January 21, 2019
Typeset using LaTeX **preprint2** style in AASTeX62

Processing JUPITER Hydrodynamics Simulation Data for Visualisation in Paraview.

Evert Nasedkin, Sup. Judit Szulágyi, Hans Martin Schmid[1]

[1]*ETH Zurich, Institute for Particle and Astrophysics*

ABSTRACT

At the present, no standard procedure exists for visualising the 3D outputs of JUPITER hydrodynamic simulations. Therefore we have developed a tool that converts the hydrodynamic fields calculated on a nested mesh into a VTK file format which can be visualised using standard open source software. We examine the use of Paraview, pythonic VTK and matplotlib for producing analyses and visualisations, and document procedures for producing publication ready figures.

1. INTRODUCTION

Hydrodynamic simulations are one of the tools used in understanding the formation and evolution of planets. They provide insight into the environment of the circumstellar and circumplanetary disks, allowing for the study of the accretion process and modelling the effects of planet-disk interactions. As observations lack the resolution to study small scale effects in circumplanetary disks or are hindered by the optical thickness of circumstellar disks, these models provide a unique means of studying these processes.

Modern techniques are 3-dimensional, and produce large datasets that require significant processing and analysis to be interpreted. This report examines the software developed to process JUPITER hydrodynamic simulation outputs to allow for visualisation using a variety of tools.

1.1. *JUPITER*

JUPITER is a 3D, nested mesh simulation program that solves the hydrodynamic and radiative transfer equations using a high order Godunov (1959) scheme described in Szulágyi et al. (2016, 2014); De Val-Borro et al. (2006). This allows a spatial resolution of about 0.8 Jupiter radii at the finest mesh level. The radiative transfer is calculated with the method of Commerçon, B. et al. (2011), with Dirichlet boundary conditions on the boundaries between mesh levels. Thus JUPITER solves for mass and momentum conservation, along with total energy, accounting for coupling between thermal and radiative energy (ϵ_{rad}). The governing equations for the program are therefore as in Szulágyi et al. (2016):

$$\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{v}) = 0 \tag{1}$$

$$\frac{\partial (\rho \mathbf{v})}{\partial t} + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) + \nabla P = - \rho \mathbf{v} \cdot \nabla \Phi + \nabla \cdot \bar{\boldsymbol{\tau}} \tag{2}$$

$$\frac{\partial E}{\partial t} + \nabla \cdot [(P\mathbb{1} - \bar{\boldsymbol{\tau}}) \cdot \mathbf{v} + E\mathbf{v}] = \rho \mathbf{v} \cdot \nabla \Phi - \rho \kappa_P c \left(\frac{B(T)}{c} - \epsilon_{rad} \right) \tag{3}$$

$$\frac{\partial \epsilon_{rad}}{\partial t} = -\nabla \cdot F_{rad} + \rho \kappa_P c \left(\frac{B(T)}{c} - \epsilon_{rad} \right) \tag{4}$$

Corresponding author: Evert Nasedkin
evertn@student.ethz.ch

The density is given by ρ, E is the total gas energy (U + K), $\mathbf{v}$ is the gas velocity, P is pressure, Φ is the gravitational potential and T is temperature. κ_P is the Planck opacity from Eqn. 6 and $B(T)$ is the thermal blackbody radiation power, given by $4\sigma T^4$. c is the speed of light and σ is the Stephan-Boltzmann constant. $\mathbb{1}$ is the identity tensor and $\bar{\boldsymbol{\tau}}$ is the stress tensor:

$$\bar{\boldsymbol{\tau}} = 2\rho\nu\left(\bar{\boldsymbol{D}} - \frac{1}{3}\left(\nabla\cdot\mathbf{v}\right)\mathbb{1}\right) \tag{5}$$

where ν is the kinematic velocity and $\bar{\mathbb{D}}$ is the strain tensor. The Planck Opacity is defined as in Bitsch et al. (2013):

$$\kappa_P = \frac{\int_0^\infty \kappa_\nu B_\nu\left(T\right)d\nu}{\int_0^\infty B_\nu\left(T\right)d\nu} \tag{6}$$

Finally F_{rad} is given by:

$$F_{rad} = -\frac{c\lambda}{\rho\kappa_R}\nabla\epsilon_{rad} \tag{7}$$

κ_R is the Rosseland mean opacity, and λ is a flux limiter to smooth the transition between optically thick and thin regions, also defined in Bitsch et al. (2013).

The equation of state of the system is taken to be

$$P = \left(\gamma - 1\right)\epsilon \tag{8}$$

where $\epsilon = \rho c_v T$ and the adiabatic index $\gamma = 1.43$.

JUPITER can then be supplied with a given set of initial conditions and system properties, and will calculate each of the hydrodynamic fields at each time step to explore planet-disk interactions and related physics. These must then be further processed for visualisation and study.

1.2. *Adaptive Mesh Refinement*

The nested mesh system used in JUPITER is an example of adaptive mesh refinement (AMR) techniques. In general, AMR improves computational speed for a given maximum resolution, as it reduces the number of mesh elements in the grid while maintaining the high resolution in the region of interest as seen in Fig. 1. A more complete treatment of adaptive or multigrid solvers can be found in Hockney & Eastwood (1988).

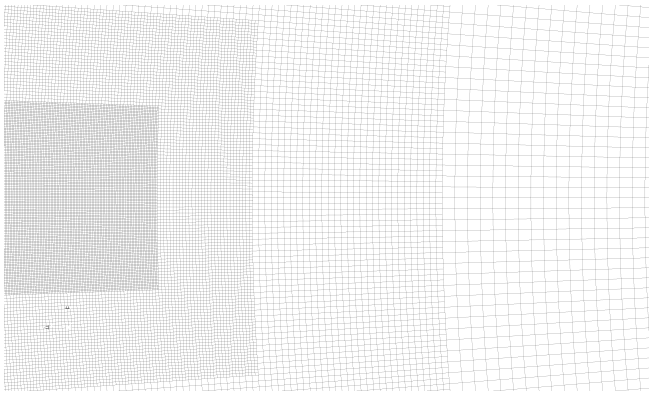

Figure 1. 2D projection of nested mesh levels around the location of the planet from a JUPITER simulation.

In JUPITER, the mesh is not truly adaptive, as the refinement regions are predefined in the region of the planet embedded in the circumstellar disk. Other AMR techniques could refine the mesh only in regions of steep density or velocity gradients. However, as circumstellar are not a static-flow environment, the location of these gradients changes over time, and it would be difficult to manage a constantly changing grid. In contrast, JUPITER uses a static grid, and allows the flow to evolve throughout the grid over time.

When using adaptive or multigrid solvers, the results of computations at one refinement level are used to provide the initial conditions for the next level to reduce computation time, and is shown in Fig. 2. Starting with the finest grid, a first solution is found. This solution is *restricted* to the overlapping region of the next finest grid. This process is repeated to the coarsest grid, where the full solution is calculated. These results are then *prolongated* down to the finer mesh, and again all the way back to the finest level, where a solution is achieved. This

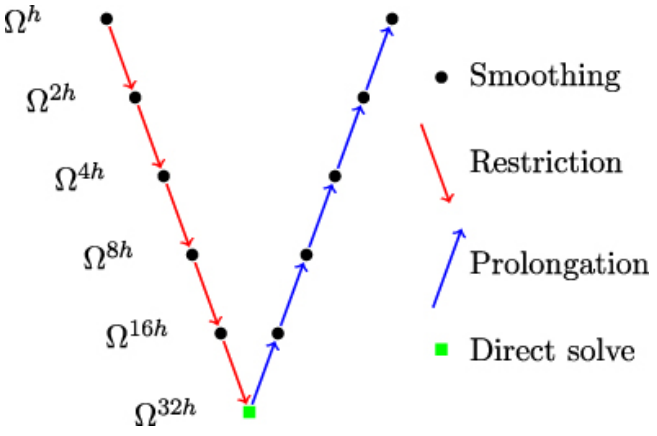

Figure 2. Example of a v-cycle for a 6 level multigrid Sampath et al. (2010).

process can then be iterated, and is known as the v-cycle Hockney & Eastwood (1988).

1.3. *Outputs*

As JUPITER relies on parallel computation, each computational core will output the hydrodynamic fields in its mesh region. These can be stitched together to create output files for each hydrodynamic field over the whole disk.

The first output file is a descriptor file that provides simulation properties, along with the positions in spherical coordinates of each of the vertices of the mesh for each mesh level. For this report ϕ represents the azimuthal angle, and θ the polar angle from the z-axis, and coordinates are generally ordered (ϕ, r, θ). The file is structured as follows:

```
[line 1]
[line 2]
[line 3]
[line 4]
[line 5]
[line 6]
[N_φ N_r N_θ]
[line 8]
[φ axis]
[r axis]
[θ axis]
```

This structure is then repeated sequentially for each mesh refinement level, from coarse to fine. The lines without descriptions are undocumented lines used within the JUPITER simulations, and are not relevant for this report. The first and last two components of each axis are ghost cells, and should not be read in. The number of components along each axis (e.g. N_ϕ) takes this accounts for this, however it assumes zero index counting. Thus if there are n ϕ components, the descriptor file will read $n-1$, and the total ϕ axis in the file will contain $n+4$ elements.

Each hydrodynamic field is stored in a data file, with a data point for each cell of the mesh. Note that while the descriptor file stores the locations of the vertices of the cells, all of the field data is defined at the barycentre of a given cell. The data is stored as binary doubles, and is ordered such that for a given coordinates (ϕ_i, r_j, θ_k), i is iterated the fastest and k the slowest. For vector data (velocities), each component of the vector is ordered as a scalar, with the azimuthal velocity listed first in its entirety, followed by the radial component, followed by the polar component. This ordering is also user adjustable in JUPITER, and may change. All data is stored in code units, and must be converted to physical units after read in.

Many of these hydrodynamic fields cannot be processed for visualisation with existing software (e.g. RadMC3D), so a conversion tool was necessary to allow for visualisation in Paraview.

2. PROCESSING TOOLS

All of the tools described on this section are available on Github: https://github.com/nenasedk/JUPITER_VTKFileConversion.

2.1. *VTK File Structure*

Paraview uses the VTK file format to store data for visualisation. This section describes the format as in Visual Toolkit Organization (2009). While a modern XML format exists, the legacy

format is simpler to use and has much more extensive documentation, and was thus chosen for this project. Future work could explore the usage of a VTK format developed for adaptive mesh grids.

The legacy VTK format is structured into 5 parts:

1. The file version and identifier, which must be exactly `# vtk DataFile Version x.x` For this project we are using VTK version 2.0.

2. The header, one line terminated by `\n` to describe the data.

3. File format. Either `ascii|binary`

4. Dataset structure. This describes the geometry and topology of the dataset, and consists of the word `DATASET` followed by a keyword description of the data. We use an Unstructured Grid to describe the data. An unstructured grid requires both a list of coordinates and a list of connections to fully specify the locations and connections of each cell in the grid.

   Following the keyword is the list of coordinates used to describe the data. This initiated by line `POINTS n datatype`, where n is the number of coordinates and datatype is a C type (we use doubles). The JUPITER coordinate grid is listed in order ϕ, r, θ, with ϕ iterating the fastest and θ the slowest. Each coordinate vector is converted to a Cartesian grid before being written to file, so the VTK file will contain the x,y,z components of each vertex in the mesh in CGS units, centred on the star. Each mesh level is appended subsequently from coarsest to finest.

   After the list of coordinates, the cells are listed. This section begins with the line `CELLS m` where m is the number of cells. Each row begins with the number of vertices of the cell, followed by the index of the coordinates of each vertex of the cell. We use a hexahedral cell type (cell type 8), and a description of the algorithm to compute the indices is found in Sec. 2. Although the JUPITER grid is based in spherical coordinates, for a small enough grid size a Cartesian grid with hexahedral cells provides a 'close-enough' approximation for practical purposes.

   Following the list of cells is the list of cell types for each cell. This is started by the line `CELL_TYPES m` where m is the number of cells. All of the cells used in JUPITER are hexahedral (type 12).

5. Dataset attributes. This can be either `POINT_DATA`, where each datapoint is located at a mesh vertex, or `CELL_DATA`, where each datapoint is located at the centre of a cell. All JUPITER hydro fields are `CELL_DATA`. Additionally, each datapoint can be either a `SCALAR` or `VECTOR`, the latter of which is used to store velocities.

An example file is included in Appendix A.

2.2. *Data Processing Methods*

Requirements

To run the python conversion script the user must have installed at minimum: Python 2.7, numpy, astropy and the python VTK package v8.1.1. It is recommended to use the environment included with the package on Github.

Overview

To process the outputs of the JUPITER simulations into the VTK file format, a python class was written. The class has a range of user settable functions, allowing the user to write a script to wrap the class for a given conversion job, or use the interactive command line interface written for the class. In general, the user must provide the input and output directories,

the output number, the number of mesh refinement levels, the hydrodynamic field to convert, the orbital radius of the planet in AU and the mass of the star in solar masses.

The other primary requirement of the tool was to be at most of O(n) time complexity, as the mesh can contain millions of elements. This restricts which algorithms can be used throughout the program.

The program flow is as follows:

1. Set-up directories and filenames, and set science variables.

2. Read in the coordinates from the descriptor file. From the axes read in, build a grid of 3D coordinates (ϕ_i, r_j, θ_k), and create an additional grid with the coordinates converted to Cartesian.

3. For each mesh level, define the boundaries of the next finest level. Cells (NOT grid points) within this region of interest (ROI) will be ignored.

4. Compute the indices of each vertex of each cell. This is achieved using a stride calculation in the `ComputeStructuredCell()` function:

   - Label the indices from 0 to 7, counter clockwise and from bottom to top as shown in Fig. 3.
   - The ϕ axis is iterated fastest and θ the slowest. Therefore index 1 will be one greater, but index 3 will be a full ϕ axis farther along than index 0. Likewise, index 4 will be an $r - \phi$ plane greater than index 0.
   - Iterate through each axis, incrementing the counters. Any cells with one or more vertices within the ROI will be skipped, but their index in the total list of cells is stored.

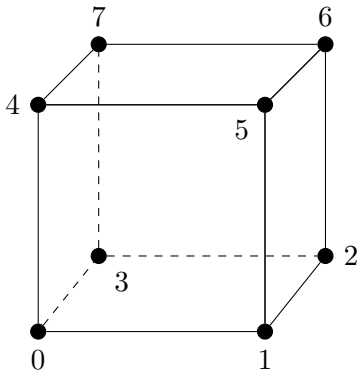

Figure 3. Labelling of hexahedral cell vertices. 0-1 lies along the azimuthal axis, 0-3 along the radial and 0-4 along the polar axis. Each vertex is labelled using the ordering of the grid read in from file, that is the index of vertex 3 is an length(azimuthal axis) greater than vertex 0, where the length of the axis is the number of points in the axis. Likewise, vertex 4 iterates over both the azimuthal and radial axes, so its index is length(azimuthal axis)×length(radial axis) greater than vertex 0.

5. Read in the data from the descriptor file, reshaping vector (velocity) data as necessary. Remove the datapoints with the indices of the removed cells.

6. Write the data to a binary or ascii VTK file. If the output file already exists, the user can choose to overwrite the file, or append new data to the existing file.

Usage

To use the command line interface simply use `python Convert.py`. Within the Convert.py script, the user can change whether to output to a binary or ascii VTK file, as well as the location of the input data from JUPITER. This data must be in a folder labelled outputXXXXX where the X's denote the simulation output number with leading zeros. The default path to this directory is the current working directory.

A second interface has been developed to allow for easier bash scripting. To use this, enter the following into the command line or a bash script:

```
python script_Convert.py [first]\
[grid level] [radius] [mass]
```

```
[-l [last]] [-v]\
[-b [binary/ascii]] [-d[dir]]\
[-f [field list]]
```

Or without all of the optional tagged arguments:

```
python script_Convert.py [first]\
[grid_level] [radius] [mass]\
[-f [field list]]
```

Tagged arguments can be placed in any order, but the field list must be the last argument passed.

- `[first]` The first simulation output to process.
- `[-l [last]]` The last simulation output to process. Assumes that all integers between first and last exist and should be processed.
- `[grid_level]` The mesh refinement level.
- `[radius]` The orbital radius of the planet in AU.
- `[mass]` The mass of the star in solar masses.
- `[-v]` Include if velocities should be planet centred.
- `[-b [binary/ascii]]` A string (b)inary or (a)scii to set the output file type. Binary is recommended, and is the default value.
- `[-d [dir]]` Directory where the output folder from JUPITER is located
- `[-f [field_list]]` A list of space separated strings denoting which hydro fields should be converted.

2.3. *Technical Challenges*

Several technical challenges needed to be overcome to implement this data processing tool.

Cell Counting—To visualise data in Paraview, cells cannot overlap, but the output from JUPITER contains overlapping cells within the ROI around the planet. This creates challenges when indexing the vertices of cells, as the axis length is not constant. However, Paraview does allow grid points to occur without being used. This allows us to store the full set of mesh points in the VTK file, and use a standard stride method with constant length axes to index cell vertices. Any cells with a vertex within the ROI are discarded, with the cell number being noted to allow for removal of the datapoint.

Velocity Vectors—The conversion of the velocity vectors from the initial data file to the VTK file was not straightforward. In the initial data file, the velocity vectors are stored component-wise, with the full list of azimuthal velocity components, followed by the radial and finally the polar velocities. These velocities have the azimuthal velocity of the planet subtracted, so we are in the co-rotating frame of the planet. These velocities are then converted from spherical to Cartesian coordinates using the following set of transformations:

$$\begin{aligned}\dot{x} = \; & \dot{r}\sin\phi\sin\theta + \\ & r\dot{\phi}\cos\phi\sin\theta + \\ & r\dot{\theta}\sin\phi\cos\theta\end{aligned} \tag{9}$$

$$\begin{aligned}\dot{y} = \; & -\dot{r}\cos\phi\sin\theta + \\ & r\dot{\phi}\sin\phi\sin\theta - \\ & r\dot{\theta}\cos\phi\cos\theta\end{aligned} \tag{10}$$

$$\dot{z} = \dot{r}\cos\theta - r\dot{\theta}\sin\theta \tag{11}$$

Where a dot denotes the time derivative of the coordinate. Note that the direction of the y component is inverted from the standard transformation.

3. VISUALISATION

3.1. *Paraview*

Paraview is an interactive visualisation program that allows the user to visualise large

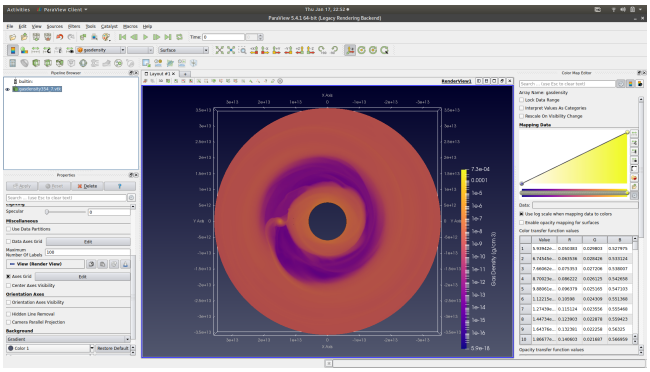

Figure 4. The Paraview 5.4.1 graphical user interface on an Ubuntu system.

datasets,with the processing done either in serial on a local machine or distributed on a server. As with the VTK file format, Paraview is an open source project primarily developed by Kitware, with additional funding and support from the Sandia and Los Alamos National Laboratories. While the full scope of Paraview's visualisation tools is extensive, and detailed in the user guide Ahrens et al. (2005), we will outline some of the common techniques relevant to this report.

The render view of the Paraview user interface is shown in Fig. 4. Once a VTK file has been produced by the conversion tool, it can be opened in Paraview, where the user may then select a suitable colour map for the data. Best practices should be followed when producing publication-ready visualisations. Colour hue and brightness should be considered to allow for accessible images that can convey the content of the data. A large range in brightness and hue will allow small variations in the data to be more visible. The luminosity should be consistent across the colour range, so as not to emphasise certain features and to maintain even perception of colour intensity. Many pre-made colour palettes are available in Paraview, including standard matplotlib palettes commonly used in python or Matlab visualisations, allowing for standardisation across various tools. Opacity scaling, and logarithmic colour scales are also useful tools when visualising complex data.

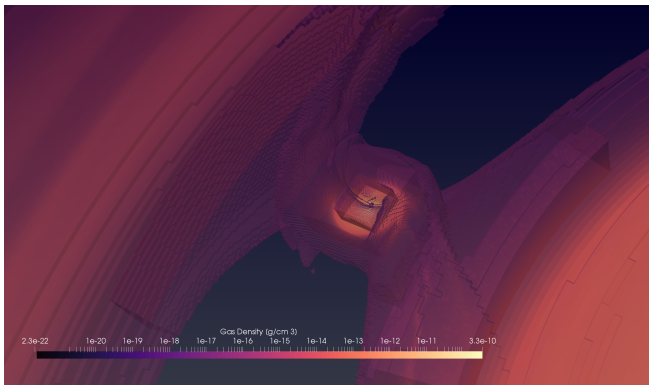

Figure 5. Example of the 3D density structure of the circumplanetary disk as rendered in Paraview.

In addition to colour, Paraview also allows the user to select various ways of representing the data in a 3D render. The default is to view the surface of the object, additional options can represent the data as points in space, a volume render, a wireframe or other options. These can be used to develop a better understanding of the 3 dimensional nature of the data, as shown in Fig. 5.

Following the initial rendering, **filters** can be applied to the data. Paraview currently includes around 120 different filters, several commonly used ones are summarised here.

Slice Extract a 2D slice from 3D data given an orientation and position.

Reflect Mirrors the data across a specified axis.

Clip Given an orientation and position, remove all cells 'outside' of the cut. To generate publication quality plots, the data should be clipped down to the region of interest, as the axes will only be generated over the extent of the data.

Threshold Remove all cells below a minimum, or above a maximum threshold.

Glyphs Visualise vector data as arrows or other objects for flow visualisation. Typically set to show every nth vector.

CellDataToPointData This filter shifts data from being located at the centre of cells to the vertices of the cell. This is required for plotting surface vectors or converting to numpy arrays.

Surface Vectors This filter projects a three dimensional vector field onto a 2D plane.

Stream Tracer Compute particle trajectories in a flow and visualise using streamlines. The user should set an initial point, with a small radius, and select whether to integrate forwards or backwards from that point (or both). For speed, an RK2 integrator is suggested. Typically only a small number of streamlines are necessary to visualise the flow. Used in conjunction with slicing and surface vectors, the stream trace filter can generate 2D streamline plots.

Tube Replace the streamlines generated by a Stream Tracer filter with 3 dimensional tubes, for easier viewing.

PlotOverLine Plots a scalar data variable along a path, such as the density along a radial line.

All of these filters can be combined in various ways to generate interesting and informative visualisations. Velocity vectors can be plotted over density fields; multiple slices can show the 3D structure of the temperature field. Additional tools include axis visibility, with customisable labelling, background settings (solid colour, gradient or image), alternate render views (slice view, charts, etc.) animations and more.

3.2. *Paraview Processing Example*

A typical example of a desired output would be a publication ready, 2D plot of a velocity vector stream on top of a scalar field such as the temperature. To produce this plot, read in both the temperature and velocity VTK data files. Slice the data along the desired axis to produce a 2D surface plot, and clip the data to the region of interest. Filter parameters can be copied and pasted, so identical filters can be applied to both datasets. Choose a colourmap and scale for the temperature field, and label the colourbar axis. Ensure the axis are visible, properly labelled and with a reasonable font size chosen. Apply the CellDataToPointData filter to the sliced and clipped velocity data, followed by the surface vector filter. Finally, generate a stream plot, and colour the streams with the magnitude of the velocity. Ensure the render background is white, and position the colourbars where desired. Save the scene in the desired file format, or export as a png screenshot. Note that both of these output options may not exactly reproduce the render view, so check the final output to ensure consistency. The output of this procedure should resemble Fig. 12.

3.3. *VTK Python*

An alternative to Paraview that allows for automation of plot generation is the python wrapper for the C++ based VTK library Avila & Kitware (2010). The python VTK package maintains the same functionality as Paraview, and produces similar results as it uses the same rendering engine. Fig. 6 outlines the process of generating a render using the VTK package. A VTK file is read in with a `reader` module. Each variation of the VTK grid structure (rectilinear, unstructured, XML, etc.) has its own reader, so the reader must match the VTK filetype being read in. The reader provides access to all of the data stored in the VTK file such as scalar or vector fields, along with the coordinate grid. It also details many of the properties of the file and the data stored within, including the number of points, the names and lengths of each data field, header information and more.

The data from the reader is typically accessed with the `reader.GetOutput()` function,

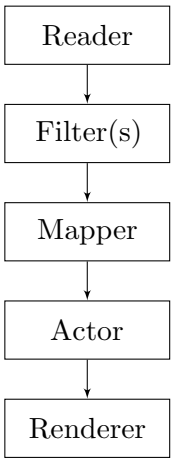

Figure 6. The pipeline for processing VTK data in python.

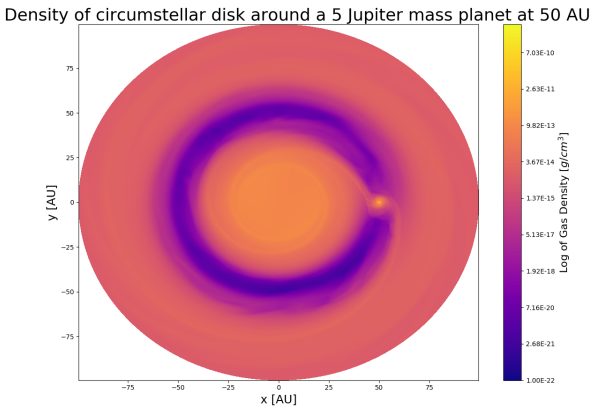

Figure 7. Matplotlib style figure of the density of a circumstellar disk.

which can be passed to filter objects for processing. Note that some filters use the function `reader.GetOutputPort()` instead. All of the filters available in Paraview are available in the VTK package. It is also possible for the user to write their own filter classes, though that is beyond the scope of this report.

After the data has been passed from the reader to the filter, it is applied using a Mapper object. This class maps the data to graphics primitives to allow for rendering. Any object to be rendered must be passed through a mapper class. The Mapper object is then passed to an Actor, which specifies the geometry and properties for the render. All Actors are then passed to the Renderer, which outputs a visualisation of the data. At each level of this pipeline, various settings are available with class functions to adjust the output of the render, for example changing the colourmap or camera position. The full documentation is available in Avila & Kitware (2010).

The use of the VTK package was explored as a means to reliably produce publication ready plots, and allow for some level of automation and reproducibility to increase the speed of generating plots. However, scripts must be tailored for specific datasets, reducing the utility of templates, and the lack of a visual reference leads to difficulties in producing the desired results. It was found that the GUI of Paraview provided a faster workflow and easier methods for generating high quality renders.

As a third alternative, the use of numpy and matplotlib was explored, as plots generated with these tools are effectively the standard in the scientific community. The use of these tools also relies upon the VTK python library, and an understanding of the VTK pipeline remains a necessity.

3.4. *Plotting with Matplotlib*

As matplotlib style plots are relatively standard, we developed a python program to take in the VTK files and output matplotlib plots. The data is read in and processed as described in §3.3 above. The data must be sliced and cut using the VTK filters prior to conversion. Once filtered and passed to a `CellDataToPointData` mapper, the mapper output can be converted to numpy arrays using the `vtk_to_numpy` function. The point coordinate grid can likewise be converted to a numpy array, and read into Cartesian axes.

We found that the function `tricontourf()` produces the best outputs for scalar fields, as many other plotting functions require 2D pixel grids for each axis. With millions of coordinate

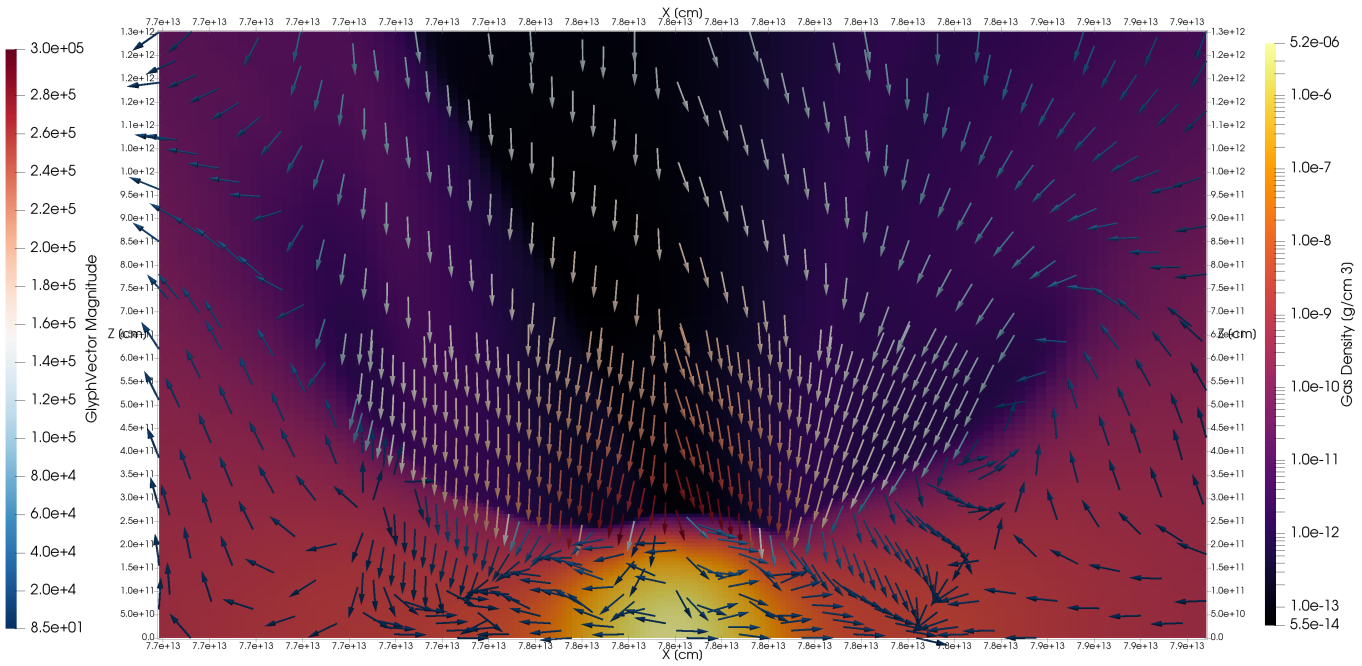

Figure 8. Vertical inflow of gas shocking near the surface of the accreting 1 M_J planet, followed by the outflow of gas in the circumplanetary disk.

points and an irregular grid, this quickly grows larger than the available memory on a consumer-grade computer. `Tricontourf` uses Delauney triangulation to create a grid from 1D coordinate axes, which can be irregular. The data is then mapped over the grid, and plotted with a filled contour. The resolution of the colourscale can also be specified. An example of the matplotlib output can be seen in Fig. 7.

However, this style of plotting also has its limitation. Vector plots are difficult, and do not produce insightful results. Stream plots are currently impossible due to the above mentioned memory issues - this is a known issue with the matplotlib Streamplot function. Therefore, while useful for generating simple plots in a standardised format, matplotlib remains less useful for more complex visualisations.

3.5. *Results*

With the conversion tool developed, we can present several examples of possible visualisations made using Paraview Ahrens et al. (2005). In this section we discuss the scientific results of a set of JUPITER simulations.

These simulations primarily consist of 100AU radius circumstellar disks with a single embedded planet of varying mass at 50AU. The gas in the simulations is considered to be ideal, with an equation of state and adiabatic coefficient as in Eqn. 8 The kinematic viscosity of the simulations is low, with a value of $10^{-5}a^2\Omega_p$, or $\alpha \approx 0.004$ at R_p. The density is close to that of a Minimum Mass Solar Nebula Hayashi (1981). The initial disk aspect ratio was set to H/r = 0.05, where H is the pressure scale-height of the disk. The upper surface of the disk is radiatively heated from stellar flux. For the opacity calculation, the dust-to-gas ratio of the disk was set to the standard value for the interstellar medium of 0.01.

3.5.1. *Circumplanetary disk flow*

During planet formation, the planet attracts gas and dust from the local environment. The JUPITER simulations show that this flow of ma-

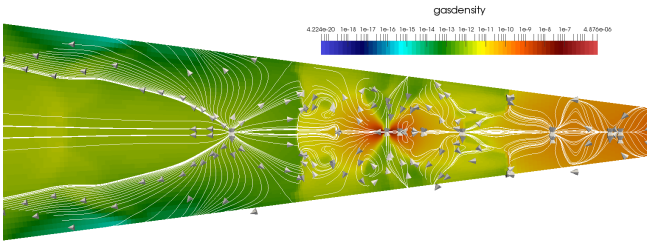

Figure 9. Meridional recirculation of gas from the CPD back into the circumstellar disk.

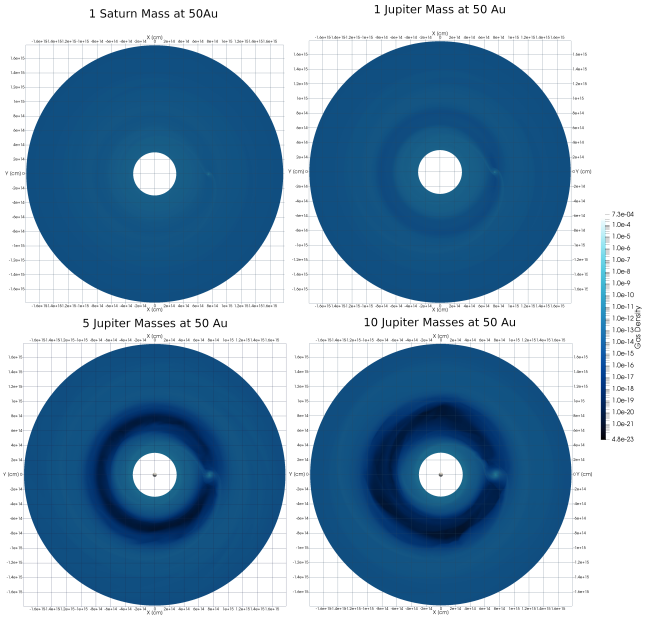

Figure 10. Variation of gap widths and depths over a range of embedded planetary masses. The colour scale is in units of g cm^{-3}. From 1 M_J to 10 M_J, the gap width increases by a factor of 2, while the depth decreases from $8{\times}10^{-16}$g cm^{-3} to $5{\times}10^{-20}$g cm^{-3}.

terial towards the planet is primarily from the vertical directions, Fig. 8. Gas from the edges of the gap where the CPD is embedded flows accelerates vertically towards accreting planet. As the gas in the region of the planet is adiabatically compressed, it heats up and forms a high pressure region surrounding the planet. The inflowing gas shocks on the upper boundary of the CPD near this region, forming a high temperature discontinuity, and losing most of its velocity Fig. 12. Some of the material is then accreted onto the planet, while the majority flows out-

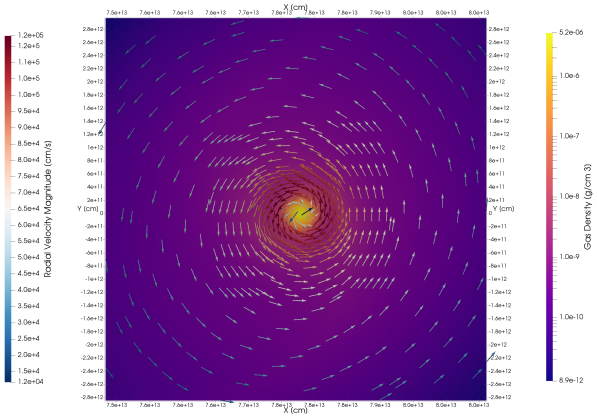

Figure 11. Radial outflow of gas through the CPD, planet mass is 1 M_J located at 5.2AU.

ward in the CPD, Fig. 11, before being recycled into the circumstellar disk, Fig. 9. The strength of this shock, along with the sharpness of the density discontinuity of the CPD and the gap in general will vary with both the planet mass and initial temperature. An exploration of the structure of CPDs can be found in Szulágyi et al. (2016).

3.5.2. *Dust traps and implications for satellite formation*

Visible in Figs. 8 and 12, there are regions where inflowing gas from the circumstellar disk and outflowing gas from the CPD meet. Inflowing gas from the gap walls flows over this high pressure region and the CPD, forming a vortex at approximately 20 planet radii from the planet. This forms a circular ring that acts as a dust trap, providing an ideal location for satellite formation. According to Cilibrasi et al. (2018), the dust-to-gas ratio in the dust trap can increase to 1 or greater. While beyond the scope of this report to analyse this phenomena in detail, it is discussed at length in Cilibrasi et al. (2018).

3.5.3. *Gap profiles*

In addition to the flow analysis, we can also examine the gap properties that arise from planet-disk interactions. In Fig. 10, we see that both the gap widths and gap depths increase as the

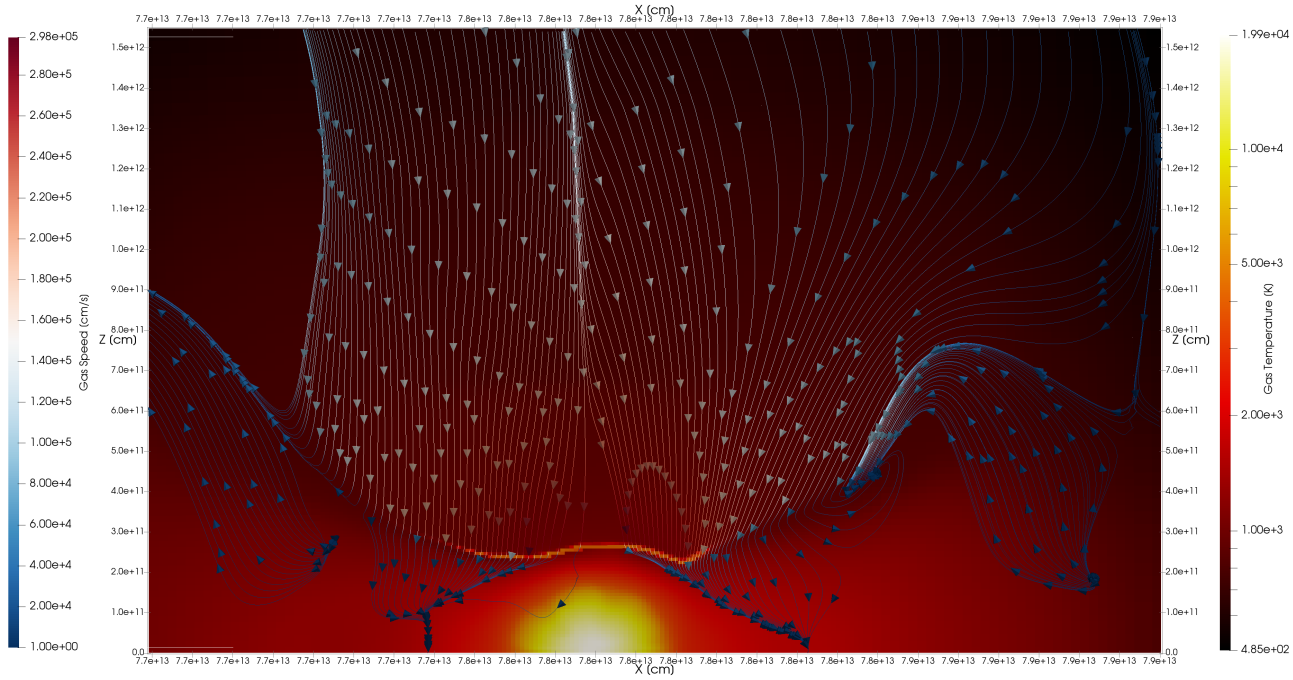

Figure 12. High temperature shock near the surface of the accreting 1 M_J planet. After shocking upper boundary of the CPD, the inflowing gas loses some of its energy and spirals out in the CPD.

planet mass increases. Comparing the 1 M_J simulation to the 10 M_J, the gap width increases by a factor of 2 while the depth decreases from 8×10^{-16}g cm^{-3} to 5×10^{-20}g cm^{-3}. Additional structure such as spiral arms or gap asymmetries also become more visible in the density profile of the disk as the planet mass increases.

However, these properties are also highly dependant on the radial position of the planet, and are also highly viscosity dependant. Other studies have shown that planetary properties can be derived from observed disk features.

4. CONCLUSIONS

We have successfully developed a data processing tool to allow JUPITER simulations results to be visualised using Paraview or other VTK software. This will allow further exploration of the physics of circumstellar and circumplanetary disks, as the increasingly complex 3D results require adequate visualisation to understand and study.

Various visualisation and analysis techniques have been examined, including Paraview, Python VTK and matplotlib. Each tool has its own strengths and weaknesses, and the choice of visualisation tool should depend on the desired outcome.

In general, we recommend the use of Paraview both as an exploratory tool to examine 3 dimensional data, but also to produce high quality renders. Python VTK is useful for the generation of simple, repetitive plots from a consistent dataset, and allows for a degree of automation. Finally, matplotlib should be used to generate plots that maintain a consistent aesthetic with other figures in scientific works.

We also present a qualitative analysis of several simulations. We note the flow patterns in 3 dimensional simulations are inconsistent with traditional 1D planet formation models, but may represent a more realistic scenario of gas flow. We also demonstrate that planet properties will have an effect on the observational properties of protoplanetary disks, though the derivation of planetary properties from disk features is not a subject of this report.

5. ACKNOWLEDGEMENTS

I would like to sincerely thank my supervisor Dr. Judit Szulágyi for her guidance, mentorship and patience during this project.

Thanks as well to Prof. Hans Martin Schmid for allowing me to be a part of the Star and Planet Formation group at ETH Zürich.

I would also like to acknowledge the Kitware Inc. for supporting the open source platforms of the Visualization ToolKit (VTK) and Paraview, upon which this work relies.

REFERENCES

Ahrens, J., Geveci, B., & Law, C. 2005, ParaView: An End-User Tool for Large Data Visualization (Elsevier)

Avila, L., & Kitware, I. 2010, The VTK User's Guide (Kitware). https://books.google.ch/books?id=6IxSewAACAAJ

Bitsch, B., Crida, A., Morbidelli, A., Kley, W., & Dobbs-Dixon, I. 2013, A&A, 549, 124

Cilibrasi, M., Szulgyi, J., Mayer, L., et al. 2018, Monthly Notices of the Royal Astronomical Society, 480, 4355. http://dx.doi.org/10.1093/mnras/sty2163

Commerçon, B., Teyssier, R., Audit, E., Hennebelle, P., & Chabrier, G. 2011, A&A, 529, A35. https://doi.org/10.1051/0004-6361/201015880

De Val-Borro, M., Edgar, R. G., Artymowicz, P., et al. 2006, Monthly Notices of the Royal Astronomical Society, 370, 529

Godunov, S. K. 1959, 47 (89), 271

Hayashi, C. 1981, Progress of Theoretical Physics Supplement, 70, 35

Hockney, R., & Eastwood, J. 1988, Computer Simulation Using Particles (CRC Press). https://books.google.ch/books?id=nTOFkmnCQuIC

Sampath, R. S., Barai, P., & Nukala, P. K. V. V. 2010, Journal of Statistical Mechanics: Theory and Experiment, 2010, P03029. http://stacks.iop.org/1742-5468/2010/i=03/a=P03029

Szulágyi, J., Masset, F., Lega, E., et al. 2016, 10, 1

Szulágyi, J., Morbidelli, A., Crida, A., & Masset, F. 2014, Astrophysical Journal, 782, arXiv:arXiv:1312.6302v2

Visual Toolkit Organization. 2009, 1

APPENDIX

A. LEGACY VTK FILE FORMAT

An example of a simple VTK file Visual Toolkit Organization (2009):

```
# vtk DataFile Version 2.0
Jupiter Simulation Data
ASCII
DATASET UNSTRUCTURED_GRID
POINTS n double
p0x p0y p0z
...
p(n-1)x p(n-1)y p(n-1)z

CELLS m
nVert0 i0 i1 i2 ... i(nVert0-1)
...
nVert(m-1) i0 i1 i2 ... i(nVert(m-1)-1)

CELL_TYPE m
type0
..
type(m-1)

CELL_DATA m
SCALARS name double
LOOKUP_TABLE default
s0
...
s(m-1)
```