SPH Visualization Code Instruction Manual

Roshan Mehta
Advisor: Miki Nakajima
University of Rochester

January 7, 2026

1 Overview

Smoothed particle hydrodynamics (SPH) is a Lagrangian technique that allows for accurate, high-resolution simulations of planetary collisions. While quantitative analysis is usually conducted, a qualitative overview is often the first step; these visualization programs make this process easy.

This manual serves as a guide to using the visualization code provided in this repository folder, which allow users to generate 3D renders, cross sections, and animations of their collisions. To begin, the user must first have data to visualize. This analysis code is programmed to run off of the Nakajima Lab's SPH code at the University of Rochester (`https://github.com/NatsukiHosono/FDPS_SPH`). For user convience, this code has been used to generate a low-resolution canonical impact scenario; one sample timestep of this simulation has been linked in the `SampleData.txt` file in the repository for users to try analyzing. While the programs are designed to run for data frames generated using the Nakajima Lab's SPH code, it can be easily be adapted to other SPH codes with minor reformatting.

The programs are all based in Python 3 and use the Matplotlib, NumPy, PyVista, VTK, and ImagIO libraries. ImagIO is used to turn snapshots into animations, and as such needs an extra installation in order to save the video as an MP4 file. Therefore, when installing this library, use the following command in terminal:

```
pip install imageio[ffmpeg]
```

Once all of these installations are complete, the user can begin visualizing their data.

2 Cross Section Visualizations

To visualize a collision as a cross section, navigate to `CSVis.py`. There are several user inputs for this program. The input, `path`, refers to the system file path to the SPH simulation data, while `outputpath` refers to the system file path that the cross section images will be stored in. Both must be entered as a Python string. The SPH code outputs files that are titled `results.XXXXX_YYYYY_ZZZZZ.dat`, where `XXXXX` is the outputnumber, `YYYYY` is the number of cores or processors used in the simulation, and `ZZZZZ` is the file number ranging from 0 to one less than the total number of cores. The input, `ncores`, should be equal to `YYYYY`. The inputs, `outputnumber1` and `outputnumber2`, refer to the first and last timestep for visualization, respectively. For example, if `outputnumber1 = 200` and `outputnumber2 = 300`, the code will generate separate cross sections for each output number between 200 and 300. These plots can then be combined into an animation (see Section 4). If the user only wants to generate a snapshot at one output number, they may enter that output number for both `outputnumber1` and `outputnumber2`. The input, `centering`, refers to if the plot will be centered at the body's center of mass; set this to `True` to center it, and `False` to leave the plot as is. The input, `axesscale`, refers to what units the plot axes will be in (i.e., 10^6 m, 10^7 m, etc.), and `axesdim` is how long each axis will be in units of `axesscale`. If the user does not want the plot axes to show,

they may set the input, `axes`, to `False`, or set it to `True` otherwise. The input, `background`, controls what the background color of the plot is. The user may enter `'Black'` or `'White'` for this, and it must be entered as a Python string. The input, `thickness`, controls how thick the cross section is in units of `axesscale`. For lower resolution simulations, a larger thickness is needed to fill in the plot. Alternatively, the user can edit the input, `particlesize`, to accomplish this, as increasing this variable will make the particles appear larger in the plot.

The remaining inputs allow the user to control how the plot is color-coded. One option is to color-code by a parameter, for example, adding a color map to visualize density. This is what the input, `parameter`, is for. The user may enter `'Temperature', 'Pressure', 'Density', 'Entropy',` or `'Energy'` for this variable as a Python string. The `minimum` and `maximum` inputs refer to what the minimum and maximum values of `parameter` are for visualization in standard SI units. The resulting color mapping will be scaled and normalized between these values. The input, `colormap`, refers to the name of the specific Matplotlib color map that will be used to visualize the data (i.e., `'inferno', 'viridis',` etc.). This again must be entered as a Python string. The list of supported color maps can be found at the attached link (`https://matplotlib.org/stable/users/explain/colors/colormaps.html`).

Let's say the user only wants a few materials to be color-coded by the colormap, and wants the others to remain a solid color. This is what the last four inputs are for. The user can set the target mantle, target core, impactor mantle, and impactor core to be solid colors by entering the corresponding color names for the inputs, `tarmantlecolor, tarcorecolor, impmantlecolor,` and `impcorecolor`, respectively, as Python strings. For a full list of available colors, please see the attached link (`https://matplotlib.org/stable/gallery/color/named_colors.html`). If the user wants one of the materials to be color-coded by the color mapping for the parameter, then they may enter `'cmap'` for that input. For example, suppose the user is doing a visualization of temperature for silicate material. They might set `parameter = 'Temperature'` with `minimum = 0`, `maximum = 7000`, and `colormap = 'afmhot'`. Since they are only visualizing silicate material, they would set `tarmantlecolor, impmantlecolor = 'cmap'` and set `tarcorecolor` and `impcorecolor` to a solid color, possibly `'gray'` (see Figure 1c).

If the user instead wanted all materials to be color-coded by the mapping, they could simply enter `'cmap'` for the last four inputs (Figure 1a). By contrast, if they wanted all materials to be solid colors, they can enter the corresponding colors for the last four inputs (Figure 1b). In this case, the inputs, `parameter, minimum, maximum`, and `colormap`, become dummy variables. However, the code still requires them to have valid values to run.

Although having several inputs may be difficult to manage at first, it allows the visualization program significant versatility. Once the user becomes familiar with how it functions, several different cross sections and data visualizations can be generated rather quickly and efficiently.

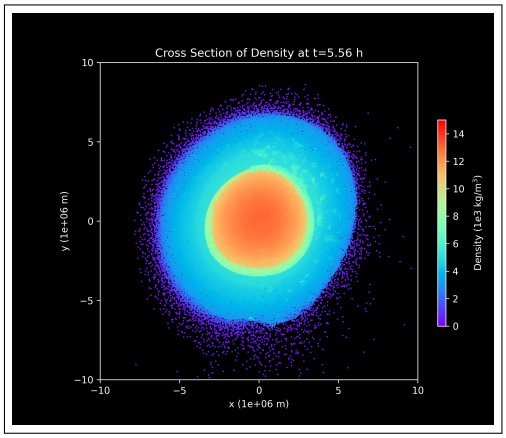
As the program runs, it will save the snapshots to the output path under the filename `CS_XXXXX.png`, where `XXXXX` is the specified outputnumber. Figure 1 shows few example cross sections that have been created for various collisions, as well as the inputs used to generate them.

3 3D Renders

In addition to making cross sections, the code allows users to generate 3D renders of a collision. To begin, navigate to the `3DRender.py` file. All inputs are defined the same way as in Section 2. The only difference is that there are three additional inputs, `camperaposition, azimuth,` and `elevation`. The input, `cameraposition`, is a tuple and refers to the viewing position for the render along the x, y, and z axes. The x and y coordinates are in units of the `axesscale`, and the z should be entered as 0 (e.g., (100,100,0)). The input, `azimuth`, represents the viewpoint's azimuthal angle, and `elevation` representing its angle above the x-y plane.

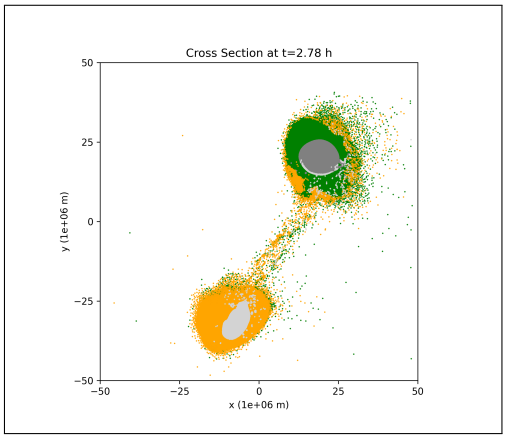
As the program runs, it will save the snapshots to the specified `outputpath` under the filename `3D_XXXXX.png`, where `XXXXX` is the specified `outputnumber`. Figure 2 shows a few example 3D renders created for various collisions, as well as the inputs used to generate them.

(a)



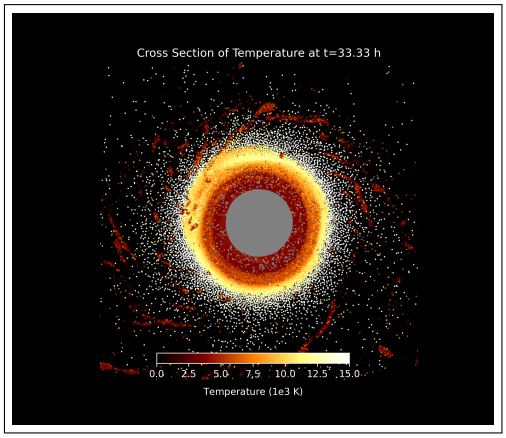

```
centering = True
axesscale = 1e6
axesdim = 20
axes = True
background = 'Black'
thickness = 2.5
particlesize = 2

parameter = 'Density'
minimum = 0
maximum = 15000
colormap = 'rainbow'

tarmantlecolor = 'cmap'
tarcorecolor = 'cmap'
impmantlecolor = 'cmap'
impcorecolor = 'cmap'
```

(b)




```
centering = False
axesscale = 1e6
axesdim = 100
axes = True
background = 'White'
thickness = 1
particlesize = 1

parameter = 'Density'
minimum = 0
maximum = 12000
colormap = 'rainbow'

tarmantlecolor = 'green'
tarcorecolor = 'gray'
impmantlecolor = 'orange'
impcorecolor = 'lightgray'
```

(c)



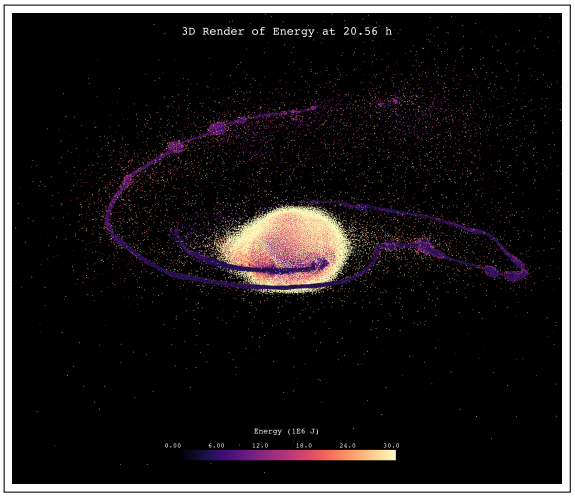

```
centering = True
axesscale = 1e6
axesdim = 40
axes = False
background = 'Black'
thickness = 1
particlesize = 1

parameter = 'Temperature'
minimum = 0
maximum = 15000
colormap = 'afmhot'

tarmantlecolor = 'cmap'
tarcorecolor = 'gray'
impmantlecolor = 'cmap'
impcorecolor = 'gray'
```

Figure 1: Example cross sections generated using `CSVis.py`.

4 Making Animations

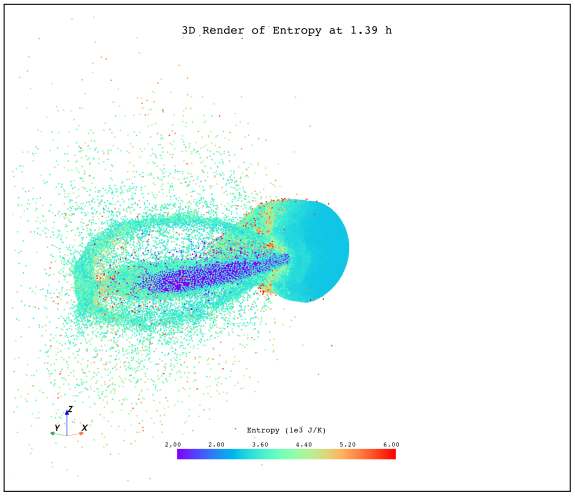
Once several successive output numbers have been visualized, either as a cross section or 3D render, the user can compile the snapshots into an animation. Navigate to the `AnimationMaker.py` file. The input, `path`, refers to the folder the snapshots are stored in, while the input, `outputpath`, refers to where the final animation will be stored. Both must be entered as Python strings). The input, `prefix`, refers to the file prefix attached to the snapshot filenames. For cross sections, the user can enter `'CS'`, and for 3D renders, they may enter `'3D'` (again, both must be entered as Python strings). The inputs, `outputnumber1` and `outputnumber2`, refer to the start and end output numbers for the animation, meaning the code will include all snapshots between these output numbers in the final video. Lastly, the input, `fps`, refers to how many frames per second the final video will be. Once the program finishes running, it will save the video as `Animation.mp4` to the specified `outputpath`.

(a)



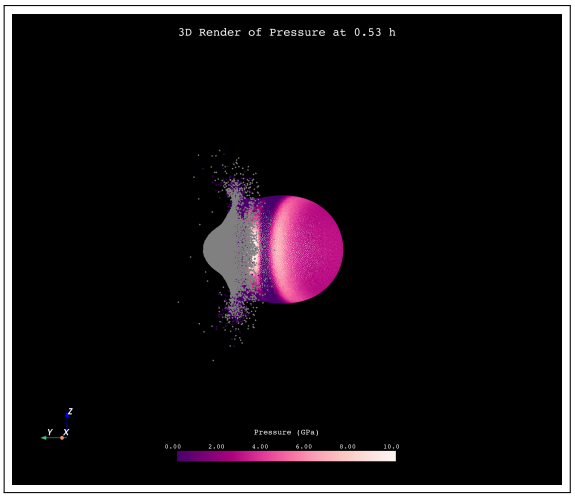

```
centering = True
axesscale = 1e6
cameraposition = (110,110,0)
axes = False
azimuth = 0
elevation = 20
background = 'Black'
particlesize = 3

parameter = 'Energy'
minimum = 0
maximum = 30e6
colormap = 'magma'

tarmantlecolor = 'cmap'
tarcorecolor = 'cmap'
impmantlecolor = 'cmap'
impcorecolor = 'cmap'
```

(b)




```
centering = True
axesscale = 1e6
cameraposition = (70,70,0)
axes = True
azimuth = 180
elevation = 10
background = 'White'
particlesize = 6

parameter = 'Entropy'
minimum = 2000
maximum = 6000
colormap = 'rainbow'

tarmantlecolor = 'cmap'
tarcorecolor = 'cmap'
impmantlecolor = 'cmap'
impcorecolor = 'cmap'
```

(c)




```
centering = True
axesscale = 1e6
cameraposition = (70,70,0)
axes = True
azimuth = 120
elevation = 0
background = 'Black'
particlesize = 6

parameter = 'Pressure'
minimum = 0
maximum = 10e9
colormap = 'RdPu_r'

tarmantlecolor = 'cmap'
tarcorecolor = 'cmap'
impmantlecolor = 'gray'
impcorecolor = 'gray'
```

Figure 2: Example 3D renders generated by `3DRender.py` .