Visualizing Astronomical Data in Houdini

August 21, 2017

Masters Project

Vijin Ravindran

MSc Computer Animation and Visual Effects 2016-17

Bournemouth University

Index

Contents

1 Abstract **4**

2 Introduction **4**

3 Technical Background **4**
- 3.1 Development . . . 5
  - 3.1.1 FITS . . . 5
  - 3.1.2 CSV . . . 10
  - 3.1.3 AMR . . . 10
- 3.2 Data Sources . . . 12
  - 3.2.1 THINGS . . . 13
  - 3.2.2 Gaia . . . 13
  - 3.2.3 HYG . . . 13
  - 3.2.4 GRS . . . 13
  - 3.2.5 VGPS . . . 14
- 3.3 External Libraries . . . 14
  - 3.3.1 YT . . . 14
  - 3.3.2 PyFITS . . . 14
  - 3.3.3 PyOpenVDB . . . 14

4 Methodology **15**

5 Previous work **16**
- 5.1 Papers . . . 16
- 5.2 Software . . . 16
  - 5.2.1 FRELLED . . . 16
  - 5.2.2 DS9 . . . 17
  - 5.2.3 FITS Liberator . . . 18

6 Implementation **18**
- 6.1 FITS . . . 18
  - 6.1.1 Algorithm . . . 21
- 6.2 CSV . . . 23
  - 6.2.1 Algorithm . . . 24
- 6.3 AMR . . . 25
  - 6.3.1 Algorithm . . . 28

7 Results **29**

8 Problems **30**

9 Conclusion **32**

10 Future Work **32**
10.1 Integration of PyOpenVDB in Houdini 32
10.2 Spiral galaxy simulations 32
10.3 Multi-threading 32

11 Acknowledgements **32**

12 References **33**

13 Appendix **36**
13.1 Synonyms 36
13.2 Code 37
13.2.1 HDA ordered menu creation 37
13.2.2 FITS Points Visualisation 38
13.2.3 FITS Volume Visualization 42
13.2.4 CSV Visualization 43
13.2.5 AMR VDB Creation (Naiman, J. P. et al, 2017) 45
13.2.6 FITS VDB Creation 47
13.2.7 Trials using inlincpp module in Python 49

1 Abstract

A project to visualize astronomical data such as star clusters, galaxies and dark matter maps using 3D data cubes, AMR files and VDBs. A pipeline tool has been developed to analyze and plot data points collected from various astronomical agencies.

2 Introduction

Astronomical data from satellites and ground telescopes are publicly available on a number of online portals and are free to use. Many interesting data, ranging from relatively small star clusters, to dark matter maps, to maps of entire sections of galaxies containing billions of stars are available. Their usage is somewhat limited, and the software to parse the data are developed specifically to serve the astronomy community. The data formats are not made for portability or speed, but for longevity and ease-of-use. Hence, the files can be a few megabytes to several gigabytes in size. Many astronomy software provides parsing and viewing of such files, but most do not cater to more than one format. The approach is more scientific, with the focus being on extraction of data points, potting graphs and in some cases, a bare-minimum 2D or 3D visualization. For this project, similar concepts have been applied, as well as general astronomical data analysis theory, to bring similar capabilities into Houdini, and build assets for use in visual effects. Three of the most widely used formats in astronomy have been chosen - FITS, CSV and AMR. The motivation for doing this project is an interest in astronomy and desire to learn Python.

3 Technical Background

The knowlede of a few basic astronomy concepts is necessary to understand this project. Astronomical data is collected using many different instruments. The most common of which are optical telescopes (reflector/reflector), which capture relatively high precision 2D images in the visible light spectrum, and radio telescopes. A radio telescope is much larger than optical telescopes because radio wavelengths are much longer than optical wavelengths. The longer wavelengths means that the radio waves have lower energy than optical light waves. Radio telescopes detect emission from cold clouds of hydrogen in the space between the stellar objects. Stars and planetary systems form in these molecular clouds (Strobel, N., 2017). The files used in this project have been obtained from similar radio telescopes.

The units of the celestial coordinate system, which is primarily used for the this type of data are Right Ascension (RA) and Declination (Dec). They are spherical coordinate systems, and can be thought of as the lines of the Earth's latitude and longitude projected outward and printed on the inside of the sky sphere. Directly out from the Earth's equator, 0° latitude, is the celestial equator, 0° declination. Lines of both right ascension and declination stay fixed with respect to the stars. They are measured in degrees, minutes and seconds.

3.1 Development

The two primary objectives during the research phase of this project was to attain a clear understanding of astronomical data formats, and learning Python scripting for Houdini.

3.1.1 FITS

FITS stands for Flexible Image Transport System. It was developed in the late 1970s to interchange astronomical image data and is the most widely used data format in astronomy. Most of the technical details of the first basic FITS agreement were developed by Don Wells and Eric Greisen (NRAO) in March 1979. It is endorsed by NASA and the International Astronomical Union and is used for the transport, analysis, and archival storage of scientific data sets such as multi-dimensional arrays - 1D spectra, 2D images, 3D+ data cubes and 2D database-style tables (McGlynn, T. A., 2016). It is a data format built on the assumption that neither the software not the hardware that wrote the data will be available when the data are read. (National Research Council, 1995).

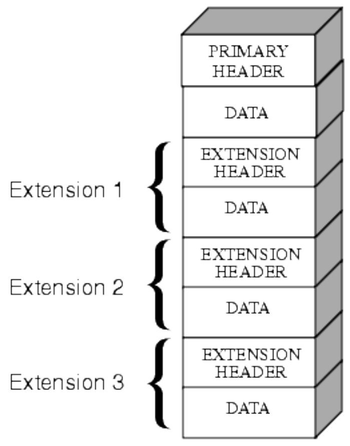
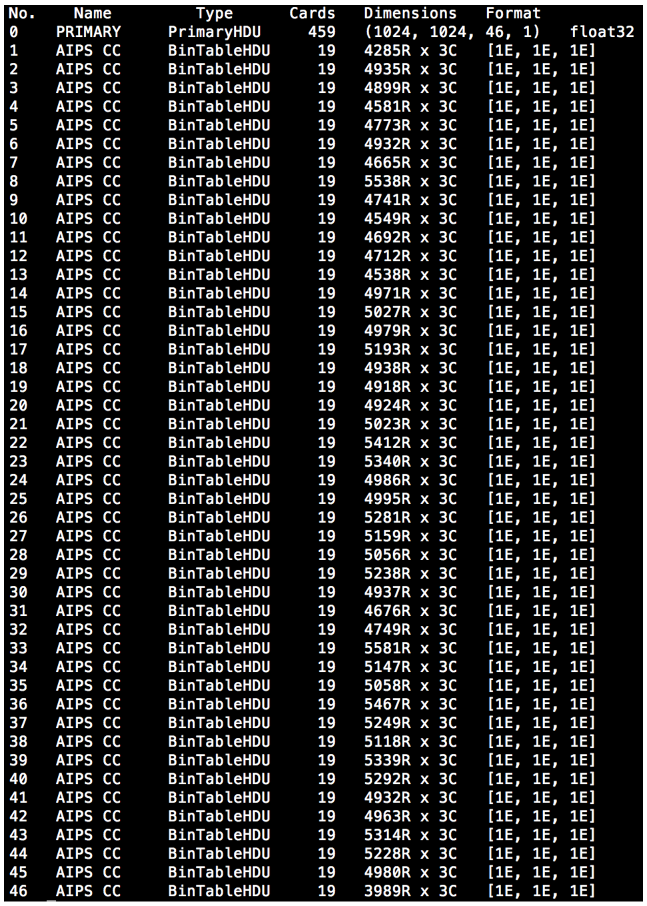


Figure 1: The FITS data structure (HST Data Handbook for WFPC2)

The precise format of 3D FITS files varies, but all of them contain the same basic structure : a 3D array of flux values where the axes are x, y (typically position on the sky, usually RA and Dec) and z (usually velocity, wavelength or frequency, which are all directly equivalent), with the value in each array cell being the measured flux value at that position and velocity. The THINGS format is a little different because it is technically 4D - it uses the 4th axis to hold polarization information (the 'Stokes' axis parameters). This is standard practise when processing data from the VLA, as some other types of observations also give other information on the degree of polarization. However for neutral hydrogen this is not the case - only the intensity is recorded. This means we can ignore the 4th axis.

The number of tables refer to the velocity/frequency 'channels' of the data. For the THINGS data, the telescope simultaneously images the sky at 46 different frequencies, each one of which is recorded as a 2D table (technically 3D, but each will have the polarization axis, which we can safely ignore). It is the channel number which gives us the precise velocity of the emission. The wavelength, frequency and velocity are directly equivalent to each other. What the instrument directly records is frequency or wavelength; how these are transformed into velocity is somewhat arbitrary. Because of the differences in receiver technology, different conventions defining velocity based on frequency or wavelength have arisen in radio and optical astronomy, however, as computing power has increased almost everyone use the 'optical' formula, even for radio observations like THINGS. Since the z axis is velocity, not distance, the separation of the image planes/voxels/vertices along that direction can be completely arbitrary.

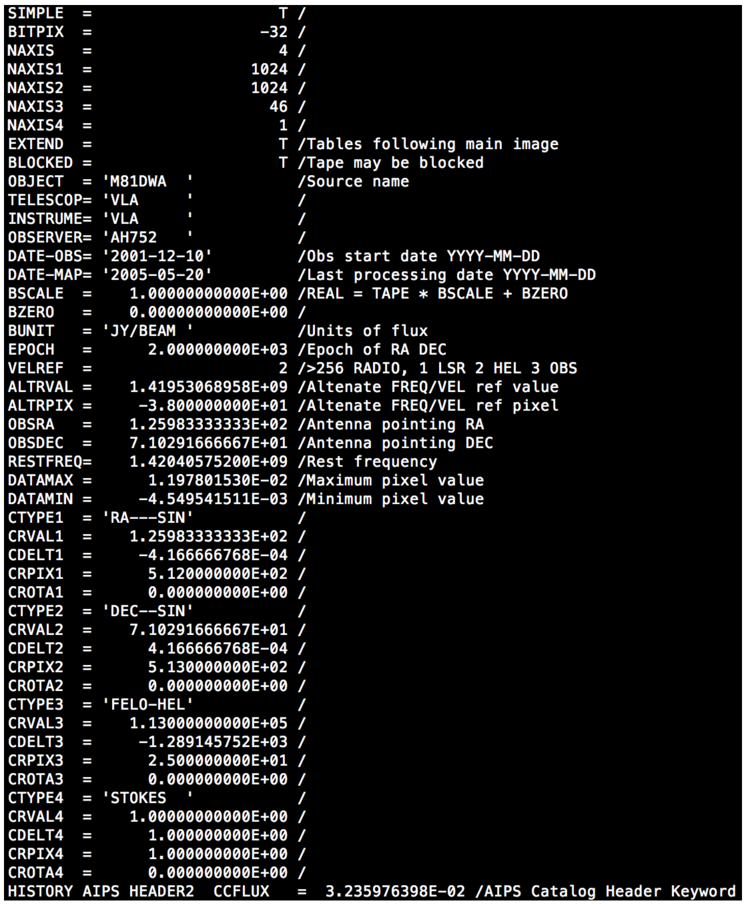
A typical FITS file contains a one or more 'header-data units', hereafter referred to as the 'HDU'. There can be one primary HDU and optional multiple extension HDU's. These HDU's can contain varying data structures such as the primary data array, random groups structures, image extensions, ASCII table extensions, or binary table extensions (Pence, W. D., et. al., 2008). The data is divided into 'FITS blocks', which is a sequence of 2880 bytes aligned on 2880 byte boundaries, mostly a header or a data block. Each HDU contains a header and a data block. The header consists of multiple 'cards' which describe the file, as well as the format of the data in the HDU, if there are data. The following example will help to understand the structure of a FITS file better.



```
No.    Name          Type       Cards   Dimensions    Format
0    PRIMARY     PrimaryHDU     459   (1024, 1024, 46, 1)   float32
1    AIPS CC     BinTableHDU     19   4285R x 3C   [1E, 1E, 1E]
2    AIPS CC     BinTableHDU     19   4935R x 3C   [1E, 1E, 1E]
3    AIPS CC     BinTableHDU     19   4899R x 3C   [1E, 1E, 1E]
4    AIPS CC     BinTableHDU     19   4581R x 3C   [1E, 1E, 1E]
5    AIPS CC     BinTableHDU     19   4773R x 3C   [1E, 1E, 1E]
6    AIPS CC     BinTableHDU     19   4932R x 3C   [1E, 1E, 1E]
7    AIPS CC     BinTableHDU     19   4665R x 3C   [1E, 1E, 1E]
8    AIPS CC     BinTableHDU     19   5538R x 3C   [1E, 1E, 1E]
9    AIPS CC     BinTableHDU     19   4741R x 3C   [1E, 1E, 1E]
10   AIPS CC     BinTableHDU     19   4549R x 3C   [1E, 1E, 1E]
11   AIPS CC     BinTableHDU     19   4692R x 3C   [1E, 1E, 1E]
12   AIPS CC     BinTableHDU     19   4712R x 3C   [1E, 1E, 1E]
13   AIPS CC     BinTableHDU     19   4538R x 3C   [1E, 1E, 1E]
14   AIPS CC     BinTableHDU     19   4971R x 3C   [1E, 1E, 1E]
15   AIPS CC     BinTableHDU     19   5027R x 3C   [1E, 1E, 1E]
16   AIPS CC     BinTableHDU     19   4979R x 3C   [1E, 1E, 1E]
17   AIPS CC     BinTableHDU     19   5193R x 3C   [1E, 1E, 1E]
18   AIPS CC     BinTableHDU     19   4938R x 3C   [1E, 1E, 1E]
19   AIPS CC     BinTableHDU     19   4918R x 3C   [1E, 1E, 1E]
20   AIPS CC     BinTableHDU     19   4924R x 3C   [1E, 1E, 1E]
21   AIPS CC     BinTableHDU     19   5023R x 3C   [1E, 1E, 1E]
22   AIPS CC     BinTableHDU     19   5412R x 3C   [1E, 1E, 1E]
23   AIPS CC     BinTableHDU     19   5340R x 3C   [1E, 1E, 1E]
24   AIPS CC     BinTableHDU     19   4986R x 3C   [1E, 1E, 1E]
25   AIPS CC     BinTableHDU     19   4995R x 3C   [1E, 1E, 1E]
26   AIPS CC     BinTableHDU     19   5281R x 3C   [1E, 1E, 1E]
27   AIPS CC     BinTableHDU     19   5159R x 3C   [1E, 1E, 1E]
28   AIPS CC     BinTableHDU     19   5056R x 3C   [1E, 1E, 1E]
29   AIPS CC     BinTableHDU     19   5238R x 3C   [1E, 1E, 1E]
30   AIPS CC     BinTableHDU     19   4937R x 3C   [1E, 1E, 1E]
31   AIPS CC     BinTableHDU     19   4676R x 3C   [1E, 1E, 1E]
32   AIPS CC     BinTableHDU     19   4749R x 3C   [1E, 1E, 1E]
33   AIPS CC     BinTableHDU     19   5581R x 3C   [1E, 1E, 1E]
34   AIPS CC     BinTableHDU     19   5147R x 3C   [1E, 1E, 1E]
35   AIPS CC     BinTableHDU     19   5058R x 3C   [1E, 1E, 1E]
36   AIPS CC     BinTableHDU     19   5467R x 3C   [1E, 1E, 1E]
37   AIPS CC     BinTableHDU     19   5249R x 3C   [1E, 1E, 1E]
38   AIPS CC     BinTableHDU     19   5118R x 3C   [1E, 1E, 1E]
39   AIPS CC     BinTableHDU     19   5339R x 3C   [1E, 1E, 1E]
40   AIPS CC     BinTableHDU     19   5292R x 3C   [1E, 1E, 1E]
41   AIPS CC     BinTableHDU     19   4932R x 3C   [1E, 1E, 1E]
42   AIPS CC     BinTableHDU     19   4963R x 3C   [1E, 1E, 1E]
43   AIPS CC     BinTableHDU     19   5314R x 3C   [1E, 1E, 1E]
44   AIPS CC     BinTableHDU     19   5228R x 3C   [1E, 1E, 1E]
45   AIPS CC     BinTableHDU     19   4980R x 3C   [1E, 1E, 1E]
46   AIPS CC     BinTableHDU     19   3989R x 3C   [1E, 1E, 1E]
```

Figure 2: Sample FITS file header-data units list

Figure 2 shows the content of the FITS file with data about a dwarf galaxy called M81 Dwarf A, in the Messier 81 galaxy group. From left to right, the columns describes the HDU number, the name of the HDU, the type, dimensions and format of the data contained in the HDU. This file has one primary HDU and 46 extension HDUs, which are binary tables. The primary HDU contains 459 'cards' which are basically 80-character keyword-value pair records contained in the HDU. The dimensions column mentions the number of dimensions of the data contained in the HDU has, which in this case is 4. This means that the data in the primary HDU is a single 'data-cube' which is 1024 X 1024 X 46 in size. The format column specifies that the values in the data array are 32-bit floating point numbers. Multidimensional data such as these are image planes stacked up against each other, forming a cube of data points. The extension HDUs are AIPS (Astronomical Image Processing System) tabular data (Bridle, A. et al., 1991). The table in the first extension HDU contains 4285 rows and 3 columns in the Fortran 1e format. To know more about the data in the HDU, the headers have to be read.



```
SIMPLE  =                    T /
BITPIX  =                  -32 /
NAXIS   =                    4 /
NAXIS1  =                 1024 /
NAXIS2  =                 1024 /
NAXIS3  =                   46 /
NAXIS4  =                    1 /
EXTEND  =                    T /Tables following main image
BLOCKED =                    T /Tape may be blocked
OBJECT  = 'M81DWA  '           /Source name
TELESCOP= 'VLA     '           /
INSTRUME= 'VLA     '           /
OBSERVER= 'AH752   '           /
DATE-OBS= '2001-12-10'         /Obs start date YYYY-MM-DD
DATE-MAP= '2005-05-20'         /Last processing date YYYY-MM-DD
BSCALE  =    1.00000000000E+00 /REAL = TAPE * BSCALE + BZERO
BZERO   =    0.00000000000E+00 /
BUNIT   = 'JY/BEAM '           /Units of flux
EPOCH   =      2.000000000E+03 /Epoch of RA DEC
VELREF  =                    2 />256 RADIO, 1 LSR 2 HEL 3 OBS
ALTRVAL =    1.41953068958E+09 /Altenate FREQ/VEL ref value
ALTRPIX =     -3.800000000E+01 /Altenate FREQ/VEL ref pixel
OBSRA   =    1.25983333333E+02 /Antenna pointing RA
OBSDEC  =    7.10291666667E+01 /Antenna pointing DEC
RESTFREQ=    1.42040575200E+09 /Rest frequency
DATAMAX =      1.197801530E-02 /Maximum pixel value
DATAMIN =     -4.549541511E-03 /Minimum pixel value
CTYPE1  = 'RA---SIN'           /
CRVAL1  =    1.25983333333E+02 /
CDELT1  =     -4.166666768E-04 /
CRPIX1  =      5.120000000E+02 /
CROTA1  =      0.000000000E+00 /
CTYPE2  = 'DEC--SIN'           /
CRVAL2  =    7.10291666667E+01 /
CDELT2  =      4.166666768E-04 /
CRPIX2  =      5.130000000E+02 /
CROTA2  =      0.000000000E+00 /
CTYPE3  = 'FELO-HEL'           /
CRVAL3  =    1.13000000000E+05 /
CDELT3  =     -1.289145752E+03 /
CRPIX3  =      2.500000000E+01 /
CROTA3  =      0.000000000E+00 /
CTYPE4  = 'STOKES  '           /
CRVAL4  =    1.00000000000E+00 /
CDELT4  =      1.000000000E+00 /
CRPIX4  =      1.000000000E+00 /
CROTA4  =      0.000000000E+00 /
HISTORY AIPS HEADER2  CCFLUX   =  3.235976398E-02 /AIPS Catalog Header Keyword
```

Figure 3: Header Information of the primary HDU

The cards in the header describe the data contained in the HDU, as well as information about he extension HDUs. Figure 3 shows the cards in the header of the primary HDU. Most of these fields are not important for visualization purposes. They describe the parameters of the data that are used for astronomical data analysis. The following are the list of fields of interest for the scope of this project:

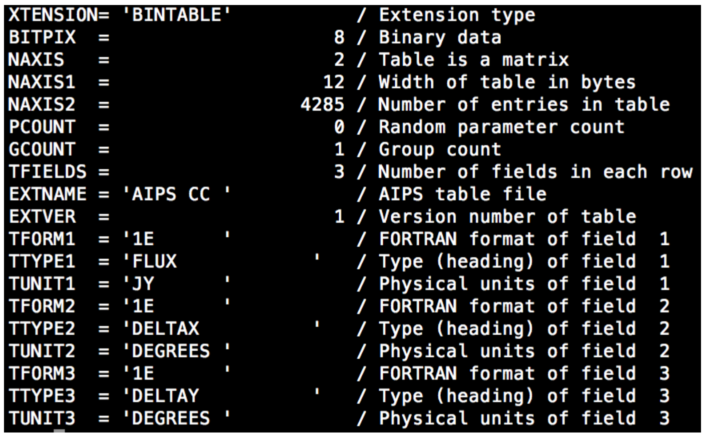
1. BITPIX - Specifies the number of bits that represent a data value in the associated data array. In this case, it is signed 32 bit floating point.

2. NAXIS - The number of axis of the data that is contained in this HDU. Here, NAXIS = 4 means that the HDU contains a 4 dimensional array.

3. NAXIS(n) - The length of axis n. Eg.: NAXIS1 = 1024 means the length of the first axis of the data array is 1024.

4. CTYPE(n) - The type for the intermediate coordinate axis n, specifying the coordinate type and a code for computing the world coordinate value. Here, CTYPE1 = RA—SIN means the first dimension of the data is the right ascension and a sin function needs to be applied to this value to get the world coordinate value.

Upon analyzing the header of the primary HDU, it can be understood that the data is a collection of 46 image planes that are 1024 X 1024 in size. The values in the array specify the intensity of each of the data points. This particular FITS dataset also has data in the 46 extension HDUs each with its own tabular data



```
XTENSION= 'BINTABLE'           / Extension type
BITPIX  =                    8 / Binary data
NAXIS   =                    2 / Table is a matrix
NAXIS1  =                   12 / Width of table in bytes
NAXIS2  =                 4285 / Number of entries in table
PCOUNT  =                    0 / Random parameter count
GCOUNT  =                    1 / Group count
TFIELDS =                    3 / Number of fields in each row
EXTNAME = 'AIPS CC '           / AIPS table file
EXTVER  =                    1 / Version number of table
TFORM1  = '1E      '           / FORTRAN format of field  1
TTYPE1  = 'FLUX            '   / Type (heading) of field  1
TUNIT1  = 'JY      '           / Physical units of field  1
TFORM2  = '1E      '           / FORTRAN format of field  2
TTYPE2  = 'DELTAX          '   / Type (heading) of field  2
TUNIT2  = 'DEGREES '           / Physical units of field  2
TFORM3  = '1E      '           / FORTRAN format of field  3
TTYPE3  = 'DELTAY          '   / Type (heading) of field  3
TUNIT3  = 'DEGREES '           / Physical units of field  3
```

Figure 4: Header Information of the extension HDU

The data contained in this HDU are 8 bit binary values in the form of a matrix.

1. NAXIS = 2 since the table has 2 dimensions

2. NAXIS1 = 12 is the width of the table in bytes.

3. NAXIS2 = 4285 specifies the number of rows in the table.

4. TFIELDS = 3 specifies the number of columns in the table.

5. TFORM(n) specifies the format of the data contained in the column

6. TTYPE(n) is the type of the data. Eg.: TTYPE1 = FLUX means the first column in the table is the flux value corresponding to the x and y coordinate point in space specified by the TTYPE2 and TTYPE3 columns.

7. TUNIT(n) is the unit of the data. JY is short for Jansky, a unit of spectral flux density.

Depending on the amount of data contained in a FITS file, the size may vary between a few megabytes to several gigabytes. The FITS file for the M81 dwarf galaxy is around 200 MB and contains image pixel data as well as coordinate and intensity information of data points. This format was chosen for the project because of it's wide use and support from most astronomical agencies. The THINGS, VGPS and Gaia data archives provide FITS data cubes of various astronomical objects.

3.1.2 CSV

CSV stands for comma-separated values. It is a file format used to store tabular data in plain text. Each line is considered as a row in the table, and the columns are separated by a character called the 'delimiter'. The delimiter is usually a comma, but since the format is not standardized, a number of special characters are commonly used. Most spreadsheet programs can import CSV data into other tabular formats using any user-inputted character as delimiters. CSV was chosen as the second file format for this project because of its ease-of-use, speed and readability. The files used for this project has been taken from the European Space Agency (ESA) Gaia missoin (https://www.cosmos.esa.int/gaia), processed by the Gaia Data Processing and Analysis Consortium (DPAC). The table in each file contains 57 columns, inclusing data such as latitude, longitude, flux, parallax, etc. of the data points, in around 135,000 rows per table.

3.1.3 AMR

AMR stands for Adaptive Mesh Refinement (Berger, M. J. et al., 1984). For solving partial differential equations (PDE) numerically, a discrete domain is selected in which algebraic analogues of the PDEs are solved. An effective method to do this is by introducing a grid and estimating the unknown values at the grid points using the solutions of these algebraic equations. The spacing of the grid points determines the local error and hence the accuracy of the solution. The spacing also determines the number of calculations to be made to cover the domain of the problem and thus the cost of the computation. For some problems, a grid of uniform mesh spacing in each of the coordinate directions gives satisfactory results. However for other more complex problems, the solution is difficult to estimate in some regions due to discontinuities, steep gradients, shocks, etc. A uniform grid having a small enough spacing can be used to minimize local errors estimated in these regions. But this approach is computationally extremely costly (Mitra, S., et al, 2001).

In the adaptive mesh refinement technique, a base coarse grid is set first. As the solution proceeds the regions requiring more resolution by some parameter characterizing the solution are identified. Then, finer sub-grids are superimposed only on those regions. Finer sub-grids are added recursively until either a given maximum level of refinement is reached or the local error has dropped below the desired level. Thus in an adaptive mesh refinement computation grid spacing is fixed for the base grid only and is determined locally for the sub-grids according to the requirements of the problem. AMR helps to track features much smaller than overall scale of the problem providing adequate spatial resolution where needed. (Guarassi, M., 2015)

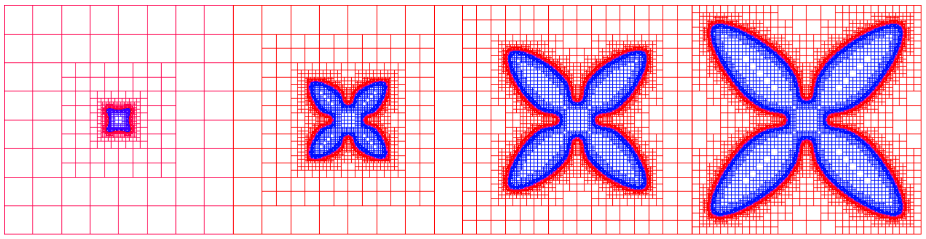


Figure 5: Adaptive Mesh Refinement (Topping, B. H. et al, 2004)

In addition to astrophysics, AMR is used in fields such as climate modeling and biophysics. The same theory can be applied to 3 dimensions as well. This is very similar to quad-trees and octrees, where a spacial area or volume is divided a number of times till the required resolution is achieved. In flocking simulation systems such as those that simulate the movement of birds in a flock, the number of neighbouring birds whose movement affects a particular bird can be limited to a within a small realistic volume using this method. Realistically, there is no need to consider the movement of all the birds in the flock to determine the movement of one. Using adaptive mesh refinement, the neighbouring birds that come within a threshold distance can be identified and used to determine the influence on flocking behaviours.

In the AMR data structure, the whole computational domain is covered by a coarse grid, representing the root node of the hierarchical data structure. In regions where higher resolution is required, finer sub-grids are created as child nodes of the root grid. This defines a new level of the hierarchy increasing the resolution of the parent grid by the 'refinement factor'. Fig. 6 shows a 2D example of an AMR grid hierarchy with a refinement factor of 2. Root grid A has one sub-grid B, which again has two children (C, D). The data values are normally stored at the grids nodes (vertex-centered) or at the centers of the cells (cell-centered). Sub-grids are completely contained within their parent grids. Sub-grids begin and end on parent cell boundaries, which means that parent grid cells are either completely subdivided or not subdivided at all. The data structures for storing AMR data are hierarchical, dynamic, and very large. For example, the simulation of some X-ray galaxy clusters use a grid hierarchy seven levels deep containing over 300 grid patches. (Norman, M. L. et al, 1999)

The AMR files tested in this project are made using Enzo, a grid-based finite-volume hydrodynamics code. That is, the domain is divided into cells, each is assigned various fluid properties (density, velocity, etc.), and at each time step fluxes of those quantities across the interfaces between cells are used to update the quantities in the cells. Enzo uses block-structured adaptive mesh refinement to provide high spatial and temporal resolution for modeling astrophysical fluid flows. The code is Cartesian, can be run in 1, 2, and 3 dimensions, and supports a wide variety of physics including hydrodynamics, and N-body dynamics (Bryan G. L., 2013)

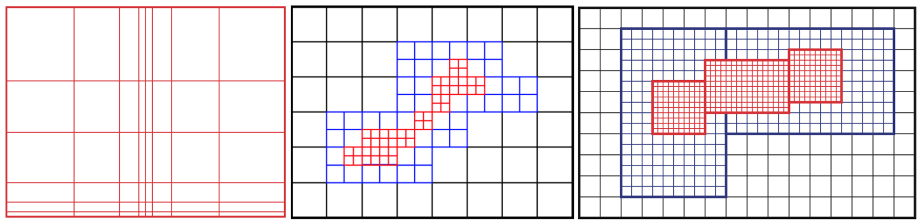


Figure 6: Types of Refinement - Mesh Distortion, Point-wise Structured and Block Structured (Almgren, A. S., 2011)

For cosmological simulations, each grid stores arrays of data describing the physical state of the cosmic fluid (density, temperature, and so on), the gravitational potential, and a list of particle positions and velocities for stars and dark matter. The cell size Δx of the grid decreases with depth in the hierarchy as 1/R level, where R is an integer refinement factor and level is the level of grid in the hierarchy. Any given grid in the hierarchy further resolves a region in its parent grid, and can also be the parent grid for a more refined child grid. Every grid contains the solution for the region it covers. That is, there are no holes in the parent grid where the child grids exist. Consequently, we can obtain an approximate representation of the global solution at any level of the hierarchy by compositing the solutions at or above that level (Norman, M. L., et al, 1999). The resolution levels of adjacent cells may differ by more than 1. This had to be taken into account for using the data for visualization using VDB volumes. The method used is discussed later.

AMR data can be extremely large and complex. Therefore, an suitable data format must be used to efficiently store and transport data. Hierarchical Data Format (HDF) is a set of file formats (HDF4, HDF5) designed to store and organize large amounts of data. Large AMR files used in astronomy are stored in HDF5 files. HDF5 includes only two major types of objects - 'Datasets', which are multidimensional arrays of a homogeneous type, and 'Groups', which are container structures which can hold datasets and other groups. This results in a truly hierarchical, filesystem-like data format.(Fox, G., 2015)

3.2 Data Sources

To test the capability of the digital asset to read and plot data from different sources, five sources were selected with different data formats.

3.2.1 THINGS

The HI Nearby Galaxy Survey is one of the largest programs undertaken at The National Radio Astronomy Observatory (NRAO) Very Large Array (VLA), U.S.A. to perform 21-cm HI observations of nearby galaxies. The HI stands for neutral hydrogen line emissions and refers to the electromagnetic radiation spectral line that is created by a change in the energy state of neutral hydrogen atoms. The project is designed to determine whether extra-galactic neutral hydrogen (HI) line emissions from nearby galaxies can be detected. A sample of 34 objects at distances between 3 to 15 Mega Parsecs have been targeted in THINGS, covering a wide range of star formation rates, total masses, absolute luminosities, evolutionary stages, metallicities, small-scale and 3D structure of ISM, dark matter distribution and the processes leading to star formation (Walter, F., 2007).

3.2.2 Gaia

Gaia is a mission to chart a 3D map of the Milky Way, to reveal the composition, formation and evolution of the galaxy. Gaia provides unprecedented positional and radial velocity measurements with the accuracies needed to produce a stereoscopic and kinematic census of about one billion stars in our Galaxy and throughout the Local Group. This amounts to about 1% of the Galactic stellar population, collected over a period of 5 years. (Europian Space Agency, 2016). The data are provided in both FITS as well as CSV formats, and a reduced dataset containing verified star positions (omitting other luminous objects) called the TGAS (Tycho-Gaia Astrometric Solution) is also provided. This is the data set that has been used in this project.

3.2.3 HYG

The HYG 3.0 database is a compilation of stellar data from a variety of catalogs. It is useful for background information on star names, positions, brightnesses, distances, and spectrum information. The database is a subset of the data in three major catalogs: the Hipparcos Catalog, the Yale Bright Star Catalog, and the Gliese Catalog of Nearby Stars. The Hipparcos catalog is the largest collection of high-accuracy stellar positional data, particularly parallaxes. The Yale Bright Star Catalog contains basic data on essentially all naked-eye stars. The Gliese catalog is the most comprehensive catalog of nearby stars (those within 75 light years of the Sun) including fainter stars not found in Hipparcos. (Nash, D., 2006)

3.2.4 GRS

The Galactic Ring Survey is a joint project of Boston University and Five College Radio Astronomy Observatory to observe the Milky Way's dominant star-forming structure, the Galactic Ring. Using the SEQUOIA multi-pixel array receiver on the FCRAO 14 m telescope, a new molecular line survey of the inner Galaxy was conducted. The project intended to catalog the molecular clouds and cloud cores, establish kinematic distances to many clouds and their associated Young Stellar Objects and determine their sizes, luminosities, and distributions, and to determine the structure of the inner Milky Way (Jackson et al., 2006).

3.2.5 VGPS

The VLA Galactic Plane Survey is a project to catalogue 21-cm line emission from neutral atomic hydrogen (HI) in the Milky Way disk. The VLA survey provides a link between the northern and southern hemisphere surveys and cover the first quadrant of the Galaxy, where the effects of star formation and the interaction between the disk and halo are expected to be dominant shapers of the inter-stelar medium (ISM) (e.g. Heiles 1984). The VGPS will also provide, together with high resolution infrared images, complete imaging of the major components of the interstellar medium in this region down to scales of a few parsecs. The data is provided in the FITS format as multiple moment maps. A moment map can be considered as a single layer of a 3D data cube along the velocity axis. (McClure-Griffiths, N. M. et al. 2001).

3.3 External Libraries

3.3.1 YT

Yt is an open-source, permissively-licensed python package for analyzing and visualizing volumetric data. Yt supports structured, variable-resolution meshes, unstructured meshes, and discrete or sampled data such as particles. Focused on driving physically-meaningful inquiry, yt has been applied in domains such as astrophysics, seismology, nuclear engineering, molecular dynamics, and oceanography. Yt provides methods to parse AMR and FITS files. However, it has been used in this project only to read AMR files.

3.3.2 PyFITS

PyFITS provides an interface to FITS formatted files in the Python scripting language. It is useful both for interactive data analysis and for writing analysis scripts in Python using FITS files as either input or output. PyFITS is a development project of the Science Software Branch at the Space Telescope Science Institute. PyFITS and all necessary modules are included with the stsci_python distribution and associated updates to it. It may be used independently as long as numpy is installed.

3.3.3 PyOpenVDB

OpenVDB is an Academy Award-winning open-source C++ library comprising a novel hierarchical data structure and a suite of tools for the efficient storage and manipulation of sparse volumetric data discretized on three-dimensional grids.

It is developed and maintained by DreamWorks Animation for use in volumetric applications typically encountered in feature film production. (Museth, K., 2013). The Python module PyOpenVDB supports a fixed set of grid types - FloatGrid, BoolGrid and Vec3SGrid.

4 Methodology

The principles of data visualization (Bernhard, J., 2012) suggests that a large amount of quantitative information can be packed into a small region. Graphing data should be an iterative, experimental process. The overall scale of the data presented by astronomical data sets is somewhat irrelevant for our purpose. However, the relative scales, coordinate systems and units have to be taken into consideration while plotting the data, to maintain the integrity of the data. Iterating through every data point in the data set is both time consuming and computationally expensive. For non-scientific purposes, it was observed that a subset of the data that was scaled down proportionally, was visually similar to the result obtained by using the entire data set. Implementing a variable resolution also helped in the testing phase, by ensuring the entire data set was not being parsed after every iteration.

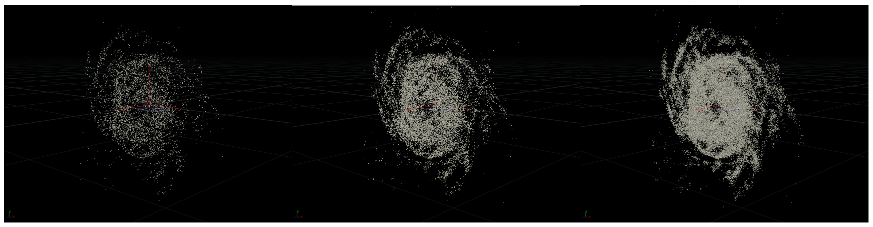


Figure 7: NGC628 FITS data cube at 10, 25 and 50 percent resolution

To plot the data points accurately, the user needs to have the flexibility to choose from the data sets what values to set as positional data, and what to set as point attributes like magnitude, flux etc. Some data sets such as those from the Gaia observations have positional data based on multiple coordinate systems and also data related to star intensity, flux density, error, etc. Therefore, it was decided to split the whole process into two parts - data analysis and data visualization. The visualization does not commence until the user selects the parameters to use for the visualization.

This method completely depends on the type of data. HI emission maps carry a large number of data points based on the flux densities of observed points in space. Star maps such as the data collected in the Gaia project represent positions of individual stars (Europian Space Agency, 2016). For visualizing AMR files, VDB volumes were used. For each refinement level of the AMR, a VDB grid is generated because the voxel size of a single volume has to be the same, unlike 3D AMR, where each sub-grid is 1/8th of the size of their parent grid within the same volume.

5 Previous work

5.1 Papers

Scientific data visualization has been implemented using Blender, a free open source software used for 3D content creation (Kent, B. R., 2013). The paper describes the methods used to read FITS data cubes and AMR files to visualize dark matter maps and star clusters. Blender has a python interface which can be used to parse and visualize data as points and volumes. Commonly used data formats used for large datasets and techniques to visualize them in Houdini has been illustrated by Ben Simons (Simons, B., 2015). Various real-time colouring and filtering algorithms that can be used for producing visually appealing renders have been illustrated in the paper Real-time colouring and filtering with graphics shaders (Vohl, D. et al, 2017). However, this approach is for purely scientific purposes, and realtime rendering is not in the scope of this project. Further research into visualization of large N-particle data and spectral cubes has been done by Amr Hassan et. al. including using high performance computing architectures (e.g: distributed processing and GPUs), collaborative astronomy visualization, the use of workflow systems to store metadata about visualization parameters, and the use of advanced interaction devices (Hassan, A. et. al., 2011). Naiman (2017) developed tools to import and manipulate astrophysical data into Houdini using the yt python library focusing primarily on visualization of adaptive mesh refinement files. More strategies for visualization of AMR files and improved scalability techniques were presented by Micheal L. Norman (1999). Data from the Gaia mission was visualized using Houdini by Niklas Rosenstein (2017).

5.2 Software

5.2.1 FRELLED

FRELLED stands for FITS Realtime Explorer of Low Latency in Every Dimension, an astronomical data viewer designed for 3D FITS files. It's mainly aimed at visualizing data cubes in realtime, interactive 3D, and is particularly geared toward HI and simulation data. It is a set of python scripts for Blender, and allows users to import 3D FITS files into Blender, where they can be viewed from any angle in realtime. (Taylor, R., 2015)

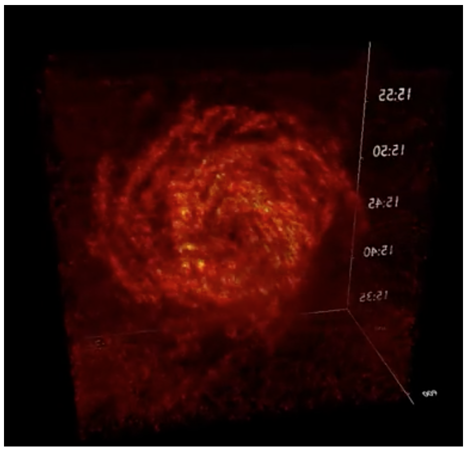


Figure 8: Visualization of NGC628 FITS data cubeusing FRELLED

5.2.2 DS9

SAOImage DS9 is an astronomical imaging and data visualization application. It supports FITS images and binary tables, multiple frame buffers, region manipulation, and many scale algorithms and colormaps. DS9 is a stand-alone application. It requires no installation or support files. DS9 supports advanced features such as 2D, 3D and RGB frame buffers, mosaic images, tiling, blinking, geometric markers, colormap manipulation, etc. The GUI for DS9 is user configurable.

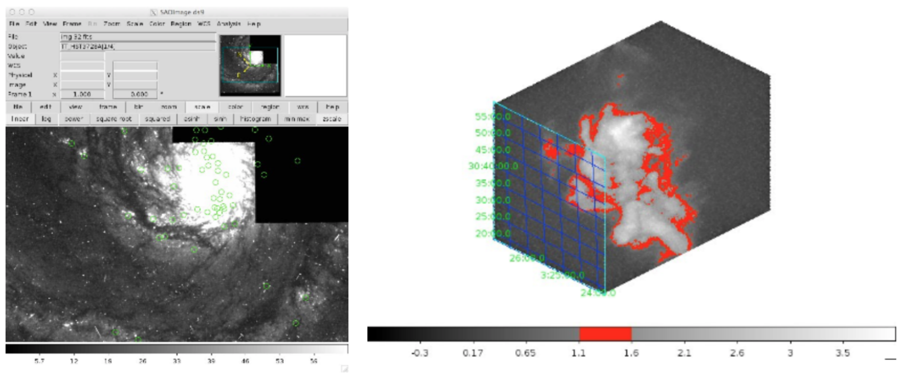


Figure 9: Visualization of FITS data cube in DS9

5.2.3 FITS Liberator

FITS Liberator is a free software for processing and editing astronomical data in the FITS format. Version 3 and later are standalone programs, while earlier versions were plugins for Adobe Photoshop. It uses NASA's CFITSIO library to read and write FITS data. As with other software mentioned above, the capabilities of this program is limited to scientific analysis and 2D viewing of FITS data cubes. However, it is extremely fast and memory efficient, and is supported by NASA and ESA.

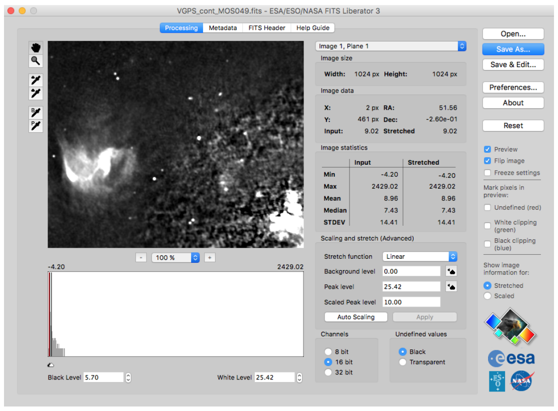


Image 1, Plane 1

Image size
Width: 1024 px Height: 1024 px

Image data
X: 2 px RA: 51.56
Y: 461 px Dec: -2.60e-01
Input: 9.02 Stretched 9.02

Image statistics

	Input	Stretched
Min	-4.20	-4.20
Max	2429.02	2429.02
Mean	8.96	8.96
Median	7.43	7.43
STDEV	14.41	14.41

Scaling and stretch (Advanced)
Stretch function: Linear
Background level: 0.00
Peak level: 25.42
Scaled Peak level: 10.00

Channels: 8 bit [✓ 16 bit] 32 bit
Undefined values: [✓ Black] Transparent

[✓ Preview]
[✓ Flip image]
Freeze settings
Mark pixels in preview:
Undefined (red)
White clipping (green)
Black clipping (blue)
Show image information for: [✓ Stretched] Scaled

100 %
-4.20 2429.02
Black Level 5.70 White Level 25.42

Figure 10: Visualization of a VGPS Data cube image planes in FITS Liberator

6 Implementation

This project aims at using real world astronomical data for visualization in motion graphics. Large datasets collected from ground and satellite telescopes can be used to accurately construct objects in space such as star clusters, dark matter clusters and entire galaxies. The algorithms presented here were used to extract data from the various sources and unify the visualization process using a custom digital asset in Houdini. Data from five different sources in three most widely used formats have been considered. The primary objective of this project is to efficiently parse data from astronomical data sources to and plot the data points on a platform where it can be used to develop visually appealing renders for visual effects.

The Python node in Houdini was used to read and plot the data. The process is divided into two - data analysis and data visualization.

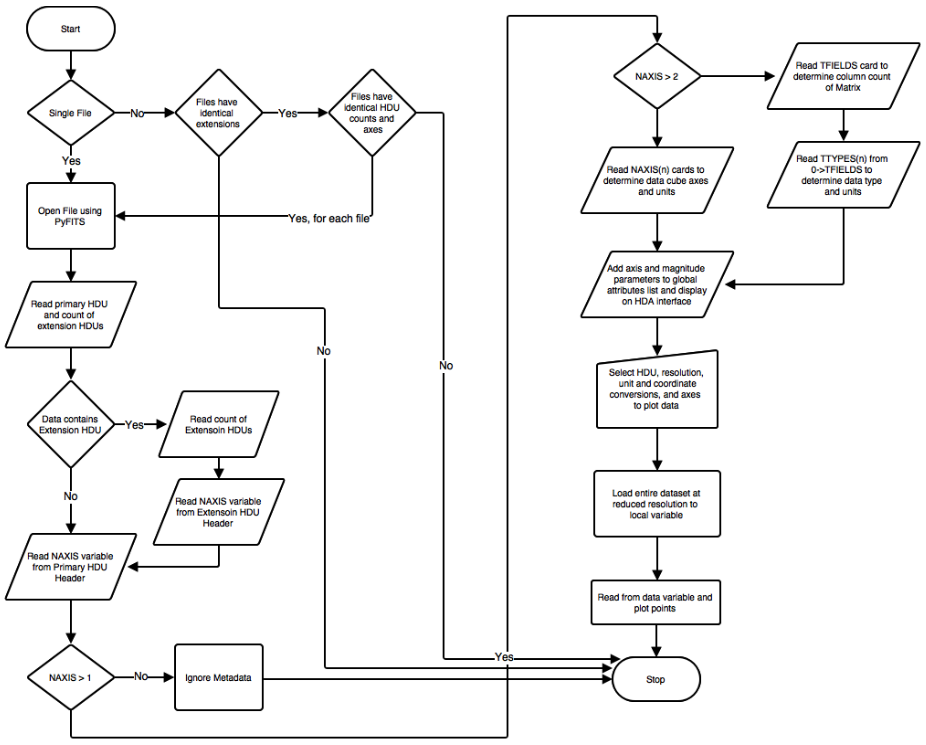
6.1 FITS

FITS files were read using the PyFITS Python library. If this library is not installed and configured for use from within Houdini's Python environment, the asset will display a message asking the user to install it before continuing.

Information about the data contained in the file is presented in human-readable ASCII format in the header section of the header-data units, and both binary and ASCII data can be held in the HDUs. Some sources provide data split into 'moment maps', which are 2D image planes contained in separate FITS files, or the entire 3D data split into multiple files for the sake of portability. Therefore, the capability to read multiple files has been included in the digital asset. If a folder is selected, all files in the folder are verified to contain identical HDU counts and axis definitions, and the data is read and plotted in a loop.

The first Python node contains code to read the data, create attributes and load the the user interface parameters. Since data variables cannot be passed between Houdini nodes, the ideal way to do it was through the attributes. It reads the number of HDU's, the number of axes in each HDU and the name of the axes. If the data is a matrix or table, the number and name of the columns would also be extracted. The order and type of data stored in FITS files vary from source to source, so it was not possible to plot the data based on a predefined axes selection.

The ordered menu on the HDA interface are biult live using Python scripts which extract these global attributes. Therefore the menus remain deactivated until a file is chosen and analysed. Several of-the-shelf software that read FITS files have predefined axis and coordinate systems, so the data cannot be plotted accurately (For example, using the DS9 program to open FITS files from the Gaia source). Therefore, the option to switch between spherical and cartesian coordinate systems and between degrees and radians were added. Once the axes are defined, the user can select what attributes to be added to the points/volume, depending on the available data in the file. For most infra red and x-ray telescopes, the apparent magnitude or luminosity is recorded, which affects the colour and visibility of the data point respectively. This has been defined as the 'magnitude' field in the point attributes. Once again, the type of data that represents the magnitude may vary depending on the source.



Start
Single File
No
Yes
Files have identical extensions
Yes
Files have identical HDU counts and axes
Open File using PyFITS
Yes, for each file
Read primary HDU and count of extension HDUs
No
No
Data contains Extension HDU
Yes
Read count of Extensoin HDUs
Read NAXIS variable from Extensoin HDU Header
No
Read NAXIS variable from Primary HDU Header
NAXIS > 1
No
Ignore Metadata
Yes
NAXIS > 2
Read TFIELDS card to determine column count of Matrix
Read NAXIS(n) cards to determine data cube axes and units
Read TTYPES(n) from 0->TFIELDS to determine data type and units
Add axis and magnitude parameters to global attributes list and display on HDA interface
Select HDU, resolution, unit and coordinate conversions, and axes to plot data
Load entire dataset at reduced resolution to local variable
Read from data variable and plot points
Stop

Figure 11: Flowchart - FITS file analysis

The second Python node reads the entire dataset and loads it into a local variable first. This data is then read out to assign point positions. This reduced the number of times the dataset had to be parsed using the external libraries. This improved the loading time considerably. However, for previewing purposes, a resolution setting has been provided, which effectively reduces the number of points by iterating through the data set at intervals higher than the default 1. This was added because it had very little effect the overall structure of the visual output, and helped to decrease the reading time. Once the data visualisation is complete, the geometry is cached out to disk using a file node, and subsequent changes on the HDA interface affects the cached out geometry. This also includes coordinate system and unit changes. It was observed that only the nodes that have calls to the the changed parameter are cooked when a particular parameter is changed. Hence, it was faster to perform some of the calculations using VEX on the cached out geometry.

6.1.1 Algorithm

Algorithm 1 Pseudocode - FITS Visualization as Points

```
create list data = []

foreach HDU in HDUlist
 append to data(HDUdata)

if NAXIS = 2 //data is a matrix:

 //buildFromMatrix()
 foreach column in data, iterate by resolution/100:
  //columns for position data from HDA interface
  x = data[HDU number][row Number][x Column Number]
  y = data[HDU number][row Number][y Column Number]
  if zAxis is arbitrary:
   //small value for image plane separation
   z = 0.001 * HDU number
  else:
   z = data[HDU number][row Number][z Column Number]

  create point with position (x, y, z)
  set attribute 'magnitude' = \
  data[HDU number][row Number][magnitude Column Number]
  //set layer count as attribute for better control
  set attributer layer = HDU number

else if NAXIS > 2 //3D data cube
 //Function buildFromDatacube()
 if NAXIS = 4:
  //ignore polarization axis
  set data = HDU data[0]
 else:
  set data = HDU data
  set x, y, z axis lengths of HDUdata
  for each x, y, z in HDUdata:
   create point with position (x, y, z)
   set attribute 'magnitude' = \
   data[HDU number][row Number][magnitude Column Number]
   set attributer layer = HDUnumber
```

For volume visualisation however, the entire data set is used to input voxel values. For all FITS data sets used for this project, 3D arrays as image plane data is contained in the primary HDU, so volume visualisation has been disabled for extension HDUs. Both VDBs and Houdini volumes can be produced.

Algorithm 2 Pseudocode - FITS Visualization as Volume

```
//buildFromVolume()
if data dimensions = 4:
 set data = HDUdata[0] //ignore polarization axis
else:
 set data = HDUdata

data = cast to float(data) //cast to float32
data = remove nan(data) //remove numpy not-a-number

//create bounding box with size = data cube.
//Houdini assumes NDC dimensions for volumes otherwise

create bounding box(xAxisLength,yAxisLength,zAxisLength)
create volume with (dataSize, bounding box)
for each voxel index (i, j, k) in data:
 set voxel value = data[i][j][k]
```

The code for generating the VDB volume can be found in a PythonModule subsection in the Scripts section of the HDA. This script is called using a callback Script that is executed when the 'Write' button is clicked under the VDB section.

Algorithm 3 Pseudocode - FITS script for generating VDB volume

```
//writeFITSVDB() HDA PythonModule
create data list = []
if data dimensions = 4:
 set data = HDUdata[0] //ignore polarization axis
else:
 set data = HDUdata

set data = cast to float32(data) //cast to float32
set data = remove nan(data) //numpy not-a-number values

create vdb FloatGrid 'dataGrid'
dataGrid.copy from array(data)
#set volume name from HDA input
set dataGrid name = volumeName
set vdbFileName = VDBPath + volumeName + '.vdb'
write volume grids = dataGrid
```

6.2 CSV

CSV tables only contains two dimensional tabular data, so the columns to visualise data from depends on the type of data. The column names are first loaded as detail attributes and read from the HDA interface into ordered menus for the user to select. As with FITS files, ordered menus are populated using python scripts.

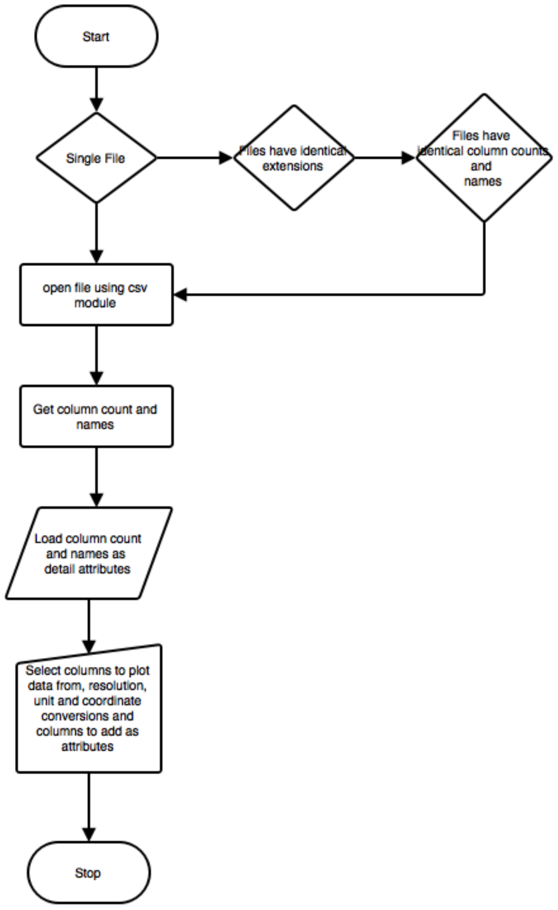

Start
Single File
Files have identical extensions
Files have identical column counts and names
open file using csv module
Get column count and names
Load column count and names as detail attributes
Select columns to plot data from, resolution, unit and coordinate conversions and columns to add as attributes
Stop

Figure 12: Flowchart - CSV file analysis

Houdini provides a 'Table Import' node built using python code, but it provides limited functionality. Multiple files cannot be read at the same time, and reading columns and rows is a static, preset process. For this project, it was decided to use Python's CSV module to give these functionalities to the user. The Gaia data is split between multiple files, and a number of attributes such as intensity and flux can be added to the data points. This is only possible if the user is able to pick the columns to be read. Attributes from specific columns can be added using the 'Add Point Attributes', by separating the column names by a comma. The user can limit the number of files and rows read for testing purposes.

6.2.1 Algorithm

Algorithm 4 Pseudocode - CSV Visualization as Points

```
# read CSV file
//readFile()
    csvfile = csv.open(file name)
    reader = csv reader
    header = first row or table

# build points from data
//buildGeometry():
# make dictionary of columns in CSV table
header = name:index for header

for each row in table rows:
 # set x, y, x as per GUI axis selection
 x = convert to float(xAxisColumn)
 y = convert to float(yAxisColumn)
 z = convert to float(zAxisColumn)

 create point
 set point position(x, y, z)
 set attribute value magnitude = convert to float(item \
 at index of magnitude column)
 # add filenumber and row number to control in wrangle node
 set Attribute Value fileNumber
 set Attribute Value rowNumber

 # add attributes from 'add point attributes' parameter \
 on HDA
 for attribute name in Attributes:
  #double check if attribute exists
  if attribute name in header:
   attribute value = convert to float(item at index of \
   attribute column)
   set point attribute Value
```

6.3 AMR

AMR files are read in the analysis node to get the refinement levels and fields contained the grids. Fields such as creation time, particle position, particle mass, density, temperature etc. are recorded in AMR files, and can be used to build the volumes, depending upon the user input.

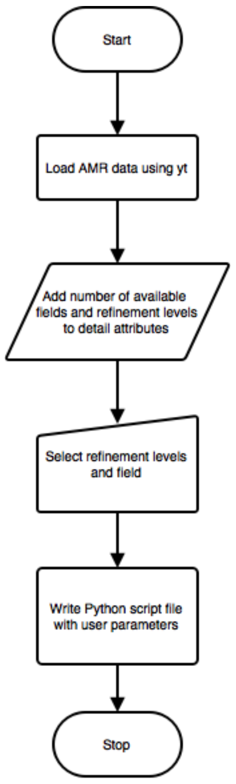

Start
Load AMR data using yt
Add number of available fields and refinement levels to detail attributes
Select refinement levels and field
Write Python script file with user parameters
Stop

Figure 13: Flowchart - AMR file analysis

The AMR module works differently from the FITS and CSV modules, because the volumes are not built on the fly from within Houdini. Although Houdini provides an python API to use VDB volumes, it was found to be very limited in terms of accessing and manipulating the voxel values. Most functions are provided to read VDB volumes and determine the voxel values in specific indices, the resolution, bounding box and transforms.

The PyOpenVDB library provided fast and easy manipulation of voxel data and the ability to copy data directly from 3 dimensional arrays into a volume. Hence, in the AMR module, the user input is used to write a python script that will build the VDB files outside of Houdini. Once the field and refinement levels are selected, the script is prepared, and the user has to run the script from a terminal in order to b build the volumes. There will be a VDB file for each refinement level in the AMR file. After this, the files have to be imported into Houdini using the file node and merged. A shader has been provided to test the built VDB volumes (Naiman, J. P. et al, 2017).

While populating voxel values into a VDB volume from the AMR data, the voxel size is always constant for a volume. After all refinement levels are parsed and the subgrids built, the VDB volumes are transformed to accurately fit in the masked regions of the grid built using the parent grid of the AMR. The following figure illustrates this process.

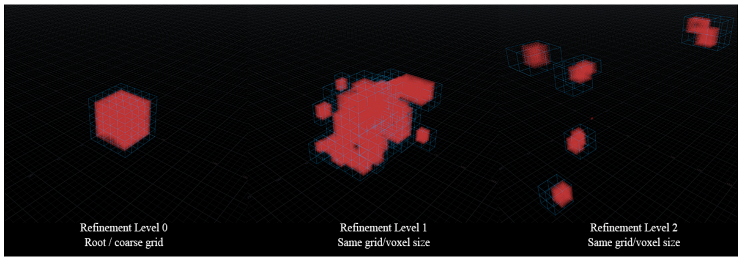


Figure 14: AMR data in VDB volumes before scaling

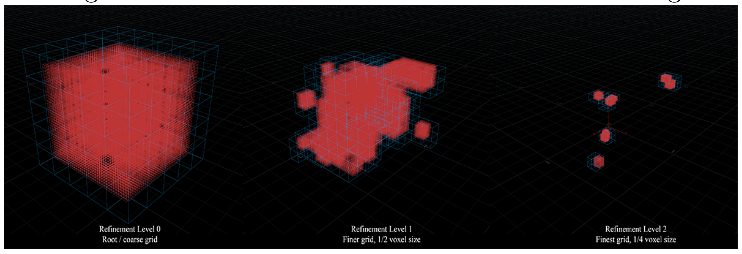


Figure 15: AMR data in VDB volumes after scaling

Notice the voxel sizes are the same in the 3 refinement levels in figure 14. These cannot be merged into a single volume to resemble an AMR cube unless they are scaled. VDB volumes were the ideal choice for this because they are sparse volumes. They are also multi-threaded, which means that AMR data of higher refinement levels (>50) can be read and loaded into VDB volumes relatively faster than with normal volumes. For multi-threaded insertion operations, separate grids are assigned to each thread and then merged as threads terminate. This technique is extremely efficient, because OpenVDB is sparse and hierarchical. (OpenVDB, 2017)

The yt Python library was chosen to read AMR files. Yt provided a robust solution to read and analyze Enzo AMR files and worked well with Houdini. PyOpenVDB provides functions to copy 3D array data into voxels of a VDB volume. Since AMR data has varying grid sizes within the same volume, the data can not be copied to a single grid of a VDB volume, since the voxel size is constant throughout any given grid. To overcome this, Yt provides the capability to build mask grids. A mask grid can be used to 'cut holes' in a VDB volume so that the voxel positions where the AMR grid contains denser sub-grids can be left with a background value. This space is then used to accurately place the sub-grids from the second refinement level of the AMR, and so on.

Enzo uses a dynamic load-balancing scheme to distribute the workload within each level of the AMR hierarchy evenly across all processors. Although each processor stores the entire distributed AMR hierarchy, not all processors contain all grid data. A grid is a real grid on a particular processor if its data is allocated to that processor, and a ghost grid if its data is allocated on a different processor. Each grid is a real grid on exactly one processor, and a ghost grid on all others. Each data field on a real grid is an array of zones with dimensionality equal to that of the simulation (typically 3D in cosmological structure formation). Zones are partitioned into a core block of real zones and a surrounding layer of ghost zones. Real zones are used to store the data field values, and ghost zones are used to temporarily store values from surrounding areas, ie, neighboring grids, parent grids or external boundary conditions, when required for updating real zones (Enzo Developers, 2017).

6.3.1 Algorithm

Algorithm 5 Copy AMR data into VDB volume

```
for level in refinement levels:
 gridSet = dataSet.index.select_grids(level)

 create mask VDB Grid
 create data VDB Grid

 for each subgrids in gridSet:
  #get field name from HDA input eg.'Density'
  set subGrid = currrent sub Grids[field]
  set subGridGhostZone = subGrids.retrieve ghost zones \
  for field

  set mask = child mask(subGrids)
  set voxelStartIndex = start voxel Index of sub grid

  mask VDB Grid.copyFromArray(mask)
  data VDB Grid.copyFromArray(subGridGhostZone)

 set voxelNumbers = dataSet dimensions * dataSet \
 refinement ^ level
 #the simulation domain is the scale used for distance \
 measurements in the AMR file.eg. kpc, au, mile etc.
 set voxelSize = cast to float(data Set domain \
 width / voxel count)
 #scale down both grids to the same voxel size
 set transform for mask VDB Grid = \
 createLinearTransform(voxelSize)
 set transform for data VDB Grid = \
 createLinearTransform(voxelSize)

 create VDB Grids list = []

 set data VDB Grid name = field
 set mask VDB Grid name = 'mask'
 append mask to VDB Grids list
 append data to VDB Grids list
 set VDB File Name = VDBPath + field + 'level' + '.vdb'
 write VDB file (vdbFileName, grids = VDB Grids)
```

7 Results

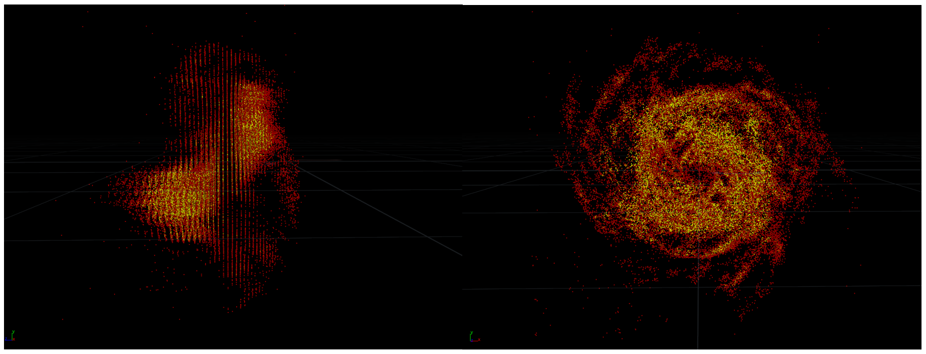


Figure 16: FITS data cube visualized as points at 50% resolution

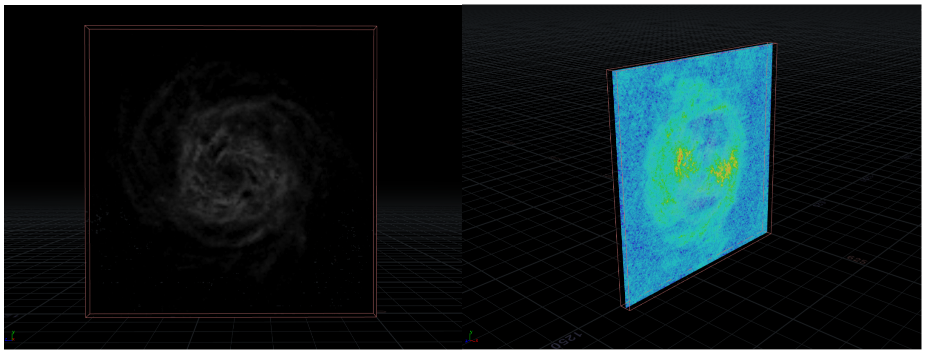
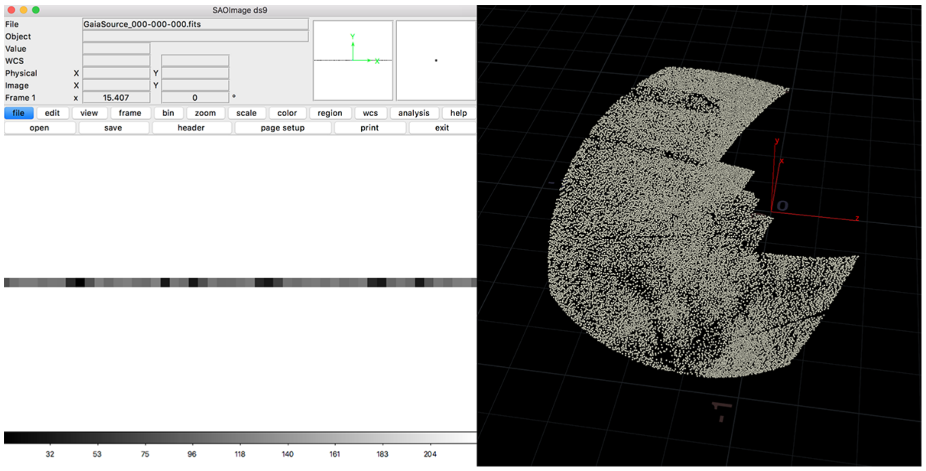


Figure 17: FITS data cube volume and histogram view



File GaiaSource_000-000-000.fits
Object
Value
WCS
Physical X Y
Image X Y
Frame 1 x 15.407 0 °

Figure 18: Gaia FITS file viewed using DS9 vs AstroVis

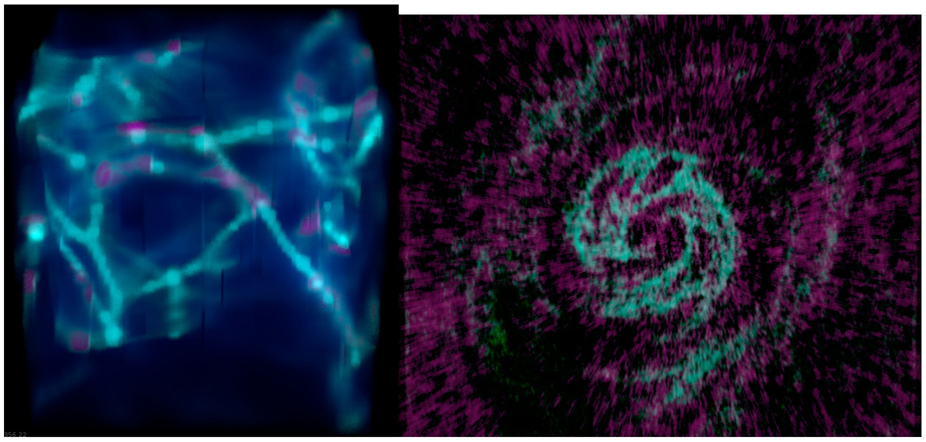


Figure 19: AMR and FITS VDB volumes

8 Problems

1. Performance - The data sets used for this project were extremely large due to the amount of data points captured by the telescopes. Most of these are extremely high resolution radio telescopes that observe and capture data over many years. The FITS file for the galaxy NGC628 contains over 60 million floating point numbers stored in a 3D array. A single CSV file from the Gaia source contains over 130,000 rows and 57 columns with floating point values. Testing the visualisation proved to be extremely slow, so most of the testing was done using a reduced resolution. However,

the final renders were done at full resolution. It was noticed that the processor-heavy activities such as reading and plotting data points from the FITS data cubes used only a single core. Python in Houdini is by default single threaded, but the multiprocessing module allows processes to be split between threads. This was initially considered to speed up the visualisation process but, upon further research, it was found that the Python interpreter in Houdini is not a separate process in Houdini. Since it's part of the same process as Houdini, the multiprocessing module can only fork the Houdini process. The numpy package in itself improved the performance to a great extent. In addition to this, the itertools module improved the performance further, and also simplified the code. Nested loops were removed and replaced by the product iterator (See Appendix 13.2.2).

2. API compatibility - Four external Python libraries were used for this project - Numpy, PyFITS, yt, and PyOpenVDB. Although yt provides functions to open and read FITS files, it was observed that PyFITS was faster, as well as easy to comprehend and integrate into Python. PyOpenVDB worked well in the native Python environment on a Macintosh, but caused Houdini to crash upon import. This was observed on both Houdini 15 and 16.

3. Houdini Inlinecpp module - To avoid running an external Python script for the AMR to VDB conversion, an attempt was made to write c++ functions using the inlinecpp Python module in Houdini. Using the c++ library for OpenVDB, three functions were written to build the volumes, insert values and write the VDB files to disk. Although it was relatively simple to pass arguments and get return values from the functions, sending data between the c++ functions proved to be nearly impossible, because iterating through the AMR refinement levels had to be done using yt in Python, and that meant moving the OpenVDB floatGrid datatype in Python, which was not possible (See Appendix 13.2.7).

4. Array Iterators: Using nested loops to iterate over 3D arrays proved to be very slow. This was required to read the FITS data cubes and create the VDB volumes. PyOpenVDB provides a function called copyfromarray that can be used to directly copy 3D array data into a volume, as long as the dimensions of the volume and the array match. However, this did not work initially, because of data-type conflicts. Since the entire data set is first loaded into a variable eliminating 'nan' values, the data was getting corrupted because of the conversion from 32 bit single precision floats to numpy floats. Therefore, the data had to be cast to float32 using Numpy's astype function. The other way to load the voxel values was to use a combination of the product iterator from itertools and input values per-voxel using an voxel accessor and the setVoxel method. This was also slower than using the copyfromarray function (See appendix 13.2.3)

9 Conclusion

The objective of this project was to extract and visualise data from astronomical data sources in a way that can be used for visual effects. Current software solutions that read such data cannot be used for this purpose, and focus on the technical aspects rather than the artistic. By using appropriate shaders and lighting, the geometries and volumes produced using this tool can be used to visualize astronomical data in an efficient, visually appealing way. Apart from a few minor caveats, the project has been a success.

10 Future Work

10.1 Integration of PyOpenVDB in Houdini

Since VDB volumes are sparse, they are considerably faster and more memory efficient than standard Houdini volumes. By removing the process of outputting a script to generate the VDB's this would be a complete self-contained tool to visualize data cubes.

10.2 Spiral galaxy simulations

An example is shown with an axis force applied to the visualised points from a FITS data cube. It would be beneficial to refine that further using better algorithms and physically observed values for the galaxy's movement. This can be used for simulations.

10.3 Multi-threading

More research into multi-threading capabilities in the Houdini python environment needs to be done. The multiprocessing module of Python is very useful when dealing with large arrays and can also help to stop Houdini from freezing while the geometry is cooking.

11 Acknowledgements

This project made use of THINGS, 'The HI Nearby Galaxy Survey' (Walter et al. 2008)

This project has made use of data from the European Space Agency (ESA) mission Gaia (https://www.cosmos.esa.int/gaia), processed by the Gaia Data Processing and Analysis Consortium (DPAC, https://www.cosmos.esa.int/web/gaia/dpac/consortium). Funding for the DPAC has been provided by national institutions, in particular the institutions participating in the Gaia Multilateral Agreement.

The shaders used for the AMR volumes were obtained from http://www.ytini.com/docs-assets/tutorial_amr/amr_shader.hdanc

Many thanks to Jon Macey, Phil Spicer and the class of 2016-17, MSc Computer Animation and Visual Effects, Bournemouth University.

12 References

Almgren, A. S., 2011. Introduction to Block-Structured Adaptive Mesh Refinement (AMR). Center for Computational Sciences and Engineering, 1-11. Available from: http://hipacc.ucsc.edu/Lecture%20Slides/2011ISSAC/Almgren HIPACC _July2011.pdf [Accessed 3 August 2017].

Berger, M. J. and Oliger, J., 1984. Adaptive mesh refinement for hyperbolic partial equations. Journal of Computational Physics, 53, 484-512.

Bernhard, J., 2012. Principles of Data Visualisation. CSE512 Data Visualization, Available from: http://stat.pugetsound.edu/courses/class13/ dataVisualization.pdf [Accessed 11 August 2017].

Bridle, A.,Nance, J., 1991. The AIPS FAQ. The 1990 AIPS Site Survey, AIPS Memo No. 70, National Radio Astronomy Observatory. Available from: http://www.aips.nrao.edu/aips_faq.html#AIPS.1 [Accessed 21 July 2017].

Bryan, G. L., Norman, M. L., O'Shea, B. W. et al., 2013. Enzo: An Adaptiv Mesh Refinement Code for Astrophysics. Instrumentation and Methods for Astrophysics (astro-ph.IM), 1. Available from: https://arxiv.org/pdf/1307.2265.pdf [Accessed 1 August 2017].

Enzo Developers, 2017. Adaptive Mesh Refinement. Available from: http:// enzo.readthedocs.io/en/latest/reference/EnzoAlgorithms.html [Accessed 3 August 2017].

Europian Space Agency, 2016. Gaia Archive. European Space Agency. Available from: https://gea.esac.esa.int/archive/ [Accessed 4 August 2017].

Fox, G., 2015. Big Data Open Source Software and Projects ABDS in Summary XI: Layer 11A, 4-5. School of Informatics and Computing Digital Science Center Indiana University Bloomington, Available from: http:// bigdataopensourceprojects.soic.indiana.edu/downloads/section2_unit8_ sp2015/ BDOSSP-Spring 2015- 02L11ASlides. pptx. [Accessed 2 August 2017].

Guarrasi, M., 2015. An introduction to Adaptive Mesh Refinement (AMR): Numerical Methods and Tools. Super Computing Applications and Innovation Department - HPC Numerical Libraries, 2. Available from: http://www.training.

prace-ri.eu/uploads/tx_pracetmo/AMRIntroHNDSCi15.pdf [Accessed 21 July 2017].

Hassan, A., Fluke, C. J., 2011. Scientific Visualization in Astronomy: Towards the Petascale Astronomy Era. Instrumentation and Methods for Astrophysics (astro-ph.IM), Available from: https://arxiv.org/abs/1102.5123 [Accessed 6 July 2017].

HST Data Handbook for WFPC2. Graphic. Space Telescope Science Institute. Available from: http://www.stsci.edu/instruments/wfpc2/Wfpc2_dhb/intro_ch23.html [Accessed 10 August 2017].

Jackson et al., 2006. Galactic Ring Survey. apjs, 163, 145, Nash, D.. Available from: https://www.bu.edu/galacticring/new_index.htm [Accessed 12 July 2017].

Kent, B. R., 2013. Visualizing Astronomical Data with Blender. Publications of the Astronomical Society of the Pacific, Available from: https://arxiv.org/abs/1306.3481 [Accessed 24 June 2017].

McClure-Griffiths, N. M.; Green, A. J.; Dickey, John M.; Gaensler, B. M.; Haynes, R. F.; Wieringa, M. H., 2001. The Southern Galactic Plane Survey: The Test Region. The Astrophysical Journal, 551, 394-412. Available from: http://www.ras.ucalgary.ca/VGPS/introduction.html [Accessed 9 August 2017].

McGlynn, T. A., 2016. The FITS Support Office. High Energy Astrophysics Science Archive Research Center, Available from: https://fits.gsfc.nasa.gov [Accessed 26 June 2017].

Meshing - Octree, Degree & Samples per Node Explained, 2015. Graphic. Available from: https://matterandform.desk.com/customer/en/portal/articles/2107547-meshing—octree-degree-samples-per-node-explained
[Accessed 10 August 2017].

Mitra, S. et.al., 2001. Adaptive Mesh Refinement. Distributed Adaptive Grid Hierarchy : User Guide, Available from: http://www.cs.utexas.edu/users/dagh/ch2.html [Accessed 19 July 2017].

Museth, K., Budsberg, J., Hankins, R., Keeler, T., Bailey, D., Hoetzlein, R., 2013. 27 VDB: High-Resolution Sparse Volumes with Dynamic Topology. Available from: http://www.museth.org/Ken/Publications_files/Museth_TOG13.pdf [Accessed 7 July 2017].

Naiman, J. P., Borkeiwicz, K., Christensen, A. J., 2017. Houdini for Astrophysical Visualisation. Publications of the Astronomical Society of Pacific, Volume 129, Issue 975, Available from: https://arxiv.org/abs/1701.01730 [Accessed 11 August 2017].

Nash, D., 2006. The HYG Database. Portland, Oregon, Nash, D.. Available from: http://www.astronexus.com/hyg [Accessed 12 July 2017].

National Research Council, 1995. Preserving Scientific Data on Our Physical Universe: A New Strategy for Archiving the Nation's Scientific Information Resources. Washington, DC, The National Academies Press. Available from: https://doi.org/10.17226/4871 [Accessed 13 July 2017].

Norman, M. L., Shalf, J., Levy, S., Daues, G., 1999. Diving Deep: Data management and visualisatoin strategies for adaptive mesh refinement simulations. 1-3. Available from: http://users. eecs.northwestern.edu/~wkliao/ENZO/ diving_deep.pdf [Accessed 2 August 2017].

OpenVDB, 2017. FAQ. OpenVDB.org. Available from: http://www. openvdb.org/documentation/doxygen/faq.html [Accessed 11 August 2017].

Pence, W. D., et. al., 2008. Definition of the Flexible Image Transport System (FITS). FITS Working Group Commission 5: Documentation and Astronomical Data International Astronomical Union. 3.0. Available from: http://fits. gsfc.nasa.gov/iaufwg/ [Accessed 29 July 2017].

Rosenstein, N., 2017. Importing Satellite Data into Houdini. Available from: https://www.niklasrosenstein.com/post/2017-07-17-entagma- tutorial- importing-satellite- data-into-houdini/ [Accessed 20 July 2017].

Simons, B., Visualisation of Large Datasets with Houdini. Sydney, Australia: Available from: http://www.utas.edu.au/__data/assets/pdf_file/0006/ 415536/BenSimons.pdf [Accessed 19 July 2017].

Strobel, N., 2017. Astronomy Notes. Available from: http://www. astronomynotes.com/telescop/chindex.htm [Accessed 10 July 2017].

Taylor, R., 2015. FRELLED - Fits Realtime Explorer of Low Latency in Every Dimension. Available from: http://www.rhysy.net/frelled-1.html [Accessed 17 July 2017].

The HI 21 cm Line. The National Radio Astronomy Observatory. Available from: http://www.cv.nrao.edu/course/astr534/HILine.html [Accessed 27 July 2017].

Topping, B. H., Soares, C. A., 2004. Progress in Engineering Computational Technology. Stirling, United Kingdom, Saxe-Coburg Publications. Available from: http://www.saxe-coburg.co.uk/pubs/contents/sle04_04.htm [Accessed 10 August 2017].

VanderPlas, J., 2012. Quad Tree Example. Graphic. Available from: http://www.astroml.org/book_figures/chapter2/fig_quadtree_example.html [Accessed 10 August 2017].

Vohl, D., Fluke, C. J., Barnes, D. G., Hassan, A., H., 2017. Real-time colouring and filtering with graphics shaders. Instrumentation and Methods for Astrophysics (astro-ph.IM), Available from: https://arxiv.org/abs/1707.00442 [Accessed 24 June 2017].

Walter, F., 2007. The HI Nearby Galaxy Survey. The National Radio Astronomy Observatory. Available from: http://www.mpia.de/THINGS/ Overview. html [Accessed 7 August 2017].

13 Appendix

13.1 Synonyms

FITS - Flexible Image Transport System
VLA - Very Large Array
VGPS - VLA Galactic plane Survey
HYG - Hipparcos, Yale Bright Star, and Gliese
AMR - Adaptive Mesh Refinement
CSV - Comma Separated Values
GRS - Galactic ring Suvey
FCRAO - Five College Radio Astronomy Observatory
NRAO - National Radio Astronomy Observatory
TGAS - Tycho-Gaia Astrometric Solution
THINGS - The HI Nearby Galaxy Survey
ISM - Inter-Stellar Medium

13.2 Code

13.2.1 HDA ordered menu creation

```
attribList = []
if (node.evalParm('FITS_data_source')=="0"):
  #0 is always primary
  dataSource_suffix="prim_"
else:
  dataSource_suffix="ext_"

if geo and geo.findGlobalAttrib(dataSource_suffix+'NAXIS'):
  #Metadata, no datatype suffix
  if (geo.attribValue(dataSource_suffix+'NAXIS')==1):
   iterateMax=1
   return ("0",geo.attribValue(dataSource_suffix+'NAXIS1'))
  elif (geo.attribValue(dataSource_suffix+'NAXIS')==2):
   #2D matrix / table / CSV
   dataType_suffix = "T"
  if geo.findGlobalAttrib(dataSource_suffix+'TFIELDS'):
   iterateMax=geo.attribValue(dataSource_suffix+ \
   'TFIELDS')+1
  else:
   dataType_suffix = "C"
   iterateMax=geo.attribValue(dataSource_suffix+'NAXIS')
if iterateMax != 1:
  attr_name = dataSource_suffix+dataType_suffix+'TYPE'
  for i in range(1, iterateMax):
   #1-indexed, add all possible coordinate formats
   attribList.append(i)
   attribList.append(geo.attribValue(attr_name+ \
   str(i)).upper())
return attribList
```

13.2.2 FITS Points Visualisation

```
# global lists
hduList = []
tFields = []
addedAttr = []

# add attributes to pass to next node
def addAttrs(hdu):
 global tFields
 global addedAttr

 #layer number to control image plane spacing
 if not geo.findPointAttrib("layer"):
  geo.addAttrib(hou.attribType.Point, "layer", 0)
 #main attribute that controls point properties
 if not geo.findPointAttrib("magnitude"):
  geo.addAttrib(hou.attribType.Point, "magnitude", 0.0)

 #add column counts and names for matrix data
 if geo.findGlobalAttrib(hdu + "_TFIELDS"):
  for i in range(1, geo.attribValue(hdu + \
  "_TFIELDS") + 1):
   if geo.findGlobalAttrib(hdu + "_TTYPE" + str(i)):
    tFields.append(geo.attribValue(hdu + "_TTYPE" + \
    str(i)).split(" : ")[0])

# add attributes from 'add point attributes'
 for attrName in extraAttr:
 if attrName.replace(" ", "").lower() in tFields:
  geo.addAttrib(hou.attribType.Point, \
  attrName.replace(" ", ""), 0.0)
  addedAttr.append(attrName.replace(" ", ""))

# build points or volume based on GUI settings
def buildGeometry(fileOrDir):

global hduList
```

FITS Points Visualisation cont.

```
#load fits data
 try:
  hduList = fits.open(fileOrDir)
 except Exception, e:
  hou.ui.displayMessage(Error Message)
  return

 #Primary HDU data, single run
 if hduNum == 0:
  addAttrs("prim")
  if geo.findGlobalAttrib('prim_NAXIS'):
   # Matrix data. Ignore 1D metadata.
   if geo.attribValue('prim_NAXIS') == 2:
    # build from single primary hdu
    build_from_matrix(hduList[0].data)

   elif geo.attribValue('prim_NAXIS') > 2:
    if geo.attribValue('prim_NAXIS') == 4:
      # ignore polarization axis (eg: STOKES)
      build_from_datacube(hduList[hduNum].data[0])
    else:
      build_from_datacube(hduList[hduNum].data)

# extension HDU data, possible multiple runs
 else:
 addAttrs("ext")
 if geo.findGlobalAttrib('ext_NAXIS'):
  # Matrix data. Ignore 1D metadata.
  if geo.attribValue('ext_NAXIS') == 2:
    # concat data from all extensions
    data = []
    for i in range(1, len(hduList)):
      currentHduData = list(hduList[i].data)
      data.append(currentHduData)

      # build from concatenated data
      build_from_matrix(data)
```

FITS Points Visualisation cont.

```
# 3D data cube
  elif geo.attribValue('ext_NAXIS') > 2 and \
   geo.attribValue('ext_NAXIS3') > 1:
    if geo.attribValue('ext_NAXIS') == 4:
     #ignore polarization axis (eg: STOKES)
     build_from_datacube(hduList[hduNum].data[0])
    else:
      build_from_datacube(hduList[hduNum].data)

# build points from 3D array (FITS data cube)
def build_from_datacube(data):
 # get np.shape of data cube
 axisLengths = list(reversed(np.shape(data)))

 # assign as per GUI axis selection
 xAxisLen = axisLengths[xAxis-1]
 yAxisLen = axisLengths[yAxis-1]
 zAxisLen = axisLengths[zAxis-1]

 # iterate through data cube and set point positions
 for x, y, z in product(range(0, xAxisLen, resolution),\
 range(0, yAxisLen, resolution), range(0, zAxisLen)):
  pt = geo.createPoint()
  # arbitrary image plane separation
  pt.setPosition((x, y, z))
  # set layer and magnitude point attributes
  pt.setAttribValue("magnitude", float(data[z][y][x]))
  pt.setAttribValue("layer", z)

# build points from 2D matrix
def build_from_matrix(data):
 global addedAttr
 global tFields
```

FITS Points Visualisation cont.

```
# set points from each HDU
 for i in range(len(data)):
  for j in range(0, len(data[i]), resolution):
   #        data[hdu][row][column]
   x = float(data[i][j][xAxis-1])
   y = float(data[i][j][yAxis-1])
   # manual z
   if zAxis == 0:
    # arbitrary image plane separation
    z = 0.001 * i
   else:
    z = float(data[i][j][zAxis-1])
    pt = geo.createPoint()
    pt.setPosition((x, y, z))
    # set layer and magnitude point attributes
    pt.setAttribValue("magnitude", \
    float(data[i][j][magnitude - 1]))
    pt.setAttribValue("layer", i)
    # add point attributes from Matrix (HDA interface)
    for attr in addedAttr:
     pt.setAttribValue(attr, data[i][j][tFields.index(attr)])

def main():
# run if 'live build' is enabled.
if node.evalParm("../FITS_live_build_points") and \
int(node.evalParm("../FITS_build_as")) == 0:# and \
(xAxis != yAxis != int(zAxis)+1 != magnitude):

 if xAxis == yAxis or xAxis == zAxis or yAxis == zAxis:
  return
 else:
  # multiple files in folder
  if os.path.isdir(fileOrDir):
   for file in os.listdir(fileOrDir):
    buildGeometry(fileOrDir + file)
  else:
   # single file
   buildGeometry(fileOrDir)
else:
    return

main()
```

13.2.3 FITS Volume Visualization

```
# create Houdini volume and populate voxel values
fileNum = 0
def buildVolume(fileOrDir):
 global fileNum
  #read FITS file
 try:
  hdu_list = fits.open(fileOrDir)
 except Exception, e:
  hou.ui.displayMessage(Error Message)
  return

#ignore polarization axis (eg: STOKES)
 if hdu_list[hduNum].data.ndim == 4:
  # avoid numpy 'not-a-number' values and cast to float
  data = nan_to_num(hdu_list[hduNum].data[0]).astype(float)
 elif hdu_list[hduNum].data.ndim == 3:
  data = nan_to_num(hdu_list[hduNum].data).astype(float)

 bbox = hou.BoundingBox(fileNum * len(data),0,0,\
 len(data),len(data[0]),len(data[0][0]))
 #create volume using data axis lengths
 vol = geo.createVolume(len(data), len(data[0]), \
     len(data[0][0]), bbox)
 #load voxel values using itertools
 for i,j,k in product(range(len(data)),\
 range(len(data[0])),range(len(data[0][0]))):
  vol.setVoxel((i,j,k), data[i][j][k])

def main():
 global fileNum
 fileNum = 0
 if hou.evalParm("../FITS_live_build_volume"):
  if os.path.isdir(fileOrDir):
    for file in os.listdir(fileOrDir):
      buildVolume(fileOrDir + file)
      fileNum += 1
  else:
    buildVolume(fileOrDir)

main()
```

13.2.4 CSV Visualization

```
fileOrDir = ""
# build points from data
def buildGeometry(reader, header, fileNum):
# make dictionary of {column index:column name}
header={name:index for (index, name) in enumerate(header)}

try:
 for i, row in enumerate(reader):
  if i < rowCount or rowCount == 0:
   #set x, y, x as per GUI axis selection
   x = float(row[header[geo.attribValue("column" + \
   str(xAxis))]])
   y = float(row[header[geo.attribValue("column" + \
   str(yAxis))]])
   z = float(row[header[geo.attribValue("column" + \
   str(zAxis))]])

   pt = geo.createPoint()
   pt.setPosition((x, y, z))
   if geo.findGlobalAttrib("column" + str(magnitude)):
    pt.setAttribValue("magnitude", \
    float(row[header[geo.attribValue("column" + \
    str(magnitude))]]))
    # add file and row number to control in wrangle node
    pt.setAttribValue("fileNum", fileNum)
    pt.setAttribValue("rowNum", i+1)

 # add attributed from 'add point attributes' parameter
 for attrName in extraAttr:
  #double check if attr exists
  if attrName in header and geo.findPointAttrib(attrName):
   attrValue = float(row[header[attrName]])
   if geo.findPointAttrib(attrName):
    pt.setAttribValue(attrName, attrValue)
except:
 hou.ui.displayMessage("Row " + str(i) + " in file no." +\
 str(fileNum)+" contains a NULLL byte.Visualization Failed.")
```

CSV Visualisation cont.

```
#read CSV file
def readFile(filename):
 csvfile = open(filename)
 #reader = csv.reader(csvfile)
 reader = csv.reader(x.replace('\0', '') for x in csvfile)
 header = next(reader)
 return reader, header

def main():
if not node.evalParm("../CSV_live_build_points") or \
xAxis == yAxis or xAxis == int(zAxis)+1 or \
yAxis == int(zAxis)+1:

 return
else:
 global fileOrDir
 try:
  fileOrDir = node.evalParm("../CSV_file_name")
 except:
  return

 if not fileOrDir:
  return
 else:
  filename, file_extension = os.path.splitext(fileOrDir)
  # single file
  if file_extension.lower() == '.csv':
    # csv reader and list of column names
    reader, header = readFile(fileOrDir)
    buildGeometry(reader, header, 0)

  # folder
  elif os.path.isdir(fileOrDir):
   header = readFile(fileOrDir + os.listdir(fileOrDir)[0])[1]
   fileCount = node.evalParm('../CSV_file_count')
   for i, file in enumerate(os.listdir(fileOrDir)):
    if i < fileCount or fileCount == 0:
      reader = readFile(fileOrDir + file)[0]
      buildGeometry(reader, header, i+1)

main()
```

13.2.5 AMR VDB Creation (Naiman, J. P. et al, 2017)

```
# write out the script to build VDB volumes from AMR files
# modified from http://www.ytini.com/tutorials/tutorial_amr.html
def write_AMR_script():
 geo = node.geometry()
 fileName = node.evalParm("AMR_file_name")
 VDBPath = node.evalParm("AMR_VDB_output")
 scriptPath = node.evalParm("AMR_script_output")

 # All refinement levels
 if int(node.evalParm("AMR_ref_levels")) == 0:
  levels = int(geo.attribValue("levels"))
 else:
  # selected level = total levels - selected menu index
  # (levels are loaded in reverse in menu script)
  levels = int(geo.attribValue("levels")) - \
  int(node.evalParm("AMR_ref_levels"))

 field = geo.attribValue("field" + \
 str(node.evalParm("build_field"))).split(':')[1]

text = """
import yt
import pyopenvdb as vdb
import numpy as np
import os

fileName = '%s'
levels = %d
field = '%s'
VDBPath = '%s'

dataSet = yt.load(fileName)

if not os.path.exists(VDBPath):
 os.makedirs(VDBPath)
```

AMR VDB Creation cont.

```
for level in range(0, levels):
gridSet = dataSet.index.select_grids(level)

maskVdbGrid = vdb.FloatGrid()
dataVdbGrid = vdb.FloatGrid()

for i in range(len(gridSet)):

 subGrids = gridSet[i]
 subGrid = subGrids[field]
 subGridGhostZone = subGrids.retrieve_ghost_zones(n_zones\
 = 1, fields = field)[field]

 mask = subGrids.child_mask
 voxelStartIndex = subGrids.get_global_startindex()

 maskVdbGrid.copyFromArray(mask, ijk = \
 (voxelStartIndex[0], voxelStartIndex[1],voxelStartIndex[2]))
 dataVdbGrid.copyFromArray(subGridGhostZone, ijk = \
 (voxelStartIndex[0],voxelStartIndex[1],voxelStartIndex[2]))

voxelNumbers = dataSet.domain_dimensions * \
pow(dataSet.refine_by, level)
voxelSize = float(float(dataSet.domain_width[0]) \
/ float(voxelNumbers[0]))
maskVdbGrid.transform = vdb.createLinearTransform(voxelSize)
dataVdbGrid.transform = vdb.createLinearTransform(voxelSize)

vdbGrids = []
dataVdbGrid.name = field
maskVdbGrid.name = 'mask'
vdbGrids.append(maskVdbGrid)
vdbGrids.append(dataVdbGrid)
vdbFileName = VDBPath+field+'_level'+str(level)+'.vdb'
vdb.write(vdbFileName, grids = vdbGrids) """

if not os.path.exists(scriptPath):
 os.makedirs(scriptPath)

with open(scriptPath + "writeAMRVDB.py", "w") as text_file:
 text_file.write(text%(fileName, levels, field, VDBPath))
```

13.2.6 FITS VDB Creation

```
def write_FITS_script():
 geo = node.geometry()
 # primary HDU
 if int(node.evalParm("FITS_data_source")) == 0:
  hduIndex = 0
  axisLength = geo.attribValue("prim_NAXIS")
 #Extension HDU
 else:
  hduIndex = 1
  axisLength = geo.attribValue("ext_NAXIS")

 #verify content is datacube
 if axisLength > 2:
  fileName = node.evalParm("FITS_file_name")
  volumeName=node.evalParm("FITS_VDB_name").replace(" ","")
  VDBPath = node.evalParm("FITS_VDB_output")
  scriptPath = node.evalParm("FITS_script_output")

  # script text
 text = """
import pyopenvdb as vdb
import pyfits as fits
import os
from numpy import float32, nan_to_num
from itertools import product

fileName = '%s'
hduIndex = %d
axisLength = %d
volumeName = '%s'
VDBPath = '%s'

 hdulist = fits.open(fileName)
 if axisLength == 3:
  data = nan_to_num(hdulist[hduIndex].data).astype(float32)
 else:
  #Ignore polarization data
  data=nan_to_num(hdulist[hduIndex].data[0]).astype(float32)
```

FITS VDB Creation cont.

```
dataGrid = vdb.FloatGrid()
dataGrid.copyFromArray(data)
dataGrid.name = volumeName
vdbFileName = VDBPath + '/' + volumeName + '.vdb'
vdb.write(vdbFileName, grids = dataGrid) """

if not os.path.exists(scriptPath):
 os.makedirs(scriptPath)

with open(scriptPath + "/writeFITSVDB.py", "w") as \
text_file:
 text_file.write(text % (fileName, hduIndex, axisLength,\
 volumeName, VDBPath))
```

13.2.7 Trials using inlincpp module in Python

```
import inlinecpp
import yt
import numpy as np
import os

node = hou.pwd()
geo = node.geometry()

# Add code to modify contents of geo.
# Use drop down menu to select examples.

#print node.evalParm("../build_field")

fileName = geo.attribValue("filename")
VDBPath = node.evalParm("../AMR_output_folder")
levels = int(node.evalParm("../AMR_ref_levels"))
field = geo.attribValue("field" + str(node.evalParm("../
build_field"))).split(':')[1]
something = 1
data = yt.load(fileName)

for level in range(0, levels + 1):
 gridSet = data.index.select_grids(level)
 voxelCount = data.domain_dimensions * pow(data.refine_by
 , level)
 voxelSize = float(float(data.domain_width[0])/float
 (voxelCount[0]))

 maskCube = vdbCreate_module.createVDB()

 for i in range(len(gridSet)):
  grid = gridSet[i]
  subGrid = grid[field]
  subGridGhost = grid.retrieve_ghost_zones(n_zones = 1,
  fields = field)[field]
  maskGrid = grid.child_mask
  ijkout = grid.get_global_startindex()

  #vdbFill_module.fillVDB(maskCube, ijkout, maskGrid)
  #vdbFill_module.fillVDB(dataCube, ijkout, subGridGhost)
```

Trials using Inlinecpp module in Python cont.

```
#vdbOutput_module.outputVDB(voxelSize, VDBPath, fieldName)

'''_____________________________________________
INPUTS: data(maskGrid and subGridGhost),
_____________________________________________
ijkout, cube(maskCube and dataCube) - 2 runs
_____________________________________________'''

vdbFill_module = inlinecpp.createLibrary(
 name = "cpp_vdbFill_library",
 include_dirs = ["/usr/local/Cellar/openvdb/4.0.1/include"],
 link_dirs = ["/usr/local/lib"],
 link_libs = ["openvdb.4.0.2"],
 includes = """
 #include <openvdb/openvdb.h>
 #include <vector>
 """,
 function_sources = [
"""
void fillVDB(openvdb::FloatGrid::Ptr cube, std::vector<int>
ijkout, std::vector< std::vector< std::vector<int> > >
amrGrid)
{
 openvdb::FloatGrid::Accessor accessor = cube->getAccessor();
 for (int i = ijkout[0]; i < amrGrid.size(); i++)
 {
  for (int j = ijkout[1]; j < amrGrid[0].size(); j++)
  {
   for (int k = ijkout[2]; k < amrGrid[0][0].size(); k++)
   {
    openvdb::Coord ijk(i, j, k);
    accessor.setValue(ijk, amrGrid[i][j][k]);
   }
  }
 }
}
"""])
```

Trials using Inlinecpp module in Python cont.

```
'''______________________________________________
INPUTS:  voxelSize ,  fieldName ,  VDBPath,
______________________________________________
dataCube ,  maskCube
______________________________________________'''

vdbOutput_module = inlinecpp.createLibrary(
 name = "cpp_vdbOutput_library",
 include_dirs = ["/usr/local/Cellar/openvdb/4.0.1/include"],
 link_dirs = ["/usr/local/lib"],
 link_libs = ["openvdb.4.0.2"],
 includes = """
 #include <openvdb/openvdb.h>
""",
function_sources = [
"""
void outputVDB(int voxelSize, std::string fieldName, std
::string vdbPath, openvdb::FloatGrid::Ptr dataCube,openvdb
::Float Grid::Ptr maskCube)
{
 maskCube->setTransform(openvdb::math::Transform::create
 LinearTransform(voxelSize));
 dataCube->setTransform(openvdb::math::Transform::create
 LinearTransform(voxelSize));
 openvdb::GridPtrVec cubes;
 dataCube->setName(fieldName);
 maskCube->setName("mask");
 cubes.push_back(dataCube);
 cubes.push_back(maskCube);
 openvdb::io::File file(vdbPath + "mygrids.vdb");
 file.write(cubes);
 file.close();
}
"""])      }
```