SPH Analysis Instruction Manual

Roshan Mehta
Advisor: Miki Nakajima
University of Rochester

January 2, 2026

1 Overview

This manual serves as a guide to using the SPH analysis code at the attached link (`https://github.com/rmehta125/SPH-Simulation-Analysis/tree/main`). SPH is a Lagrangian method of simulating particle systems as fluids. Here, it treats the collision between two planets as such, and generates data frames that can be analyzed for various properties post-impact. The provided code allows users to estimate the amount of target mantle melting that occurs, determine how much of the impactor's core mixes with the target body's mantle, as well as visualize the pressure, temperature, entropy, etc. as a cross section or 3D render.

To begin, the user must first have data to analyze. 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`). One sample timestep from a low resolution canonical impact scenario using this code has been uploaded at the attached Dropbox link for users to try analyzing (`https://www.dropbox.com/scl/fo/omhu89fb82087rrr11p68/ABkQkBSbzLosi3UvUJS1GYU?rlkey=178ipj8oxtva3upz7obvhy8qn&st=1ub217ll&dl=0`). While the programs are designed to run for data frames generated using the Nakajima Lab's SPH code, many of the core ideas used can be adapted to other SPH codes with minor reformatting.

The programs are all based in Python 3 and use the Matplotlib, NumPy, SciPy, 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 analyzing their data.

2 Sorting Particle Data

The first step in estimating the amount of melting and mixing that occurs is sorting the particle data into planetary, disk, and escaping material. To do this, navigate to the `SortParticles.py` file. At the top of the document, the program requires seven user inputs: `path, outputpath, outputnumber, ncores, xlim, percentile,` and `iterations`.

The input, `path`, represents the system file path to the folder the SPH data is stored in (i.e., "`/home/[user]/collision_data/`"). Similarly, the `outputpath` represents the system file path to where the sorted data should be stored. It is recommended to make the `outputpath` the same as `path` so that all data is stored in the same place.

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 `ncores` input should be equal to `YYYYY`. For example, the canonical impact data provided with the code has files that are titled

`results.00740_00300_ZZZZZ.dat`. Therefore, if analyzing this data, the user should enter `outputnumber = 740` and `ncores = 300` into the program.

The inputs `xlim, percentile,` and `iterations`, are specific to how the program works. Once the code is run, the program first reads through all the data and centers the particles at the main body's center of mass. It then plots the distribution of particle distances from the center of mass in a histogram. `xlim` controls the cutoff distance for the x-axis on the histogram, and should be entered in units of 10^6 m. The code will then prompt the user to double-click on a point in the histogram which represents an initial guess for the planetary radius. For collisions with no resulting disk, this estimate is easy. However, if there is significant debris, it can be difficult to determine an initial guess. A good place to click is at the base of the main particle distribution, as shown in Figure 1.

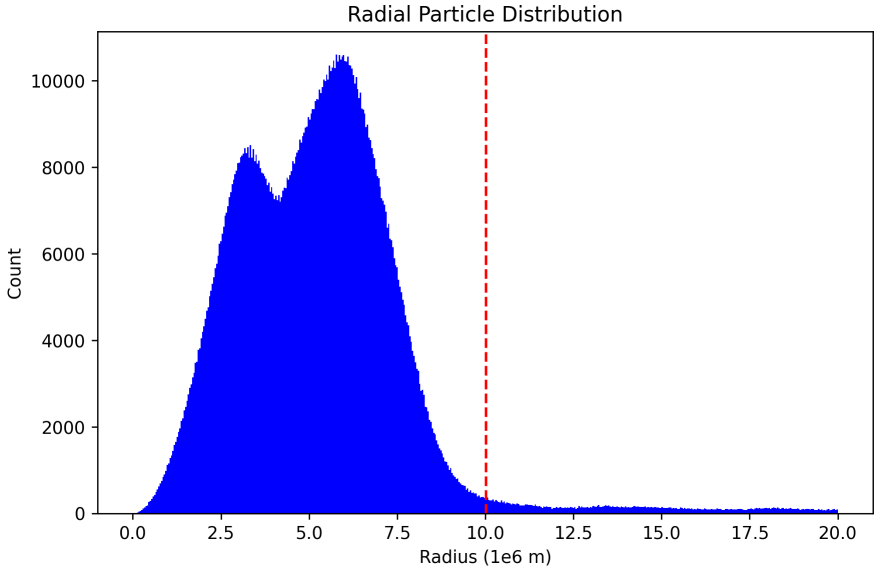


Figure 1: This figure shows the histogram generated for a collision that has significant disk material, as seen by the trailing right end of the histogram. A good initial guess for the radius is at the base of the main distribution, represented by the red line.

The program then proceeds to determine which particles are part of the planet, which are part of the orbiting disk, and which escape the main body entirely using an iterative scheme proposed by Canup (2004). First, an initial value for the main body's mass, M, is calculated by summing each particle's mass within the initial guess for the planetary radius. Then, each individual particle is tracked using classical orbital dynamics. Its energy is calculated with

$$E = \frac{1}{2}v^2 - \frac{GM}{r}, \tag{1}$$

where v and r are the particle's velocity and radial distance from the center of mass, respectively, and G is the gravitational constant. The particle's periapsis is also calculated with

$$r_p = a(1 - \varepsilon), \tag{2}$$

where a and ε are the orbital semi-major axis and eccentricity, respectively. These are calculated with

$$a = -\frac{GM}{2E} \tag{3}$$

$$\varepsilon = \sqrt{1 + \frac{2El^2}{(GM)^2}}, \tag{4}$$

with l being the particle's specific angular momentum. The program then determines if each particle is escaping or bound based on if its energy is positive or negative, respectively. Of those that are bound,

a particle is considered as planet material if its periapsis is less than or equal to the initial guess for the planetary radius, and disk material otherwise. A better estimate for the planetary radius can then be obtained by taking a percentile measure of the planet particles' distances from the center of mass. This is what the input, `percentile`, does. For example, if `percentile = 99`, the program will take the 99^{th} percentile of the planet particles' radial distances as a better estimate for the main body radius. The default value of 99 tends to work well, however, it can be adjusted if necessary. This entire process of tracking the particle orbits and sorting the particles is then repeated on a second iteration using this new radius value. This again returns a better estimate which can be used for a third iteration, and so on. Ideally, the iterations continue until the radius and mass values converge, which can be seen in the program's print statements. The user can control how many times this process occurs with the `iterations` input (for example, setting `iterations = 10` means the process will repeat 10 times). For systems with no disk, usually three to five iterations is sufficient to achieve convergence. However, for collisions with significant orbiting debris, five to ten iterations may be necessary.

One important note is that this process is not designed for oblique impacts at low velocities (which is about equal to the system escape velocity). This is because these impacts often begin with an initial grazing-impact, with the majority of the impactor staying intact and re-colliding with the planet later on. If the sorting program is run before this second collision occurs, the radius value will expand until it encompasses the orbiting impactor. This happens because its periapsis is less than the planetary radius, since it will eventually re-collide with the target. This of course gives a nonphysical value for the planetary radius that cannot be used. Additionally, it might not be ideal to run the simulation until this second collision occurs, as this can take several hours to happen. Even if this is feasible to simulate, the effect of numerical viscosity becomes non-negligible near 25 hours of simulated time, meaning this option is not optimal (Canup, 2004). Instead, for these types of collisions, it is best to set `iterations` to `0`. This makes it so that the final radius used is the same as the initial one the user inputs as a guess.

Once the program finishes running, it will output one file titled `SortedData_XXXXX.txt` to the specified `outputpath`, where `XXXXX` is the specified `outputnumber`. The file header is formatted as follows:

```
<simulation time (s)>
<total number of particles>
<planetary radius (m)>
<planetary mass (kg)>
<x center of mass (m)> <y center of mass (m)> <z center of mass (m)>
```

After the header is the particle data. Each line represents a different particle, and is formatted as follows:

```
<id> <tag> <mass (kg)> <x position (m)> <y position (m)> <z position (m)>
<x velocity (m/s)> <y velocity (m/s)> <z velocity (m/s)> <density kg/m^3)>
<internal energy (J)> <pressure (Pa)> <potential energy (J)> <entropy (J/K)>
<temperature (K)> <fate>
```

The particle data is formatted the same way it is outputted from the SPH code. The only change is that after `temperature`, a new data category, `fate`, is added. This value can be 0, 1, or 2, which indicates that the particle is planet, disk, or escaping material, respectively.

Now that the data has finished sorting, the amount of melting and mixing that occurs can be analyzed.

3 Analyzing Melting

The melting code provided allows users to estimate how much of the planetary mantle is melted during impact, and provides a cross section plot to visualize the melt mass.

To begin, navigate to the `AnalyzeMelting.py` file. This program prompts the user for several inputs: `path, outputpath, outputnumber, gamma, angle, axesdim, thickness` and `cutoff`. The input, `path`, refers to where the `SortedData_XXXXX.txt` file is stored from Section 2, and the input, `outputpath`, is where

the output plot will be stored. The input, `outputnumber`, is defined as it is in Section 2. The inputs, `gamma` and `angle`, are needed for the melting plot title, and represent the impactor-to-total mass ratio and the collision angle in degrees, respectively. The inputs, `axesdim` and `thickness`, are needed for generating the melting plot, and respectively represent the length of each axis and the thickness of the cross section, both in units of 10^6 m. Lastly, the input, `cutoff`, is related to whether or not the collision is an oblique, low-velocity impact, as discussed in Section 2. If it is not, set `cutoff` to `False` (we will will discuss what to do if it is later on).

The program will proceed by calculating a melting temperature, T_m for each gravitationally bound silicate particle. This criteria is defined by Rubie, et al. (2015) as

$$T_m = \begin{cases} 1874 + 55.43P - 1.74P^2 + 0.0193P^3, & P \leq 24 \text{ GPa} \\ 1249 + 58.28P - 0.395P^2 + 0.0011P^3, & P > 24 \text{ GPa} \end{cases} , \tag{5}$$

where P is the particle's pressure in GPa. If a particle's temperature is greater than its melting temperature, it is considered molten. Otherwise, it is considered solid. Partial melting is not considered. The program will finish by saving a plot to the specified `outputpath`, titled `Melting_XXXXX.png`, where `XXXXX` is the specified `outputnumber`. In the plot, blue particles represent solid material, while red particles represent molten material (see Figure 2 for reference). The program will also print the final melting percentage in the terminal.

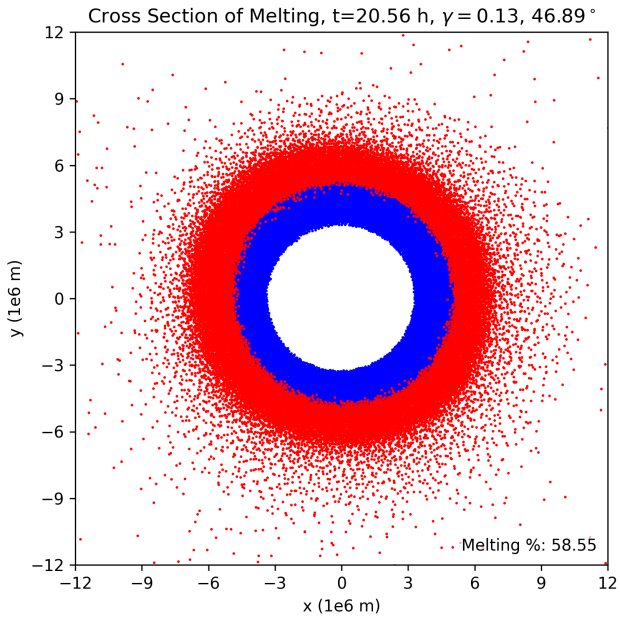


Figure 2: This figure shows the melt-mass plot generated for the sample canonical impact data provided. Red particles represent molten material, while blue particles represent solid material. The melt-mass percentage is shown in the bottom right.

Now, if the collision being analyzed is an oblique, low-velocity impact, the user may set `cutoff` to `True`. The melting analysis will function the same as before, however, the code will first return a histogram showing the radial particle distribution at the output number being analyzed. There will most likely be two main distributions, one representing the main body, and one representing the impactor that will eventually re-collide. The user will need to double-click on the histogram at a point that represents a cutoff for analysis. The program will then only analyze particles within that cutoff radius for melting. A point selected just before the impactor's orbital distance should suffice (see Figure 3 for reference). If `cutoff` was set to `False`,

the code would analyze the orbiting impactor silicate for melting, since it is gravitationally bound to the main body. This may be desireable; however, in the case that it is not, setting `cutoff = True`, allows the user to analyze the melting after the first collision only.

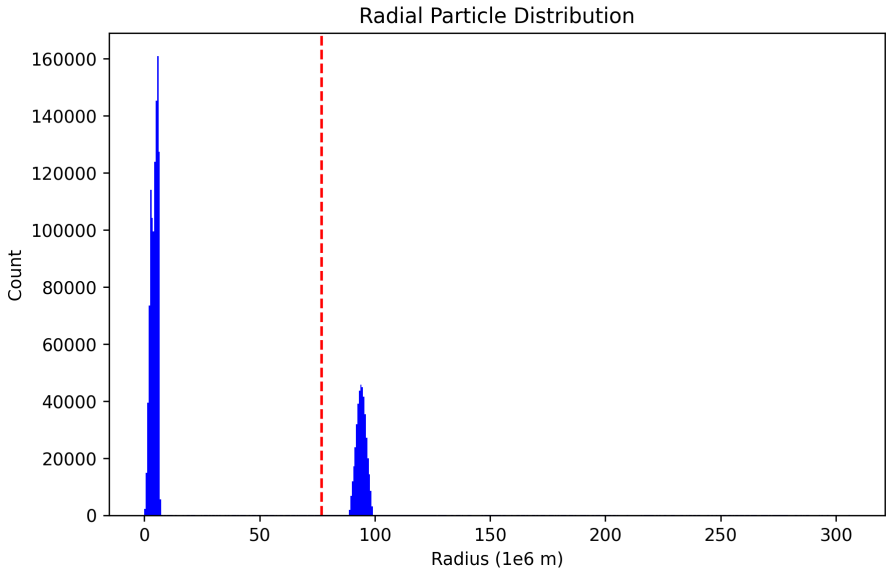


Figure 3: This figure shows the histogram generated when `cutoff` is set to `True` for an oblique, low-velocity impact. There are two main distributions; one represents the main body, while the other represents the impactor. Choosing a cutoff just less the impactor's orbital distance, as shown by the red line, works well for analyzing melting after the first collision.

4 Analyzing Mixing

The mixing code allows users to determine how much of the impactor's core merges with the target's mantle post-collision. It outputs a mixing percentage, as well as two visualization plots; one is a projection of particles onto the x-y plane, and the other is a 3D render.

To begin, navigate to the `AnalyzeMixing.py` file. The program prompts the user for eight inputs: `path, outputpath, outputnumber, gamma, angle, axesdim, azimuth,` and `elevation`. The first six are defined as they are in Section 3. The inputs `azimuth` and `elevation` correspond to the 3D plot; `azimuth` refers to the viewpoint's azimuthal angle, and `elevation` refers to its angle above the x-y plane.

When the program is run, the code will first store the particle data. It will then go through each iron particle and determine how many other iron particles are within 0.05 times the planetary radius from it using a SciPy tree. This is known as the number of "neighbors" that particle has. After sorting through this data, the code will generate a histogram of the distribution of neighbors for each iron particle. There should be a main distribution off the right. This represents iron particles in the resulting body's core, as these particles will have a large number of neighbors. Additionally, there will likely be a spike towards the left. This represents parts of the impactor core that have mixed with the target mantle, as there are likely far fewer neighbors for these iron particles. The program will ask the user to double-click on a point in the histogram that represents a cutoff value that will be used to analyze mixing. Specifically, if an impactor core particle has fewer neighbors than the cutoff, it will be considered as mixed with the target mantle. Otherwise, it will be considered as mixed with the target core. A good point to choose is right before the main distribution, as this is the natural separation between the mantle and core mergers (see Figure 4 for reference).

After this, the code will sort through the impactor core particles and determine which ones have a number of neighbors above versus below the chosen cutoff value. In the specified `outputpath`, it will

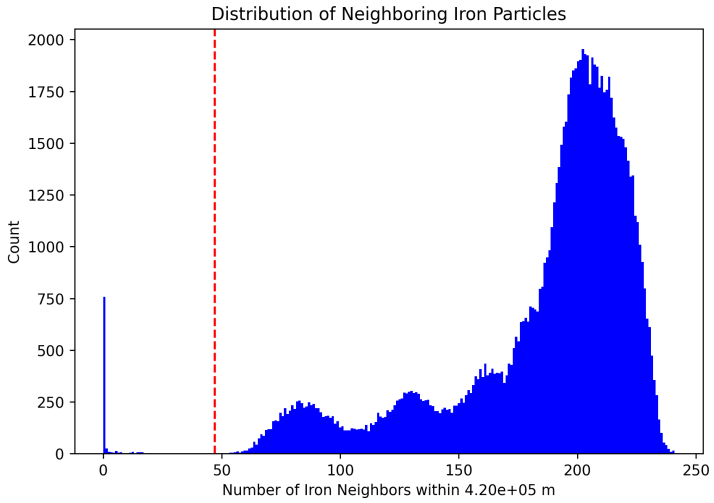
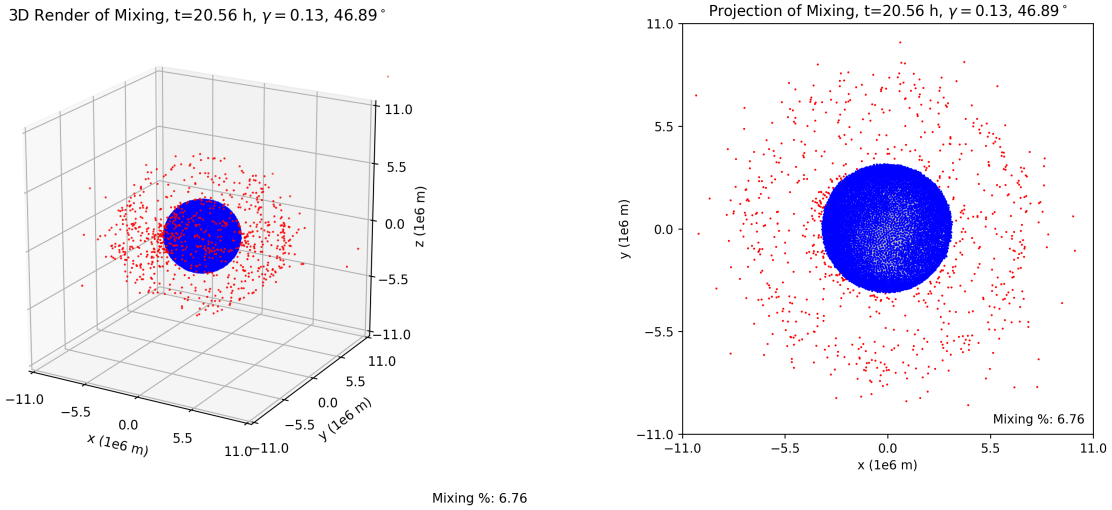


Figure 4: This figure shows the histogram of neighboring iron particles generated by the mixing code for the sample canonical impact data provided. The main distribution off to the right represents the core of the main-body, while the spike towards the left represents the impactor core particles that have merged with the target mantle. A good point to choose as the cutoff is at the start of the main distribution.

then save a 3D render and projection onto the x-y plane of the results, titled `Mixing3D_XXXXX.png` and `MixingProj_XXXXX.png`, respectively, where `XXXXX` is the specified `outputnumber`. Examples of these plots are shown in Figure 5. In both, only iron is plotted; red dots represent the impactor core particles that have mixed with the target mantle, and the blue dots represent those that have sunk to the target core. Beneath the layer of blue particles, the target core is plotted in a semi-transparent gray color. For collisions with large impactors, the target core will likely no longer be visible, as the impactor core tends to coat the outside of it. However, it will appear for collisions with smaller impactor masses. In addition to these plots, the code will output the final mixing percentage in the terminal.



(a) **(b)**

Figure 5: This figure shows a 3D render and projection of mixing generated for the sample canonical impact data provided. Blue dots represent impactor core particles that have sunk to the target core, while red dots represent those that have merged with the target mantle. The fraction of red particles to total impactor core particles is represented by the mixing percentage at the bottom right of each plot.

5 Cross Section Visualizations

In addition to analyzing the mixing and melting that occurs, the code provided allows users to generate cross section visualizations of the collision. These cross sections can visualize temperature, pressure, density, energy, etc. and can be used to generate animations as well.

To begin, navigate to `CSVis.py`. There are several user inputs for this program. The input, `path`, refers to where the original data files from the SPH simulation are stored, as this program does not read from the sorted data. The inputs, `outputpath` and `ncores`, again refer to where the output plots will be stored, and the total number of processors used for the simulation, respectively. 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 7). 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. 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. 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.). The list of supported color maps can be found at the attached link (`https://matplotlib.org/stable/users/explain/colors/colormaps.html`).

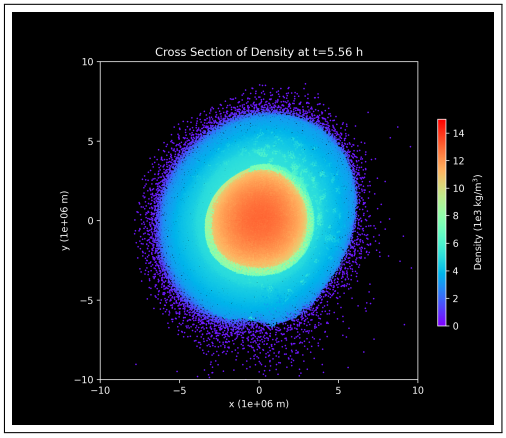
Let's say the user only wants a few materials to be color-coded by the colormap, and wants the others to remain a solid color. This is what the last four inputs are for. The user can set the target mantle, target core, impactor mantle, and impactor core to be a solid colors by entering the corresponding color names for the inputs, `tarmantlecolor, tarcorecolor, impmantlecolor,` and `impcorecolor`, respectively. 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 = 'magma'`. 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 6c).

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 6a). By contrast, if they wanted all materials to be solid colors, they can enter the corresponding colors for the last four inputs (Figure 6b). 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 6 shows few example cross sections from the sample canonical impact data, as well as the inputs used to generate them.

(a)

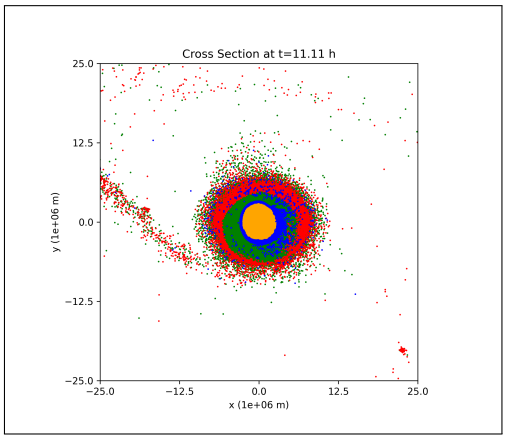



```
centering = True
axesscale = 1e6
axesdim = 20
axes = True
background = 'Black'
thickness = 2.5
particlesize = 2

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

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

(b)

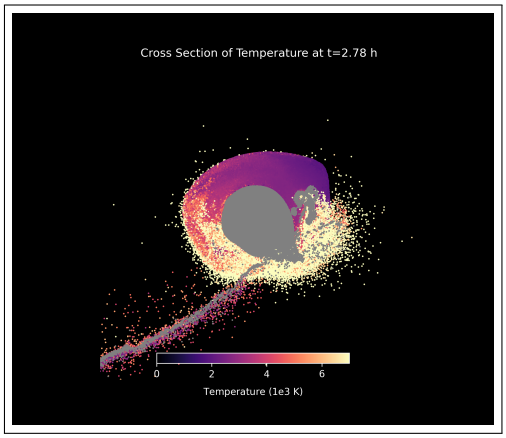



```
centering = True
axesscale = 1e6
axesdim = 50
axes = True
background = 'White'
thickness = 2.5
particlesize = 2

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

tarmantlecolor = 'green'
tarcorecolor = 'orange'
impmantlecolor = 'red'
impcorecolor = 'blue'
```

(c)




```
centering = True
axesscale = 1e6
axesdim = 30
axes = True
background = 'Black'
thickness = 2.5
particlesize = 2

parameter = 'Temperature'
minimum = 0
maximum = 7000
colormap = 'magma'

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

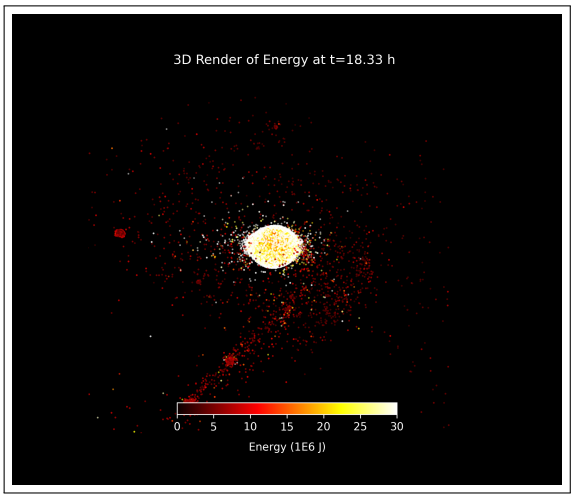
Figure 6: This figure shows a few example cross sections that have been generated for the sample canonical impact data.

6 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 they are in Section 5. The only difference is that there are two additional inputs, `azimuth` and `elevation`. These are defined the same as they are in Section 4, with `azimuth` representing 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 7 shows a few example 3D renders from the canonical impact data, as well as the inputs used to generate them.

(a)

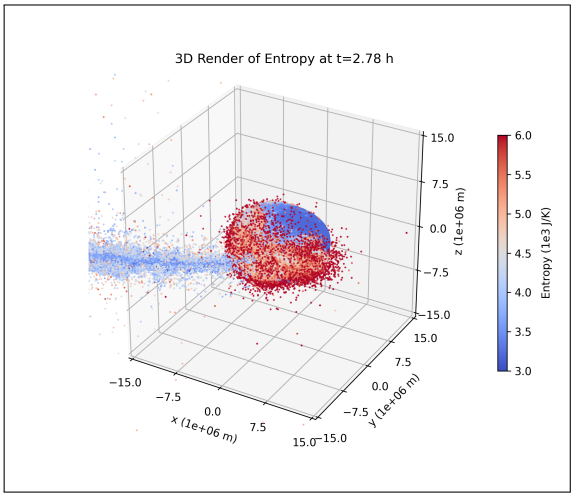



```
centering = True
axesscale = 1e6
axesdim = 75
axes = False
azimuth = -180
elevation = 30
background = 'Black'
particlesize = 2.5


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


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

(b)

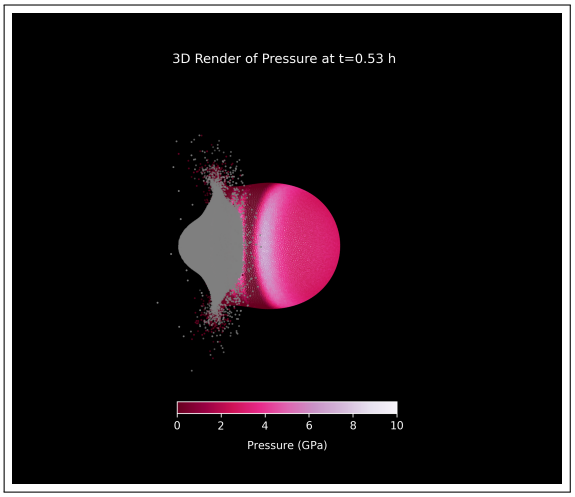



```
centering = True
axesscale = 1e6
axesdim = 30
axes = True
azimuth = -60
elevation = 30
background = 'White'
particlesize = 2.5


parameter = 'Entropy'
minimum = 3000
maximum = 6000
colormap = 'coolwarm'


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

(c)




```
centering = True
axesscale = 1e6
axesdim = 20
axes = False
azimuth = -185
elevation = 0
background = 'Black'
particlesize = 2.5


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


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

Figure 7: This figure shows a few example 3D renders that have been generated for the sample canonical impact data.

7 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. 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'`. 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`.

8 References

All programs and data visualized in this project were created and generated by Roshan Mehta under the supervision of Miki Nakajima at the University of Rochester. Data was generated using the Nakajima Lab's SPH code (`https://github.com/NatsukiHosono/FDPS_SPH`). Additional references are listed below.

[1] Canup, R. M. 2004, Simulations of a late lunar-forming impact, Icarus, 168, 433–456, `https://doi.org/10.1016/j.icarus.2003.09.028`

[2] Marchi, S., Canup, R. M., & Walker, R. J. 2018, Heterogeneous delivery of silicate and metal to the Earth by large planetesimals, Nat. Geosci., 11, 77–81, `https://doi.org/10.1038/s41561-017-0022-3`

[3] Nakajima, M., Golabek, G. J., Wünnemann, K., Rubie, D. C., Burger, C., Melosh, H. J., Jacobson, S. A., Manske, L., & Hull, S. D. 2021, Scaling laws for the geometry of an impact-induced magma ocean, Earth Planet. Sci. Lett., 568, 116983, `https://doi.org/10.1016/j.epsl.2021.116983`