Visualization of binary neutron-star merger simulations

Nathanyel Schut
12907995

Report Bachelor Project Physics and Astronomy

Size	15 EC
Conducted between	04-04-2022 and 12-06-2022
Institute	Anton Pannekoek Institute
Universities	Vrije Universiteit Amsterdam and University of Amsterdam
Faculties	Faculty of Sciences
Supervisor	Dr. Philipp Moesta
Daily Supervisor	Dr. Pablo Bosch Gomez
Second examiner	Dr. Samaya Nissanke

VU VRIJE UNIVERSITEIT AMSTERDAM

Universiteit van Amsterdam

Summary

This research presents a new three-dimensional data visualization tool in the form of a `Python` package, called `FieldVis`. `FieldVis` is designed to visualize numerical simulation data of binary neutron star merger remnants, but is also expandable to other types of data. The goal of `FieldVis` is to create fast, customizable and scientifically meaningful visualizations, in the form of plots or animations of streamlines for vector field data and volume renderings for scalar field data. Presented in this report are example plots of the magnetic field lines, density and the Bernoulli criterion, a criterion for investigating outflows, of the binary neutron star merger simulation data. The current version achieves loading times in the order of 10 seconds for still images of both available visualization types on an average personalized computer in a standard use case. Further developments on `FieldVis` can aim to improve stability of the code, variety in the types of visualizations and additional built-in support for other data types.

Samenvatting

In deze scriptie wordt de ontwikkeling van een computer programma voor data visualizatie omschreven. Het programma is bedoeld om een beeld te vormen van specifieke natuurkundige computer simulaties. Deze simulaties bootsen de omstandigenheden na van een samensmelting van twee neutronensterren, oftewel hele zware sterren die zijn uitgebrand en vervolgens zijn ingestort. Met dit computer programma kan dan onder andere het magneetveld en de dichtheid van deze sterren worden weergegeven in een afbeelding of een animatie. Het programma is geschreven in de programmeertaal `Python` en heeft als doel om snel nieuwe afbeeldingen of animaties te kunnen maken, zodat de gebruiker van dit programma hier niet lang op hoeft te wachten. Op een gemiddelde computer duurt het ongeveer 10 seconden om een afbeelding te maken en ongeveer 5 minuten voor een animatie, wat sneller is dan sommige andere data visualizatie programma's.

Contents

Introduction

Motivation

With the developments of computer hardware and software in the last century numerical simulations have started to play a progressively larger role in physics and astrophysics research. Nowadays, very complex simulations, where many different physical processes need to be taken into account in multiple dimensions, are possible thanks to the advancements made in numerical methods and computer hardware.

But with the increasing complexity of numerical simulations comes the challenge of visualizing the simulation data. While it can be sufficient to only look at the numbers themselves, in most cases it is crucial to create a rendered image or even an animation of the simulation data at hand to get an understanding of the underlying physical processes. Understanding jet formation, for example, requires a look into the shape and flow of both the present magnetic field and the matter around the compact object.

Hence, visualization tools have been developed to make the process of visualizing simulation or even observational data easier. While these visualization tools are capable of creating good looking visualizations, there are certain areas where they are lacking. An example of such a tool is `VisIt`, which is an open-source data visualization and analysis tool [1]. `VisIt` is capable of reading a large variety of data types and comes with a rich feature set. However, for the data types used in this research loading in data and creating visualizations was a very slow process, sometimes taking multiple hours. Another problem arised with creating streamline visualizations of magnetic fields. The final streamlines were very dependant of their initial starting positions, which made creating an animation of the simulation data in time a very time consuming process as new starting points had to be selected for each frame. So in order to create the desired visualizations for the data used in this research a new tool needs to be developed which can quickly create customizable and scientifically meaningful images and animations.

Theoretical background

In this section a theoretical background will be provided on the relevant fields in astrophysics and computational physics that show up in this research. First an overview of the physical processes will be given, after which the numerical simulations made prior to this work will be discussed.

While the general use of the desired visualization tool mentioned in the Motivation section can be quite broad, its use in this research will be limited to visualizations of binary neutron star (BNS) merger remnants. Neutron stars (NS) are among the most extreme objects in the universe having incredible densities, in the order of the density of atomic nuclei [6], and magnetic field strengths as strong as 10^{15} Gauss [5]. NSs remain an active topic in today's research with open questions like 'What is the relation between pressure and mass-energy density inside a NS (equation of state)?' [6], and 'Can neutron stars produce short gamma ray bursts (sGRBs)?' [7].

BNS merger events, the inspiralling of two NSs, provide a new aspect to NS research as these events allow us to probe the inside of a NS by measuring the gravitational waves coming from the mergers [11]. The inspiralling of the NSs is a result of a loss in angular momentum, caused by the emission of gravitational waves [12]. After merging, the remnant can follow

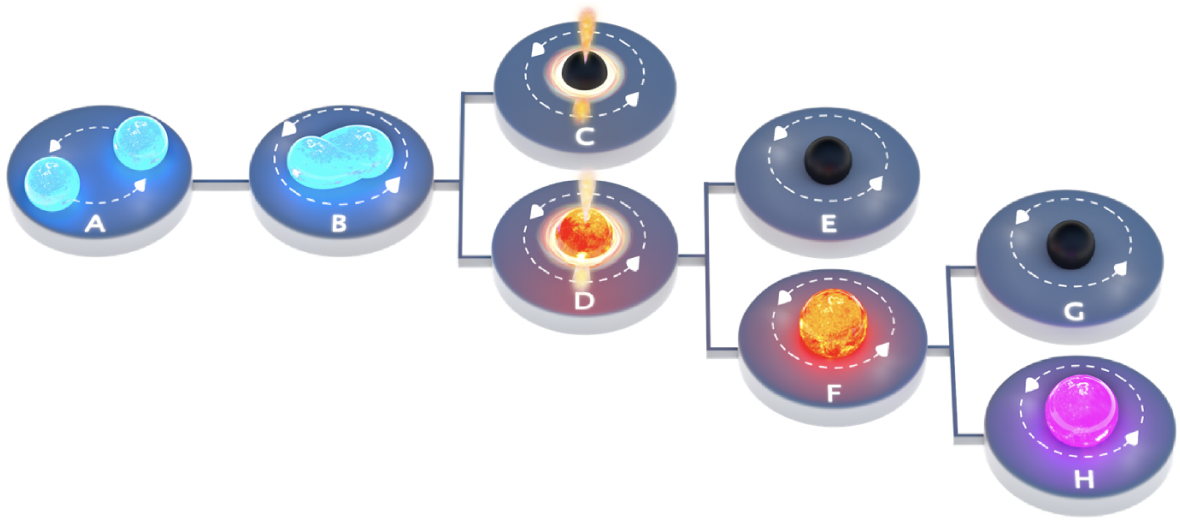


Figure 1: An overview of the different types of evolutions for binary neutron star mergers from [12]. The paths are as follows: direct collapse to black hole (A-B-C); formation of hypermassive neutron star and collapse to black hole (A-B-D-E); formation of supramassive neutron star and collapse to black hole (A-B-D-F-G); formation of stable neutron star (A-B-D-F-H).

different paths in its evolution depending on its mass and the assumed equation of state. The different paths include:

- direct collapse into a black hole;
- surviving the merger as a hypermassive neutron star (HMNS) before collapsing into a black hole;
- surviving the merger as a supramassive neutron star (SMNS) before collapsing into a black hole;
- surviving the merger entirely forming a stable neutron star.

The difference between the HMNS and the SMNS is the collapse timescales, with the latter taking far longer to collapse. An overview of the different evolution paths is provided in figure 1. HMNSs form the relevant group of merger remnants for this research.

As can be seen in diagram A and B in figure 1, it is believed that in some cases two collimated and opposite beams, called jets, can be formed after the neutron star merger. Current research suggests that the present magnetic field plays a crucial role in the emission of these outflows and so naturally visualization of the magnetic field and outflows is desirable [3]. Understanding the possible jets that can be formed by BNS merger remnants is essential for answering the question of whether hypermassive neutron stars can be engines for sGRBs. It is believed that jets can be the origin for sGRBs provided that these jets have sufficient energy and collimation [2, 3].

The numerical simulations providing the data for the visualizations in this research have been made using a so called General Relativistic Magnetohydrodynamics (GRMHD) simulation code. GRMHD simulation codes evolve general relativistic spacetime configurations containing both matter and magnetic fields in the ideal magnetohydrodynamics approximation. In this approximation fluids are assumed to have infinite conductivity and charge separation is

left out of the picture. For the simulations prior to this work a GRMHD code called `GRHydro` is used. `GRHydro` is a component for `The Einstein Toolkit`, an open-source community-driven software platform for computational tools, developed by Mösta et al. (2013) [8, 9].

The BNS merger remnant simulations use a technique called adaptive mesh refinement (AMR), where subvolumes of the total simulation volume that may be of greater interest have finer grids than the main grid of the total volume. A common application of AMR in the case of astrophysical simulations is to increase grid resolution for a number of subvolumes closer to and centered around the astrophysical object of interest. For a deeper read into the simulations used in this work see [10].

Scientific questions / Research objective

Since creating good looking visualizations is a rather subjective matter, the objective of this research will be to develop a tool with which a user can create custom visualizations of both vector fields by showing its streamlines and several different scalar fields by making use of volume renderings. The tool should meet three requirements which are as follows. Firstly, this tool has to be flexible enough that the user can make plots to their own liking. Changing aspects of the plots like background color or color maps should be intuitive. Secondly, the tool needs to be fast enough to provide a decent workflow. This means that creating a static rendering needs to take at most a minute and creating an animation of about 100 frames should take around 10 to 15 minutes. Lastly, it is important that scientifically relevant and accurate images can be generated to be useful for astrophysical research in general.

Approach

The approach for creating the desired visualization software is to create a `Python` package containing various function that can aid the user in easily creating the demanded images. The main advantages of using the `Python` programming language are that it's widely used in the research community, that it can be fast enough to provide the wanted workflow and that it is easily expandable.

To create the 3d renderings the new code will make use of an existing `Python` package called `PyVista`. `PyVista` is wrapper itself for `The Visualization Toolkit`, which is an open-source software designed for displaying three-dimensional scientific data [13, 14]. `PyVista` provides key features included in `The Visualization Toolkit` in a pythonic way with thorough documentation. While `PyVista` in itself is a useful package for visualizing three-dimensional simulation data, creating plots and animations can be quite complex and it lacks the intuitiveness that available visualization software like `VisIt` has.

Outlook

Looking ahead, the work done in this research will leave room for improvement and will aim to be easily upgradable. The visualization tool in this research will focus only on streamline plots and volume renderings, but other types of visualizations such as iso-surface plots or vector arrow plots might also be desired. Therefore, the code needs to be written in such a way, that a user is able to add these functionalities themselves.

Description of work and results

In this section the various stage of development of the python package for creating the visualizations of the BNS merger simulations, called `FieldVis`, will be described. The different stages include data processing, plot creation and animation. After discussing the challenges and methods in each of these stages, the structure of `FieldVis` will be explained. Finally, a few usage examples will be presented as well as an overview of the performance of `FieldVis`.

Data processing

The data processing required for visualizing the simulation data can be split into two parts. The first part consists of turning the raw data into more useable `Python` objects which in this case will be three-dimensional `NumPy` arrays. The second part of the data processing will be converting this `NumPy` array into a `PyVista` object. The key difference between these two parts is that the first is specific to this research, whereas the second part is a necessary general step. In this way users with differently structured data can still make use of this package's functionality, provided that they have converted their data to `NumPy` arrays themselves.

Data description
In order to describe the conversion of the raw data to `NumPy` arrays, one must understand the raw data and how it is generated. In this case the raw data is provided by `The Einstein Toolkit` and is stored in HDF5 files. HDF5 files have a hierarchical structure starting from the so-called *root group*, which can contain three types of *information sets*: HDF5 groups, HDF5 datasets and HDF5 datatype objects [4]. The data at hand only has HDF5 datasets in its root group, which are presented as an array variable that might have various data attributes describing the array variable.

Each dataset represents a cuboid-shaped subvolume of the total simulation volume, called a component, with dimensions of roughly 10 datapoints in each direction. The simulation regime is split into these components to parallelize the computation by distributing components to different computing cores. Between two neighbouring components a layer of overlapping edge grid points is required as boundary conditions may be needed to evolve a single grid point. These overlapping grid points are called ghost zones and they provide the necessary data communication between two computing cores.

The structure of the simulation data becomes a bit more complex when taking adaptive mesh refinement (AMR) into account (see Theoretical background). AMR is applied in this case by creating 5 different grids centered around the BNS merger remnant, with each grid having a different total volume and a different grid resolution. The levels of coarseness are called reference levels and are labeled with a number from 0 to 4 with 0 being the least fine grid and 4 the finest grid. Naturally, reference level 0 is simulated over the largest volume and reference level 4 is simulated over the smallest volume. Each reference level is split into 960 components giving a total of 4800 components per point in time.

The different time points, or iterations, of the simulation are then binned with each bin being represented by 120 data files. Each of these data files contains a certain selection of components for each reference level and each point in time of the corresponding bin. In the simulations done for this work every 512 iterations of the simulation the data points are given as an output. A visual representation of the data structure is provided in table 1.

File																															
														0																	
Iteration					0									...													39936				
Reference level		0			...			4				0		...				4				0				...				4	
Component	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	0	...	7	

File																															
														...																	
Iteration					0									...													39936				
Reference level		0			...			4				0		...				4				0				...				4	
Component	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	473	...	480	

File																															
														119																	
Iteration					0									...													39936				
Reference level		0			...			4				0		...				4				0				...				4	
Component	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	952	...	959	

Table 1: Overview of the hierarchical structure of example data produced by `The Einstein Toolkit`. The table represents a single time bin containing 120 files, of which three are shown in this table. As can be seen in this table each file has the same iterations and reference levels, but different components are present in each file. Iterations later than the ones shown in this table can be found in other time bins.

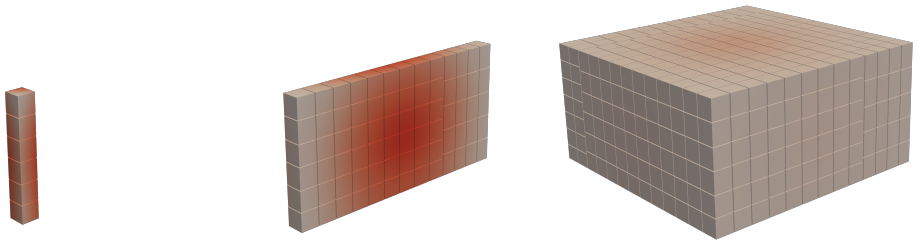
Conversion to `NumPy` arrays

The first step in converting the raw data files into `NumPy` arrays is to create new data files which contain all components for a single point in time instead of all time points for a selection of components. A separate `Python` script has been written for converting the raw files into the time sliced files. This script iterates over the separate raw files, and iteratively writes all components for a single time point to the corresponding time sliced file. By first converting the raw data to time sliced files, the process of creating a plot or animation becomes much faster as only a single file has to be read into memory instead of 120 files, which is necessary to provide a decent workflow for the user.

The time sliced files still have the data for each reference level split into components and so the next step would be to combine all 960 components per reference level and point in time into single datasets. In order to combine these components, their position in the total simulation volume has to be known. Consecutive components will first stack on top of each other in the z-direction until the maximum z-coordinate is reached. The next component will then move over the distance of a single component in the x-direction where a new stack in the z-direction is formed. When the maximum x-coordinate is reached, the next component will move back to the lowest x-coordinate but the distance of one component further in the y-direction. The structure of the components in the total volume is shown in figure 2. Before merging the components, the individual components are first converted to `NumPy` arrays and all the ghost zones are removed. By then concatenating the components in the right order the final `NumPy` array will be achieved.

Conversion to `PyVista` objects

As a final preparation for visualizing the simulation data, the data needs to be converted to a `PyVista` object so that the `PyVista` visualization functions can be applied to the data. The `PyVista` package contains a few types of data objects among which are PointSets, DataSets and various grid types. As the grids used in the BNS merger simulations are evenly spaced, the `PyVista UniformGrid` class is the most logical object type. To create a `UniformGrid` with a custom coordinate system, the grid dimensions, spacing and origin are required, which in this case are stored in the attributes of the HDF5 datasets representing the data components. After creating a grid, the vector or scalar field data can then be loaded into the grid.



(a) Rendering of the first 6 components.

(b) Rendering of the first 74 components.

(c) Rendering of the final cube containing all components 960.

Figure 2: Renderings of the first iteration of the magnetic field strength data of one of the binary neutron star merger simulations. In this figure the different stages of combining components into a single dataset are shown. In figure 2a, 2b and 2c the maximum z, x and y-coordinates are reached respectively.

Plot creation

This research focuses on two types of three-dimensional plots: streamline plots for vector field data and volume renderings for scalar field data. The general process of creating these plots will be explained in this section. For rendering the visualizations, the `PyVista Plotter` class will be used, which contains attributes and functions for creating the setting of the plot and rendering the final image.

For creating the streamline plots the `streamlines` function from `PyVista` will be applied to the `UniformGrid` containing the vector field data. The `streamlines` function creates a sphere of randomly ordered points, or seeding points, from which the vector field is integrated to create the streamlines. The number of seeding points and the radius and center point of the seeding points sphere can be passed to `streamlines` as arguments to customize the streamline plots. The streamlines are then stored as `PyVista PolyData` objects, which is a special dataset for containing surface geometry data such as lines and faces. Line data in the form of `PolyData` objects can be rendered as tubes using the `PyVista tube` function, which accepts the tube radius as an argument.

Volume renderings can be directly made from a `UniformGrid` containing scalar field data. The `UniformGrid` can be passed directly to a `Plotter` object along with a colormap and an opacity mapping, a mapping of transparency values to data values. `FieldVis` allows the option for both the volume and streamlines plots to be mirrored in the xy-plane. This option is included, since the BNS merger simulations have been done on only one half of the merger remnant, with the merger remnant sliced orthogonally on the spin axis. The other half should not provide additional information given that the merger remnant is symmetric in the xy-plane.

`FieldVis` aims to be a flexible tool, which means that plots should be easily customizable. To this end, FieldVis makes use of `Python` dictionaries to alter the settings of a plot. For each set of field data, a separate dictionary can be passed as well as a single dictionary for the general settings passed to the `Plotter` object. Example settings that can be changed in these dictionaries are the number of streamlines, the opacity mapping for volume renderings, colormaps and background color of the plot. An example of what such a dictionary might look like is shown in Appendix 1.

Animation

While creating animations is possible with `PyVista`, it is rather complex and the rendering speeds can be optimized. In this section the process of creating an animation in `FieldVis` will be presented and how it improves upon `PyVista`.

To create animations `FieldVis` renders each frame of the animation individually, after which a movie can be created from the individual frames using the open-source audio and video conversion software `Ffmpeg`. Currently the conversion with `Ffmpeg` is not included in `FieldVis`, and so this has to be done manually by the user in a command prompt. Before creating an animation, the user needs to specify the number of frames worth of data that is available, the path to the data, the settings per plot object, the general plot settings and finally a function object which takes as input the frame number and returns the data for a single plot object in a three-dimensional `NumPy` array. The `animator` function in `FieldVis` will then iteratively call the function object providing the data, process the data by either creating streamlines for vector field data or a `UniformGrid` for scalar field data and finally pass the plot objects to the `plotter` function from `FieldVis` along with the settings for the final image.

`FieldVis` makes use of the built-in `Python` package `multiprocessing`, a tool for parallel processing, which provides improvements on two fronts, speed and memory usage. By dividing the total number of frames over a number of different computing cores, the duration of creating all frames decreases by a factor equal to the number of available computing cores. Memory usage is also decreased by using `multiprocessing`, since all memory is cleared once a single process, in this case creating a frame, is finished. This is not always the case in `Python` due to the way its garbage collection system works. Performing the frame generation sequentially in a for loop might result into memory usage steadily increasing, which can ultimately lead to `Python` shutting down once the available memory is filled.

As an additional feature, `FieldVis` provides the ability to create an intro to the animations where the plot scene is shown from multiple positions with the camera moving smoothly between these positions. Such an intro can help the viewer understand the perspective better, as this can sometimes be difficult when dealing with two-dimensional projections of a three-dimensional plot.

FieldVis structure

The `FieldVis` package contains three modules, `et_reader`, `field_dp` and `field_plot`, that together provide the functionalities mentioned in the previous sections. For each module, its functions will be presented with a short description of their usage.

et_ reader
The design philosophy of `FieldVis` is that users themselves are responsible for providing their data in the right format, so that the usability of `FieldVis` isn't limited to a single data format. However, `FieldVis` does come with a built-in data converter for simulation data made with `The Einstein Toolkit`, in the form of the `et_reader` module.

The `et_reader` module comes with three functions: `ET_file_parser`, `ET_get_grid_info` and `ET_to_numpy`. `ET_file_parser` takes an HDF5 file object as an input and returns the name of the variable, as well as the index numbers for the iteration, time level, reference levels and components of the provided file. To obtain the attributes of a grid corresponding

to a certain reference level such as the grid spacing, number of ghost zones, dimensions and minimum and maximum coordinates the user can make use of the `ET_get_grid_info`. The function takes as an input the variable name, iteration, time level, reference level, a list of component indexes and an HDF5 file object and returns the aforementioned grid attributes. Finally, the `ET_to_numpy` function returns the full dataset of a single reference level in the form of a `NumPy` array. The user needs to provide the full paths to the files containing the data for a single iteration (three names are required for vector field data and a single name is required for scalar field data) as well as the reference level for which the data is requested.

field_dp
`field_dp` is responsible for the general data processing after the data has been processed to a `NumPy` array. The function `create_pyvista_grid` converts the `NumPy` array to a `PyVista UniformGrid`. Naturally, this function takes a `NumPy` array as input and has optional arguments for the grid spacing and origin. The two functions `get_streamlines` and `get_volume` call `create_pyvista_grid` and process the `UniformGrid` containing the field data further, by for example calculating the streamlines, taking the logarithm of the field data or mirroring the data in the xy-plane.

The remaining functions are all tools that can aid the user with the use of the other functions in `field_dp`. Two functions are provided for getting lists of files with `find_it_files` returning the files corresponding to a given iteration and `find_iterations` returning all iteration numbers as well as the full file names for a given variable. For both functions the name of the variable in the file names and the path to the directory containing the data files needs to be provided. Other remaining functions include `get_plot_object`, a function that calls the correct function for data processing (either `get_streamlines` or `get_volume`) based on the provided data and `custom_opacity`, a tool for generating custom opacity mappings and colormaps.

field_plot
The final module of `FieldVis` provides the functions for creating the three-dimensional plots and animations. Three functions are included for creating different types of visualizations and the rest of the functions are helper functions for these three functions. The main functions are `plot_slice`, `plotter` and `animator`.

`plot_slice` creates a two-dimensional plot of a slice of the field data provided using the plotting package `Matplotlib`. The user can change the coordinate and axis on which the data is sliced and can customize the plot with settings like the colormap and logarithmic scale. The `plotter` function handles the rendering of the three-dimensional plot scene and can return this in the form of an interactive plot window or a png file. `plotter` requires the plot objects as a `PyVista` object contained in a `Python` list as well as a list containing the plot object settings in the form of a `Python` dictionary. Optionally, another dictionary can be provided containing general plot settings. Finally, `animator` creates all frames of an animation and requires the user to provide a list containing the functions that convert the raw data to `NumPy` arrays, a list containing the plot object settings and the number of frames worth of data available. Optional settings include the name for a new directory where the frames will be stored and the number of cores to use for the frame generation.

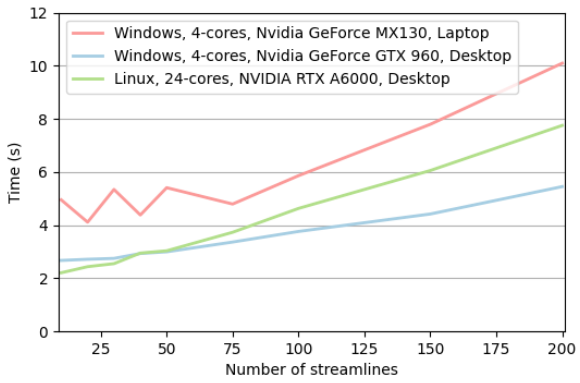
Performance

The overall performance of `FieldVis` is an important aspect of this research. Therefore, various performance test results will be presented in this section. These performance tests aim to provide a rough estimate of overall loading times for the general use of `FieldVis`. Performance will be tested by measuring the total run time for creating different plots on different machines.

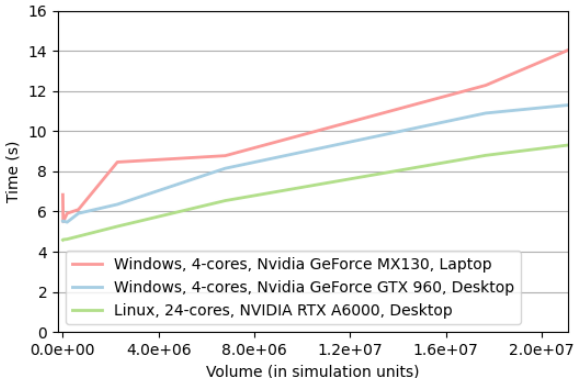
Plot types that were tested include still images of vector field streamlines and a scalar field volume rendering. The computers running the tests each have a different set of hardware to give a performance estimate for multiple levels of hardware quality. The weakest computer is a laptop running Windows 11 with an Intel Core i7-8565U quad core CPU and a Nvidia GeForce MX130 GPU. The next computer is a desktop running Windows 10 with an Intel Core i5-4690K quad core CPU and a Nvidia GeForce GTX 960 GPU. Finally, a very high-end computer, designed for data visualization, is included in the tests running Ubuntu 18.04.6 LTS with an AMD EPYC 7F72 24-core CPU and a Nvidia RTX A6000 GPU, which is roughly equivalent to a GPU node of a supercomputer.

For each visualization either the number of streamlines or the total rendered volume was varied to see how `FieldVis` scales with more difficult renders. The loading times for each value of the two variables were averaged over five different iterations. The remaining parameters that might affect loading times remained constant and were chosen to represent a standard use case of `FieldVis`. Each plot was rendered with a black background and a resolution of 800x1000 pixels, with the volume rendering done using the GPU. Field data values were mapped to a colormap, which is shown in a scalar bar. The data files containing the scalar field data or a component of the vector field data are HDF5 files of about 385 megabytes each.

The results of the performance tests are shown in figure 3. The graphs in figure 3a and 3b show a linear upward trend in loading times as a function of either the number of streamlines or the total rendered volume. In both tests the Windows laptop performs the worst, with the Windows desktop performing the best in the streamlines test and the Linux desktop performing the best in the volume rendering test. A possible explanation for the Windows desktop outperforming the Linux desktop in the streamlines test is that the Linux desktop is a shared workstation, which means that the full capabilities of the CPU might not be used. The fact that the Linux desktop still performs best in the volume rendering test can be explained by the fact that the volume rendering test relies more on GPU capabilities.

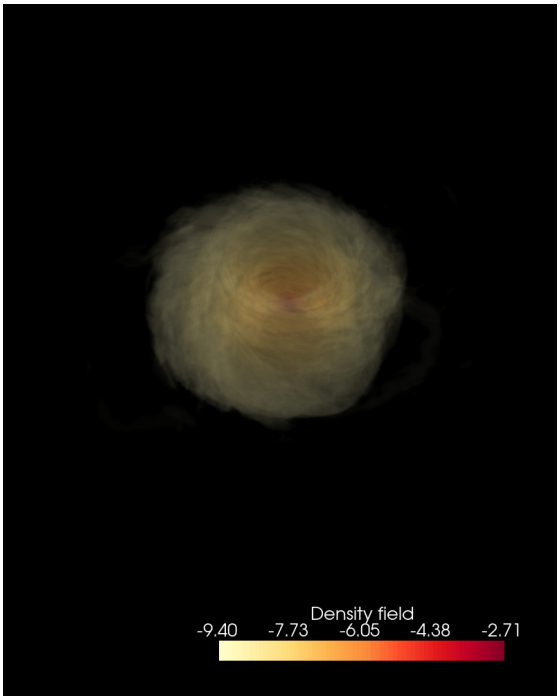


(a) Single streamline plot performance comparison. The x-axis shows the number of streamlines rendered.



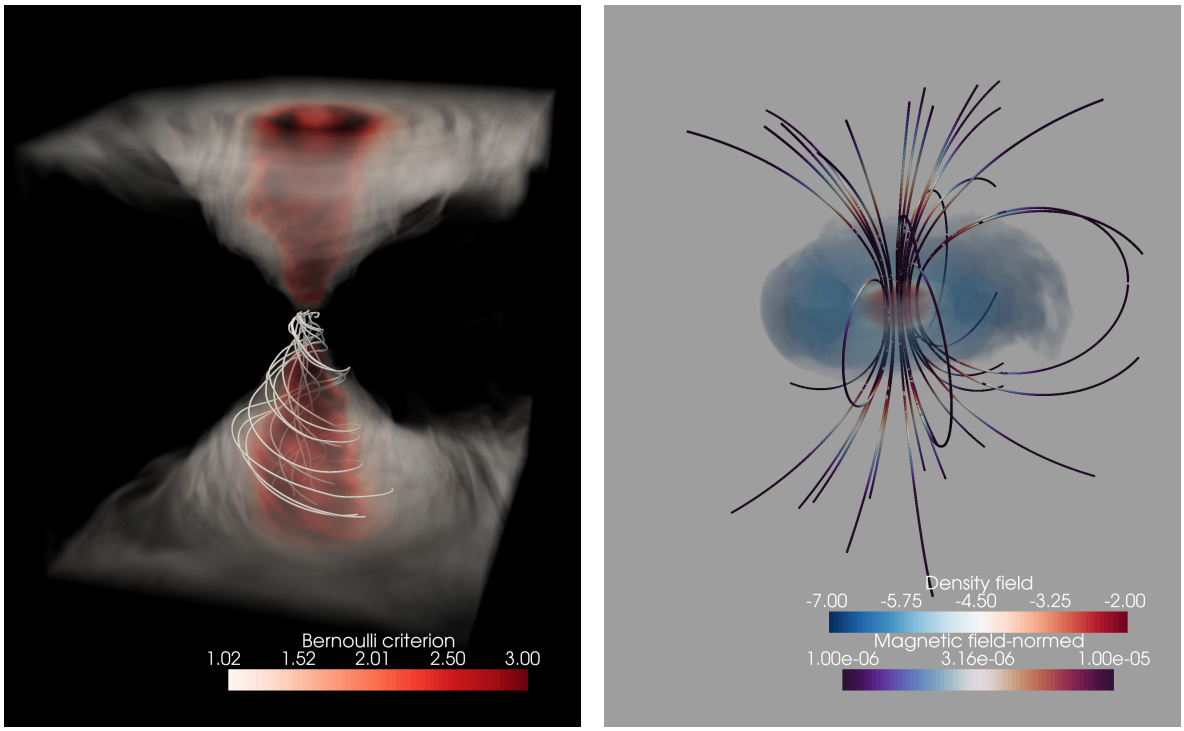
(b) Single volume plot performance comparison. The x-axis shows the total rendered volume.

(c) Example rendering of the streamline performance test. This rendering contains 40 streamlines.



(d) Example rendering of the volume performance test. This rendering contains a total volume of $2.3 \cdot 10^6$ in simulation units^3.

Figure 3: Performance comparisons of streamline and volume plots made in `FieldVis`. Three different machines performed the same renderings. The machine indicated in red is a laptop running Windows 11 with an Intel Core i7-8565U quad core CPU and a Nvidia GeForce MX130 GPU. The machine in blue is a desktop, also running Windows 10, but with an Intel Core i5-4690K quad core CPU and a Nvidia GeForce GTX 960 GPU. Finally, the machine in green runs on Ubuntu 18.04.6 LTS with an AMD EPYC 7F72 24-core CPU and a Nvidia RTX A6000. For both plots an image of 800x1000 pixels has been rendered with a black background and a scalar bar was included. Figures 3a and 3b show the loading times per number of streamlines and total rendered volume respectively, while figures 3c and 3d show example renderings made in the performance tests.



(a) Frame of an example animation visualizing the Bernoulli criterion and magnetic field lines of a binary neutron star merger remnant simulation made with `FieldVis`. In this plot the structure of two jets produced by the BNS merger remnant is visible as well as the helical structure formed by the magnetic field lines around a jet.

(b) Frame of an example animation visualizing the density field and magnetic field lines of a binary neutron star merger remnant simulation made with `FieldVis`. In this plot the structure of the accretion disk around the BNS merger remnant is visible as well as the initial poloidal magnetic field of the simulation.

Figure 4: Single frames of two example animations made with `FieldVis`.

Example visualizations

The motivation for the development of `FieldVis` was to create visualizations of binary neutron star merger remnant simulation data and so a few examples of such visualizations will be presented in this section. The examples will focus on the visualization of the magnetic field, density and the Bernoulli criterion. The Bernoulli criterion is formulated as $-h \cdot u_t > 1$, with $h = \left(1 + \epsilon + P + \frac{b^2}{2}\right)/\rho$ being the relativistic enthalpy and u_t begin the time component of the velocity four-vector [10]. If matter obeys this criterion, it is by definition unbound and so the criterion can be used to investigate outflows of the BNS merger remnant.

Single frames of the example animations can be seen in figure 4. Both animations as well as the code for generating them can be found at `https://github.com/nathanyelschut/Visualization-of-binary-neutron-star-merger-simulations`. In figure 4a a visualization is shown of the Bernoulli criterion and magnetic field lines of one of the BNS merger remnant simulations. Here the Bernoulli criterion is mapped to the 'Reds' colormap provided by `Matplotlib`. Streamlines are colored white and are only generated for one half of the plot to make the Bernoulli criterion more visible in the other half. A minimum and maximum threshold is applied to the Bernoulli criterion to filter out bound and loosely unbound matter

and to add more contrast in the colormap. The second example is shown in figure 4b and visualizes the density of the merger remnant as well as the accretion disk around it using a different color scheme than figure 4a. In figure 4b the initial poloidal magnetic field is shown in both halves of the plot with a colormap instead of a single color. The animation demonstrates the intro functionality of `FieldVis` by showing the scene from three different angles.

Discussion

The main benchmark for `FieldVis` will be the performance of the alternative visualization tool `VisIt`, as this tool has been used previously to generate visualizations of similar BNS merger remnant simulations. Both tools will be compared in this section on several quality indicators. As will become evident in this section, the tools can differ strongly for certain quality indicators and so only rough estimates for these indicators will be provided as this will be sufficient for creating an understanding of where each tool is better suited.

Firstly, loading times will be taken into account. For the simulation data used in this research, loading in a single HDF5 data file of about 390 megabytes in `VisIt` took around five minutes on the Windows 10 desktop mentioned in the Performance section. Performance for creating visualizations was much better, with loading times in the order of a few seconds. For the same data file, `FieldVis` was able to load the data and create a volume rendering of the data in around 9 seconds, depending on the total volume from the data that was rendered.

Secondly, the feature set will be compared. Whereas `FieldVis` is only limited to streamline plots, volume renderings and two-dimensional slice plots, `VisIt` comes with a wide range of visualization types, such as contour plots, iso-surface or iso-volume plots and vector arrow plots. `FieldVis` can be expanded to include such features, but this requires a decent understanding of `Python` coding and is not user-friendly.

Another quality indicator for data visualization tools is their versatility in data reading. `VisIt` claims to read over 130 different file formats [1]. In contrast, `FieldVis` provides built-in support only for a specific type of HDF5 data files, namely those provided by `The Einstein Toolkit`. However, `FieldVis` provides the ability for the user to submit their own data reader, meaning that `FieldVis` is accessible for anyone that can convert their own data to a `NumPy` array using `Python`. In the end, built-in support for many data types is still much preferred over the ability to provide a self-made data reader.

`VisIt` has been released in 2012 and so since then it has had the time to be updated several times, which means plenty of bug fixes have been done and the chance for new bugs to be present is low. `FieldVis` on the other hand is yet to be released to the public and so currently bugs are still likely to occur.

Finally, the quality of the visualizations will be discussed. Both visualization tools are capable of producing high-quality streamline and volume rendering plots. Example visualizations created with `FieldVis` are shown in figure 4 and similar visualizations created with `VisIt` can be seen in figure 6 of [10]. Both visualization tools allow for multiple volume rendering techniques. `FieldVis` has the 'fixed point' method for rendering using a CPU and the 'gpu' or 'open-gl' methods for rendering with a GPU, giving a higher quality render. `VisIt` allows for five different rendering options: a default rendering option and four ray-casting based options. The ray-casting options of `VisIt` provide high-quality renders, but at the cost of computing time.

Conclusion

In conclusion, the goal of producing a fast and scientifically useful visualization tool has been achieved, by creating a `Python` package called `FieldVis`. The package was developed for visualizations of binary neutron star merger remnant simulation data made in `The Einstein Toolkit`, giving built-in support for this data type. For other data types a user has to provide a data reader themselves, which can be easily passed to the `FieldVis` package. The visualizations supported by `FieldVis` include streamline plots for vector field data and volume rendering plots for scalar field data, with both visualizations being customizable in a user-friendly way using `Python` dictionaries.

Loading times for an average still image produced with `FieldVis` are in the order of 10 seconds on a personalized computer, giving an estimated loading time for an animation of 100 frames of around 5 minutes using 4 computing cores. With these loading times, `FieldVis` achieves its target loading times of one minute and 10 minutes for a still image and an animation respectively, which had been selected based on performance seen in the alternative visualization tool `VisIt`.

Currently, `FieldVis` has only been tested on four different machines and so instabilities such as bugs and errors might occur on other machines. Furthermore, `FieldVis` lacks in data reading support and variety in visualization types. However, the areas where `FieldVis` falls short are beyond the intended purpose of `FieldVis`, which is to create streamline and volume rendering plots of binary neutron star merger remnant simulations. Nonetheless, further developments for `FieldVis` can aim to improve where `FieldVis` is lacking to create a more mature visualization tool.

Acknowledgements

Finally, I would like to take this opportunity to thank dr. Philipp Mösta, dr. Pablo Bosch Gomez and Sebastiaan de Haas for their amazing effort in guiding me through the project. I also want to thank them for the warm welcome I received when I joined their research group to do this project.

References

[1] Hank Childs, Eric Brugger, Brad Whitlock, Jeremy Meredith, Sean Ahern, David Pugmire, Kathleen Biagas, Mark Miller, Cyrus Harrison, Gunther H. Weber, Hari Krishnan, Thomas Fogal, Allen Sanderson, Christoph Garth, E. Wes Bethel, David Camp, Oliver Rübel, Marc Durant, Jean Favre, and Návratil Paul. Visit: An end-user tool for visualizing and analyzing very large data. *Lawrence Berkeley National Laboratory*, 2012.

[2] Riccardo Ciolfi. Short gamma-ray burst central engines. *International Journal of Modern Physics D*, 27(13):1842004, 2018.

[3] Riccardo Ciolfi. The key role of magnetic fields in binary neutron star mergers. *General Relativity and Gravitation*, 52(6), 2020.

[4] Mike Folk, Gerd Heber, Quincey Koziol, Elena Pourmal, and Dana Robinson. An overview of the hdf5 technology suite and its applications. *Proceedings of the EDBT/ICDT 2011 Workshop on Array Databases - AD '11*, 2011.

[5] R. O. Gomes, S. Schramm, and V. Dexheimer. Modeling magnetic neutron stars: a short overview, 2018.

[6] J. M. Lattimer and M. Prakash. The physics of neutron stars. *Science*, 304(5670):536–542, 2004.

[7] William H Lee and Enrico Ramirez-Ruiz. The progenitors of short gamma-ray bursts. *New Journal of Physics*, 9(1):17–17, 2007.

[8] Frank Löffler, Joshua Faber, Eloisa Bentivegna, Tanja Bode, Peter Diener, Roland Haas, Ian Hinder, Bruno C Mundim, Christian D Ott, Erik Schnetter, et al. The einstein toolkit: a community computational infrastructure for relativistic astrophysics. *Classical and Quantum Gravity*, 29(11):115001, 2012.

[9] Philipp Moesta, Bruno C Mundim, Joshua A Faber, Roland Haas, Scott C Noble, Tanja Bode, Frank Loeffler, Christian D Ott, Christian Reisswig, and Erik Schnetter. Grhydro: a new open-source general-relativistic magnetohydrodynamics code for the einstein toolkit. *Classical and Quantum Gravity*, 31(1), 2013.

[10] Philipp Moesta, David Radice, Roland Haas, Erik Schnetter, and Sebastiano Bernuzzi. A magnetar engine for short grbs and kilonovae. *The Astrophysical Journal*, 901(2), 2020.

[11] David Radice, Sebastiano Bernuzzi, and Albino Perego. The dynamics of binary neutron star mergers and gw170817. *Annual Review of Nuclear and Particle Science*, 70(1):95–119, 2020.

[12] Nikhil Sarin and Paul D. Lasky. The evolution of binary neutron star post-merger remnants: a review. *General Relativity and Gravitation*, 53(6), 2021.

[13] C. Sullivan and Alexander Kaszynski. Pyvista: 3d plotting and mesh analysis through a streamlined interface for the visualization toolkit (vtk). *Journal of Open Source Software*, 4(37):1450, 2019.

[14] Graham Wills. Visualization toolkit software. *Wiley Interdisciplinary Reviews: Computational Statistics*, 4(5):474–481, 2012.

Appendices

Appendix 1

Example usage of plot settings dictionaries in `FieldVis`.

```
plot_settings = {
    'window_size': [1200, 2000],
    'position': (245.23, 840.74,
    -315.04),
    'focal_point': (-6.21, 11.91,
    35.24),
    'up': (-0.11, -0.36, -0.93),
    'background_color': 'w',
    'off_screen': True,
    'screenshot': 'Thesis_plot1.png'
}

volume_settings = {
    'spacing': [1.2, 1.2, 1.2],
    'origin': [-138, -138, -138],
    'cmap': 'jet',
    'clim': [1.02, 3],
    'mapper': 'gpu',
    'scalar_bar_args': {'use_opacity':
    False, 'color':'k'},
    'opacity': opacity,
    'n_colors': 256,
    'shade': True,
    'name': 'Bernoulli criterion',
    'mirror_z': True
}

field_line_kwargs = {
    'spacing': [1.2, 1.2, 1.2],
    'origin': [-138, -138, 0],
    'source_center': [0, 0, 0],
    'n_points': 30,
    'radius': 0.7,
    'source_radius': 9,
    'color': '#949494',
    'mirror_z': False
}
```

Listing 1: Example `Python` dictionaries for selecting plot settings in `FieldVis`. The variable 'opacity' is a linear opacity mapping defined outside the code that is shown. A full explanation on the possible settings is included in the documentation for FieldVis at `https://github.com/nathanyelschut/Visualization-of-binary-neutron-star-merger-simulations`.

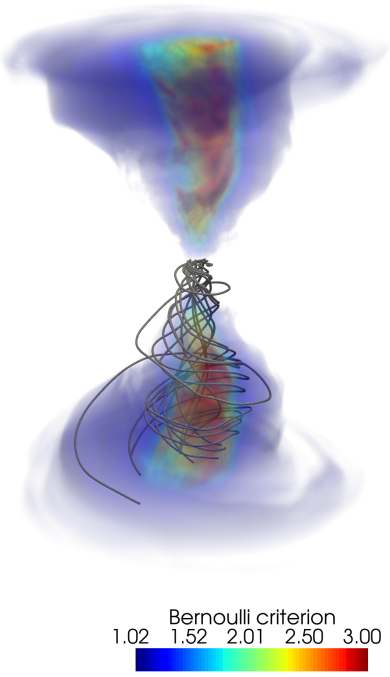


Figure 5: Output image corresponding to the settings shown in Listing 1.