**Abstract**

All current ray tracing programs use straight lines to approximate the behavior of light in space. This is problematic, however, if one wants to visualize space that is not necessarily flat, such as near a massive object. Because of this, the goal of this project was to create a program that can simulate the paths (geodesics) of particles, light or otherwise, in curved spacetimes defined by the user with a metric tensor. For this to be possible, a CUDA program was first coded to solve the geodesic equations using the Runge-Kutta fourth order method across multiple GPU cores. This solved data is then passed to a C++ program that was created to act as a translator between Python and CUDA called by Python. The main Python code was made next to calculate initial values, run the C++ code, and calculate holes, splines, coordinate transformations, and collisions called by the user Python program. This program was coded to parse the input Jupyter notebook and json settings file into the metric tensor, surfaces, coordinate transformation functions, and program parameters. Next, a CUDA file writer was created to write to and compile the CUDA program with Christoffel Symbols derived from the metric tensor. The program could then run the main Python code and output a picture or plot. The resulting program was successful in its task and achieved the project’s goal. A few problems were encountered, like memory management, but were solved, resulting in a functional program, that accurately simulated curved spacetime.

**Introduction**

Since the inception of light ray tracing techniques in the late 1900s, many developments have taken place to allow it to be commonplace in the scientific, commercial, and personal communities. The process of ray tracing involves calculating the directions of linear light rays when they interact with different materials, such as glass, plastics, metals, etc. to allow for close-to-life visualizations of the digital world (Rademacher, 1990). These techniques are used in industries that rely on photorealistic visualizations, such as architectural firms, advertisers, movie studios, and others, and can be used for scientific research, especially in the fields of material science and optics. In the personal sector, tools such as Cycles, the built-in engine in Blender 3D, take advantage of ray tracing. The domain of ray tracing has only increased in recent years, including its implementation into many video games and the open-source CUDA toolkit provided by Nvidia with their recent real time ray tracing capable RTX graphics cards allow anyone to develop ray traced graphics. The pitfalls of ray tracing, however, are the computational strain it requires, and the fact that it only allows for light rays to be perfectly straight lines. The former is problematic because space, and time by extension, is curved.

The prospect of spacetime being curved given by Einstein in 1915 changed the view of physics from spacetime being flat, as shown by Newton in 1687, to spacetime being curved like a sheet of bent wax paper (Cook & Faller, 2019). This curvature can be caused by the presence of any mass or energy in the universe, including anything from subatomic particles to galaxies and electromagnetic fields to dark matter and dark energy. Einstein showed that spacetime can be modelled by a pseudo-Riemannian manifold or a Lorentzian manifold (which is defined by a specific metric tensor). A manifold, as defined in differential geometry, is a space with topology that is locally flat, or Euclidian. A Lorentzian manifold is differentiable and is defined in four dimensions, one temporal (time) and three spatial. A metric on a Lorentzian manifold generalizes properties of spacetime such as time flow, distance, volume, etc. A specific metric can be found using the Einstein field equations from general relativity, which are ten coupled, nonlinear partial differential equations using information gathered using a stress-energy tensor. This tensor consists of data on the mass and energy content of space that causes curvature (Perkowitz, 2019). The distance property of a metric tensor can be used to find the position of a particle over the curve’s arc length using the geodesic equations. A geodesic on a manifold is a generalization of the curve of shortest length between two points on the manifold, on which a particle will travel. The geodesic equations allow one to solve for this curve in four dimensions. A geodesic is categorized as timelike, lightlike, or spacelike based on the magnitude of the initial velocity of a particle. If the particle has a speed less than the speed of light, the geodesic is timelike, if the particle has a speed equal to the speed of light, the geodesic is lightlike, and if the particle has a speed greater than the speed of light, the geodesic is spacelike, but this is impossible. The equations are a set of four second order ordinary differential equations, or eight first order ordinary differential equations, which cannot be solved easily except in the most basic of cases. The geodesic equations use the Christoffel symbols, which describe parallel transportation along a manifold. Because of the complexity of solving the geodesic equations exactly, a numerical approach to integrate the equations is necessary (Lecture Notes on General Relativity, 2013), for which there are two methods that are relatively simple that can be used to solve such a problem, with the first being Euler’s method. This method is simple by only requiring a few calculations per step, but quickly accumulates error over a long interval, for each new step relies on the results of the last step. A second method that can be used is a Runge-Kutta fourth order method (RK4). This method uses four calculated values offset from the initial value and has a fixed step size. A Runge-Kutta method of any order uses a weighted average of multiple values given by the differential equation, which vastly reduces error over a long interval compared to Euler’s method. (Fourth Order Runge-Kutta, 2016) For the movie *Interstellar* (2014), the team at *Double Negative Visual Effects* produced a program that calculated geodesics for a spinning black hole in the movie called *Gargantua*. This was, at the time, the most accurate simulation of a spinning black hole that has been produced, and provided, for the general public, a glimpse into the workings of the universe. The visualization had been somewhat proven by the images produced by the Event Horizon Telescope in 2019 of the supermassive black hole at the center of Messier 87. (James, von Tunzelmann, Franklin, & Thorn, 2015)

There is currently no program that can take input of any spacetime metric, proven or arbitrary, and produce the geodesic curve of a particle given initial conditions. This is problematic because institutions that study the effects of general relativity on either micro or macro scales do not have the proper tools to simulate this phenomenon. Also, in the current state of physics, the connection between quantum field theory and the theory of general relativity has not been found, so such a program could help shine light on the topic. Given all previous information, the purpose of this paper was to describe the work done to create a program that can produce geodesic curves of a particle on a manifold using a user-given spacetime metric tensor.

The program was written using a combination of the Python and C++ coding languages using the Python libraries NumPy, SciPy, SymPy, MatPlotLib, and CPython, and the CUDA toolkit provided by Nvidia. The program took input of the specific spacetime metric and the type of simulation needed, either an output of light rays or particle paths. If the type was for light rays, the user inputted the desired parameters of the image, parameters for the RKF45 solver, and other information regarding the layout of the three-dimensional scene. If the type was for particle paths, the user inputted a list of particles to be solved, motion information for each particle, physical information for each particle (mass, size, etc.), and parameters for the RKF45 solver. The program then solved the geodesic equations accordingly using the RKF45 solver, calculated collisions, accelerations, velocities, positions, and forces, and outputted the results in the desired format. This paper will describe and explain more thoroughly the processes that needed to take place to create this program and will expand upon the importance of such a program.

**Purpose and Engineering Goal**

The goal of this project was to build a program that could utilize CUDA and different algorithms related to general relativity to calculate geodesics of particles on a Lorentzian manifold and find points of collision with surfaces. The purpose of this is to create a program that could be used in scientific research of curved spaces such as: black holes (spinning, charged, both, or neither), wormholes, Alcubierre drives, expanding spaces, etc. The metric tensor of the manifold, surfaces, and coordinate transformations from program coordinates to metric coordinates were user-defined in a Jupyter notebook file formatted in LaTeX. The program parameters were set through a json file, including number of solver steps, resolution, camera position, program mode, etc. If the mode was equal to zero, the program was in ray-tracing image mode, that is, it used camera data to output an image using collisions and geodesic paths. If the mode was equal to one, the program was in particle mode. This mode calculated the geodesics using a csv file containing the four-positions, and four-velocities of individual particles and outputted a three-dimensional plot and saved the solved data to a file.

**Materials and Methods**

The specific computer hardware that the program ran on for the finally-produced images and plots used a Ryzen 7 3800X (8-core, 16-thread), Nvidia RTX 2070 8GB (36 SM, 2304 CUDA cores), 32 GB DDR4-3000 RAM, and an NVMe SSD (4x PCIe 3). This hardware was necessary to run the program in a reasonable amount of time with the final testing cases used. The following programs were used to aid in the development process: Microsoft Visual Studio 2019, Anaconda 3, Spyder IDE, CUDA Toolkit 10.2, Jupyter Lab, and Microsoft Excel. The Python code used the following third-party libraries to run: ScyPy, NumPy, SymPy, MatPlotLib, and built-in libraries (time, math, cmath, os, json, csv). The C++ and CUDA code used the Python library from the Anaconda installation, CUDA-runtime, and built-in libraries (windows, cmath, string, sstream, iostream, fstream, chrono, vector).

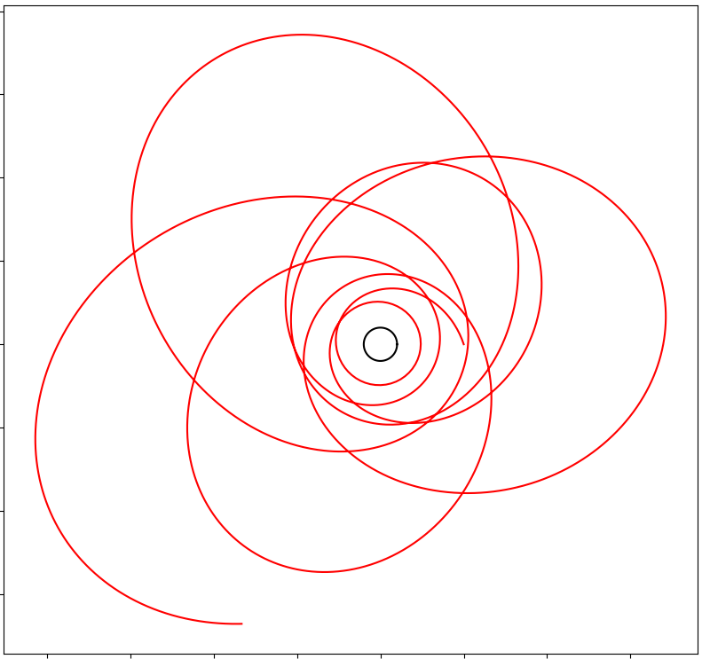
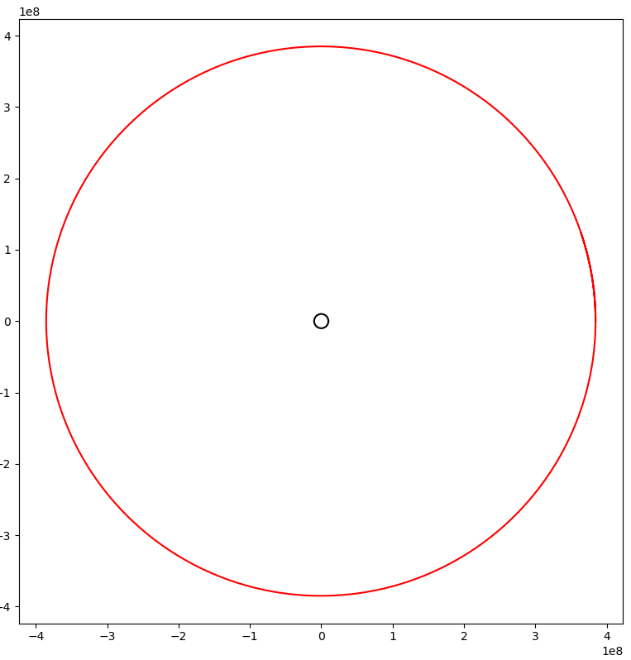
The methods that follow were undertaken to achieve the engineering goal. The CUDA code was written first, with the implementation of the Runge-Kutta Fourth Order (RK4) method and the geodesic equation parallelized on many threads on the GPU. This code took input from the C++ code and returned the solved data. As per the CUDA architecture, the GPU has a grid, which is split into blocks, which are then split into threads. On the RTX 2070 used for this project, each block can contain a maximum of 2048 threads, but the grid can have an effectively unlimited number of blocks. Next, the C++ code, working as a bridge between the Python and CUDA code, was written. It was written so that the main function could be called by Python with arguments that were Python objects. These objects were then converted to C++ types, such as vectors and double precision floating points and passed to the CUDA code. The results from the CUDA code were converted to Python objects and returned to the Python program. The Python code was separated into two parts, the main code and the user-accessible code. The main was written to fist calculate the initial positions and velocities of particles from the camera-related data such as: position point, point where camera is facing, up vector, resolution, and horizontal field of view or use the data from a csv file. The main code then runs the C++ code using these starting values. Next, undefined spaces, defined by the user as hole-type surfaces, are used in a collision solver to adjust the range of values returned from the CUDA code to exclude any nonexistent values. A normal surface collision solver was then written, and all solved interpolated splines and collisions are outputted. All surfaces were defined using a surface class, containing the surface function and other parameters. The user Python code was then written to be able to be ran from a command line. First, a Jupyter notebook reader was written to extract and parse the metric tensor, surfaces, and coordinate transformations. The program then used the metric tensor to calculate the Christoffel symbols, and, using a file writing interface, wrote these to the CUDA file and compiled if necessary. Second, any coordinate transformations were processed, and first order derivatives were calculated. Third, all program parameters were read from an input .json file and stored to the respective variables. Fourth, the Surface class instances were defined using the parsed data. Finally, the main code was run, the data was stored, and the collisions were used to output the final image or plot.

A picture containing guitar

Description automatically generatedA picture containing drawing

Description automatically generatedA close up of a logo

Description automatically generated**Data**

**Figure 1 (left).** Schwarzschild metric in spherical coordinates with a resolution of 200x200 and 500 solver steps. **Figure 2 (middle).** Schwarzschild metric in isotropic coordinates with a resolution of 100x100 and 300 solver steps. **Figure 3 (right).** Schwarzschild metric in isotropic coordinates with a resolution of 100x100, 300 solver steps, and at a different angle.

**Figure 4 (left).** Schwarzschild metric in spherical coordinates with a particle located at [5, 0, 0] and a velocity of [-0.1, 0.3, 0.0] with 4000 solver steps. **Figure 5 (right).** Orbit of the Moon around the Earth. The black circle is the Earth (enlarged to see clearly), the red path is the path of the Moon. Earth Schwarzschild radius: 0.00887 meters, Earth-Moon distance: 384.4 million meters, Moon tangential velocity: 1019 meters per second, speed of light: 299,792,458 meters per second, 20000 solver steps.

**Results**

All the engineering objectives for the program were met. The CUDA code was successful in the parallelization of calculating geodesics using the geodesic equation and the Runge-Kutta fourth order method. This code was able to be scaled effectively in accordance to the CUDA architecture, was efficient memory-wise, and was able to be run by both a GTX 1060 and an RTX 2070 without any errors. The C++ code worked effectively as a translator between Python and CUDA and was able to be called directly from Python as a library. Python objects were able to be passed as arguments and converted to C++ types to be passed to the CUDA code. The C++ code was then able to convert the return arrays from the CUDA code back to Python types and returned. The main Python code could calculate initial values, either from camera data or direct user input, and pass to the C++ library. The code could then calculate any holes in the geodesic paths using a user-defined surface, generate interpolated splines, transform coordinates, and calculate collisions with surfaces. A Surface class was successfully defined, containing methods to calculate holes and collisions. The user Python code was able to be run by the user of the program through a command line. The program could read and parse a Jupyter notebook file into the metric tensor, surface definitions, and coordinate transformations. The metric tensor was then used to generate the Christoffel symbols of the manifold, using that to write and compile the CUDA file if necessary. All parameters for the program were able to be read from a json file and the program could define any number of surfaces. The program was then able to pass all parameters and surfaces to the main Python code and get the return arrays. The return arrays were then able to be used to either plot geodesic paths or output an image, depending on the set mode. The ability of the program to use different coordinate systems is exemplified by the use of spherical coordinates in figure 1, and the use of isotropic rectangular coordinates in figures 2 and 3. Also, its ability to output both an image, as in the first three figures, and a plot, as in the last two figures, was validated. Overall, the project was very successful and achieved all goals.

**Conclusions and Discussion**

To conclude, the goal of coding a program that can utilize CUDA to calculate geodesics on a Lorentzian manifold with any user-defined metric using the geodesic equation and the Runge-Kutta fourth order method was realized. This program could be used in the research of general relativity, due to the possibility of very high accuracy on more powerful hardware, such as a supercomputer with multiple GPUs and CPUs.

Many different skills were learned so that this project could be successful. Regarding CUDA programming, proper memory management and the ins and outs of parallelization on the CUDA architecture were learned. To speak of the C++ code, the usage of the CPython API to expose C++ functions to Python and translate between Python types and C++ types. Also, effective management of Visual Studio projects and solutions was learned. For the Python code, proper writing of classes, json and csv file reading, Jupyter notebook parsing, usage of differential geometry methods from SymPy, running C++ libraries from Python, and producing images using MatPlotLib were learned. These skills were crucial to completing this project and could be used in a professional setting.

Problems were encountered in the course of this project, however. With the CUDA code, the main problem was related to how memory was allocated for objects such that objects on the host were not passed correctly to the device. This problem was solved by keeping the host and device memory spaces separate and copying memory as necessary. This also reduced the amount of memory used by the program. With the C++ code, there were problems in the compiling of the code with CUDA and CPython, but this was solved by adjusting compilation arguments. With the main Python code, there were problems in the detection of collisions and changing of coordinates. First, the collisions were not correctly identifying were the solutions did not exist, but this was solved by writing a function that detected any non-finite values. The return value of this function was then compared to the return value of the hole solver to produce an index to crop the solution arrays to real values. Second, the coordinate functions were not being parsed correctly, but this was solved by adjusting how the program detected what symbols corresponded to each coordinate. With the user Python code, there were very minor problems, but none that are noteworthy.

There are future considerations for this project. First, a more efficient or accurate differential equation solver could be used in place of the RK4 method. This could be the Runge-Kutta-Fehlburg adaptive method, an Adams-Moulton linear multistep method, etc. Second, the collision solver, spline generator, and coordinate transformer could be implemented for parallelization across all CPU threads using the Python multiprocessing library. Alternatively, these processes could be implemented in CUDA to allow for GPU parallelization for massive performance benefits. Third, in both the C++ and CUDA code, better memory management could be done by using more memory efficient methods and using smaller size data types, but at the cost of accuracy. Fourth, the collision solver could be written to use a calculated, continuous spline, rather than discrete points to determine points of intersection between a geodesic and a surface. With this, the surfaces could be written parametrically, in addition to implicitly, so that more complex surfaces could be easily represented, and the exact point of collision could be found. Fifth, a GUI interface, in addition to a command line interface, could be written to allow the program to be more user friendly. This could be done in Qt or similar systems. Lastly, more ray tracing centered features could be added, such as: shadow calculation, volumes, texturing, emission, lighting, meshes, etc. Texturing, shadows, lighting, and emission would allow a more defined way to show the effects of curved space, e.g. textures compressing and stretching. Volumes, such as accretion disks, could be modeled using particles interpolated from solutions of the geodesic solver. Meshes would allow for complex surfaces not easily modelled with mathematics to be used using connected triangles.

**Literature Cited**

Cook, A. H., & Faller, J. E. (2019, June 20). *Gravitational fields and the theory of general relativity*. Retrieved from Encyclopaedia Britannica: https://www.britannica.com/science/gravity-physics/Gravitational-fields-and-the-theory-of-general-relativity

*Fourth Order Runge-Kutta.* (2016). Retrieved from University of Münster: https://www.uni-muenster.de/imperia/md/content/physik\_tp/lectures/ss2016/num\_methods\_ii/rkm.pdf

James, O., von Tunzelmann, E., Franklin, P., & Thorn, K. S. (2015, February 13). *Gravitational lensing by spinning black holes in astrophysics, and in the movie Interstellar.* Retrieved from IOPScience: https://iopscience.iop.org/article/10.1088/0264-9381/32/6/065001/pdf

*Lecture Notes on General Relativity.* (2013, January 16). Retrieved from Columbia University: https://web.math.princeton.edu/~aretakis/columbiaGR.pdf

Perkowitz, S. (2019, April 10). *Relativity*. Retrieved from Encyclopaedia Britannica: https://www.britannica.com/science/relativity/Experimental-evidence-for-general-relativity

Rademacher, P. (1990). *Ray Tracing: Graphics for the Masses*. Retrieved from University of North Carolina: https://wwwx.cs.unc.edu/~rademach/xroads-RT/RTarticle.html

**Appendix A. Some details of LMGE (Lorentzian Manifold Geodesic Engine)**

*A.1. The Lorentzian metric and the geometry of spacetime*

A metric tensor, in its most basic form, is essentially a generalization of a dot product on a manifold. With this property, it can be used to find the distance between two points in much the same way that one could find the arc length of a vector-valued function between two bounds, such that

can be compared with

where

is the metric tensor in matrix form and is the line element form of the metric. The Minkowski metric for flat spacetime in line element form is which could be used to find the same path length as the vector-valued function method. The Schwarzschild metric for a spherically symmetric mass in a vacuum is given by:

or in isotropic coordinates:

*A.2. The geodesic equation and the Runge-Kutta fourth order method*

Using the action given by:

The principle of least action when fully simplified gives:

This equation is the geodesic equation, which minimizes the distance between two events in spacetime when solved given the initial four-position and four-velocity. The geodesic equation is a set of four coupled, nonlinear, second order differential equations that, when integrated twice, give the positions in each of the four coordinates for the parameter . Because of this, the geodesic equation can be related to the acceleration of the particle, and its derivative the velocity. These equations are very difficult to solve analytically, except in the most basic cases, such as the Minkowski metric (which the solution of the geodesic equation would be a straight line). So, in order to solve these equations in any spacetime case, a numerical method must be implemented: this is the Runge-Kutta fourth order (RK4) method.

The RK4 method can be compared to Euler’s method, which, in fact, is a first order Runge-Kutta method. In using Euler’s method, a single tangent line to a curve is calculated, using the differential equation that is being solved in the form:

where is the step size of integration. This method is very prone to accumulated error because every subsequent step uses the values of the last step. The RK4 method uses this concept of tangent lines to the curve, but instead of one line, the method uses four and a weighted sum. This method takes the form:

This method produces very little accumulated error, as opposed to Euler’s method, and can use fewer steps to produce the same degree of accuracy. As such, it was chosen to solve the geodesic equation, for it could be expanded to solve a system of second order differential equations using eight RK4 methods subsequently, for each second order differential equation can be split into two first order equations.

*A.3. Collision solver algorithm*

The collision solver is used to find if and where a geodesic calculated from the geodesic equation intersects with a surface defined by a function . If the surface is defined to be less than or equal to a number , and so that:

This solver is iterated through all indexes of in code and stops when the condition is held true. If the condition is not met at any index, then there is no collision. Some examples of surface equations are as follows: