started, and thus the program will continue as if it was a normal sequential program.

To view the full RK4 code along with the A and B functions, please see appendices A.3 and A.4.

4.5 Parallel Reduction revisited

The parallel reduction method outlined above applied when one does not have any restrictions on the array. In our case (equations 2.6 and 2.7) we have certain restrictions, multiple arrays and multiple indices on these arrays. For this parallel reduction, we require a temp variable with `extern` and `shared` keywords which had shared memory size. The restrictions can be dealt with by using a couple of if statements in the first for loop.

```
if(l < size_x){
    for(int i = threadIdx.x; i < size_x; i += blockDim.x){
    for(int j = threadIdx.y; j < size_x; j += blockDim.y){
    int diff = i+j-l;
    if(i != l && j != l && diff >= 0 && diff < size_x){
```

Given that the S array had values that were dependent on four indices, it was required to find the correct position for the giving i, j. This was done by clever manipulations of sums. This was needed since the S array had conditions (given by 3) which gave non-trivial values. Once it was found, these positions where stored in a variable. See section 3.2.2 and B.1 for details. To find the position variable we used a copy of the position function code within the sum. The S array was loaded into the temp variable via,

```
temp[blockDim.y*threadIdx.x+threadIdx.y] += -divS[l]*Svals[position]
*A[i]*A[j]*A[diff]*sin(B[l]+B[diff]-B[i]-B[j]);
```

for the A sum. The B sum was similar except for a different division as well as the sine was changed into a cosine.

The R array was to be dealt with next. Since this array was already flattened from when reading in the co-oefficents we simply apply the parallel sum algorithm. For parallel reduction, we need a temp variable with extern and shared keywords with shared memory size.

```
for(int j = threadIdx.y; j < size_x; j += blockDim.y){
if(j != l){
temp[threadIdx.y] += -divR[l]*Rvals[j*size_x + l]*A[j]*A[j];
}
}
__syncthreads();
```

for $B[l]$. $A[l]$ did not have an R term and thus was not needed in the $A[l]$ function. This R sum is only a single sum and hence we only need one dimension of the thread and block index.

The double parallel reduction sum, that was first discussed in 3.3, took the form,

```
if(l < size_x){
for(int size = blockDim.x/2; size > 0; size /= 2){
if(threadIdx.x < size){
temp[blockDim.y*threadIdx.x+threadIdx.y] +=
temp[blockDim.y*(threadIdx.x+size)+threadIdx.y];
}
__syncthreads();
}
for(int size = blockDim.y/2; size > 0; size /=2){
if(threadIdx.y < size){
temp[threadIdx.y] += temp[threadIdx.y + size];
}
__syncthreads();
}
}
```

There was a single sum for each block (i.e. each l value).

```
if(l < size_x){
if(threadIdx.x == 0 && threadIdx.y == 0){
result[blockIdx.x] = temp[0];
}
}
```

for A, and

```
if(l < size_x){
if(threadIdx.x == 0 && threadIdx.y == 0){
result[blockIdx.x] = temp[0] - divT[l]*Tvals[l]*A[l]*A[l];
}
}
```

for B.

Because we compiled using a makefile we had to implement a wrapper function to call the sums for the A and B differential equations.

```
void Array_sum_A_GPU(int size_x, double* A, double* B, double* dividerS,
```

```
double* Tvals, double* Rvals, double* Svals, double* sum){

    int ThreadPerBlock = 16;
    dim3 dimBlock(ThreadPerBlock, ThreadPerBlock, 1);
    dim3 dimGrid(size_x, 1, 1);

    divAS<<<dimGrid,1>>>(size_x, dividerS);

    reduced_sum_A<<<dimGrid, dimBlock, ThreadPerBlock*ThreadPerBlock*
sizeof(double)>>>(size_x, A, B, dividerS, Tvals, Rvals, Svals, sum);
}
```

and similarly for the B and `add_array` functions.

These wrapper functions are called in the RK4 function of the C++ code. Note that the functions `divAS, divBS, divBR` and `divBT` took care of the divisions.

4.6 Managing Data

After each time step, we write the data to two files to be analyzed later, one file for the A variables and one for the B variables. Remember that at each time step we have lists of numbers instead of a single number. After writing to the files we can analyze it using python. To write to a file we use the ofstream library of C++. This is done as

```
    ofstream RK4_Afunc_Data;

    RK4_Afunc_Data.open("RK4_Afunc_Data.txt",ios::app);
    RK4_Afunc_Data << (t+1)*h << ", ";

    for(int i = 0; i < array_size_x; i++){
      RK4_Afunc_Data << A_GPU[i];
       if(i != array_size_x - 1){
        RK4_Afunc_Data << ", ";
    }
    }
```

```
RK4_Afunc_Data << "\n";

RK4_Afunc_Data.close();
```

and similarly for B. This is done inside the loop over time.

Note that we create and open two separate files, one for the A values and one for the B values. The formatting is in the form time, $A[0], A[1], ...$ and similarly for the B values. For the full code please see A.5.

Chapter 5

Results

5.1 Testing the Program

The differential equations 2.6 and 2.7 have double sums on the right hand side. Before programming the RK4, parallel and sequential code was written that calculated these sums. These sums were compared and was found that the relative difference between the GPU and the CPU was lesss than than 10^{-11}, which shows that the reduction CUDA sum was working.

Now that the sums matched the sequential part of the code, the RK4 algorithm was implemented on the CPU. Our main program, which uses the benefits of CUDA and parallel programming, should excel at preforming the calculations quicker and to more accuracy. Nonetheless, it still requires substantial computing time and power. In light of this, it would be appropriate to test the code against known solutions and the sequential version. Due to this we rewrote the RK4 algorithm in C++ sequentially, which included the A and B function declarations, the coefficient arrays and the RK4 algorithm. Once it ran the sequential code, like the parallel version, produced two text files, one for the A values and one for the B values.

To compare the sequential code with the parallel code a python file was created which read in the two files and stored the data values in the corresponding lists.

Due to the fact that many files needed to be compared, the analysis file was made such that when running it on the terminal, it required an additional two arguments which were the two text files with data, that was produced from the main RK4 code. These two text files were then analyzed.

Once it ran, this python program stored the values in four lists. Two lists were the times and respective values for one file and the other two lists were the times and respective values for the second file. For each line in the data file, the first value gives the time and the rest of the values on that line give the A or B values for that time. It then called the analysis function which took in

those four lists and compared them. Since the times where the same, there was no need to compare the times.

In the analysis function a zip function was used to unpack the four lists. Then it calculated the relative difference between the values corresponding to the A and B array values. From here it found the minimum and maximum values as well as their positions in the original array. An important quantity that was calculated was the root mean square value of the relative differences. The root mean square value of the relative differences here quantifies how much the sequential results differ from the parallel results. So a low RMS is desirable since we want the values in both the CPU and the GPU to match.

A dictionary (i.e. a map) was then created to store these results for each time step. The results, minimum value, maximum values, relative differences with their locations and the RMS was finally printed into a text file for each time, from which one could analyze it. To see the python code please see A.8.

This analysis was done for both AdS4 and AdS5 space-times with 16×16 2-D blocks for the sums and 1-D block for the add array function, which simply added to arrays in parallel. The parallel code took seconds to run for both space-times, the sequential code took longer to run totaling approximately one hour. This shows that the GPU calculations indeed does run faster than the sequential counterpart given the same data and length of data values. The relative errors between the sequential code and the parallel code for each time was much less than one, meaning that the data for both sequential and parallel code agrees and thus our GPU code works.

Our initial conditions for the AdS4 space-time was $A[l] = 10^{-6}/(3 + 2l)$ for $l \geq 2$, $A[0] = \frac{1}{3}$, $A[1] = \frac{1}{5}$ and $B[l] = 0$. This was done with an array size of 100, for 10^3 time steps and with the step-size being 10^{-6}. The minimum (most negative) relative difference was found for $l = 98$ at time $t = 0.000751$, when there was a RMS difference of 2.73. The maximum relative difference was found at $l = 98$ at time $t = 0.000861$ with a value of 0.982, where there was a RMS difference of 0.33. The fact that the RMS differences are much smaller in magnitude than the minimum and maximum relative differences shows that the relative differences must be much smaller for the other l values.

Our initial conditions for the AdS5 space-time was $A[l] = 10^{-20}$ for $l \geq 2$, $A[0] = \frac{1}{4}$, $A[1] = \frac{1}{6}$ and $B[l] = 0$. This was done with an array size of 100, for 10^7 time steps and with the step-size being 10^{-6}. For the AdS5 space-time the sequential code was compared to the parallel code and was found that the relative differences was of the order 10^{-2} with the minimum relative difference of around -344.23 at $t = 0.000341$, which has a root mean squared value of about 34.42 located at $l = 98$ and a maximum relative difference of around 11.28 at $t = 0.000801$, which has a root mean

squared value of about 1.15 located at $l = 97$. The root mean squared values fluctuate quickly throughout time.

5.2 Analysis

Now that our main program is complete and working, all that is left is to run it on the GPU with various initial conditions and analyze the results. When running the Runge-Kutta algorithm on the GPU we have the freedom to vary five parameters. These parameters are the array size (n, which corresponds to the number of data points used, i.e. i_{max}), the max time (amount of time to run the program), h (the step-size of the RK4 algorithm), and the initial conditions for the A and B arrays. These values are set before the main loop of the algorithm. Note that because our values are arrays, we must provide initial conditions for the entire array of size `array_size_x`. These conditions are hard-coded into the program.

After this we run the program on the GPU and it produces two text files with data. The two text files are for the two (array) variables, A and B. Via the way the output was programmed, the text files produces data on each line, each line representing all the values for the given array at one time step. The time is given as the first element of each line when we imported the data into the python script. For each line in the data file, the spaces were removed using the `.strip()` command and the values were put into two lists using the comma as the separation via the `.split(',')` command. The first list contained all the time values, while the second list contained the values of A_l. This was done for both AdS4 and AdS5 space-times, which means we have four arrays. Two for AdS4 and two for AdS5.

At each time, we fit A_l to the curve described in section 2.2, that is $A_l = \alpha l^{-\beta} e^{-\gamma l}$ where α, β and γ are the fit parameters. This is the fit used since we expect the coefficients to decay exponentially for smooth solution, but to become algebraic if the system develops a singularity. γ measures the distance from the real axis to the closest singularity in the complex plane. If γ goes to zero, the system develops a singularity and coincides with the black hole formation in general relativity. Since the first value of each line corresponded to $l = 0$ and the fit had negative l powers to avoid division by zero when calling scipy's curve fit function we had to start at index one as follows:

```
parameters_A, covariance_A = curve_fit(fit, l_vals[1:], A_vals[1:])
```

The important information (the numerical values for the three fit parameters) were stored in the `parameters_A` variable. The `covariance_A` variable stored the error which was not used.

Since the behavior of the amplitudes of A_l determines if the scalar field is getting concentrated only the time evolution of A_l needed to be analyzed. No fit was required for B_l.[1]

In this simulation, two important quantities are conserved by the differential equations 2.6 and 2.7, the energy at each time and the number of exitations at each time. The energy of the system, at each time, is defined as $E_T = \sum_l \omega_l^2 |A_l|^2$ and the number of exitations of the system, at each time, is defined as $N_T = \sum_l \omega_l |A_l|^2$. Therefore, whether these quantities are constant over time in our output is an important consistency check of our code.

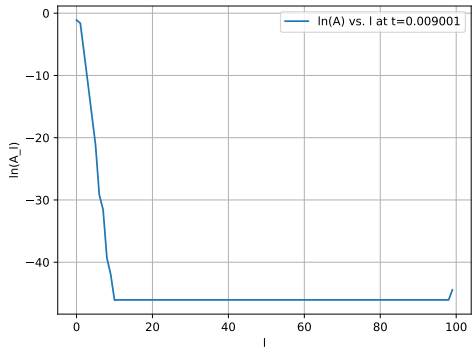
This data was analyzed in python. The python script was written in which we looped over each time and summed the energy and number of exitations. The summation was done via a simple for loop where the loop went over all the array values of A_l. That is, for each time t, we looped and summed the values of A_l to form the energy and number of exitations. After this we simply printed the total energy and total excitation for each time, t. At the end we simply printed the energy and number of exitations at each time into the terminal. To view the full python files see A.8.

5.3 Results and Plots

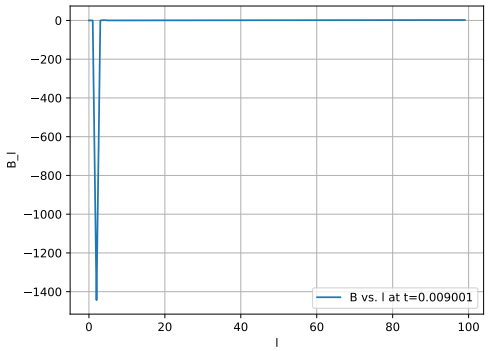
Now that we have the text files for $A[l]$ and $B[l]$ for both space-times, we can now plot and analyze them where t was the first value in each line. Due to the size of the A values for the A vs. l plots we plotted the log of A vs. l. Four plots were made at the specific time-steps $t = 10$, $t = 50$ and $t = 140$. We did two calculations for the AdS4 with two initial conditions, where the first initial condition puts equal amounts of energy into $A[0]$ and $A[1]$. As time evolves the error located at $l = 98$ spread, giving NANs by the end of the simulation, which resulted in the mode number and energy number to be zero, signifying data was crashed, as can be seen in 5.7a and 5.7b. These NANs seems to be coming from the GPU itself rather than the code because the parallel code was verified against the sequential code with high degree of accuracy. This GPU error made the numerical calculations differ from the sequential part and as time went on this error magnified, eventually crashing. This happened for all four simulations which resulted in us unable to do any curve fitting.

For the AdS4 space-time, A and B plots for $t = 0.009001$ are given in figure 5.1, for $t = 0.499001$ are given in figure 5.2 and for $t = 0.999001$ are given in figure 5.3. This was done with array size 100, 10^7 time steps (one time step is one loop in the RK4 algorithm, which corresponds to a time of $t =$ time step $* h$), $h = 10^{-6}$ with initial conditions $A[0] = \frac{1}{3}$, $A[1] = \frac{1}{5}$, $A[l] = 10^{-20}$

[1] Since the two ordinary differential equations were coupled, one could not get away with ignoring the B_l variable entirely.



(a) How the log of A values vary with l in AdS4 space-time.



(b) How the B values vary with l in AdS4 space-time.

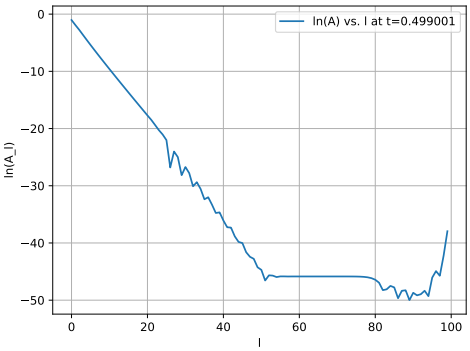
Figure 5.1: Plots showing the variation of $\log(A)$ with l and B with l in AdS4 space-time at $t = 0.009001$.

for $l \geq 2$ and $B[l] = 0$ for all l, to make the energy mode small. These initial conditions have equal energies in the $l = 0$ and $l = 1$ modes and approximately zero in the other modes. The figures show an increase in the A_l values for larger l as time passes.

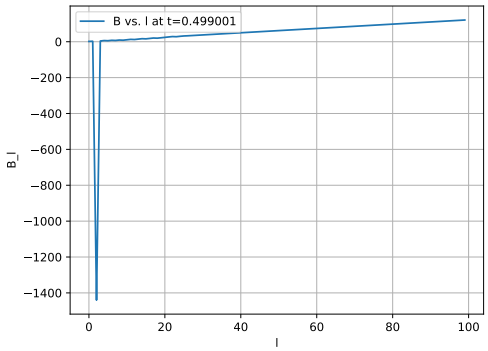
We can understand this increase quantitatively by fitting $A[l]$ for $l \in [10, 40]$. At time $t = 0.499001$, the fit is $A_l = 0.053l^{-1.33}e^{-0.958l}$. For time $t = 0.999001$, the fit is $A_l = 0.25l^{0.692}e^{-0.253l}$. The coefficient in the exponent gets closer to zero as time passes, which indicates progress towards forming a black hole. However, we also note an increase in A_l near mode $l = 98$, indicating an error due to the GPU. As time goes on the differential equation causes this error to spread, which eventually causes the simulation to crash.

We also preformed calculations for AdS5. We chose similar initial conditions with equal energy in the $l = 0$ and $l = 1$ modes. These initial conditions were $A[0] = \frac{1}{4}$, $A[1] = \frac{1}{6}$, $A[l] = 10^{-20}$ for $l \geq 2$ and $B[l] = 0$ for all l. A and B plots for $t = 0.009001$ are shown in figure 5.4, figure 5.5 are shows for $t = 0.149001$ and for time $t = 0.299001$ the plots are given in 5.6. These plots show that the result is similar to AdS4 in that for higher l values the curve gets more horizontal indicating the formation of a black hole. However similar to AdS4, the error in the GPU ultimately caused the simulation to diverge and crash.

However in both space-times the conservation of energy and mode number throughout the simulation seemed to be conserved before the numerical instability. As seen below in figure 5.7, both energy and mode number are nearly constant throughout our AdS4 calculations until late times. As the numerical instability, due to the GPU, increases the conservation of the energy and



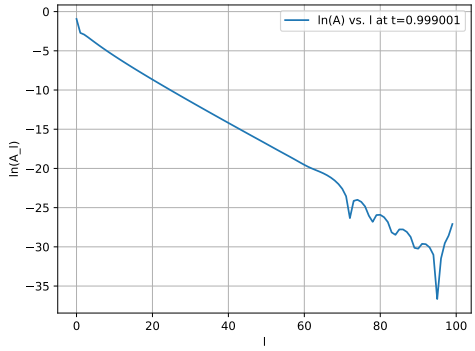
(a) How the log of A values vary with l in AdS4 space-time.



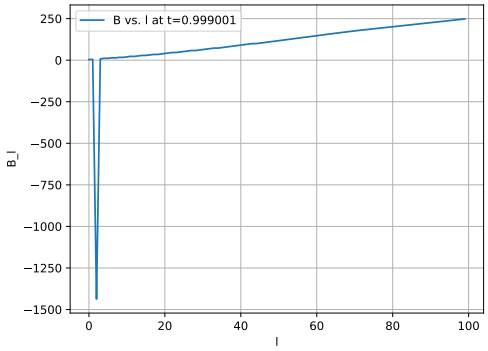
(b) How the B values vary with l in AdS4 space-time.

Figure 5.2: Plots showing the variation of $\log(A)$ with l and B with l at $t = 0.499001$ in AdS4 space-time.

mode number fails. In the figures the curve spikes upward indicating the growth of the error, eventually crashing to zero. The zero value is due to fact that the calculation has failed completely and is yielding NaNs.

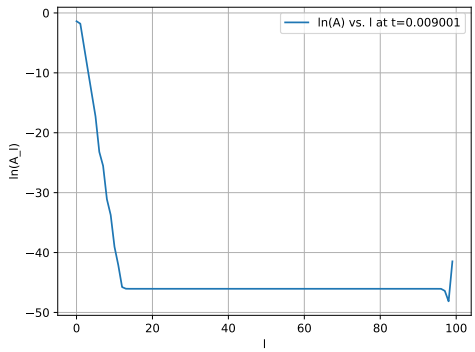


(a) How the log of A values vary with l at $t = 0.999001$ in AdS4 space-time.

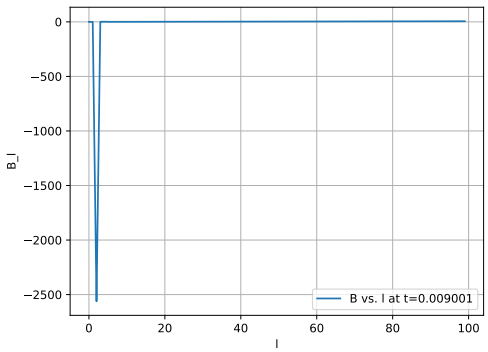


(b) How the B values vary with l at $t = 0.999001$ in AdS4 space-time.

Figure 5.3: Plots showing the variation of $\log(A)$ with l and B with l at time step 140 in AdS4 space-time.

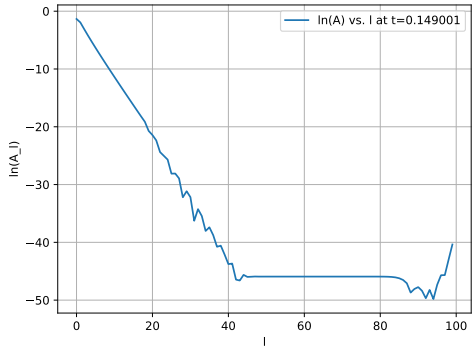


(a) How the log of A values vary with l in AdS5 space-time.

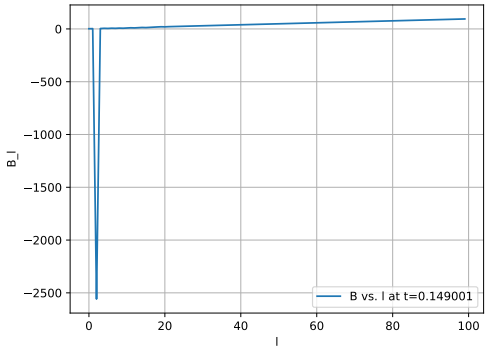


(b) How the B values vary with l in AdS4 space-time.

Figure 5.4: Plots showing the variation of $\log(A)$ with l and B with l in AdS5 space-time at $t = 0.009001$.

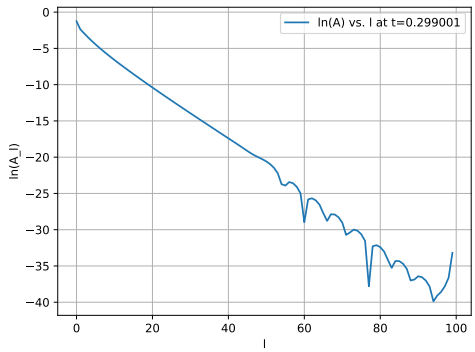


(a) How the log of A values vary with l in AdS5 space-time.

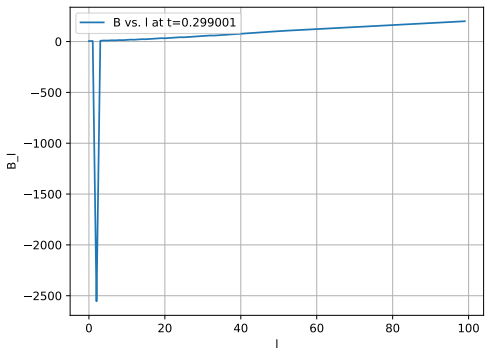


(b) How the B values vary with l in AdS5 space-time.

Figure 5.5: Plots showing the variation of $\log(A)$ with l and B with l at $t = 0.149001$ in AdS5 space-time.

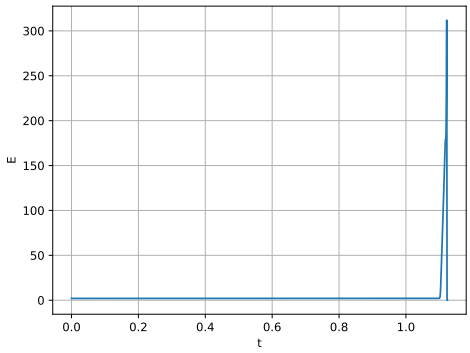


(a) How the log of A values vary with l at $t = 0.299001$ in AdS5 space-time.

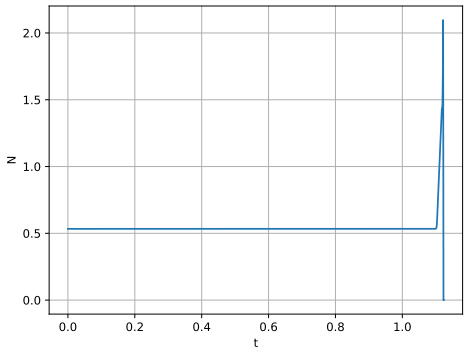


(b) How the B values vary with l at $t = 0.299001$ in AdS5 space-time.

Figure 5.6: Plots showing the variation of $\log(A)$ with l and B with l at time step 140 in AdS5 space-time.



(a) Conservation of energy in AdS4 space-time



(b) Conservation of mode number in AdS4 space-time

Figure 5.7: Conservation of energy and mode number for AdS4

Chapter 6

Conclusion

The problem of gravitational collapse in AdS4 and AdS5 space-times has been a fascinating study in the recent years. Specifically to scalar field collapse it was shown that the two governing equations for the perturbation version of the Klein-Gordon equation produced, 2.6 and 2.7. To solve these equations one must resort to numerical methods. Previously this was solved parallely on the CPU. Here, we solved the same problem on the GPU, showing that the GPU sped up the calculations enormously and agreed with the sequential counterpart except at high l values. The speed up was significant, the CPU calculations took hours while the same calculations on the GPU took minutes. The solutions to these differential equations, namely, A_l and B_l were solved as a function of the mode number l. These were shown in section 5.3 along with the conservation of energy and mode number, which were conserved.

Some future work might include extending the parallel computing to higher thread per block count and more data values (larger array size). The divergence at high l values need to be analyzed and understood so that it can be corrected. Lastly, the numerical integration method used was the classic RK4 method, which while strong could be substituted to a more sophisticated method.

A Code

A.1 Read in Coefficient

```
void read_in_coefficients_T4_CPU(int n, double* Tval){
    // T is a one indexed array where the index is read in first

    FILE* Coefficients = fopen("AdS4_T_j400.bin", "rb");

    int i;

    double dummy;
    size_t err;

    while(true){
        err = fread(&i,sizeof(int),1,Coefficients);
        if(err != 1 and feof(Coefficients)){
            break;
        }

        if(i < n){
            err = fread(&(Tval[i]),sizeof(double),1,Coefficients);
        } else {
           err = fread(&(dummy),sizeof(double),1,Coefficients);
        }
        if(err != 1 && feof(Coefficients)){
            break;
        }
    }
    fclose(Coefficients);
}

 void read_in_coefficients_R4_CPU(int n, double* Rval){
     // R is a two indexed array where the two indices are read in first
```

```
    FILE *Coefficients = fopen("AdS4_R_j400.bin", "rb");

    int i;
    int j;

    double dummy;
    size_t err;

    while(true){
        err = fread(&i,sizeof(int),1,Coefficients);
        if(err != 1 && feof(Coefficients)){
            break;
        }
        err = fread(&j,sizeof(int),1,Coefficients);
        if(err != 1 && feof(Coefficients)){
            break;
        }

        if(i < n && j < n){
            err = fread(&(Rval[i*n+j]),sizeof(double),1,Coefficients);
    }
    else{
          err = fread(&(dummy),sizeof(double),1,Coefficients);
        }
        if(err != 1 && feof(Coefficients)){
            break;
        }
    }
    fclose(Coefficients);
}

 void read_in_coefficients_S4_CPU(int n, double* Sval){
     // S is a four indexed array where the four indices are read in first
```

```
FILE *Coefficients = fopen("AdS4_S_j400.bin", "rb");

int i;
int j;
int k;
int l;

double dummy;
size_t err;

while(true){
    err = fread(&i,sizeof(int),1,Coefficients);

    if(err != 1 && feof(Coefficients)){
       break;
    }
    err = fread(&j,sizeof(int),1,Coefficients);

    if(err != 1 && feof(Coefficients)){
      break;
    }
    err = fread(&k,sizeof(int),1,Coefficients);

    if(err != 1 && feof(Coefficients)){
       break;
    }
    err = fread(&l,sizeof(int),1,Coefficients);

        if(err != 1 && feof(Coefficients)){
       break;
    }

    if(i >= 0 && j >= 0 && k >= 0 && l >= 0 && i < n && j < n && k < n
```

```
 && l < n && k+l == i+j && i != l && j != l){
        err = fread(&(Sval[positions_in_S_array_CPU(i,j,l,n)]),
sizeof(double),1,Coefficients);
        }
    else {
          err = fread(&(dummy),sizeof(double),1,Coefficients);
        }
        if(err != 1 && feof(Coefficients)){
           break;
        }
  }
    fclose(Coefficients);
}
```

A.2 Parallel sum

```
extern __shared__ double temp[];

for(int i = threadIdx.x; i < sizeA; i += blockDim.x){
temp[threadIdx.x] += A[i];
}
__syncthreads();
for(int size = ThreadPerBlock/2; size > 0; size /=2){
if(threadIdx.x < size){
temp[threadIdx.x] += temp[threadIdx.x+size];
__syncthreads();
}
}
*result = temp[0];
```

```
  __global__ void reduced_sum_A(int size_x,double* A,double* B,double* divS
  double* Tvals,double* Rvals,double* Svals,double* result){
    extern __shared__ double temp[];
```

```
temp[blockDim.y*threadIdx.x+threadIdx.y] = 0.0;
__syncthreads();
int position;

for(int size = blockDim.x/2; size > 0; size /= 2){
   if(threadIdx.x < size){
     temp[blockDim.y*threadIdx.x+threadIdx.y] +=
     temp[blockDim.y*(threadIdx.x+size)+threadIdx.y];
   }
   __syncthreads();
}
for(int size = blockDim.y/2; size > 0; size /=2){
   if(threadIdx.y < size){
      temp[threadIdx.y] += temp[threadIdx.y + size];
   }
   __syncthreads();
}



if(threadIdx.x == 0 && threadIdx.y == 0){
   result[blockIdx.x] = temp[0];
}
}
   }
   }
   }
}
}
```

A.3 A function and B function declartion

```
__global__ void reduced_sum_A(int size_x,double* A,double* B,double* divS,
 double* Tvals,double* Rvals,double* Svals,double* result){
   extern __shared__ double temp[];
```

```
    int l = blockIdx.x;

    temp[blockDim.y*threadIdx.x+threadIdx.y] = 0.0;
    __syncthreads();
    int position;
    if(l < size_x){
       for(int i = threadIdx.x; i < size_x; i += blockDim.x){
       for(int j = threadIdx.y; j < size_x; j += blockDim.y){
       int diff = i+j-l;
       if(i != l && j != l && diff >= 0 && diff < size_x && i+j-l < size_x
 && i+j-l >= 0 && i != l && j != l){
    if(i < l){
    if(j < l){
    position = (3*pow(i,2)-6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)
+3*pow(l,2)*size_x+3*l*pow(size_x,2)-12*l*size_x+2*l)/6;
    }
    else if(j > l){
    position = (3*pow(i,2)-6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)
+3*pow(l,2)*size_x+3*l*pow(size_x,2)-12*l*size_x+2*l-6)/6;
    }
    }
    if(i > l){
    if(j < l){
    position = (-3*pow(i,2)+6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)
+3*pow(l,2)*size_x-6*pow(l,2)+3*l*pow(size_x,2)-12*l*size_x+2*l
-6*size_x-6)/6;
    }
    else if(j > l){
    position = (-3*pow(i,2)+6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)
+3*pow(l,2)*size_x-6*pow(l,2)+3*l*pow(size_x,2)
-12*l*size_x+2*l-6*size_x)/6;
    }
    }
```

```
temp[blockDim.y*threadIdx.x+threadIdx.y] += -divS[l]*Svals[position]
*A[i]*A[j]*A[diff]*sin(B[l]+B[diff]-B[i]-B[j]);
}
}
}
}
    __syncthreads();

    if(l < size_x){
    for(int size = blockDim.x/2; size > 0; size /=2){
       if(threadIdx.x < size){
          temp[blockDim.y*threadIdx.x+threadIdx.y] +=
 temp[blockDim.y*(threadIdx.x+size)+threadIdx.y];
       }
       __syncthreads();
    }
    for(int size = blockDim.y/2; size > 0; size /=2){
       if(threadIdx.y < size){
          temp[threadIdx.y] += temp[threadIdx.y + size];
       }
       __syncthreads();
    }
    }

    if(l < size_x){
    if(threadIdx.x == 0 && threadIdx.y == 0){
       result[blockIdx.x] = temp[0];
    }
    }
    }


__global__ void divAS(int size_x, double* result){
```

```
    int l = blockIdx.x;

    if(l < size_x){
    result[l] = 1.0/(2*(2.0*l + 3.0));
}
}

__global__ void divBS(int size_x, double* A, double* result){
    int l = blockIdx.x;

    if(l < size_x){
    result[l] = 1.0/(2*A[l]*(2.0*l + 3.0));
}
}

__global__ void divBR(int size_x, double* result){
    int l = blockIdx.x;

    if(l < size_x){
    result[l] = 1.0/(2*(2.0*l + 3.0));
}
}
__global__ void divBT(int size_x, double* result){
    int l = blockIdx.x;

    if(l < size_x){
    result[l] = 1.0/(2*(2.0*l + 3.0));
}
}

__global__ void add_array(int size, double mult_factor1, double* array_1,
 double mult_factor2, double* array_2, double* result){

        int k = threadIdx.x;
```

```
        if(k < size){
        result[k] = mult_factor1*array_1[k] + mult_factor2*array_2[k];
        }
}

__global__ void reduced_sum_B(int size_x,double* A,double* B,double* divS,
double* divR, double* divT,double* Tvals,double* Rvals,double* Svals,
double* result){
    extern __shared__ double temp[];

    int l = blockIdx.x;

    temp[blockDim.y*threadIdx.x+threadIdx.y] = 0.0;
    __syncthreads();
    int position;
    if(l < size_x){
       for(int i = threadIdx.x; i < size_x; i += blockDim.x){
       for(int j = threadIdx.y; j < size_x; j += blockDim.y){
       int diff = i+j-l;
       if(i != l && j != l && diff >= 0 && diff < size_x &&
i+j-l < size_x && i+j-l >= 0 && i != l && j != l){
    if(i < l){
    if(j < l){
    position = (3*pow(i,2)-6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)+
3*pow(l,2)*size_x+3*l*pow(size_x,2)-12*l*size_x+2*l)/6;
    }
    else if(j > l){
    position = (3*pow(i,2)-6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)+
3*pow(l,2)*size_x+3*l*pow(size_x,2)-12*l*size_x+2*l-6)/6;
    }
    }
    if(i > l){
    if(j < l){
    position = (-3*pow(i,2)+6*i*l+6*i*size_x-3*i+6*j-2*pow(l,3)+
```

```
3*pow(l,2)*size_x -6*pow(l,2)+3*l*pow(size_x,2)-12*l*size_x+2*l
-6*size_x -6)/6;
    }
    else if(j > l){
    position = (-3*pow(i,2)+6*i*l+6*i*size_x -3*i+6*j-2*pow(l,3)+
3*pow(l,2)*size_x -6*pow(l,2)+3*l*pow(size_x,2)-12*l*size_x+2*l
-6*size_x )/6;
    }
    }
    if(A[l] != 0){
          temp[blockDim.y*threadIdx.x+threadIdx.y] +=
-divS[l]*Svals[position]*A[i]*A[j]*A[diff]*cos(B[l]+B[diff]-B[i]-B[j]);
       }
}
}
}
    __syncthreads();

    for(int j = threadIdx.y; j < size_x; j += blockDim.y){
    if(j != l){
    temp[threadIdx.y] += -divR[l]*Rvals[j*size_x + l]*A[j]*A[j];
}
}
__syncthreads();

    if(l < size_x){
    for(int size = blockDim.x/2; size > 0; size /= 2){
       if(threadIdx.x < size){
          temp[blockDim.y*threadIdx.x+threadIdx.y] +=
 temp[blockDim.y*(threadIdx.x+size)+threadIdx.y];
       }
       __syncthreads();
    }
    for(int size = blockDim.y/2; size > 0; size /=2){
```

```
        if(threadIdx.y < size){
            temp[threadIdx.y] += temp[threadIdx.y + size];
        }
        __syncthreads();
    }
    }

    if(l < size_x){
    if(threadIdx.x == 0 && threadIdx.y == 0){
        result[blockIdx.x] = temp[0] - divT[l]*Tvals[l]*A[l]*A[l];
    }
    }
    }
}

void Array_sum_A_GPU(int size_x, double* A, double* B,double* dividerS,
double* Tvals,double* Rvals,double* Svals, double* sum){

    int ThreadPerBlock = 16;
    dim3 dimBlock(ThreadPerBlock,ThreadPerBlock,1);
    dim3 dimGrid(size_x,1,1);

    divAS<<<dimGrid,1>>>(size_x,dividerS);

    reduced_sum_A<<<dimGrid,dimBlock,
ThreadPerBlock*ThreadPerBlock*sizeof(double)>>>(size_x,A,B,dividerS,Tvals,
Rvals,Svals,sum);
}

void Array_sum_B_GPU(int size_x, double* A, double* B,double* dividerS,
double* dividerR,double* dividerT,double* Tvals,double* Rvals,
double* Svals, double* sum){

    int ThreadPerBlock = 16;
```

```
    dim3 dimBlock(ThreadPerBlock, ThreadPerBlock, 1);
    dim3 dimGrid(size_x, 1, 1);

    divBS<<<dimGrid,1>>>(size_x, A, dividerS);
    divBR<<<dimGrid,1>>>(size_x, dividerR);
    divBT<<<dimGrid,1>>>(size_x, dividerT);

    reduced_sum_B<<<dimGrid, dimBlock, ThreadPerBlock*ThreadPerBlock
*sizeof(double)>>>(size_x, A, B, dividerS, dividerR,
dividerT, Tvals, Rvals, Svals, sum);
}

void Add_Array(int size_x, double mult_factor1, double* array_1,
 double mult_factor2, double* array_2, double* result){

    int ThreadPerBlock = 16;
    dim3 dimBlock(ThreadPerBlock, ThreadPerBlock, 1);
    dim3 dimGrid(size_x, 1, 1);

    add_array <<<1,512,ThreadPerBlock*sizeof(double)>>>(size_x, mult_factor1,
array_1, mult_factor2, array_2, result);
}
```

A.4 Runga-Kutta 4

```
void RK_4(int array_size_x, double h, double* A_GPU, double* B_GPU,
double* A_GPU_new, double* B_GPU_new, double* divAS, double* divBS,
 double* divBR, double* divBT, double* Tvals, double* Rvals,
double* Svals, double* k1, double* k2, double* k3, double* k4, double* l1,
 double* l2, double* l3, double* l4){
  Array_sum_A_GPU(array_size_x, A_GPU, B_GPU, divAS, Tvals, Rvals, Svals, k1);
 // Gives k1
    cudaDeviceSynchronize();
    Array_sum_B_GPU(array_size_x, A_GPU, B_GPU, divBS, divBR, divBT, Tvals,
```

```
Rvals,Svals,l1); // Gives l1
    cudaDeviceSynchronize();

    Add_Array(array_size_x,1.0,A_GPU,h/2.0,k1,A_GPU_new);
// Adds old A_GPU to h*k1/2 and stores it in A_GPU_new
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU,h/2.0,l1,B_GPU_new);
// Adds old B_GPU to h*l1/2 and stores it in B_GPU_new
    cudaDeviceSynchronize();

    Array_sum_A_GPU(array_size_x,A_GPU_new,B_GPU_new,divAS,Tvals,Rvals,
Svals,k2); // Gives k2
    cudaDeviceSynchronize();
    Array_sum_B_GPU(array_size_x,A_GPU_new,B_GPU_new,divBS,divBR,divBT,
Tvals,Rvals,Svals,l2); // Gives l2
    cudaDeviceSynchronize();


    Add_Array(array_size_x,1.0,A_GPU,h/2.0,k2,A_GPU_new);
 // Adds old A_GPU to h*k2/2 and stores it in A_GPU_new
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU,h/2.0,l2,B_GPU_new);
// Adds old B_GPU to h*l2/2 and stores it in B_GPU
    cudaDeviceSynchronize();


    Array_sum_A_GPU(array_size_x,A_GPU_new,B_GPU,divAS,Tvals,Rvals,
Svals,k3);// Gives k3
    cudaDeviceSynchronize();
    Array_sum_B_GPU(array_size_x,A_GPU_new,B_GPU,divBS,divBR,divBT,
Tvals,Rvals,Svals,l3); // Gives l3
    cudaDeviceSynchronize();

    Add_Array(array_size_x,1.0,A_GPU,h,k3,A_GPU_new);
```

```
 // Adds old A_GPU to h*k3 and stores it in A_GPU_new
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU,h,l3,B_GPU_new);
// Adds old B_GPU to h*l3 and stores it in B_GPU_new
    cudaDeviceSynchronize();

    Array_sum_A_GPU(array_size_x,A_GPU_new,B_GPU_new,divAS,Tvals,Rvals,
Svals,k4); // gives k4
    cudaDeviceSynchronize();
    Array_sum_B_GPU(array_size_x,A_GPU_new,B_GPU_new,divBS,divBR,divBT,
Tvals,Rvals,Svals,l4); // Gives l4
    cudaDeviceSynchronize();

    // Updates A_GPU
    Add_Array(array_size_x,1.0,A_GPU,h/6.0,k1,A_GPU_new);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,A_GPU_new,2*h/6.0,k2,A_GPU);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,A_GPU,2*h/6.0,k3,A_GPU_new);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,A_GPU_new,h/6.0,k4,A_GPU);
    cudaDeviceSynchronize();

    // Updates B_GPU
    Add_Array(array_size_x,1.0,B_GPU,h/6.0,l1,B_GPU_new);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU_new,2*h/6.0,l2,B_GPU);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU,2*h/6.0,l3,B_GPU_new);
    cudaDeviceSynchronize();
    Add_Array(array_size_x,1.0,B_GPU_new,h/6.0,l4,B_GPU);
    cudaDeviceSynchronize();
}
```

A.5 Managing Data

```
        ofstream RK4_Afunc_Data;
    ofstream RK4_Bfunc_Data;
    RK4_Afunc_Data.open("RK4_Afunc_Data.txt");
    RK4_Bfunc_Data.open("RK4_Bfunc_Data.txt");

    for(int t = 0; t < max_time; t++){
      RK_4(array_size_x,h,A_GPU,B_GPU,A_func_new,B_func_new,divAS,
divBS,divBR,divBT,Tvals,Rvals,Svals,k1,k2,k3,
k4,l1,l2,l3,l4);
    cudaDeviceSynchronize();

    RK4_Afunc_Data << (t+1) << ", ";

    for(int i = 0; i < array_size_x; i++){
      RK4_Afunc_Data <<  A_GPU[i];
       if(i != array_size_x - 1){
        RK4_Afunc_Data << ", ";
    }
    }
       RK4_Afunc_Data << "\n";

    RK4_Bfunc_Data << (t+1) << ", ";

    for(int i = 0; i < array_size_x; i++){
      RK4_Bfunc_Data <<  B_GPU[i];
       if(i != array_size_x - 1){
        RK4_Bfunc_Data << ", ";
    }
    }
    RK4_Bfunc_Data << "\n";
    }
```

```
RK4_Afunc_Data.close();
RK4_Bfunc_Data.close();
```

A.6 S position

```
int positions_in_S_array_CPU(int i, int j, int l, int n){
    int position;
    if(i+j-l < n && i+j-l >= 0 && i != l && j != l){
    if(i < l){
    if(j < l){
      position = (3*pow(i,2)-6*i*l+6*i*n-3*i+6*j-2*pow(l,3)
      +3*pow(l,2)*n+3*l*pow(n,2)-12*l*n+2*l)/6;
    }
    else if(j > l){
      position = (3*pow(i,2)-6*i*l+6*i*n-3*i+6*j-2*pow(l,3)
      +3*pow(l,2)*n+3*l*pow(n,2)-12*l*n+2*l-6)/6;
    }
    }
    if(i > l){
    if(j < l){
      position = (-3*pow(i,2)+6*i*l+6*i*n-3*i+6*j-2*pow(l,3)
      +3*pow(l,2)*n-6*pow(l,2)+3*l*pow(n,2)-12*l*n+2*l-6*n-6)/6;
    }
    else if(j > l){
      position = (-3*pow(i,2)+6*i*l+6*i*n-3*i+6*j-2*pow(l,3)
      +3*pow(l,2)*n-6*pow(l,2)+3*l*pow(n,2)-12*l*n+2*l-6*n)/6;
    }
    }
    return position;
    }
    else{
    return -1;
    }
```

```
}
4
```

A.7 Makefile

```
BASE = $(shell /bin/pwd)
EXED = $(BASE)
OBJD = $(BASE)
SRCD = $(BASE)
INCD = $(BASE)
# UW
#CUDA = /usr/local/cuda
# Graham
CUDA = /cvmfs/soft.computecanada.ca/easybuild/software/2023/x86-64-v3/
Core/cudacore/12.2.2

#CUDASAM = $(CUDA)/cuda-samples

# Files
CPP_FILE = RK-4.cpp
CUDA_FILE = RK4.cu

# Compiler of C++
CC=g++

# CUDA compiler
NVCC = $(CUDA)/bin/nvcc -ccbin $(CC)
NVCCFLAGS = -m64 -Xcompiler

# Flags -I tells which directory to look for the .h files
CFLAGS = -Wall -fopenmp -I $(INCD) -O3 -L$LD_LIBRARY_PATH
CUFLAGS = -I$(CUDA)/include -I$(INCD) #-I$(CUDASAM)
CUFLAGS += -Wno-deprecated-declarations
```

```
# Libraries (Math)
LIBS = -Xpreprocessor -fopenmp -lm
# CUDA Libraries (as needed)
CULIBS = -lcudart -lnvJitLink

# Set of headers .h files
DEPS = $(wildcard $(INCD)/*.h)

# Set of source .cpp files
SRC = $(CPP_FILE)

# CUDA source files
CUSRC = $(CUDA_FILE)

# Object files
OBJS = $(SRC:.cpp=.o)
CUOBJS = $(CUSRC:.cu=.o)

EXECTBL = RK4

all: $(EXECTBL)

# Compile C++
%.o: %.cpp $(DEPS)
	@echo ''
	@echo '============ ...building' $*.o '============'
	@echo ''
	$(CC) -o $@ -c $< $(LIBS) $(CFLAGS) $(CUFLAGS) $(CULIBS)

# Compile CUDA code
%.o: %.cu $(DEPS)
	@echo ''
	@echo '============ ...building' $*.o '============'
	@echo ''
```

```
	$(NVCC) $(NVCCFLAGS) $(CUFLAGS) -c $< -o $@

$(EXECTBL): $(CUOBJS) $(OBJS)
	@echo ''
	@echo '============ ...building the executable ============'
	@echo ''
	cd $(OBJD); $(CC) $(CFLAGS) -DCUDA_DEBUG -L$(CUDA)/lib64 -Wl,-rpath
$(CUDA)/lib64 -o $(EXECTBL) $(OBJS) $(CUOBJS) $(LIBS) $(CULIBS)
	@echo ''
	@echo '============ ...Done! ============'

flush:
	rm -f $(EXECTBL) *.dat fort.* *.o *.kmo *.mod *.d *.pc*

clean:
	rm -f *.dat *.yg fort.* *.o *.kmo *.mod *.d *.pc* $(EXECTBL)
```

A.8 Data Analysis

```
import numpy as np
import matplotlib.pyplot as plt

file_path = 'RK4_Afunc_Data.txt'
#file_path = 'RK4_Bfunc_Data.txt'

A_vals = []
#B_vals = []
l_vals = []
t_vals = []
E_vals = []
N_vals = []

with open(file_path,"r") as Data_File:
    with open('data.txt','w') as data:
```

```
        for index, line in enumerate(Data_File):
        values = [float(value.strip()) for value in
          line.strip().split(',')]
            t = values[0]
            A_vals = values[1:]
            l_vals = np.arange(len(A_vals))

            E = 0
            N = 0
            for l in range(len(A_vals)):
                omega_l = 2*l + 3
                E += omega_l**2*abs(A_vals[l])**2
                N += omega_l*abs(A_vals[l])**2

            t_vals.append(t)
            E_vals.append(E)
            N_vals.append(N)

            data.write(f"{t},{E},{N}\n")

plt.figure()
plt.plot(t_vals,E_vals)
plt.savefig(f'energy vs. time.pdf')
plt.figure()
plt.plot(t_vals,N_vals)
plt.savefig(f'mode number vs. time.pdf')

E_init = E_vals[0]
N_init = N_vals[0]

E_rel_diff = [(E-E_init)/E_init for E in E_vals]
N_rel_diff = [(N-N_init)/N_init for N in N_vals]

plt.plot(t_vals,E_rel_diff,label='Relative difference in E',color='blue')
```

```
plt.xlabel("t")
plt.ylabel("Delta E/E")
plt.legend()
plt.grid()

plt.plot(t_vals,N_rel_diff,label='Relative difference in N',color='red')

plt.xlabel("t")
plt.ylabel("Delta N/N")
plt.legend()
plt.grid()
```

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
from scipy.optimize import curve_fit

file_path = 'RK4_Afunc_Data.txt'
# file_path = 'RK4_Bfunc_Data.txt'

A_vals = []
t_vals = []
alpha_vals = []
beta_vals = []
gamma_vals = []

def fit(l, alpha, beta, gamma):
    y = alpha * l**-beta * np.exp(-gamma * l)
    return y

with open(file_path, "r") as Data_File:
    with open('analysis_data.txt', 'w') as data:
        for line in Data_File:
```

```
        values = [float(value.strip()) for value in
          line.strip().split(',')]
            t = values[0]
            t_vals.append(t)
            A_vals = values[1:]
            l_vals = np.arange(len(A_vals))


            parameters_A, covariance_A = curve_fit(fit, l_vals[1:],
            A_vals[1:])


            alpha_vals.append(parameters_A[0])
            beta_vals.append(parameters_A[1])
            gamma_vals.append(parameters_A[2])

            for t, alpha, beta, gamma in zip(t_vals, alpha_vals, beta_vals,
            gamma_vals):
            data.write(f"t={t}, alpha={alpha}, beta={beta},
            gamma={gamma}\n")
```

```
import sys
import math

file_path_1 = sys.argv[1]
file_path_2 = sys.argv[2]

def read_data(file_path1,file_path_2):
    time_values_1 = []
    values_1 = []

    time_values_2 = []
    values_2 = []
```

```
    with open(file_path_1, "r") as data_file_1:
        for line in data_file_1:
            stripped_line = line.strip()
            if stripped_line:
                parts = stripped_line.split(',')
                time_values_1.append(parts[0])
                value_list = []
                for value in parts[1:]:
                    value_list.append(float(value.strip()))

                values_1.append(value_list)

    with open(file_path_2, "r") as data_file_2:
        for line in data_file_2:
            stripped_line = line.strip()
            if stripped_line:
                parts = stripped_line.split(',')
                time_values_2.append(parts[0])
                value_list = []
                for value in parts[1:]:
                    value_list.append(float(value.strip()))

                values_2.append(value_list)

    return time_values_1, values_1, time_values_2, values_2

def analysis(times_1, values_1, times_2, values_2):
    analysis_results = []

    for t_1, v_1, t_2, v_2 in zip(times_1, values_1, times_2, values_2):
        if v_1 and v_2:
            try:
                differences = [(v2 - v1)/v2 for v1, v2 in zip(v_1, v_2)]
```

```
                min_value = min(differences)
                max_value = max(differences)
                min_index = differences.index(min_value)
                max_index = differences.index(max_value)

                RMS_squared = sum(value**2 for value in differences)
                / len(differences)
                RMS = math.sqrt(RMS_squared)

                analysis_results.append({
                    'time': t_1,  # t_2
                    'min_value': min_value,
                    'min_index': min_index,
                    'max_value': max_value,
                    'max_index': max_index,
                    'RMS': RMS
                })
            except ZeroDivisionError as e:
                print("Division by zero!")
                break

    return analysis_results

times_1, values_1, times_2, values_2 = read_data(file_path_1,file_path_2)

analysis_results = analysis(times_1, values_1, times_2, values_2)

FP1 = file_path_1.rsplit('.',1)[0]
FP2 = file_path_2.rsplit('.',1)[0]

with open(f'Data_analysis_for_{FP1}_vs_{FP2}.txt', 'w') as analysis_file:
    for result in analysis_results:
        analysis_file.write(
        f"Time: {result['time']}, Min: {result['min_value']}
```

```
(Index: {result['min_index']}), "
f"Max: {result['max_value']}
(Index: {result['max_index']}), RMS: {result['RMS']}\n"
)
```

B Calculations

B.1 Positions in S array

We have a four indexed object S_{ijkl} where $0 \leq i, j, k, l \leq n-1$. The conditions are that:

1. $i, j \neq k, l$
2. $i + j = k + l$

Need an explicit formula, $f(i, j, k, l)$ which tells where the co-efficient is in the S array... We will preform multiple sums where the summand is 1.

i)

Given i, l and n how many allowed values of j are there?

If there were not any restrictions on i, j, k, l then S would have:

$$\sum_{l=0}^{n-1} \sum_{k=0}^{n-1} \sum_{j=0}^{n-1} \sum_{i=0}^{n-1} 1 = n(n)(n)(n) = n^4 \tag{1}$$

terms.

However given the restrictions above, we find that we need to sum over j. There will be three cases depending on the sign of $i - l$:

Case 1: $i - l < 0$

$$\sum_{j=l+1}^{n-1} 1 + \sum_{j=l-i}^{l-1} 1 = n - l + i - 1 \tag{2}$$

Case 2: $i - l = 0$

$$\sum_{j=0}^{i-1} 1 + \sum_{j=i+1}^{n-1} 1 = i + n - i - 1 = n - 1 \tag{3}$$

Case 3: $i - l > 0$

$$\sum_{j=0}^{l-1} 1 + \sum_{j=l+1}^{n-1+l-i} 1 = l + n - i - 1 \tag{4}$$

ii)

Given l and n how many allowed values of j and i are there?

$$\begin{aligned} \sum_{i=0}^{l-1} n - l + i - 1 + \sum_{i=l+1}^{n-1} n + l - i - 1 &= \frac{-1}{2}(l(l - 2n + 3)) - \frac{1}{2}(l - n + 1)(l + n - 2) \\ &= \frac{n^2}{2} + n\left(l - \frac{3}{2}\right) - l^2 - l + 1 \end{aligned} \tag{5}$$

The total number of values in S, given n is

$$\sum_{l=0}^{n-1} \frac{n^2}{2} + n\left(l - \frac{3}{2}\right) - l^2 - l + 1 = \frac{2}{3}\left(n^3 - 3n^2 + 2n\right) = \frac{2}{3}n\left(n - 1\right)\left(n - 2\right) \tag{6}$$

Where we summed over all the l values...

Hence given n, the total values in S_{ijkl} is $\frac{2}{3}n(n-1)(n-2)$.

Now we want to find the number of allowed index values that 'come before' a particular n. We split the sum into three cases depending on if $i < l$, $i = l$, $i > l$.

If $i < l$, we sum 6 over $\tilde{l} = 0 \cdots l-1$ plus sum over 2 over $\tilde{i} = 0 \cdots i-1$ plus sum over 1 over $\tilde{j} = l+1 \cdots j-1$.

If $i > l$, we sum 6 over $\tilde{l} = 0 \cdots l-1$ plus sum over 2 over $\tilde{i} = 0 \cdots l-1$ plus sum over 4 over $\tilde{i} = l+1 \cdots i-1$ plus sum over 1 over $\tilde{j} = 0 \cdots j-1$.

If $i = l$ there is no sum.

Case 1: $i - l < 0$

Here we will have three cases depening in the sign of $j - l$.

Case i: $j - l < 0$

$$\sum_{\tilde{l}=0}^{l-1}\left(\frac{n^2}{2}+n\left(\tilde{l}-\frac{3}{2}\right)-\tilde{l}^2-\tilde{l}+1\right)+\sum_{\tilde{i}=0}^{i-1}\left(n-l+\tilde{i}-1\right)+\sum_{\tilde{i}=l-i}^{j-1}1= \tag{7}$$

$$-\frac{l}{6}\left(2l^2-3ln-3n^2+12n-8\right)$$

$$+i\left(n-l-\frac{3}{2}\right)+\frac{i^2}{2}+j-l+i=$$

$$\frac{1}{6}(3i^2-6il+6in-3i+6j-2l^3+3l^2n+3ln^2-12ln+2l)$$

Case ii: $j - l = 0$

N/A

Case iii: $j - l > 0$

$$\sum_{\tilde{l}=0}^{l-1}\left(\frac{n^2}{2}+n\left(\tilde{l}-\frac{3}{2}\right)-\tilde{l}^2-\tilde{l}+1\right)+\sum_{\tilde{i}=0}^{i-1}\left(n-l+\tilde{i}-1\right)+\sum_{\tilde{j}=l-i}^{l-1}1+\sum_{\tilde{j}=l+1}^{j-1}1= \tag{8}$$

$$-\frac{l}{6}\left(2l^2-3ln-3n^2+12n-8\right)$$

$$+i\left(n-l-\frac{3}{2}\right)+\frac{i^2}{2}+j-l+i-1=$$

$$\frac{1}{6}(3i^2-6il+6in-3i+6j-2l^3+3l^2n+3ln^2-12ln+2l-6)$$

Case 2: $i-l=0$

N/A

Case 3: $i-l>0$

Here we will have three cases depening in the sign of $j-l$.

Case i: $j-l<0$

$$\sum_{\tilde{l}=0}^{l-1}\left(\frac{n^2}{2}+n\left(\tilde{l}-\frac{3}{2}\right)-\tilde{l}^2-\tilde{l}+1\right)+\sum_{\tilde{i}=0}^{l-1}\left(n-l+\tilde{i}-1\right)+\sum_{\tilde{i}=l+1}^{i-1}\left(l+n-\tilde{i}-1\right)+\sum_{\tilde{j}=0}^{j-1}1= \tag{9}$$

$$-\frac{l}{6}(2l^2-3ln-3n^2+12n-8)-\frac{l}{2}(l-2n+3)-\frac{1}{2}(i-l-1)(i-l-2n+2)+j=$$

$$\frac{1}{6}(-3i^2+6il+6in-3i+6j-2l^3+3l^2n+3ln^2-6l^2-12ln+2l-6n+6)$$

Case ii: $j-l=0$

N/A

Case iii: $j-l>0$

$$\sum_{\tilde{l}=0}^{l-1}\left(\frac{n^2}{2}+n\left(\tilde{l}-\frac{3}{2}\right)-\tilde{l}^2-\tilde{l}+1\right)+\sum_{\tilde{i}=0}^{l-1}\left(n-l+\tilde{i}-1\right)+\sum_{\tilde{i}=l+1}^{i-1}\left(l+n-\tilde{i}-1\right)+\sum_{\tilde{j}=0}^{l-1}1+\sum_{\tilde{j}=l+1}^{j-1}1 \quad (10)$$

$$=-\frac{l}{6}(2l^2-3ln-3n^2+12n-8)-\frac{l}{2}(l-2n+3)-\frac{1}{2}(i-l-1)(i-l-2n+2)+l+j-l-1$$

$$=\frac{1}{6}(-3i^2+6il+6in-3i+6j-2l^3+3l^2n-6l^2+3ln^2-12ln+2l-6n)$$

B.2 Klein-Gordon equation

The Klein-Gordon equation of a scalar field, $\phi(x,t)$, for a general space-time with an arbitrary metric is given as,

$$\Box\phi(x,t)+m^2\phi(x,t)=0 \quad (11)$$

where $\Box\equiv\nabla^\mu\nabla_\mu$ is the d'Alembert operator, and ∇_μ is the covariant derivative.

Since we are interested in a mass-less scalar field, $m=0$. Thus 11 becomes:

$$\nabla^\mu\nabla_\mu\phi(x,t)=0 \quad (12)$$

Now, the covariant derivative of a scalar is simply the regular partial derivative, while the covariant derivative of a vector is given as: $\nabla_\mu A_\nu=\partial_\mu A_\nu-\Gamma^\alpha_{\mu\nu}A_\alpha$ where $\Gamma^\alpha_{\mu\nu}=\frac{1}{2}g^{\alpha\beta}(\partial_\mu g_{\beta\nu}+\partial_\nu g_{\beta\mu}-\partial_\beta g_{\mu\nu})$ is the Christoffel symbols. The covariant derivatives and the Christoffel symbols are needed to account for the curvature of a general space-time.

Our AdS space-time line element is give by 1.1, i.e.

$$ds^2=\sec^2 x[-dt^2+dx^2+\sin^2 x d\Omega^2_{n-1}]. \quad (13)$$

Since we are working in AdS_4, $d\Omega^2_{n-1}$ is the usual 2-sphere. So,

$$ds^2=\sec^2 x[-dt^2+dx^2+\sin^2 x(d\theta^2+\sin^2\theta d\phi^2)]. \quad (14)$$

Thus our non zero metric components are: $g_{00} = -\sec^2 x$, $g_{11} = \sec^2 x$, $g_{22} = \sec^2 x \sin^2 x = \tan^2 x$ and $g_{33} = \sec^2 x \sin^2 x \sin^2 \theta = \tan^2 x \sin^2 \theta$ where $(t, x, \theta, \phi) \equiv (0, 1, 2, 3)$.

Going back to equation 12 we can simplify this as:

$$\nabla^\mu \nabla_\mu \phi(x,t) = g^{\mu\nu} \nabla_\nu \partial_\mu \phi(x,t) = g^{\mu\nu} \left[\partial_\mu \partial_\nu - \Gamma^\alpha_{\mu\nu} \partial_\alpha\right] \phi(x,t) = 0 \tag{15}$$

where all the indices go from 0 to 3.

We need to explicitly calculate the Christoffel symbols. Realizing that $\Gamma^\alpha_{\mu\nu} = \Gamma^\alpha_{\nu\mu}$ we have:

1. $\Gamma^0_{00} = \frac{1}{2} g^{0\beta}[\partial_0 g_{\beta 0} + \partial_0 g_{\beta 0} - \partial_\beta g_{00}] = \frac{1}{2} g^{00} \partial_0 g_{00} = 0$

2. $\Gamma^0_{01} = \Gamma^0_{10} = \frac{1}{2} g^{0\beta}[\partial_0 g_{\beta 1} + \partial_1 g_{\beta 0} - \partial_\beta g_{01}] = \frac{1}{2} g^{00}[\partial_0 g_{01} + \partial_1 g_{00} - \partial_0 g_{01}] = \frac{1}{2} g^{00} \partial_1 g_{00} = \frac{1}{-2\sec^2 x}[\partial_x(-\sec^2 x)] = \tan x$

3. $\Gamma^0_{02} = \Gamma^0_{20} = \frac{1}{2} g^{0\beta}[\partial_0 g_{\beta 2} + \partial_2 g_{\beta 0} - \partial_\beta g_{02}] = \frac{1}{2} g^{00}[\partial_0 g_{02} + \partial_2 g_{00} - \partial_0 g_{02}] = 0$

4. $\Gamma^0_{03} = \Gamma^0_{30} = \frac{1}{2} g^{0\beta}[\partial_0 g_{\beta 3} + \partial_3 g_{\beta 0} - \partial_\beta g_{03}] = \frac{1}{2} g^{00}[\partial_0 g_{03} + \partial_3 g_{00} - \partial_0 g_{33}] = 0$

5. $\Gamma^0_{11} = \frac{1}{2} g^{0\beta}[\partial_1 g_{\beta 1} + \partial_1 g_{\beta 1} - \partial_\beta g_{11}] = \frac{1}{2} g^{00}[\partial_1 g_{01} + \partial_1 g_{01} - \partial_0 g_{11}] = 0$

6. $\Gamma^0_{12} = \Gamma^0_{21} = \frac{1}{2} g^{0\beta}[\partial_1 g_{\beta 2} + \partial_2 g_{\beta 1} - \partial_\beta g_{12}] = \frac{1}{2} g^{00}[\partial_1 g_{02} + \partial_2 g_{01} - \partial_0 g_{12}] = 0$

7. $\Gamma^0_{13} = \Gamma^0_{31} = \frac{1}{2} g^{0\beta}[\partial_1 g_{\beta 3} + \partial_3 g_{\beta 1} - \partial_\beta g_{13}] = \frac{1}{2} g^{00}[\partial_1 g_{03} + \partial_3 g_{01} - \partial_0 g_{31}] = 0$

8. $\Gamma^0_{22} = \frac{1}{2} g^{0\beta}[\partial_2 g_{\beta 2} + \partial_2 g_{\beta 2} - \partial_\beta g_{22}] = \frac{1}{2} g^{00}[\partial_2 g_{02} + \partial_2 g_{20} - \partial_0 g_{22}] = 0$

9. $\Gamma^0_{23} = \Gamma^0_{32} = \frac{1}{2} g^{0\beta}[\partial_2 g_{\beta 3} + \partial_3 g_{\beta 2} - \partial_\beta g_{23}] = \frac{1}{2} g^{00}[\partial_2 g_{03} + \partial_3 g_{02} - \partial_0 g_{23}] = 0$

10. $\Gamma^0_{33} = \frac{1}{2}g^{0\beta}[\partial_3 g_{\beta 3} + \partial_3 g_{\beta 3} - \partial_\beta g_{33}] = \frac{1}{2}g^{00}[\partial_3 g_{03} + \partial_3 g_{03} - \partial_0 g_{33}] = 0$

11. $\Gamma^1_{00} = \frac{1}{2}g^{1\beta}[\partial_0 g_{\beta 0} + \partial_0 g_{\beta 0} - \partial_\beta g_{00}] = -\frac{1}{2}g^{11}\partial_1 g_{00} = \tan x$

12. $\Gamma^1_{01} = \Gamma^1_{10} = \frac{1}{2}g^{1\beta}[\partial_0 g_{\beta 1} + \partial_1 g_{\beta 0} - \partial_\beta g_{01}] = \frac{1}{2}g^{11}[\partial_0 g_{11} + \partial_1 g_{10} - \partial_1 g_{01}] = 0$

13. $\Gamma^1_{02} = \Gamma^1_{20} = \frac{1}{2}g^{1\beta}[\partial_0 g_{\beta 2} + \partial_2 g_{\beta 0} - \partial_\beta g_{02}] = \frac{1}{2}g^{11}[\partial_0 g_{12} + \partial_2 g_{10} - \partial_1 g_{02}] = 0$

14. $\Gamma^1_{03} = \Gamma^1_{30} = \frac{1}{2}g^{1\beta}[\partial_0 g_{\beta 3} + \partial_3 g_{\beta 0} - \partial_\beta g_{03}] = \frac{1}{2}g^{11}[\partial_0 g_{13} + \partial_3 g_{01} - \partial_1 g_{03}] = 0$

15. $\Gamma^1_{11} = \frac{1}{2}g^{1\beta}[\partial_1 g_{\beta 1} + \partial_1 g_{\beta 1} - \partial_\beta g_{11}] = \frac{1}{2}g^{11}[\partial_1 g_{11}] = \frac{1}{2\sec^2 x}\partial_x \sec^2 x = \tan x$

16. $\Gamma^1_{12} = \Gamma^1_{21} = \frac{1}{2}g^{1\beta}[\partial_1 g_{\beta 2} + \partial_2 g_{\beta 1} - \partial_\beta g_{12}] = \frac{1}{2}g^{11}[\partial_1 g_{12} + \partial_2 g_{11} - \partial_1 g_{12}] = 0$

17. $\Gamma^1_{13} = \Gamma^1_{31} = \frac{1}{2}g^{1\beta}[\partial_1 g_{\beta 3} + \partial_3 g_{\beta 1} - \partial_\beta g_{13}] = \frac{1}{2}g^{11}[\partial_1 g_{13} + \partial_3 g_{11} - \partial_1 g_{31}] = 0$

18. $\Gamma^1_{22} = \frac{1}{2}g^{1\beta}[\partial_2 g_{\beta 2} + \partial_2 g_{\beta 2} - \partial_\beta g_{22}] = \frac{1}{2}g^{11}[\partial_2 g_{12} + \partial_2 g_{12} - \partial_1 g_{22}] = \frac{-1}{2\sec^2 x}\partial_x \sec^2 x \sin^2 x = -\tan x$

19. $\Gamma^1_{23} = \Gamma^1_{32} = \frac{1}{2}g^{1\beta}[\partial_2 g_{\beta 3} + \partial_3 g_{\beta 2} - \partial_\beta g_{23}] = \frac{1}{2}g^{11}[\partial_2 g_{13} + \partial_3 g_{12} - \partial_1 g_{23}] = 0$

20. $\Gamma^1_{33} = \frac{1}{2}g^{1\beta}[\partial_3 g_{\beta 3} + \partial_3 g_{\beta 3} - \partial_\beta g_{33}] = \frac{1}{2}g^{11}[\partial_3 g_{13} + \partial_3 g_{13} - \partial_1 g_{33}] = \frac{-1}{2\sec^2 x}\partial_x(\sec^2 x \sin^2 x \sin^2\theta) = -\tan x \sin^2\theta$

21. $\Gamma^2_{00} = \frac{1}{2}g^{2\beta}[\partial_0 g_{\beta 0} + \partial_0 g_{\beta 0} - \partial_\beta g_{00}] = \frac{1}{2}g^{22}\partial_2 g_{00} = 0$

22. $\Gamma^2_{01} = \Gamma^2_{10} = \frac{1}{2}g^{2\beta}[\partial_0 g_{\beta 1} + \partial_1 g_{\beta 0} - \partial_\beta g_{01}] = \frac{1}{2}g^{22}[\partial_0 g_{21} + \partial_1 g_{20} - \partial_2 g_{01}] = 0$

23. $\Gamma^2_{02} = \Gamma^2_{20} = \frac{1}{2}g^{2\beta}[\partial_0 g_{\beta 2} + \partial_2 g_{\beta 0} - \partial_\beta g_{02}] = \frac{1}{2}g^{22}[\partial_0 g_{22} + \partial_2 g_{20} - \partial_2 g_{02}] = 0$

24. $\Gamma^2_{03} = \Gamma^2_{30} = \frac{1}{2}g^{2\beta}[\partial_0 g_{\beta 3} + \partial_3 g_{\beta 0} - \partial_\beta g_{03}] = \frac{1}{2}g^{22}[\partial_0 g_{23} + \partial_3 g_{20} - \partial_2 g_{03}] = 0$

25. $\Gamma^2_{11} = \frac{1}{2}g^{2\beta}[\partial_1 g_{\beta 1} + \partial_1 g_{\beta 1} - \partial_\beta g_{11}] = \frac{1}{2}g^{22}[\partial_1 g_{21} + \partial_1 g_{21} - \partial_2 g_{11}] = 0$

26. $\Gamma^2_{12} = \Gamma^2_{21} = \frac{1}{2}g^{2\beta}[\partial_1 g_{\beta 2} + \partial_2 g_{\beta 1} - \partial_\beta g_{12}] = \frac{1}{2}g^{22}[\partial_1 g_{22} + \partial_2 g_{21} - \partial_2 g_{12}] = \frac{1}{2\sin^2 x \sec^2 x}\partial_x(\sin^2 x \sec^2 x) = \frac{1}{\cos x \sin x}$

27. $\Gamma^2_{13} = \Gamma^2_{31} = \frac{1}{2}g^{2\beta}[\partial_1 g_{\beta 3} + \partial_3 g_{\beta 1} - \partial_\beta g_{13}] = \frac{1}{2}g^{22}[\partial_1 g_{23} + \partial_3 g_{21} - \partial_2 g_{31}] = 0$

28. $\Gamma^2_{22} = \frac{1}{2}g^{2\beta}[\partial_2 g_{\beta 2} + \partial_2 g_{\beta 2} - \partial_\beta g_{22}] = \frac{1}{2}g^{22}[\partial_2 g_{22} + \partial_2 g_{22} - \partial_2 g_{22}] = 0$

29. $\Gamma^2_{23} = \Gamma^2_{32} = \frac{1}{2}g^{2\beta}[\partial_2 g_{\beta 3} + \partial_3 g_{\beta 2} - \partial_\beta g_{23}] = \frac{1}{2}g^{22}[\partial_2 g_{23} + \partial_3 g_{22} - \partial_2 g_{23}] = 0$

30. $\Gamma^2_{33} = \frac{1}{2}g^{2\beta}[\partial_3 g_{\beta 3} + \partial_3 g_{\beta 3} - \partial_\beta g_{33}] = \frac{1}{2}g^{22}[\partial_3 g_{23} + \partial_3 g_{23} - \partial_2 g_{33}] = -\frac{1}{2\tan^2 x}\partial_\theta(\tan^2 x \sin^2\theta) = -\sin\theta\cos\theta$

31. $\Gamma^3_{00} = \frac{1}{2}g^{3\beta}[\partial_0 g_{\beta 0} + \partial_0 g_{\beta 0} - \partial_\beta g_{00}] = \frac{1}{2}g^{33}\partial_3 g_{00} = 0$

32. $\Gamma^3_{01} = \Gamma^3_{10} = \frac{1}{2}g^{3\beta}[\partial_0 g_{\beta 1} + \partial_1 g_{\beta 0} - \partial_\beta g_{01}] = \frac{1}{2}g^{33}[\partial_0 g_{31} + \partial_1 g_{30} - \partial_3 g_{01}] = 0$

33. $\Gamma^3_{02} = \Gamma^3_{20} = \frac{1}{2}g^{3\beta}[\partial_0 g_{\beta 2} + \partial_2 g_{\beta 0} - \partial_\beta g_{02}] = \frac{1}{2}g^{33}[\partial_0 g_{32} + \partial_2 g_{30} - \partial_3 g_{02}] = 0$

34. $\Gamma^3_{03} = \Gamma^3_{30} = \frac{1}{2}g^{3\beta}[\partial_0 g_{\beta 3} + \partial_3 g_{\beta 0} - \partial_\beta g_{03}] = \frac{1}{2}g^{33}[\partial_0 g_{33} + \partial_3 g_{30} - \partial_0 g_{33}] = 0$

35. $\Gamma^3_{11} = \frac{1}{2}g^{3\beta}[\partial_1 g_{\beta 1} + \partial_1 g_{\beta 1} - \partial_\beta g_{11}] = \frac{1}{2}g^{33}[\partial_1 g_{31} + \partial_1 g_{31} - \partial_3 g_{11}] = 0$

36. $\Gamma^3_{12} = \Gamma^3_{21} = \frac{1}{2}g^{3\beta}[\partial_1 g_{\beta 2} + \partial_2 g_{\beta 1} - \partial_\beta g_{12}] = \frac{1}{2}g^{33}[\partial_1 g_{32} + \partial_2 g_{31} - \partial_3 g_{12}] = 0$

37. $\Gamma^3_{13} = \Gamma^3_{31} = \frac{1}{2}g^{3\beta}[\partial_1 g_{\beta 3} + \partial_3 g_{\beta 1} - \partial_\beta g_{13}] = \frac{1}{2}g^{33}[\partial_1 g_{33} + \partial_3 g_{31} - \partial_3 g_{13}] = \frac{1}{2\tan^2 x \sin^2\theta}\partial_x(\tan^2 x \sin^2\theta) = \frac{1}{\sin x \cos x}$

38. $\Gamma^3_{22} = \frac{1}{2}g^{3\beta}[\partial_2 g_{\beta 2} + \partial_2 g_{\beta 2} - \partial_\beta g_{22}] = \frac{1}{2}g^{33}[\partial_2 g_{32} + \partial_2 g_{32} - \partial_3 g_{22}] = 0$

39. $\Gamma^3_{23} = \Gamma^3_{32} = \frac{1}{2}g^{3\beta}[\partial_2 g_{\beta 3} + \partial_3 g_{\beta 2} - \partial_\beta g_{23}] = \frac{1}{2}g^{33}[\partial_2 g_{33} + \partial_3 g_{32} - \partial_3 g_{23}] = \frac{2\tan^2 x \sin\theta\cos\theta}{2\tan^2 x \sin^2\theta} = \frac{1}{\tan\theta}$

40. $\Gamma^3_{33} = \frac{1}{2}g^{3\beta}[\partial_3 g_{\beta 3} + \partial_3 g_{\beta 3} - \partial_\beta g_{33}] = \frac{1}{2}g^{33}[\partial_3 g_{33} + \partial_3 g_{33} - \partial_3 g_{33}] = 0$

Expanding (15) we get:

$$
\begin{aligned}
& g^{00}\partial_0\partial_0\phi(x,t) + g^{01}\partial_0\partial_1\phi(x,t) + g^{10}\partial_1\partial_0\phi(x,t) + g^{02}\partial_0\partial_2\phi(x,t) + g^{03}\partial_0\partial_3\phi(x,t) \quad (16)\\
+ \;& g^{10}\partial_1\partial_0\phi(x,t) + g^{11}\partial_1\partial_1\phi(x,t) + g^{12}\partial_1\partial_2\phi(x,t) + g^{13}\partial_1\partial_2\phi(x,t) + g^{20}\partial_2\partial_0\phi(x,t)\\
+ \;& g^{21}\partial_2\partial_1\phi(x,t) + g^{22}\partial_2\partial_2\phi(x,t) + g^{23}\partial_2\partial_3\phi(x,t) + g^{30}\partial_3\partial_0\phi(x,t) + g^{31}\partial_3\partial_1\phi(x,t)\\
+ \;& g^{32}\partial_3\partial_2\phi(x,t) + g^{33}\partial_3\partial_3\phi(x,t) - g^{00}\Gamma^0_{00}\partial_0\phi(x,t) - g^{00}\Gamma^1_{00}\partial_1\phi(x,t) - g^{00}\Gamma^2_{00}\partial_2\phi(x,t)\\
- \;& g^{00}\Gamma^3_{00}\partial_3\phi(x,t) - g^{11}\Gamma^0_{11}\partial_0\phi(x,t) - g^{11}\Gamma^1_{11}\partial_1\phi(x,t) - g^{11}\Gamma^2_{11}\partial_2\phi(x,t) - g^{11}\Gamma^3_{11}\partial_3\phi(x,t)\\
- \;& g^{22}\Gamma^0_{22}\partial_0\phi(x,t) - g^{22}\Gamma^1_{22}\partial_1\phi(x,t) - g^{22}\Gamma^2_{22}\partial_2\phi(x,t) - g^{22}\Gamma^3_{22}\partial_3\phi(x,t)\\
- \;& g^{33}\Gamma^0_{33}\partial_0\phi(x,t) - g^{33}\Gamma^1_{33}\partial_1\phi(x,t) - g^{33}\Gamma^2_{33}\partial_2\phi(x,t) - g^{33}\Gamma^3_{33}\partial_3\phi(x,t) = 0
\end{aligned}
$$

However all the off-diagonal terms in the metric are 0 and the only non-zero Christoffel symbols are: $\Gamma^0_{01}, \Gamma^0_{10}, \Gamma^1_{00}, \Gamma^1_{11}, \Gamma^1_{22}, \Gamma^1_{33}, \Gamma^2_{12}, \Gamma^2_{21}, \Gamma^2_{33}, \Gamma^3_{31}, \Gamma^3_{13}, \Gamma^3_{23}, \Gamma^3_{32}$, thus the above reduces to:

$$
\begin{aligned}
& g^{00}\partial_0\partial_0\phi(x,t) + g^{11}\partial_1\partial_1\phi(x,t) + g^{22}\partial_2\partial_2\phi(x,t) + g^{33}\partial_3\partial_3\phi(x,t) \quad (17)\\
- \;& g^{11}\Gamma^1_{11}\partial_1\phi(x,t) - g^{22}\Gamma^1_{22}\partial_1\phi(x,t) - g^{33}\Gamma^1_{33}\partial_1\phi(x,t) - g^{33}\Gamma^2_{33}\partial_2\phi(x,t) - g^{00}\Gamma^1_{00}\partial_1\phi(x,t) = 0
\end{aligned}
$$

which upon substituting the coefficients we get:

$$
\begin{aligned}
- & \frac{1}{\sec^2 x}\frac{\partial^2\phi(x,t)}{\partial t^2} + \frac{1}{\sec^2 x}\frac{\partial^2\phi(x,t)}{\partial x^2} - \frac{1}{\sec^2 x}\tan x\frac{\partial\phi(x,t)}{\partial x} \\
- & \frac{1}{\sin^2 x \sec^2 x}(-\tan x)\frac{\partial\phi(x,t)}{\partial x} \\
- & \frac{1}{\sin^2 x \sec^2 x \sin^2\theta}((-\tan x)\sin^2\theta)\frac{\partial\phi(x,t)}{\partial x} - \frac{-1}{\sec^2 x}\tan x\frac{\partial\phi(x,t)}{\partial x} = 0
\end{aligned}
\tag{18}
$$

since our field is independent of θ and ϕ. Simplifiying gives:

$$
- \quad \frac{\partial^2\phi(x,t)}{\partial t^2} + \frac{\partial^2\phi(x,t)}{\partial x^2} + \frac{2\tan x}{\sin^2 x}\frac{\partial\phi(x,t)}{\partial x} = 0 \tag{19}
$$

which becomes:

$$
- \quad \frac{\partial^2\phi(x,t)}{\partial t^2} + \frac{\partial^2\phi(x,t)}{\partial x^2} + \frac{2}{\sin x\cos x}\frac{\partial\phi(x,t)}{\partial x} = 0 \tag{20}
$$

Recall that $\hat{L} = -(\tan^{1-n} x)\frac{\partial}{\partial x}(\tan^{n-1} x\frac{\partial}{\partial x})$.

Now:

$$
\begin{aligned}
\hat{L}\phi(x,t) &= -(\tan^{1-n} x)\frac{\partial}{\partial x}(\tan^{n-1} x\frac{\partial}{\partial x})\phi(x,t) = -\frac{\partial^2\phi(x,t)}{\partial x^2} \\
&- \tan^{1-n} x(n-1)\frac{\tan^{n-2} x}{\cos^2 x}\frac{\partial\phi(x,t)}{\partial x} = -\frac{\partial^2\phi(x,t)}{\partial x^2} - \frac{(n-1)}{\tan x\cos^2 x}\frac{\partial\phi(x,t)}{\partial x} \\
&= -\frac{\partial^2\phi(x,t)}{\partial x^2} - \frac{(n-1)}{\sin x\cos x}\frac{\partial\phi(x,t)}{\partial x}
\end{aligned}
\tag{21}
$$

which is the spatial derivatives of 2.2 for $n = 3$ spatial dimensions. Hence in 4 space-time dimensons,

$$
-\frac{\partial^2\phi(x,t)}{\partial t^2} + \frac{\partial^2\phi(x,t)}{\partial x^2} + \frac{2}{\sin x\cos x}\frac{\partial\phi(x,t)}{\partial x} = -\frac{\partial^2\phi(x,t)}{\partial t^2} - \hat{L}\phi(x,t) = 0 \tag{22}
$$

B.3 Runge-Kutta 4 derivation

To derive the RK4 algorithm, we start with the Taylor series of $y(t_i)$. Since the RK4 algorithm is of fourth order, the Taylor series must be expanded to fourth order. The method is based on computing the $(i+1)^{\text{th}}$ iteration, y_{i+1} via the recurrence relation,

$$y_{i+1} = y_i + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4 \tag{23}$$

where the w_i's are to be determined, $y_i \equiv y(t_i)$ and

1. $k_1 = hf(t_i, y_i)$
2. $k_2 = hf(t_i + a_1 h, y_i + b_1 k_1)$
3. $k_3 = hf(t_i + a_2 h, y_i + b_2 k_1 + b_3 k_2)$
4. $k_4 = hf(t_i + a_3 h, y_i + b_4 k_1 + b_5 k_2 + b_6 k_3)$

where the a_i's and b_i's are to be determined.
The Taylor series of $y(t_i)$ to fourth order is given by:

$$y(t_{i+1}) = y(t_i) + h\frac{dy}{dt}\bigg|_{t=t_i} + \frac{h^2}{2!}\frac{d^2y}{dt^2}\bigg|_{t=t_i} + \frac{h^3}{3!}\frac{d^3y}{dt^3}\bigg|_{t=t_i} + \frac{h^4}{4!}\frac{d^4y}{dt^4}\bigg|_{t=t_i} + \mathcal{O}(h^5) \tag{24}$$

Now, since the ODE is $\frac{dy(t)}{dt} = f(t, y(t))$ we can find the higher order derivatives by implicit differentiation.

$$\frac{d^2y}{dt^2} = \frac{df(t,y)}{dt} = \frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\frac{dy}{dt} = \frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}$$

$$
\begin{aligned}
\frac{d^3y}{dt^3} &= \frac{d}{dt}\left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right) = \frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y}\frac{dy}{dt} \\
&+ \left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right)\frac{\partial f(t,y)}{\partial y} + f(t,y)\left(\frac{\partial^2 f(t,y)}{\partial y \partial t} + \frac{\partial^2 f(t,y)}{\partial y^2}\frac{dy}{dt}\right) \\
&= \frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y} f(t,y) + \left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right)\frac{\partial f(t,y)}{\partial y} \qquad (25) \\
&+ f(t,y)\left(\frac{\partial^2 f(t,y)}{\partial y \partial t} + \frac{\partial^2 f(t,y)}{\partial y^2} f(t,y)\right) \\
&= \frac{\partial^2 f(t,y)}{\partial t^2} + 2f(t,y)\frac{\partial^2 f(t,y)}{\partial y \partial t} + f(t,y)^2\frac{\partial^2 f(t,y)}{\partial y^2} + f(t,y)\left(\frac{\partial f(t,y)}{\partial t}\right)^2 + \frac{\partial f(t,y)}{\partial t}\frac{\partial f(t,y)}{\partial y}
\end{aligned}
$$

$$
\begin{aligned}
\frac{d^4y}{dt^4} &= \frac{d}{dt}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + 2f(t,y)\frac{\partial^2 f(t,y)}{\partial y \partial t} + f(t,y)^2\frac{\partial^2 f(t,y)}{\partial y^2}\right. \\
&\left. + f(t,y)\left(\frac{\partial f(t,y)}{\partial t}\right)^2 + \frac{\partial f(t,y)}{\partial t}\frac{\partial f(t,y)}{\partial y}\right) \qquad (26) \\
&= \frac{\partial^3 f(t,y)}{\partial t^3} + \frac{\partial^3 f(t,y)}{\partial t^2 \partial y}\frac{dy}{dt} + 2\left(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\frac{dy}{dt}\right)\frac{\partial^2 f(t,y)}{\partial y \partial t} \\
&+2f(t,y)\left(\frac{\partial^3 f(t,y)}{\partial y \partial t^2} + \frac{\partial^3 f(t,y)}{\partial t \partial y^2}\frac{dy}{dt}\right) + 2f(t,y)\left(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}\frac{dy}{dt}\right)\frac{\partial^2 f(t,y)}{\partial y^2} \\
&+f(t,y)^2\left(\frac{\partial^3 f(t,y)}{\partial t \partial y^2} + \frac{\partial^3 f(t,y)}{\partial y^3}\frac{dy}{dt}\right) + \left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right)\left(\frac{\partial f(t,y)}{\partial t}\right)^2 \\
&+f(t,y)\left(2\frac{\partial f(t,y)}{\partial t}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y}\frac{dy}{dt}\right) + \frac{\partial f(t,y)}{\partial y}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y}\frac{dy}{dt}\right)\right. \\
&\left. + \frac{\partial f(t,y)}{\partial t}\left(\frac{\partial^2 f(t,y)}{\partial y \partial t} + \frac{\partial^2 f(t,y)}{\partial y^2}\frac{dy}{dt}\right)\right) \qquad (27)
\end{aligned}
$$

Thus plugging these back into 24 gives:

$$
\begin{aligned}
y(t_{i+1}) = y(t_i) &+ hf(t,y)\Big|_{t=t_i} \\
&+ \frac{h^2}{2!}\left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right)\Big|_{t=t_i} \\
&+ \frac{h^3}{3!}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + 2f(t,y)\frac{\partial^2 f(t,y)}{\partial y \partial t} + f(t,y)^2\frac{\partial^2 f(t,y)}{\partial y^2}\right. \\
&\left. + f(t,y)\left(\frac{\partial f(t,y)}{\partial t}\right)^2 + \frac{\partial f(t,y)}{\partial t}\frac{\partial f(t,y)}{\partial y}\right|_{t=t_i} \\
&+ \frac{h^4}{4!}\left(\frac{\partial^3 f(t,y)}{\partial t^3} + f(t,y)\frac{\partial^3 f(t,y)}{\partial t^2 \partial y}\right. \\
&+ 2\left(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}f(t,y)\right)\frac{\partial^2 f(t,y)}{\partial y \partial t} \\
&+ 2f(t,y)\left(\frac{\partial^3 f(t,y)}{\partial y \partial t^2} + \frac{\partial^3 f(t,y)}{\partial t \partial y^2}\right) \\
&+ 2f(t,y)\left(\frac{\partial f(t,y)}{\partial t} + \frac{\partial f(t,y)}{\partial y}f(t,y)\right)\frac{\partial^2 f(t,y)}{\partial y^2} \\
&+ f(t,y)^2\left(\frac{\partial^3 f(t,y)}{\partial t \partial y^2} + \frac{\partial^3 f(t,y)}{\partial y^3}\right) \\
&+ \left(\frac{\partial f(t,y)}{\partial t} + f(t,y)\frac{\partial f(t,y)}{\partial y}\right)\left(\frac{\partial f(t,y)}{\partial t}\right)^2 \\
&+ f(t,y)\left(2\frac{\partial f(t,y)}{\partial t}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y}f(t,y)\right)\right. \\
&+ \frac{\partial f(t,y)}{\partial y}\left(\frac{\partial^2 f(t,y)}{\partial t^2} + \frac{\partial^2 f(t,y)}{\partial t \partial y}f(t,y)\right) \\
&\left.\left. + \frac{\partial f(t,y)}{\partial t}\left(\frac{\partial^2 f(t,y)}{\partial y \partial t} + \frac{\partial^2 f(t,y)}{\partial y^2}f(t,y)\right)\right)\right|_{t=t_i} \\
&+ \mathcal{O}(h^5)
\end{aligned}
\tag{28}
$$

Next we need to Taylor expand the recurrence relation, that is, $y_{i+1} = y_i + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4$. i.e. we Taylor expand k_1, k_2, k_3, k_4 about h. k_1 is already explanded so we continue with k_2.

$$
\begin{aligned}
w_2k_2 &= w_2h\Bigg[f(t,y)+\frac{\partial f(t,y)}{\partial t}a_1h+\frac{\partial f(t,y)}{\partial y}b_1k_1 \\
&+\frac{1}{2}\left[\frac{\partial^2 f(t,y)}{\partial t^2}a_1^2h^2+2\frac{\partial^2 f(t,y)}{\partial t\partial y}a_1hb_1k_1+\frac{\partial^2 f(t,y)}{\partial y^2}b_1^2k_1^2\right] \\
&+\frac{1}{6}\left[\frac{\partial^3 f(t,y)}{\partial t^3}a_1^3h^3+3\frac{\partial^3 f(t,y)}{\partial t^2\partial y}a_1^2h^2b_1k_1+3\frac{\partial^3 f(t,y)}{\partial t\partial y^2}a_1hb_1^2k_1^2+\frac{\partial^3 f(t,y)}{\partial y^3}b_1^3k_1^3\right] \\
&+\frac{1}{24}\left[\frac{\partial^4 f(t,y)}{\partial t^4}a_1^4h^4+4\frac{\partial^4 f(t,y)}{\partial t^3\partial y}a_1^3h^3b_1k_1+6\frac{\partial^4 f(t,y)}{\partial t^2\partial y^2}a_1^2h^2b_1^2k_1^2+4\frac{\partial^4 f(t,y)}{\partial t\partial y^3}a_1hb_1^3k_1^3\right. \\
&\left.+\frac{\partial^4 f(t,y)}{\partial y^4}b_1^4k_1^4\right]\Bigg) \qquad (29)
\end{aligned}
$$

Now Taylor expanding k_3 gives:

$$
\begin{aligned}
k_3 &= w_3h\Bigg[f(t,y)+\frac{\partial f(t,y)}{\partial t}a_2h+\frac{\partial f(t,y)}{\partial y}(b_2k_1+b_3k_2) \\
&+\frac{1}{2}\left[\frac{\partial^2 f(t,y)}{\partial t^2}a_2^2h^2+2\frac{\partial^2 f(t,y)}{\partial t\partial y}a_2h(b_2k_1+b_3k_2)\right. \\
&\left.+\frac{\partial^2 f(t,y)}{\partial y^2}(b_2k_1+b_3k_2)^2\right] \\
&+\frac{1}{6}\left[\frac{\partial^3 f(t,y)}{\partial t^3}a_2^3h^3+3\frac{\partial^3 f(t,y)}{\partial t^2\partial y}a_2^2h^2(b_2k_1+b_3k_2)\right. \\
&\left.+3\frac{\partial^3 f(t,y)}{\partial t\partial y^2}a_2h(b_2k_1+b_3k_2)^2+\frac{\partial^3 f(t,y)}{\partial y^3}(b_2k_1+b_3k_2)^3\right] \\
&+\frac{1}{24}\left[\frac{\partial^4 f(t,y)}{\partial t^4}a_2^4h^4+4\frac{\partial^4 f(t,y)}{\partial t^3\partial y}a_2^3h^3(b_2k_1+b_3k_2)\right. \\
&+6\frac{\partial^4 f(t,y)}{\partial t^2\partial y^2}a_2^2h^2(b_2k_1+b_3k_2)^2 \\
&\left.+4\frac{\partial^4 f(t,y)}{\partial t\partial y^3}a_2h(b_2k_1+b_3k_2)^3+\frac{\partial^4 f(t,y)}{\partial y^4}(b_2k_1+b_3k_2)^4\right]\Bigg) \qquad (30)
\end{aligned}
$$

Lastly, expanding k_4 we find:

$$
\begin{aligned}
k_4 = w_4 h \Bigg[& f(t,y) + \frac{\partial f}{\partial t} a_3 h \\
& + \frac{\partial f}{\partial y}(b_4 k_1 + b_5 k_2 + b_6 k_3) \\
& + \frac{1}{2}\left[\frac{\partial^2 f(t,y)}{\partial t^2} a_3^2 h^2 + 2\frac{\partial^2 f(t,y)}{\partial t \partial y} a_3 h (b_4 k_1 + b_5 k_2 + b_6 k_3) \right. \\
& \left. + \frac{\partial^2 f(t,y)}{\partial y^2}(b_4 k_1 + b_5 k_2 + b_6 k_3)^2 \right] \\
& + \frac{1}{6}\left[\frac{\partial^3 f(t,y)}{\partial t^3} a_3^3 h^3 + 3\frac{\partial^3 f(t,y)}{\partial t^2 \partial y} a_3^2 h^2 (b_4 k_1 + b_5 k_2 + b_6 k_3) \right. \\
& \left. + 3\frac{\partial^3 f(t,y)}{\partial t \partial y^2} a_3 h (b_4 k_1 + b_5 k_2 + b_6 k_3)^2 + \frac{\partial^3 f(t,y)}{\partial y^3}(b_4 k_1 + b_5 k_2 + b_6 k_3)^3 \right] \\
& + \frac{1}{24}\left[\frac{\partial^4 f(t,y)}{\partial t^4} a_3^4 h^4 + 4\frac{\partial^4 f(t,y)}{\partial t^3 \partial y} a_3^3 h^3 (b_4 k_1 + b_5 k_2 + b_6 k_3) \right. \\
& + 6\frac{\partial^4 f(t,y)}{\partial t^2 \partial y^2} a_3^2 h^2 (b_4 k_1 + b_5 k_2 + b_6 k_3)^2 \\
& \left. + 4\frac{\partial^4 f(t,y)}{\partial t \partial y^3} a_3 h (b_4 k_1 + b_5 k_2 + b_6 k_3)^3 + \frac{\partial^4 f(t,y)}{\partial y^4}(b_4 k_1 + b_5 k_2 + b_6 k_3)^4 \right] \Bigg) \qquad (31)
\end{aligned}
$$

Now by matching the coefficients 29 with 30 - 31 with the Taylor series we get the following system of equations:

1. $b_1 = a_1$
2. $b_2 + b_3 = a_2$
3. $b_4 + b_5 + b_6 = a_3$
4. $w_2 a_1 + w_3 a_2 + w_4 a_3 = \frac{1}{2}$
5. $w_2 a_1^2 + w_3 a_2^2 + w_4 a_3^2 = \frac{1}{3}$
6. $w_2 a_1^3 + w_3 a_2^3 + w_4 a_3^3 = \frac{1}{4}$
7. $w_3 a_1 b_3 + w_4 a_1 b_5 + w_4 a_2 b_6 = \frac{1}{6}$
8. $w_3 a_1 a_2 b_3 + w_4 a_1 a_3 b_5 + w_4 a_2 a_3 b_6 = \frac{1}{8}$
9. $w_3 a_1^2 b_3 + w_4 a_1^2 b_5 + w_4 a_2^2 b_6 = \frac{1}{12}$

10. $w_4 a_1 b_3 b_6 = \frac{1}{24}$

Since we have 11 equations and 13 unknowns, two additional conditions must be given. Let $a_1 = \frac{1}{2}$ and $b_2 = 0$, then the system is solved as:

$a_1 = \frac{1}{2}, a_2 = \frac{1}{2}, a_3 = 1, b_1 = \frac{1}{2}, b_2 = 0, b_3 = \frac{1}{2}, b_4 = 0, b_5 = 0, b_6 = 1, w_1 = \frac{1}{6}, w_2 = \frac{1}{3}, w_3 = \frac{1}{3}, w_4 = \frac{1}{6}$. Subsituting these into

$$y_{i+1} = y_i + w_1 k_1 + w_2 k_2 + w_3 k_3 + w_4 k_4 \tag{32}$$

1. $k_1 = hf(t_i, y_i)$
2. $k_2 = hf(t_i + a_1 h, y_i + b_1 k_1)$
3. $k_3 = hf(t_i + a_2 h, y_i + b_2 k_1 + b_3 k_2)$
4. $k_4 = hf(t_i + a_3 h, y_i + b_4 k_1 + b_5 k_2 + b_6 k_3)$

gives 3.4.

Bibliography

[1] J. Fung and S. Mann, "Openvidia: parallel gpu computer vision." 2005. [Online]. Available: https://dl.acm.org/doi/10.1145/1101149.1101334

[2] J. Maldacena, *International Journal of Theoretical Physics*, vol. 38, no. 4, p. 1113–1133, 1999. [Online]. Available: http://dx.doi.org/10.1023/A:1026654312961

[3] O. Aharony, S. S. Gubser, J. Maldacena, H. Ooguri, and Y. Oz, "Large n field theories, string theory and gravity," *Physics Reports*, vol. 323, no. 3–4, p. 183–386, Jan. 2000. [Online]. Available: http://dx.doi.org/10.1016/S0370-1573(99)00083-6

[4] D. Garfinkle and L. A. Pando Zayas, "Rapid thermalization in field theory from gravitational collapse," *Physical Review D*, vol. 84, no. 6, Sep. 2011. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.84.066006

[5] A. Buchel, L. Lehner, and S. L. Liebling, "Scalar collapse in ads spacetimes," *Physical Review D*, vol. 86, no. 12, Dec. 2012. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.86.123011

[6] B. Cownden, N. Deppe, and A. R. Frey, "Phase diagram of stability for massive scalars in anti–de sitter spacetime," *Physical Review D*, vol. 102, no. 2, Jul. 2020. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.102.026015

[7] P. Bizoń and A. Rostworowski, "Weakly turbulent instability of anti–de sitter spacetime," *Physical Review Letters*, vol. 107, no. 3, Jul. 2011. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.107.031102

[8] J. Jałmużna, A. Rostworowski, and P. Bizoń, "Ads collapse of a scalar field in higher dimensions," *Physical Review D*, vol. 84, no. 8, Oct. 2011. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.84.085021

[9] D. Garfinkle, L. A. Pando Zayas, and D. Reichmann, "On field theory thermalization from gravitational collapse," *Journal of High Energy Physics*, vol. 2012, no. 2, Feb. 2012. [Online]. Available: http://dx.doi.org/10.1007/JHEP02(2012)119

[10] O. J. C. Dias, G. T. Horowitz, and J. E. Santos, "Gravitational turbulent instability of anti-de sitter space," *Classical and Quantum Gravity*, vol. 29, no. 19, p. 194002, Aug. 2012. [Online]. Available: http://dx.doi.org/10.1088/0264-9381/29/19/194002

[11] "Nvidia, c. u. d. a. "compute unified device architecture programming guide"." 2007. [Online]. Available: https://www.slideshare.net/slideshow/nvidia-cuda-programming-guide-10/10882167

[12] *Programming Massively Parallel Processors*, 3rd ed. Kaley Birtcher, 2017.

[13] V. Balasubramanian, A. Buchel, S. R. Green, L. Lehner, and S. L. Liebling, "Holographic thermalization, stability of anti–de sitter space, and the fermi-pasta-ulam paradox," *Physical Review Letters*, vol. 113, no. 7, Aug. 2014. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.113.071601

[14] B. Craps, O. Evnin, and J. Vanhoof, "Renormalization group, secular term resummation and ads (in)stability," *Journal of High Energy Physics*, vol. 2014, no. 10, Oct. 2014. [Online]. Available: http://dx.doi.org/10.1007/JHEP10(2014)048

[15] ——, "Renormalization, averaging, conservation laws and ads (in)stability," *Journal of High Energy Physics*, vol. 2015, no. 1, Jan. 2015. [Online]. Available: http://dx.doi.org/10.1007/JHEP01(2015)108

[16] N. Deppe, "Resonant dynamics in higher dimensional anti–de sitter spacetime," *Physical Review D*, vol. 100, no. 12, Dec. 2019. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.100.124028

[17] A. Buchel, S. L. Liebling, and L. Lehner, "Boson stars in ads spacetime," *Physical Review D*, vol. 87, no. 12, Jun. 2013. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.87.123006

[18] N. Deppe and A. R. Frey, "Classes of stable initial data for massless and massive scalars in anti-de sitter spacetime," *Journal of High Energy Physics*, vol. 2015, no. 12, p. 1–31, Dec. 2015. [Online]. Available: http://dx.doi.org/10.1007/JHEP12(2015)004

[19] A. Buchel, S. R. Green, L. Lehner, and S. L. Liebling, "Conserved quantities and dual turbulent cascades in anti–de sitter spacetime," *Physical Review D*, vol. 91, no. 6, Mar. 2015. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.91.064026

[20] S. R. Green, A. Maillard, L. Lehner, and S. L. Liebling, "Islands of stability and recurrence times in ads," *Physical Review D*, vol. 92, no. 8, Oct. 2015. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.92.084001

[21] P. Bizoń and A. Rostworowski, "Comment on "holographic thermalization, stability of anti–de sitter space, and the fermi-pasta-ulam paradox"," *Physical Review Letters*, vol. 115, no. 4, Jul. 2015. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.115.049101

[22] F. V. Dimitrakopoulos, B. Freivogel, J. F. Pedraza, and I.-S. Yang, "Gauge dependence of the ads instability problem," *Physical Review D*, vol. 94, no. 12, Dec. 2016. [Online]. Available: http://dx.doi.org/10.1103/PhysRevD.94.124008

[23] V. Balasubramanian, A. Buchel, S. R. Green, L. Lehner, and S. L. Liebling, "Balasubramanianet al.reply:," *Physical Review Letters*, vol. 115, no. 4, Jul. 2015. [Online]. Available: http://dx.doi.org/10.1103/PhysRevLett.115.049102

[24] "Runge-kutta methods," https://web.mit.edu/10.001/Web/Course_Notes/Differential_Equations_Notes/node5.html.

[25] https://docs.alliancecan.ca/wiki/Graham.