ARL-CR-0880 ● FEB 2025

Automated Measurement for the Length of Connected, Flagged Elements Reported from Finite Element Simulation in 3D Space: A Fracture Length Calculation Implementation

prepared by Alan R. Goertz
SURVICE Engineering Company
4695 Millennium Dr
Belcamp, MD 21017

under contract W15P7T-19-D-0126

NOTICES

Disclaimers

The research reported in this document was performed in connection with contract/instrument W15P7T-19-D-0126 with the U.S. Army Combat Capabilities Development Command (DEVCOM) Army Research Laboratory.

The findings in this report are not to be construed as an official Department of the Army position unless so designated by other authorized documents. The views and conclusions contained in this document are those of SURVICE Engineering Company and the DEVCOM Army Research Laboratory.

Citation of manufacturer's or trade names does not constitute an official endorsement or approval of the use thereof.

Destroy this report when it is no longer needed. Do not return it to the originator.

ARL-CR-0880 ● FEB 2025

Automated Measurement for the Length of Connected, Flagged Elements Reported from Finite Element Simulation in 3D Space: A Fracture Length Calculation Implementation

prepared by Alan R. Goertz
SURVICE Engineering Company
4695 Millennium Dr
Belcamp, MD 21017

under contract W15P7T-19-D-0126

REPORT DOCUMENTATION PAGE

1. REPORT DATE	2. REPORT TYPE	3. DATES COVERED	
		START DATE	END DATE
February 2025	Contractor Report	3/13/2024	1/15/2025

4. TITLE AND SUBTITLE
Automated Measurement for the Length of Connected, Flagged Elements Reported from Finite Element Simulation in 3D Space: A Fracture Length Calculation Implementation

5a. CONTRACT NUMBER	5b. GRANT NUMBER	5c. PROGRAM ELEMENT NUMBER
W15P7T-19-D-0126		
5d. PROJECT NUMBER	**5e. TASK NUMBER**	**5f. WORK UNIT NUMBER**

6. AUTHOR(S)
Alan R. Goertz

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)	8. PERFORMING ORGANIZATION REPORT NUMBER
SURVICE Engineering Company 4695 Millennium Dr Belcamp, MD 21017	

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)	10. SPONSOR/MONITOR'S ACRONYM(S)	11. SPONSOR/MONITOR'S REPORT NUMBER(S)
DEVCOM Army Research Laboratory ATTN: FCDD-RLA-TB Aberdeen Proving Ground, MD 21005		ARL-CR-0880

12. DISTRIBUTION/AVAILABILITY STATEMENT
DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.

13. SUPPLEMENTARY NOTES
ORCID: Alan R. Goertz, 0000-0003-3389-500X

14. ABSTRACT
Researchers commonly measure the lengths of fracture traces, either individually or summed up, as a metric to assess damage in fractured brittle or semibrittle materials like ceramics or bone. It is not unusual for brittle materials to fracture in highly complex patterns that may include hundreds of nonlinear fracture traces. Manual measurement of complex fracture patterns is both a tedious and time-consuming process. This report describes the implementation of an automated skull-fracture length measuring program that generates an interactive 3D image of skull-fracture traces predicted by LS-DYNA finite element simulations. The program directly reads LS-DYNA simulation output files and sums fracture lengths for behind-helmet blunt trauma injury assessment. The program's algorithm is capable of handling heavily comminuted fractures that are multiple elements thick. The program is written in Python and uses publicly available Python packages.

15. SUBJECT TERMS
fracture, trace measurement, injury assessment, human body simulation, LS-DYNA, d3plot, Python, Terminal Effects

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES
a. REPORT	b. ABSTRACT	C. THIS PAGE		
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UU	54

19a. NAME OF RESPONSIBLE PERSON	19b. PHONE NUMBER *(Include area code)*
Alan R. Goertz	(810) 444-6792

STANDARD FORM 298 (REV. 5/2020)
Prescribed by ANSI Std. Z39.18

Contents

List of Figures

List of Tables

Acknowledgments

I acknowledge the contributions of Samantha Wozniak and Timothy Baumer of ARL for their helpful feedback and suggestions through the composition of this report. Additionally, Samantha's efforts in patiently testing the code on hundreds of simulations during the debugging process were indispensable.

This work was supported in part by high-performance computer time and resources from the DoD High Performance Computing Modernization Program.

1. Introduction

There are several different types of injury models that can be used to assess and optimize Solider protection. One such model is the finite element (FE) method, which allows researchers to examine the unique loading environment experienced by Soldiers. FE models (FEMs) can be used to better understand injury-inducing events, including nonpenetrating impacts from helmets that cause behind-helmet blunt trauma (BHBT). Techniques for modeling brain tissue damage and neurological consequences are in their infancy, but modeling bone fracture has advanced to a much further state, making it a viable method for head injury assessment. Literature suggests that intracranial hemorrhaging and neurologic injuries are more likely to occur when a skull fracture is present (Wiederholt et al. 1989; Hofman et al. 2000; Bullock et al. 2006; Ono et al. 2007; Muñoz-Sánchez et al. 2009; Tseng et al. 2011; Loftis et al. 2020; Matheis et al. 2021); therefore, being able to model skull fracture may be relevant for a spectrum of possible head injuries.

The classification of injury severity will be dependent on several factors including total fracture length, fracture patterns, and the location of fracture(s). A simple linear fracture could easily be measured manually in a model; however, it is not uncommon for the flat bones of the skull (e.g., the frontal, occipital, and parietal bones) to undergo complex comminuted fracture patterns under impact loading. Complex fractures could potentially require measuring and summing scores of individual fracture traces, which could be a daunting task to perform manually. The task is further complicated by the curvature of the skull, which dictates measuring in 3D.

A Python program titled Automated Fracture Trace Measuring (AFTM) was developed to automate the process of measuring fracture trace lengths in the skull. The AFTM program utilizes freely available Python packages to directly read LS-DYNA simulation output files and generates an interactive 3D image of the fracture traces and calculated fracture lengths. The AFTM program is tailored to directly read simulation files output by LS-DYNA's explicit FE solver (Ansys) but can be easily adapted to read any trace data that can be continuously mapped into a 3D voxel volume (i.e., a 3D matrix).

2. Methods, Assumptions, and Procedures

2.1 Methods

In this study, an FEM of the human head previously developed to investigate blast loading (Zhang et al. 2013) was used to evaluate skull fracture induced from blunt impact loading (Figure 1). A range of fracture extents was generated by prescribing varied blunt impacts to the front of the head model. For the purpose of developing a program to measure the length of the failed elements, validated patterns of failure were not a requirement; therefore, results in this report may be outside the bounds of blunt impact experimental results.

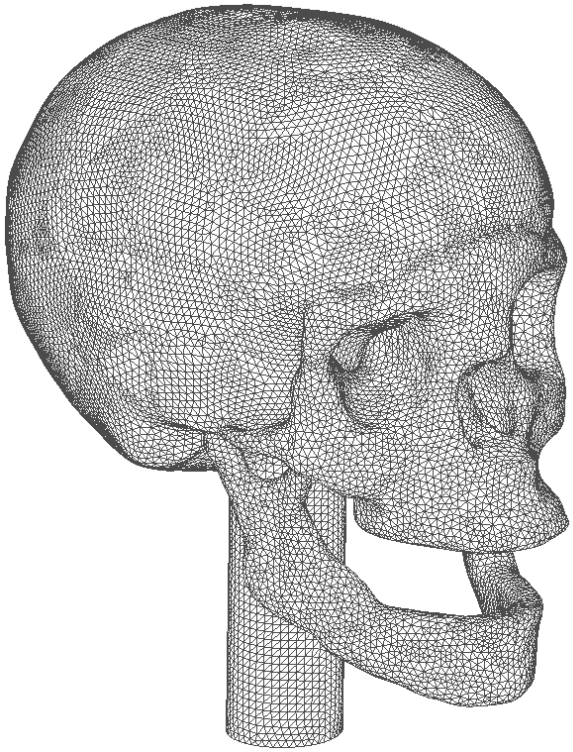

Figure 1. FE skull model.

The skull mesh described in the present study was defined by a single part with one material model—though for postprocessing, with respect to the fracture analysis, the elements were separated into one of three regions. Briefly, a skull consists of three distinct layers through the thickness: the inner and outer tables (composed of cortical bone) and a more porous middle layer called the diploë. For this study, the elements that made up the outer and inner surfaces of the skull were manually determined and noted to be part of the outer and inner tables, respectively, and the rest of the elements in the skull mesh were assumed to be part of the middle layer. The LS-DYNA element numbers for the respective regions were listed, one per

line, in separate text files as an optional input for the AFTM program. These files will be described in more detail in Section 2.3.2.

The native LS-DYNA failure mode flags an element once it has met a certain criterion, such as excessive stress or strain, and removes the element from the simulation going forward. The present analyses relied on a user-defined LS-DYNA material model that zeros the element stiffness upon exceeding a predefined threshold such that the element loses its ability to carry load but is not removed from the simulation (Alexander and Weerasooriya 2020). Additionally, each failed element is assigned a flag to a history variable that indicates whether the element failed in tension, compression, or shear. For all simulations, LS-DYNA outputs the coordinate and status (e.g., deleted or flagged) of every element through the duration of the simulation. For the purposes of measuring fracture length, simply reading the failure status of an element at the last state was sufficient since elements do not "unfail" and the last state provides the final failure state. The Python "lsreader" package (Ansys) provides an application programming interface (API) that can read native LS-DYNA "d3plot" simulation output files that contain all pertinent element information, including element location, failure status, and a connected element list for each element. The lsreader API is also available in C and C++ versions, which are beyond the scope of this report.

The skull was meshed with tetrahedral elements, which are far less labor-intensive to mesh than hexahedral elements for irregularly shaped objects. From a standpoint of fracture-pattern analysis, tetrahedral meshes are more challenging to process than hexahedral meshes because the elements are not regularly aligned (e.g., planes and columns). A second, and perhaps more formidable challenge, was identifying and reducing fracture traces that were multiple-element-wide into "discrete" lines. An example of a heavily convoluted skull fracture with numerous wide fracture traces is shown in Figure 2. Processing elements that were not on the fracture margins (transition from failed to intact elements) and were surrounded entirely by failed elements was nontrivial since those elements have no information pertaining to the directionality of the fracture trace. Both these challenges were overcome by first mapping the skull elements into a 3D box of voxels such that the size of each voxel approximated the size of a typical tetrahedral mesh element and then applying a "skeletonize" routine to reduce fracture traces to single-element-wide traces (Figure 3).

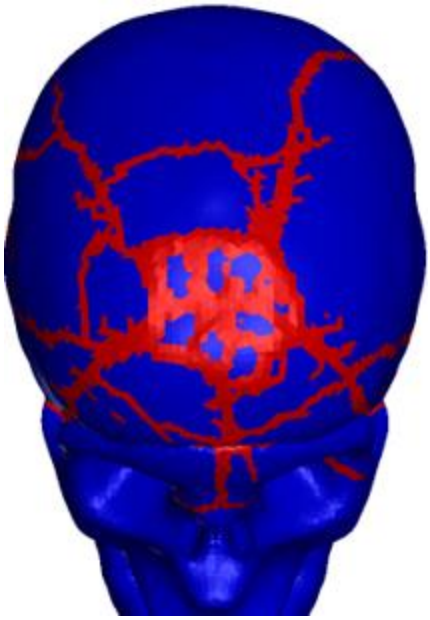

Figure 2. Comminuted skull fracture.

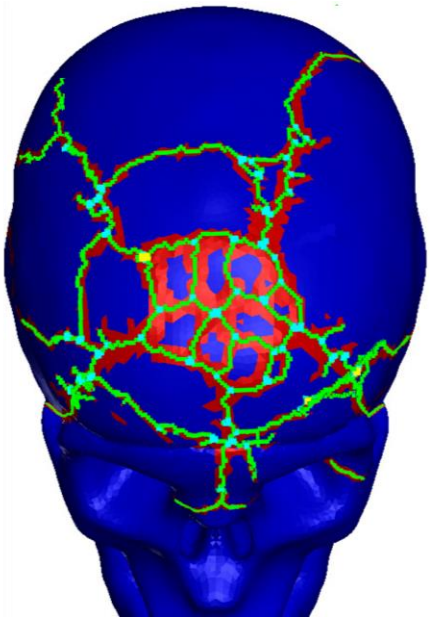

Figure 3. "Skeletonized" fracture pattern.

Once the fracture traces were reduced to single-wide traces, they could be traversed to determine the length of each fracture. Several simple rules were implemented to determine how failed elements were handled. First, if the current failed element had two failed (2-failed) element neighbors, the element was in the middle of a fracture trace and traversing continued. Second, if the current element only had a single failed (1-failed) element neighbor, then the element was the terminating element of

a fracture trace and traversing stopped. Third, if the current failed element had more than two failed element neighbors, then it was part of a junction or intersection of multiple fracture traces. Junction elements were treated similarly to terminating elements with traversing also terminating at junctions.

To prevent fractured elements from being traversed and measured multiple times, each failed element was assigned a counter that was initialized to its number of adjacent failed neighbor elements. Each time an element was traversed to or from, the counter was decremented. After its counter reached zero, an element could no longer be traversed. For 2-failed neighbor elements, the counter was decremented once when the element was traversed to from the incoming neighbor and decremented a second time when it traversed to the outgoing neighbor, which resulted in a zeroed counter and that element would not be traversed again. The counters for terminating failed elements (1-failed neighbor) were zeroed at the first traversing encounter. The junction elements were decremented once when traversed for each neighboring fracture trace.

After all the elements have reached a counter state of zero, the AFTM program prints out the total length of the summed fractures and generates an interactive 3D plot of the fracture(s) (Figure 4) with each fracture trace length labeled (Figure 5).

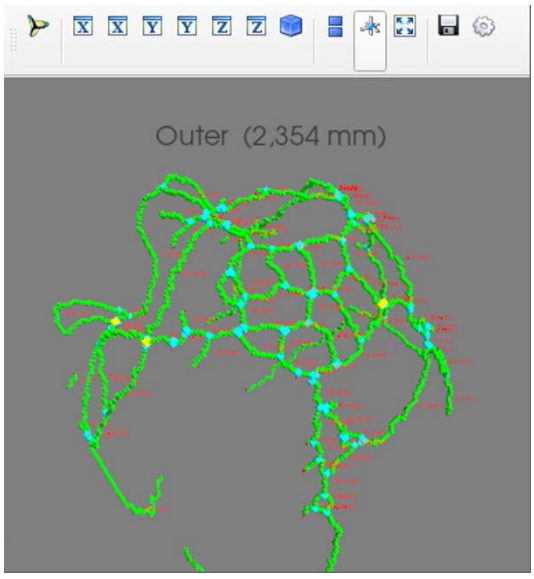

Figure 4. Interactive 3D plot of fracture traces.

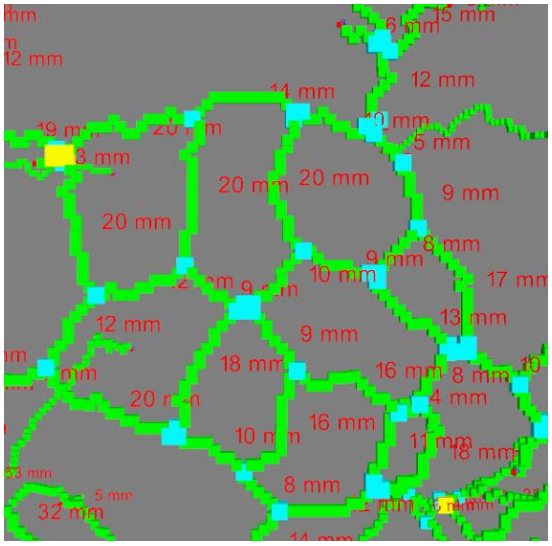

Figure 5. Zoomed-in view of the interactive 3D plot of fracture traces.

The relative size of the individual voxels in the 3D voxel space compared with the FE mesh element sizes can have a significant and unpredictable effect on the quality of the fracture trace analysis. For example, overly small voxels compared with the simulation's FE element size can result in split traces, frequently forming small loops, that artificially inflate the fracture lengths (Figure 6). Moreover, the same voxel ratio parameter may cause split traces in one analysis and none in another. By default, the AFTM program analyzes a simulation multiple times over a range of voxel ratio values and selects the value that generates the least number of fracture trace segments with the longest total fracture trace length. The premise is the analysis with the least number of fracture trace segments is least likely to have split traces. As a best practice, it is recommended that the analyst visually review the model to confirm the fracture analysis is plausible and does not contain anomalies such as split traces.

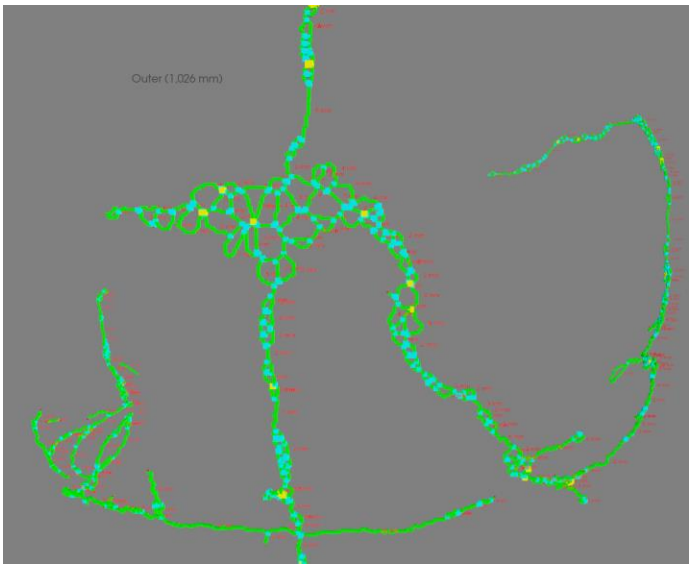

Figure 6. Split, looping fracture traces generated from undersized voxels caused by setting the voxel ratio parameter too high.

2.2 Program Algorithm

An overview of the program's algorithm is as follows:

1) Define program parameters (command line and predefined).
2) Read last state of simulation (d3plot).
3) Build universe of elements (UE) subset defined in "options" (default: whole part).
4) Link UE to their fracture outcome and connected nodes and isolate elements of interest (EI) (e.g., fractured elements).
5) Generate list of nodes that belong to EI and list of node and element IDs.
6) Generate list of CONNECTED EI by matching SHARED nodes.
7) Generate list of consolidated EI centroid coordinates, estimated radius, fracture outcome, and element IDs.
8) For each voxel ratio parameter in the voxel ratio list, do the following:
    a. Generate low- (1×) and high-resolution (2×) voxel cubes.
    b. Populate 1× voxel cubes with fractured elements and, if necessary, include populating all voxel cubes in-between fractured elements.
    c. If conducting validation analysis, overwrite voxel cubes with analytic data (e.g., circles and lines).
    d. Skeletonize 1× voxel cubes to contract wide fractures to single-element-wide.
    e. Map skeletonized 1× voxel cubes into 2× voxel cubes and skeletonize 2× voxel cubes.

f. Generate list of fractured 2× voxel cube metrics (including voxel IDs, centroid, fractured neighbor list, and untraversed counts).
g. Recursive analysis:
    i. While list of fractured 2× voxel cubes have untraversed neighbors, do the following:
        1. Sort list by untraversed neighbor count (smallest-to-largest).
        2. If available, choose voxel cubes with one neighbor (indicating end of chain).
        3. Otherwise, choose voxel with the most neighbors (if > 2, this indicates a junction).
        4. For each chosen voxel cube, do the following:
            a. Start a new fracture trace.
            b. Build fracture trace by recursively traversing the next untraversed neighbor until the chain terminates at a junction or end of chain.
h. Calculate total fracture length associated with voxel ratio parameter.
i. If the last voxel ratio parameter is in the voxel ratio parameter list, do the following:
    i. Select voxel ratio parameter that produced the least number of fracture segments and the longest total length of fracture traces.
    ii. If the selected voxel ratio parameter analysis is not the last analysis, do the following:
        1. Redefine the voxel ratio list to the selected voxel ratio parameter and repeat analysis.

9) If not in batch mode, do the following:
    a. Generate interactive fracture trace plot.

2.3 Program Setup and Operation

The AFTM program was written in Python version (v) 3.11 and has been successfully tested with v3.9 but fails to run properly under v3.6. Python version compatibility is limited by the availability of compatible package versions. For instance, at the time of this writing, the lsreader package only supports Python v3.6–3.11 (under Microsoft Windows), which was obtained directly from Ansys as a library package. In the future, Ansys may provide support for pip or conda installation. Compatibility may also be limited by other package dependencies, which were not exhaustively tested.

2.3.1 Python Setup

The program is dependent on the os, sys, operator, math, time, argparse, bisect, numpy, pandas, matplotlib, mayavi, scikit-image, and lsreader packages. Most of these packages should already be installed with Python, with exception of lsreader, which may have to be manually installed, and mayavi. Users can check which packages are already installed using the "pip list" or "conda list" commands and omit those packages from the installation command line. A typical installation command to install any missing packages using the pip installer would be as follows:

```
pip install mayavi scikit-image
```

If the user does not have system administrative privileges, the above command may need to be postpended with "--user".

If running under Anaconda, the command is as follows:

```
conda install mayavi scikit-image
```

Anaconda does not support --user installation but instead uses environments. Refer to Anaconda documentation for instructions on how to use environments.

For Linux, the lsreader package is installed by adding the path of the lsreader.so library file to the "PYTHONPATH" system environment variable. This could be accomplished by adding the following to the user's .bashrc file:

```
export PYTHONPATH=$PYTHONPATH:~/.local/lib
```

then copying the lsreader.so file to the user's ~/.local/lib subdirectory.

For Microsoft Windows, the path to the Python version-appropriate subdirectory of the lsreaderdist is added to the PYTHONPATH system environment variable. For example, the directory could be the following for Python v3.11 implementation:

```
%HOMEPATH%\lsreader_174\pythondist\lib\windows\cp311
```

Refer to Microsoft Windows documentation for instructions on creating an environmental variable. If the user is running Python inside the Spyder Integrated Development Environment, the PYTHONPATH path can be specified by the "PYTHONPATH manager" under the "Tools" tab.

If a user does not need to read LS-DYNA d3plot files, the lsreader package does not need to be installed. However, they will have to modify, comment, or delete code that references lsreader commands.

2.3.2 Model Definition Setup

The AFTM program was designed to analyze element subsets of simulations, which may be a subset of a part, a subset of combined parts, or multiple subsets of a part. If subsets of parts are used, the list of LS-DYNA element IDs must be stored in a text file, one element ID per line, with its file name template matching "measElem_id.txt", where "id" is a single character that will be included as an option in the command line for conducting the analysis. The AFTM program reads a file named "measDict.txt" to associate labels with ids defined in the command line. For instance, a file containing the following (lines beginning with "#" are ignored):

```
#format: id,description
o,Outer_Table/
m,Middle
i,Inner_Table
```

The above example specifies that the file named "measElem_o.txt" contains the "Outer Table" element IDs, "measElem_m.txt" contains the "Middle" element IDs, and "measElem_i.txt" contains the element IDs associated with "i".

2.3.3 Running and Command Line Arguments

The program runs from the command line with the following format:

```
python measfrac.py    [-h] [-p <partListStr>][-r <d3plotDirectory>]
[-d <fullPath>] [-e <abc..xyz012..9>] [-hv <hv#>]
```

See Table 1 for a list of options.

Table 1. Command line arguments.

Command	Description
-h, --help	Show this help message and exit.
-p, --parts <partListStr>	Part number(s) (concatenate multiple with "+"NO spaces, default = 0, whole model).
-r, --rootPath <d3plotDirectory>	d3plot directory path (default = <current directory>).
-d, --d3plot <d3plotName>	d3plot file name (default = d3plot).
-e, --elemListChar <abc..xyz012..9>	Element list file infix character(s), no spaces, for file(s) "measElem_ id.txt".
-l, --log <logfile>	Output to log file (default = none).
-v, --voxelRatioList <voxelRatioList>	Voxel ratio, larger shrinks voxels size. Format #.## or #.##-#.## (e.g., 1.10 or 1.00–1.20). Voxel ratio ranges are stepped through at 0.01 increments (default = 1.00–1.20).
-b, --batch	Batch mode, suppress 3D interactive window (default: no batch).
-S, --Save3dPlot	Save 3D plot. Suitable for running analyses in background (default: no saved plot).
-s, --saveInt3dPlot	Save all interim vox ratio 3D plots. Suitable for running analyses in background (default: no save plots).
-P, --ProgressBars	Display progress output. Not suitable for running analyses in background (default: no progress bars).
-V, --verbose	Verbose output (default: false).

For the present study, the skull model had three layers: "o" or outer, "m" or middle, and "i" or inner. These parameters, along with their descriptions, need to be included in a file named "measDict.txt" in the same directory as the d3plot files. In this case, the element list file infix characters would be any combination of the characters "i," "m," and "o" (without spaces or quotes). Similarly, multiple part numbers can be included in the analysis by concatenating multiple part numbers together separated by "+" characters. For example, the following command would measure the fracture metric of history variable 1 for middle and outer tables of parts 35 and 36 of the d3plot file in the analyst's home directory (Linux):

```
python measfrac.py --parts 35+36 -r /home/user -d d3plot -e im -hv hv1
```

The analyst may need to prepend the Python program (measfrac.py) with the path of the program's directory. By adding "-l output.log -b", the program suppresses displaying the interactive 3D plot of the fractures and appends the fracture analysis summary to a log file. This mode of program operation is suitable for running unattended multiple model analyses.

3. Results and Discussion

Several simulations with varying levels of skull fracture were analyzed with the AFTM program. The subset of traces that were manually assessed for accuracy was accurate with respect to the size of the voxel grid. The measurements of simulation fracture trace lengths were approximated since the length measures were calculated from one voxel centroid to another, forming a path that does not perfectly track the path of the overall fracture. Furthermore, the fractured mesh elements were mapped in the closest voxel, which is also a potential source for error. A separate analytical assessment was conducted to quantitively evaluate this limitation.

The AFTM program was assessed for accuracy by generating a sample 3D environment populated with semicircular arcs (n = 5 at 157 mm each) and straight lines (n = 3 at 100 mm each) in varying orientations of pseudo-fractures that were defined using analytic equations with known lengths (Figure 7). The total analytic length of the fractures was 1,085 mm and the AFTM program calculated a total length of 1,139 mm, which was 4.9% greater than the analytic length. The additional lengths were primarily caused by the zigzagging paths of out-of-plane fracture traces. This model was defined with relatively coarse 1-mm voxels. A follow-up analysis was conducted with 0.5-mm voxels with the AFTM program calculating a total fracture length of 1,147 mm, which was 5.7% greater than the analytic length. Two of the three straight 100-mm traces were in plane for at least one axis and both had 100% accuracy. The third trace was out of plane for all three axes and had a slight corkscrew path, which added 8 mm to its calculated length. The circular paths tended to overestimate the analytic length by 5%–10%.

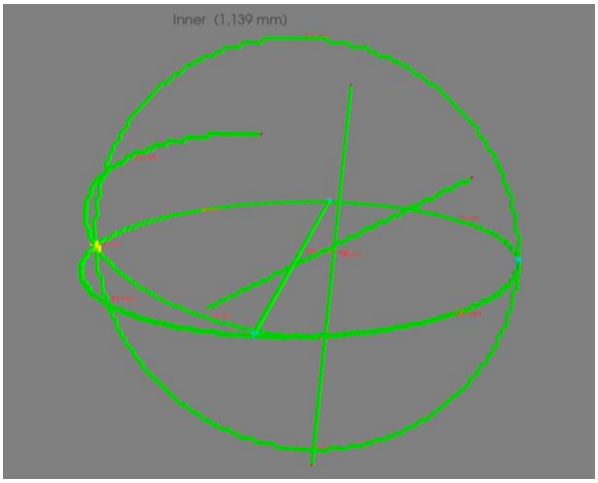

Figure 7. Validation pseudo-fracture model.

Though not ideal, a 5% overestimate of fracture length is probably not significant considering the width of some fracture traces and amount of uncertainty introduced

during the process of reducing them to a single-voxel-width trace (Figure 3). It is possible, and probably good practice, for the analyst to visually inspect the model for traces that appear to take arduous paths, and based on those findings, scale down the length estimate accordingly.

Examination of the 3D interactive plot revealed instances where traces intermittently deviated from in plane where the analytic equation defined an in-plane trace path (Figure 8). These artifacts were introduced by the 3D skeletonized routine. It would be relatively straightforward to add an interim processing routine to remove the linear deviation artifacts from the validation case but, when analyzing real-world model data, distinguishing artifacts from the real fracture path is not straightforward.

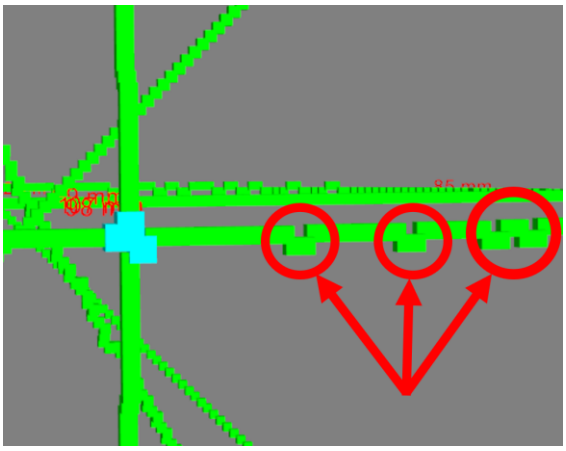

Figure 8. Out-of-plane deviation artifacts.

4. Conclusions

The AFTM program provides an automated tool for quantitative assessment of trace lengths of complex fracture patterns. It incorporates a method that reduces arbitrarily wide fracture traces to singular-width traces that provide linear paths for traversing and measuring. The fracture traces are mapped into a cubic voxel volume with a coordinate defined at the center of the voxel. The center-voxel coordinate restriction introduces error when the actual path does not directly pass through voxel centers. This error may be zero for a fracture that passes "in-line" with a column of voxels. While the validation study showed the worst-case error amounted to approximately 10% for a single fracture path that was obtuse to all planes, an overall composite error of 5% was observed.

The program is currently coded to read input data from LS-DYNA d3plot files for its underlying data but can be easily modified to handle any other sources of data, provided the data can be mapped into a 3D matrix. The automated fracture measuring analysis starts in the voxel cube volume and is not dependent on the

source or mechanism of how the volume was populated. The only stipulation is that each voxel cube in the volume has a quantitative flag set to identify whether it should be included in the measurement. An analyst seeking to modify the program for another application could refer to the validation sections of the program where analytic equations were utilized to define a pseudo-fracture pattern.

5. References

Alexander SL, Weerasooriya T. Microstructure-based model of the deformation and failure response of the human skull under uniaxial compression, DEVCOM Army Research Laboratory (US); 2020. Report No.: ARL-TR-8961.

Bullock M et al. Surgical management of depressed cranial fractures. Neurosurgery. 2006;58(suppl 3):S2–56.

Hofman P, Nelemans P, Kemerink GJ, Wilmink JT. Value of radiological diagnosis of skull fracture in the management of mild head injury: meta-analysis. J Neurol Neurosurg Psychiatry. 2000;68(4):416–422.

Loftis KL, Matheis E, Rafaels KA. Assessment of head injuries: blunt versus penetrating. Personal Armour System Symposium (PASS 2020): Personal Armour, Postponed from 2020; 2021 Oct 11–15; Copenhagen, Denmark. Royal Military Academy (Belgium); c2020. p. 253–260.

Matheis E et al. Under-reporting of focal brain injuries in emergency department records versus hospital inpatient records within the National Trauma Data Bank. DEVCOM Army Research Laboratory (US); 2021. Report No.: ARL-TR-9228.

Muñoz-Sánchez M et al. Skull fracture, with or without clinical signs, in mTBI is an independent risk marker for neurosurgically relevant intracranial lesion: a cohort study. Brain Injury. 2009;23(1):39–44.

Ono K, Wada K, Takahara T, Shirotani T. Indications for computed tomography in patients with mild head injury. Neurol Med Chir. 2007;47(7):291–298.

Tseng W et al. The association between skull bone fractures and outcomes in patients with severe traumatic brain injury. J Trauma Acute Care Surg. 2011;71(6):1611–1614.

Wiederholt W et al. Short-term outcomes of skull fracture: a population-based study of survival and neurologic complications. Neurology. 1989;39(1):96–102.

Zhang TG, Satapathy SS, Dagro AM, McKee PJ. Numerical study of head/helmet interaction due to blast loading. Proceedings of the ASME 2013 International Mechanical Engineering Congress and Exposition; 2013 Nov 15–21; San Diego, CA. ASME; c2013. https://doi.org/10.1115/IMECE2013-63015

Appendix. Python Code

NOTE: Below code listing contains line wrapping that may have to be manually removed to maintain correct Python syntax.

```
###############################################################################
# Requirements: Tested for Python 3.9 and 3.11, testing on 3.6 failed
# Usage: python measfrac.py [o][m][i] <full_path_to_d3plot>
#        where,
#                  o : outer table
#                  m : diploe (middle)
#                  i : inner table
#
# Requred Pacakges:
# $ pip install --upgrade <pkg> --user
#   <pkg> = mayavi
#           scikit-image

#   lsreader is a manual install. lsreader.so is installed in:
#   /p/app/projects/bioshare/scripts/python/lsreaderdist_0174_06Mar2024/lib
#   add following to your login script (e.g., ~/.bashrc):
#                                                                       export
PYTHONPATH=/p/app/projects/bioshare/scripts/python/lsreaderdist_0174_06Mar2024/lib
#   lsreader to be an available module.
#
#   Tested on Narwhal with "module load cray-python"
#
#   Programmer: Alan Goertz, SURVICE Engineering Company
#               (contractor to DEVCOM Army Research Laboratory)
#
###############################################################################
import os                           # OS system environment variables
import sys                          # reading command line params and error handling
from copy import deepcopy           # forcing data copy (versus pointer to same)
from tqdm import tqdm               # a fast, extensible progress bar
from lsreader import *              # reading LS-DYNA d3plot files
from operator import itemgetter  # for sorting
import numpy as np
from math import pi, sin, cos, sqrt
import time                         # analysis time tracking
import argparse                     # reading command line options
from argparse import RawDescriptionHelpFormatter
import pandas as pd                 # data handling
import matplotlib.pyplot as plt  # plotting
from mpl_toolkits.mplot3d import Axes3D
from bisect import bisect_left
import skimage
# mayavi not loaded in batch mode due to X error in gnu screen
# import mayavi.mlab as mlab       # 3D interactive Plotting

#------------------------------------------------------------------------------
class voxelUniverse:
#------------------------------------------------------------------------------
    # 3D-array of voxels encompassing the volume of the model with supporting
    # functions

    def                     __init__(self,xdim=500,ydim=500,zdim=500,llf=[-50,-50,-
50],urb=[50,50,50]):
        self.xdim = xdim
        self.ydim = ydim
        self.zdim = zdim
        self.voxels = np.zeros((self.xdim,self.ydim,self.zdim),dtype='int8')
        self.closestElem                                                    =
np.zeros((self.xdim,self.ydim,self.zdim),dtype='uint64')
        self.closestDist                                                    =
np.zeros((self.xdim,self.ydim,self.zdim),dtype='float64')
        self.Xmin = llf[0]
        self.Ymin = llf[1]
        self.Zmin = llf[2]
        self.Xmax = urb[0]
        self.Ymax = urb[1]
        self.Zmax = urb[2]
```

```
        # List of X, Y, Z indeces of fractured voxels.
        self.xx = []
        self.yy = []
        self.zz = []
        self.es = []

        # the length, width, and height of the voxelUniverse in world units.
        self.Xspan = self.Xmax - self.Xmin
        self.Yspan = self.Ymax - self.Ymin
        self.Zspan = self.Zmax - self.Zmin

    def xyz2IDX(self,X=0,Y=0,Z=0):
        # Convert model x, y, and z 'world coordinates' to voxelUniverse array
        # x, y, and z indeces.

        # Scale the X,Y,Z coordinates and round to nearest whole number
        xidx = int(round(self.xdim*(X-self.Xmin)/self.Xspan))
        yidx = int(round(self.ydim*(Y-self.Ymin)/self.Yspan))
        zidx = int(round(self.zdim*(Z-self.Zmin)/self.Zspan))

        # Set indeces to -1 if they fall outside the voxelUniverse
        if (xidx < 0) | (xidx > self.xdim-1):
            xidx = -1
        if (yidx < 0) | (yidx > self.ydim-1):
            yidx = -1
        if (zidx < 0) | (zidx > self.zdim-1):
            zidx = -1
        return xidx, yidx, zidx

    def IDX2xyz(self,xidx,yidx,zidx):
        # Convert voxelUniverse array x, y, and z indeces to
        # model x, y, and z 'world coordinates'.

        X = self.Xmin + (self.Xspan*xidx)/self.xdim
        Y = self.Ymin + (self.Yspan*yidx)/self.ydim
        Z = self.Zmin + (self.Zspan*zidx)/self.zdim
        return X, Y, Z

    def joinElems(self,df_fx,elem1,elem2,val=0,useSize=False):
        # set voxelUniverse voxels at and between two mesh elements (defined
        # by element ID)

        e1 = df_fx[df_fx["elID"] == elem1]
        e2 = df_fx[df_fx["elID"] == elem2]
        # Confirm e1 and e2 are populated, otherwise error.
        if (len(e1) < 1) | (len(e2) < 1):
            print(f"{elem1} ={len(e1)}    {elem2} ={len(e2)}   less than 1 error")
            return False
        X1 = e1.iloc[0].x
        Y1 = e1.iloc[0].y
        Z1 = e1.iloc[0].z
        X2 = e2.iloc[0].x
        Y2 = e2.iloc[0].y
        Z2 = e2.iloc[0].z
        if useSize:
            size = (e1.iloc[0].elRad,e2.iloc[0].elRad)/2
        else:
            size = 0

        # Identify indeces of both elements and define a range of indeces
        # between the element indeces to ensure any voxels between the
        # element voxels are included
        x1idx,y1idx,z1idx = self.xyz2IDX(X1,Y1,Z1)
        x2idx,y2idx,z2idx = self.xyz2IDX(X2,Y2,Z2)
        deltad   =   int(round(1.5*max(abs(x1idx-x2idx),abs(y1idx-y2idx),abs(z1idx-
z2idx))))
        xrange = np.linspace(X1,X2,deltad)
        yrange = np.linspace(Y1,Y2,deltad)
        zrange = np.linspace(Z1,Z2,deltad)
```

```
        # loop through the two element voxels, set them and any voxels in
        # between them (if not already set)
        for ii in range(len(xrange)):
            xidx,yidx,zidx = self.xyz2IDX(xrange[ii],yrange[ii],zrange[ii])
            if (xidx >= 0) & (yidx >= 0) & (zidx >= 0):

                # Don't set a voxel to fractured if already set
                if self.voxels[xidx,yidx,zidx] != val:
                    # Update closest element ID and distance
                    x,y,z = self.IDX2xyz(xidx,yidx,zidx)
                    curDist = np.sqrt((x-X1)**2 + (y-Y1)**2 +(z-Z1)**2)
                    if  (self.closestElem[xidx,yidx,zidx]  ==  0)  |  (curDist  <
self.closestDist[xidx,yidx,zidx]):
                        self.closestDist[xidx,yidx,zidx] = curDist
                        self.closestElem[xidx,yidx,zidx] = elem1
                    curDist = np.sqrt((x-X2)**2 + (y-Y2)**2 +(z-Z2)**2)
                    if  (self.closestElem[xidx,yidx,zidx]  ==  0)  |  (curDist  <
self.closestDist[xidx,yidx,zidx]):
                        self.closestDist[xidx,yidx,zidx] = curDist
                        self.closestElem[xidx,yidx,zidx] = elem2

                    self.voxels[xidx,yidx,zidx] = val
                    self.xx.append(xidx)
                    self.yy.append(yidx)
                    self.zz.append(zidx)
                    self.es.append(size)
        return

    def setvox(self,val=0,X=0,Y=0,Z=0,elid=0,size=0):
        # set voxelUniverse voxel corresponding to X, Y, Z world coordinates

        xidx,yidx,zidx = self.xyz2IDX(X,Y,Z)
        if (xidx < 0) | (yidx < 0) | (zidx <0):
            print(f"Voxel X={X} y={Y} Z={Z} IDX out-of-bounds, skipping")
            return False

        if self.voxels[xidx,yidx,zidx] != val:

            # Update closest element ID and distance
            x,y,z = self.IDX2xyz(xidx,yidx,zidx)
            curDist = np.sqrt((x-X)**2 + (y-Y)**2 +(z-Z)**2)
            if    (self.closestElem[xidx,yidx,zidx]    ==    0)    |    (curDist    <
self.closestDist[xidx,yidx,zidx]):
                self.closestDist[xidx,yidx,zidx] = curDist
                self.closestElem[xidx,yidx,zidx] = elid

            self.voxels[xidx,yidx,zidx] = val
            self.xx.append(xidx)
            self.yy.append(yidx)
            self.zz.append(zidx)
            self.es.append(size)
            if size > 0:
                for ii in [X-size,X,X+size]:
                    for jj in [Y-size,Y,Y+size]:
                        for kk in [Z-size,Z,Z+size]:
                            xxidx             =             int(round(self.xdim*(ii-
self.Xmin)/self.Xspan))
                            yyidx             =             int(round(self.ydim*(jj-
self.Ymin)/self.Yspan))
                            zzidx             =             int(round(self.zdim*(kk-
self.Zmin)/self.Zspan))
                            if (xxidx >= 0) & (xxidx < self.xdim) & (yyidx >= 0) \
                                & (yyidx < self.ydim) \
                                & (zzidx >= 0) & (zzidx < self.zdim) \
                                & ((ii != xidx) | (jj != yidx) | (kk != zidx)):
                                if self.voxels[xxidx,yyidx,zzidx] != val:

                                    # Update closest element ID and distance
                                    x,y,z = self.IDX2xyz(xxidx,yyidx,zzidx)
                                    curDist  =  np.sqrt((x-X)**2  +  (y-Y)**2  +(z-
Z)**2)
```

```
                                    if (self.closestElem[xxidx,yyidx,zzidx] == 0) |
(curDist < self.closestDist[xxidx,yyidx,zzidx]):
                                        self.closestDist[xxidx,yyidx,zzidx]       =
curDist
                                        self.closestElem[xxidx,yyidx,zzidx] =elid

                                    self.voxels[xxidx,yyidx,zzidx] = val
                                    self.xx.append(xxidx)
                                    self.yy.append(yyidx)
                                    self.zz.append(zzidx)
                                    self.es.append(size)

        return True

    def getvox(self,X=0,Y=0,Z=0):
        # Report whether voxel at world coordinates X, Y, and Z is set.

        xidx = int(round(self.xdim*(X-self.Xmin)/self.Xspan))
        yidx = int(round(self.ydim*(Y-self.Ymin)/self.Yspan))
        zidx = int(round(self.zdim*(Z-self.Zmin)/self.Zspan))
        if xidx < 0 | xidx > xdim-1:
            return -128
        if yidx < 0 | yidx > ydim-1:
            return -128
        if zidx < 0 | zidx > zdim-1:
            return -128
        return self.voxels[xidx,yidx,zidx]

    def plotvox(self, val=1, sf=1, isRebuild=False):

        if isRebuild:
            # Plot w/data rebuild
            xx = []
            yy = []
            zz = []
            progBarTtl = "Plot Vox Setup"
            for                                     xi                                      in
tqdm(range(self.xdim),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):
                for yi in range(self.ydim):
                    for zi in range(self.zdim):
                        if self.voxels[xi,yi,zi] == val:
                            xx.append(xi)
                            yy.append(yi)
                            zz.append(zi)

            fig = plt.figure(1)
            ax = fig.add_subplot(111, projection='3d')

            mlab.points3d(xx, yy, zz, scale_factor=sf, mode='cube')  # works, all
white cubes

            ax.set_xlabel('X Label')
            ax.set_ylabel('Y Label')
            ax.set_zlabel('Z Label')

            mlab.show()
        else:
            mlab.points3d(self.xx, self.yy, self.zz, scale_factor=sf, mode='cube')
# works, all white cubes
            # Alternate plotting modes
            #      mlab.points3d(self.xx,      self.yy,      self.zz,      self.es,
scale_mode="scalar", scale_factor=1, mode='cube')
            #   mlab.points3d(self.xx,   self.yy,   self.zz,   scale_mode="scalar",
scale_factor=15, mode='cube')  # works, all white cubes
            # mlab.points3d(self.xx, self.yy, self.zz, self.es, mode='cube')   #
works, large colored cubes
            mlab.show()

        return

    # __enter__ and __exit__ for using 'with' statement
```

```
    def __enter__(self,x,y,z):
        return x,y,z

    def __exit__(self,x,y,z):
        return x,y,z

#-------------------------------------------------------------------------------
class fxElem:
#-------------------------------------------------------------------------------
    def __init__(self,idx,id,elID,hv,vol=0,elRad=0,x=0,y=0,z=0,ang=0,dang=0,
                 rad=0,rad2=0,radAdj=-1,nx=0,ny=0,nz=0,mx=0,my=0,mz=0,mang=0,
                 X=0,Y=0,Z=0):
        self.idx = idx
        self.id = id
        self.elID = elID
        self.hv = hv               # LS-DYNA history variable
        self.vol = vol             # vol (mm^3)
        self.elRad = elRad         # element radius (m)
        self.x = x
        self.y = y
        self.z = z
        self.ang = ang
        self.dang = dang
        self.rad = rad
        self.rad2 = rad2            # radius calculated from average node distance
from centroid
        self.radAdj = radAdj         # radius after 'flattening', used by distal
neighbor elements for their new radius.
        self.nx = nx               # normal Unit Vector x, y, z
        self.ny = ny               # normal Unit Vector x, y, z
        self.nz = nz               # normal Unit Vector x, y, z
        self.mx = mx               # normal Unit Vector mapped to radial plane
        self.my = my               # normal Unit Vector mapped to radial plane
        self.mz = mz               # normal Unit Vector mapped to radial plane
        self.mang = mang           # angle between mapped and 'up'
        self.X = X                 # transformed x, y, z
        self.Y = Y                 # transformed x, y, z
        self.Z = Z                 # transformed x, y, z

    def __enter__(self,idx,id,elID,hv,vol=0,elRad=0,x=0,y=0,z=0,ang=0,dang=0,
                  rad=0,rad2=0,radAdj=-1,nx=0,ny=0,nz=0,mx=0,my=0,mz=0,mang=0,
                  X=0,Y=0,Z=0):
        return idx,id,elID,hv,vol,elRad,x,y,z,ang,dang,rad,rad2,radAdj,\
               nx,ny,nz,mx,my,mz,mang,X,Y,Z

    def __exit__(self,idx,id,elID,hv,vol=0,elRad=0,x=0,y=0,z=0,ang=0,dang=0,
                 rad=0,rad2=0,radAdj=-1,nx=0,ny=0,nz=0,mx=0,my=0,mz=0,mang=0,
                 X=0,Y=0,Z=0):
        return idx,id,elID,hv,vol,elRad,x,y,z,ang,dang,rad,rad2,radAdj,\
               nx,ny,nz,mx,my,mz,mang,X,Y,Z

#-------------------------------------------------------------------------------
class skelel2:
#-------------------------------------------------------------------------------
    # structure for tracking fractured voxels and neighbor counts/lists
    # (availCnt decremented as fracture lines are parsed)
    def
__init__(self,xidx=0,yidx=0,zidx=0,voxID=0,x=0,y=0,z=0,neighborCnt=0,neighborList=
[]):
        self.xidx = xidx               # Voxel X-index in volume
        self.yidx = yidx               # Voxel Y-index in volume
        self.zidx = zidx               # Voxel Z-index in volume
        self.voxID = voxID             # arbitrary voxle ID
        self.x = x                     # Voxel X centroid coordinate
        self.y = y                     # Voxel Y centroid coordinate
        self.z = z                     # Voxel Z centroid coordinate
        self.neighborCnt = neighborCnt    # number of fractured voxel neighbors
        self.neighborList = neighborList          # element ID list of connected
elements
        self.availCnt = neighborCnt       # how many unexplored neighbors
        self.usedCnt = 0               # how many explorer neighbors
```

```
#------------------------------------------------------------------------------

#------------------------------------------------------------------------------
#%% Global Constants (variables)
#------------------------------------------------------------------------------
# target either 'cpu', 'parallel' or 'cuda'
TARGET='parallel'
STRAIN_OVERFLOW = np.float32(3)
STRESS_OVERFLOW = np.float32(1e12)
SEP_CHR = ','
TQDM_COLS = 90
UNIT_CONV = 1000   # m to mm
IS_VALIDATE = False    # Analyze analytically generated spherical/linear patterns
SKIP_LT_2_mm = True      # Omit fracture lengths less than 2 mm

#%% Funtions/Procedures
#------------------------------------------------------------------------------
def listPartsDesc(p,dr,numParts,part_ids):
#------------------------------------------------------------------------------
    print("Parts:")
    p.ipart_user = -1
    for i in range(numParts):
        p.ipart = i
        pName = dr.get_data(DataType.D3P_PART_NAME, p)

    print(f'{part_ids[i]:8d}  {pName}')

    print(' ')

#------------------------------------------------------------------------------
def
calcCentroid(dr,p,partNumElements,histVars,element_ids,element_conn,node_ids,node_
pos,partType,isVerbose=False):
#------------------------------------------------------------------------------
# calculate the x,y,z coordinates of each element's centroid and estimate an
# element radius (presuming volume approximates a sphere) for elemental scale

    units_m = 'm^3'
    units_mm = 'mm^3'
    if partType == 'SOLID':
        regionType = 'volumes'
        if element_conn[0].Node(3) == element_conn[0].Node(4):
            isTet = True
            elType = 'Tetrahedra'
        else:
            isTet = False
            elType = 'Hex-/Pent-ahedra'
    else:
        isTet = False
        elType = 'Quad'
        # lsreader shellThk doesn't appear to work.
        # shellThk = dr.get data(DataType.D3P_SHELL_THICKNESS,p)
        regionType = 'areas'
        shellThk = np.zeros(partNumElements)
        if shellThk[0] == 0:
            shellThk = np.ones(len(shellThk))
    if isVerbose:
        print(f'Calculating element {regionType}. Type: {elType}',flush=True)

    # fxarray:  list[fxElem] = []
    fullarray:  list[fxElem] = []


    elemVol = np.zeros(partNumElements,dtype=np.float32)
    totVol = 0
    totRad = 0
    totRad2 = 0
    idxFX = -1
    for        i        in        tqdm(range(partNumElements),desc='Calc        Vol
Progress',ncols=TQDM_COLS,disable=isQuiet):
```

```
        ec = element_conn[i]
        if partType == 'SOLID':
            # Node4 == Node5 --> Tet
            if ec.Node(3) == ec.Node(4):
                x,y,z = elemTetCentroid(ec)
                r2 = elemTetRadius(ec,x,y,z)
                vol = elemTetVolCalc_mm(ec)
                elemVol[i] = vol
                elRad = np.cbrt((3*vol)/(4*pi))/1000
                tempElem                                                    =
fxElem(idxFX,i,element_ids[i],histVars[i],vol,elRad,x,y,z,rad=0,rad2=r2)
            # Node5 != Node6 --> Hex (not Pent)
            elif ec.Node(4) != ec.Node(5):
                elemVol[i]=elemHexVolCalc_mm(ec)
            # Node5 == Node6 --> Pent
            elif ec.Node(4) != ec.Node(5):
                elemVol[i]=elemPentVolCalc_mm(ec)
            # Shouldn't reach here, use tet
            else:
                elemVol[i]=elemTetVolCalc_mm(ec)
        else: # shell
            elemVol[i] = elemQuadAreaCalc_mm(ec) * shellThk[i]

        totVol += elemVol[i]
        totRad += elRad
        totRad2 += r2

        fullarray.append(tempElem)

        if isVerbose and False:
            print(f'i={i}  elID={element_ids[i]}')
            sc = element_conn[i]
            print(f'Elem[{i}] Nodes:')
            print(f'  sCon.Nd={element_conn[i].Node(0)}')
            print(f'  sc.Nd={sc.Node(0)}')
            print(f'  NID={node_ids[element_conn[i].Node(0)]}')
            print(f'      sCon.Nd={element_conn[i].Node(0)}      sc.Nd={sc.Node(0)}
NID={node_ids[element_conn[i].Node(0)]}')
            print(f'      sCon.Nd={element_conn[i].Node(1)}      sc.Nd={sc.Node(1)}
NID={node_ids[element_conn[i].Node(1)]}')
            print(f'      sCon.Nd={element_conn[i].Node(2)}      sc.Nd={sc.Node(2)}
NID={node_ids[element_conn[i].Node(2)]}')
            print(f'      sCon.Nd={element_conn[i].Node(3)}      sc.Nd={sc.Node(3)}
NID={node_ids[element_conn[i].Node(3)]}')
            print(f'      sCon.Nd={element_conn[i].Node(4)}      sc.Nd={sc.Node(4)}
NID={node_ids[element_conn[i].Node(4)]}')
            print(f'      sCon.Nd={element_conn[i].Node(5)}      sc.Nd={sc.Node(5)}
NID={node_ids[element_conn[i].Node(5)]}')
            print(f'      sCon.Nd={element_conn[i].Node(6)}      sc.Nd={sc.Node(6)}
NID={node_ids[element_conn[i].Node(6)]}')
            print(f'      sCon.Nd={element_conn[i].Node(7)}      sc.Nd={sc.Node(7)}
NID={node_ids[element_conn[i].Node(7)]}')

    if isVerbose:
        print(f"        Volume: Total = {totVol:,.2f} {units_mm} ({totVol/1e9:.5e}
{units_m}) "
              +f"Mean    Radius    =    {totRad/partNumElements*1000:.3f}    mm    &
{totRad2/partNumElements*1000:.3f} mm")
    return fullarray   #, fxarray

#-------------------------------------------------------------------------------
def printProgressBar (iteration, total, prefix = '', suffix = '', decimals = 0,
length = 100, fill = '#'):
#-------------------------------------------------------------------------------
# Print iterations progress
    """
    Call in a loop to create terminal progress bar
    @params:
        iteration   - Required  : current iteration (Int)
        total       - Required  : total iterations (Int)
        prefix      - Optional  : prefix string (Str)
```

```
        suffix      - Optional  : suffix string (Str)
        decimals    - Optional  : positive number of decimals in percent complete
(Int)
        length      - Optional  : character length of bar (Int)
        fill        - Optional  : bar fill character (Str)
    """
    modFactor=total // 1000
    if modFactor == 0:
        modFactor = total // 100
    if modFactor == 0:
        modFactor = 2
    if iteration == total or iteration % modFactor == 0:
        percent  =  ("{0:."  +  str(decimals)  +  "f}").format(100  *  (iteration  /
float(total)))
        filledLength = int(length * iteration // total)
        bar = fill * filledLength + '-' * (length - filledLength)
        # '\033[F' cursor up on line (for multiline)
        # '\033[K' clear to end of line
        print('\r%s |%s| %s%% %s\033[K' % (prefix, bar, percent, suffix), end =
'\r')
        # Print New Line on Complete
        if iteration == total:
            print()

#-------------------------------------------------------------------------------
def elemTetCentroid(elemConn):
#-------------------------------------------------------------------------------
    # Determine the X, Y, and Z coordinates of the centroid of a tetrahedron

    N1 = node_pos[elemConn.Node(0)-1]
    N2 = node_pos[elemConn.Node(1)-1]
    N3 = node_pos[elemConn.Node(2)-1]
    N4 = node_pos[elemConn.Node(3)-1]
    X = (N1.X() + N2.X() + N3.X() + N4.X())/4
    Y = (N1.Y() + N2.Y() + N3.Y() + N4.Y())/4
    Z = (N1.Z() + N2.Z() + N3.Z() + N4.Z())/4
    return X, Y, Z

#-------------------------------------------------------------------------------
def elemTetRadius(elemConn,x,y,z):
#-------------------------------------------------------------------------------
    # Determine the X, Y, and Z coordinates of the centroid of a tetrahedron

    N1 = node_pos[elemConn.Node(0)-1]
    N2 = node_pos[elemConn.Node(1)-1]
    N3 = node_pos[elemConn.Node(2)-1]
    N4 = node_pos[elemConn.Node(3)-1]
    R1 = sqrt((N1.X()-x)**2 + (N1.Y()-y)**2 + (N1.Z()-z)**2)
    R2 = sqrt((N2.X()-x)**2 + (N2.Y()-y)**2 + (N2.Z()-z)**2)
    R3 = sqrt((N3.X()-x)**2 + (N3.Y()-y)**2 + (N3.Z()-z)**2)
    R4 = sqrt((N4.X()-x)**2 + (N4.Y()-y)**2 + (N4.Z()-z)**2)
    return (R1+R2+R3+R4)/4

#-------------------------------------------------------------------------------
def side(n1,n2):
#-------------------------------------------------------------------------------
    # Determine the distance between two nodes

    return np.sqrt((n1.X()-n2.X())**2 +(n1.Y()-n2.Y())**2 +(n1.Z()-n2.Z())**2)

#-------------------------------------------------------------------------------
def tetVol(a1,a2,a3,a4,a5,a6):
#-------------------------------------------------------------------------------
    # Determine the volume of a tetrahedron given the six edge lengths

    A1=a1**2; A2=a2**2; A3=a3**2; A4=a4**2; A5=a5**2; A6=a6**2
    V=(A1*A5*(A2+A3+A4+A6-A1-A5) + \
       A2*A6*(A1+A3+A4+A5-A2-A6) + \
       A3*A4*(A1+A2+A5+A6-A3-A4) - \
       A1*A2*A4 - A2*A3*A5 - A1*A3*A6 - A4*A5*A6) / 144
    return np.sqrt(V)
```

```
#-------------------------------------------------------------------------------
def elemTetVolCalc_mm(elemConn):
#-------------------------------------------------------------------------------
    # Determine the volume of a tetrahedron for a given element

    N1 = node_pos[elemConn.Node(0)-1]
    N2 = node_pos[elemConn.Node(1)-1]
    N3 = node_pos[elemConn.Node(2)-1]
    N4 = node_pos[elemConn.Node(3)-1]
    a1 = side(N1,N4)
    a2 = side(N1,N3)
    a3 = side(N1,N2)
    a4 = side(N3,N4)
    a5 = side(N2,N3)
    a6 = side(N2,N4)
    volume=tetVol(a1,a2,a3,a4,a5,a6)
    return volume*1e9

#-------------------------------------------------------------------------------
def subTetVolCalc_mm(elemConn,a,b,c,d):
#-------------------------------------------------------------------------------
    # Determine the volume of a tetrahedron for a given four nodes

    N1 = node_pos[elemConn.Node(a)-1]
    N2 = node_pos[elemConn.Node(b)-1]
    N3 = node_pos[elemConn.Node(c)-1]
    N4 = node_pos[elemConn.Node(d)-1]
    a1 = side(N1,N4)
    a2 = side(N1,N3)
    a3 = side(N1,N2)
    a4 = side(N3,N4)
    a5 = side(N2,N3)
    a6 = side(N2,N4)
    volume=tetVol(a1,a2,a3,a4,a5,a6)
    return volume*1e9

#-------------------------------------------------------------------------------
def elemHexVolCalc_mm(elemConn):
#-------------------------------------------------------------------------------
    # Determine the volume of a hexahedron by decomposing into 5 tetrahedrons
    # and summing their volumes

    v1 = subTetVolCalc_mm(elemConn,0,1,3,4)
    v2 = subTetVolCalc_mm(elemConn,1,2,3,6)
    v3 = subTetVolCalc_mm(elemConn,1,4,5,6)
    v4 = subTetVolCalc_mm(elemConn,3,4,6,7)
    v5 = subTetVolCalc_mm(elemConn,1,3,4,6)
    return v1+v2+v3+v4+v5

#-------------------------------------------------------------------------------
def elemPentVolCalc_mm(elemConn):
#-------------------------------------------------------------------------------
    # Determine the volume of a pentahedron by decomposing into 3 tetrahedrons
    # and summing their volumes

    v1 = subTetVolCalc_mm(elemConn,0,1,2,4)
    v2 = subTetVolCalc_mm(elemConn,0,2,3,5)
    v3 = subTetVolCalc_mm(elemConn,0,2,4,5)
    return v1+v2+v3

#-------------------------------------------------------------------------------
def det(a):
#-------------------------------------------------------------------------------
    # Calculate the determinant of 3x3 vector

    return a[0][0]*a[1][1]*a[2][2] + a[0][1]*a[1][2]*a[2][0] \
         + a[0][2]*a[1][0]*a[2][1] - a[0][2]*a[1][1]*a[2][0] \
         - a[0][1]*a[1][0]*a[2][2] - a[0][0]*a[1][2]*a[2][1]

#-------------------------------------------------------------------------------
```

```
def unit_normal(a, b, c):
#-------------------------------------------------------------------------------
    # unit normal vector of plane defined by points a, b, and c

    x = det([[1,a[1],a[2]],
             [1,b[1],b[2]],
             [1,c[1],c[2]]])
    y = det([[a[0],1,a[2]],
             [b[0],1,b[2]],
             [c[0],1,c[2]]])
    z = det([[a[0],a[1],1],
             [b[0],b[1],1],
             [c[0],c[1],1]])
    magnitude = (x**2 + y**2 + z**2)**.5
    return (x/magnitude, y/magnitude, z/magnitude)
#-------------------------------------------------------------------------------
#dot product of vectors a and b
def dot(a, b):
#-------------------------------------------------------------------------------
    return a[0]*b[0] + a[1]*b[1] + a[2]*b[2]
#-------------------------------------------------------------------------------
#cross product of vectors a and b
def cross(a, b):
#-------------------------------------------------------------------------------
    x = a[1] * b[2] - a[2] * b[1]
    y = a[2] * b[0] - a[0] * b[2]
    z = a[0] * b[1] - a[1] * b[0]
    return (x, y, z)
#-------------------------------------------------------------------------------
def polyArea(poly):
#-------------------------------------------------------------------------------
    # Determine the area of a four-node segment.

    total = [0, 0, 0]
    for i in range(len(poly)):
        vi1 = poly[i]
        if i is len(poly)-1:
            vi2 = poly[0]
        else:
            vi2 = poly[i+1]
        prod = cross(vi1, vi2)
        total[0] += prod[0]
        total[1] += prod[1]
        total[2] += prod[2]
    result = dot(total, unit_normal(poly[0], poly[1], poly[2]))
    return abs(result/2)

#-------------------------------------------------------------------------------
def elemQuadAreaCalc_mm(elemConn):
#-------------------------------------------------------------------------------
    # Determine the area of

    N1 = node_pos[elemConn.Node(0)-1]
    N2 = node_pos[elemConn.Node(1)-1]
    N3 = node_pos[elemConn.Node(2)-1]
    N4 = node_pos[elemConn.Node(3)-1]
    return polyArea([[N1.X(),N1.Y(),N1.Z()], \
                     [N2.X(),N2.Y(),N2.Z()], \
                     [N3.X(),N3.Y(),N3.Z()], \
                     [N4.X(),N4.Y(),N4.Z()]])

#-------------------------------------------------------------------------------
def initD3plot(plotPathName,partList,listParts,metricList,ihv,isVerbose=False):
#-------------------------------------------------------------------------------
    # read d3plots and define element parameters based on element type
    # (eg, hex, tet, shell, ...)

    p = D3P_Parameter()
    try:
        dr = D3plotReader(plotPathName)
        # Read num states
```

```
    numStates = dr.get_data(DataType.D3P_NUM_STATES,p)
except:
    print(f"Error reading d3plot file states ({plotPathName}).")
    sys.exit("Exiting (code 755)...")

if False:
    print("****************************************************************")
    print("****************************************************************")
    print("                 D E B U G   M O D E")
    numStates = 80
    print("****************************************************************")
    print("****************************************************************")
# Read nodes
numNodes = dr.get_data(DataType.D3P_NUM_NODES, p)
node_ids = dr.get_data(DataType.D3P_NODE_IDS, p)
node_pos = dr.get_data(DataType.D3P_NODE_INITIAL_COORDINATES, p)
# Read parts
numParts = dr.get_data(DataType.D3P_NUM_PARTS, p)
part_ids = dr.get_data(DataType.D3P_PART_IDS, p)

# Initialize D3P for part number
if listParts:
   listPartsDesc(p,dr,numParts,part_ids)
   sys.exit("Listed parts only, exiting...")

p.ipart_user = partList[0]
p.ask_for_numpy_array = True
p.ipt = 0
p.ist = 0
p.ihv = 0
if 'H' in metricList:
    p.ihv = ihv
partName = dr.get_data(DataType.D3P_PART_NAME,p)
p.ipartset_user = partList

# Determine part type
if dr.get_data(DataType.D3P_IS_SOLID,p):
    partType = 'SOLID'
    d3p_num_elements = DataType.D3P_NUM_SOLID
    d3p_element_connectivity_mat = DataType.D3P_SOLID_CONNECTIVITY_MAT
    d3p_element_ids = DataType.D3P_SOLID_IDS
    d3p_element_stress = DataType.D3P_SOLID_STRESS
    d3p_element_vm_stress = DataType.D3P_SOLID_VON_MISES_STRESS
    d3p_element_signvm_stress = DataType.D3P_SOLID_SIGNED_VON_MISES_STRESS
    d3p_element_1princ_stress = DataType.D3P_SOLID_1ST_PRINCIPAL_STRESS
    d3p_element_2princ_stress = DataType.D3P_SOLID_2ND_PRINCIPAL_STRESS
    d3p_element_3princ_stress = DataType.D3P_SOLID_3RD_PRINCIPAL_STRESS
    d3p_element_tresca_stress = DataType.D3P_SOLID_TRESCA_STRESS

    d3p_element_eps = DataType.D3P_SOLID_EFFECTIVE_PLASTIC_STRAIN

    d3p_element_strain = DataType.D3P_SOLID_STRAIN
    d3p_element_vm_strain = DataType.D3P_SOLID_VON_MISES_STRAIN
    d3p_element_signvm_strain = DataType.D3P_SOLID_SIGNED_VON_MISES_STRAIN
    d3p_element_maxprinc_strain = DataType.D3P_SOLID_MAX_PRINCIPAL_STRAIN
    d3p_element_2princ_strain = DataType.D3P_SOLID_2ND_PRINCIPAL_STRAIN
    d3p_element_minprinc_strain = DataType.D3P_SOLID_MIN_PRINCIPAL_STRAIN
    d3p_element_tresca_strain = DataType.D3P_SOLID_TRESCA_STRAIN
    d3p_element_plast_strain = DataType.D3P_SOLID_PLASTIC_STRAIN
    d3p_element_hist_var = DataType.D3P_SOLID_HISTORY_VAR
    d3p_element_inf_strain = DataType.D3P_SOLID_INFINITESIMAL_STRAIN
    d3p_element_green_strain = DataType.D3P_SOLID_GREEN_STRAIN
    d3p_element_almansi_strain = DataType.D3P_SOLID_ALMANSI_STRAIN
    d3p_element_rate_strain = DataType.D3P_SOLID_STRAIN_RATE

    d3p_element_deletion = DataType.D3P_SOLID_DELETION
elif dr.get_data(DataType.D3P_IS_SHELL,p):
    partType = 'SHELL'
    d3p_num_elements = DataType.D3P_NUM_SHELL
    d3p_element_connectivity_mat = DataType.D3P_SHELL_CONNECTIVITY_MAT
    d3p_element_ids = DataType.D3P_SHELL_IDS
```

```
        d3p_element_stress = DataType.D3P_SHELL_STRESS
        d3p_element_vm_stress = DataType.D3P_SHELL_VON_MISES_STRESS
        d3p_element_signvm_stress = DataType.D3P_SHELL_SIGNED_VON_MISES_STRESS
        d3p_element_1princ_stress = DataType.D3P_SHELL_1ST_PRINCIPAL_STRESS
        d3p_element_2princ_stress = DataType.D3P_SHELL_2ND_PRINCIPAL_STRESS
        d3p_element_3princ_stress = DataType.D3P_SHELL_3RD_PRINCIPAL_STRESS
        d3p_element_tresca_stress = DataType.D3P_SHELL_TRESCA_STRESS

        d3p_element_eps = DataType.D3P_SHELL_EFFECTIVE_PLASTIC_STRAIN

        d3p_element_strain = DataType.D3P_SHELL_STRAIN
        d3p_element_vm_strain = DataType.D3P_SHELL_VON_MISES_STRAIN
        d3p_element_signvm_strain = DataType.D3P_SHELL_SIGNED_VON_MISES_STRAIN
        d3p_element_maxprinc_strain = DataType.D3P_SHELL_MAX_PRINCIPAL_STRAIN
        d3p_element_2princ_strain = DataType.D3P_SHELL_2ND_PRINCIPAL_STRAIN
        d3p_element_minprinc_strain = DataType.D3P_SHELL_MIN_PRINCIPAL_STRAIN
        d3p_element_tresca_strain = DataType.D3P_SHELL_TRESCA_STRAIN
        d3p_element_plast_strain = DataType.D3P_SHELL_PLASTIC_STRAIN
        d3p_element_hist_var = DataType.D3P_SHELL_HISTORY_VAR
        d3p_element_inf_strain = DataType.D3P_SHELL_INFINITESIMAL_STRAIN
        d3p_element_green_strain = DataType.D3P_SHELL_GREEN_STRAIN
        d3p_element_almansi_strain = DataType.D3P_SHELL_ALMANSI_STRAIN
        d3p_element_rate_strain = DataType.D3P_SHELL_STRAIN_RATE

        d3p_element_deletion = DataType.D3P_SHELL_DELETION
    else:
        print(f'***Error*** Unrecognized part type ({partList})')
        sys.exit("Terminating")

    # Read elements
    partNumElements = np.int32(dr.get_data(d3p_num_elements,p))
    element_ids = dr.get_data(d3p_element_ids,p)
    element_conn = dr.get_data(d3p_element_connectivity_mat,p)

    if isVerbose:
        print(f"Model:      States={numStates}            Part      Count={numParts}
Elements={partNumElements}  Nodes={numNodes}")
        print(f'Part  #:  {partList}  [{partName.rstrip()}]      Type:  {partType}
Metric: {metricList}')
        if False:
            for i in range(numParts):
                print("i = ",i,"   part id = ",part_ids[i])

    return p,dr,numStates,numNodes,node_ids,node_pos,numParts,part_ids,\
           partType,partNumElements,element_ids,element_conn,d3p_element_stress,\
           d3p_element_1princ_stress,                    d3p_element_2princ_stress,
d3p_element_3princ_stress,\
           d3p_element_tresca_stress,                        d3p_element_vm_stress,
d3p_element_signvm_stress,\
           d3p_element_strain,       d3p_element_eps,       d3p_element_vm_strain,
d3p_element_signvm_strain,\
           d3p_element_maxprinc_strain,                  d3p_element_2princ_strain,
d3p element minprinc strain,\
           d3p_element_tresca_strain,                     d3p_element_plast_strain,
d3p_element_hist_var,\
           d3p_element_inf_strain,                        d3p_element_green_strain,
d3p_element_almansi_strain,\
           d3p_element_rate_strain, d3p_element_deletion

#-------------------------------------------------------------------------------
def voxSegUsed(voxID,nextVoxID,voxSegHist):
#-------------------------------------------------------------------------------
    # Search voxSegHist for prior use of sorted segment voxID-nextVoxID
    # segment recorded low-to-high for search simplicity and because segments
    # are not directional dependent.

    minTarget, maxTarget = sorted([voxID,nextVoxID])
    lowKey = list(map(itemgetter(0), voxSegHist))
    voxSegUsed = False
    # print(f"lowKey-{lowKey}  minTarget={minTarget}  maxTarget={maxTarget}")
    for i in [k for k,x in enumerate(lowKey) if x == minTarget]:
```

```
        if voxSegHist[i][1] == maxTarget:
            voxSegUsed = True
    return voxSegUsed

#--------------------------------------------------------------------------------
def voxUsedinChain(nextVoxID,voxIDchain):
#--------------------------------------------------------------------------------
    # Determine if nextVoxID has been used in the current voxIDchain, which
    # would indicate a loop.

    try:
        usedVox = voxIDchain.index(nextVoxID)
        voxUsedinChain = True
    except (ValueError, IndexError):
        voxUsedinChain = False
    return voxUsedinChain

#--------------------------------------------------------------------------------
def neighborAvailQuery(curNeighborList,df_skel2xList):
#--------------------------------------------------------------------------------
    # Using the x,y,z voxel indices, determine the voxID and available linked voxel
count.

    [xidx,yidx,zidx] = curNeighborList
    df_neighborVox                                                              =
df_skel2xList[(df_skel2xList["xidx"]==xidx)&(df_skel2xList["yidx"]==yidx)&(df_skel
2xList["zidx"]==zidx)]
    if len(df_neighborVox) == 0:
        # print(f"*** WARNING *** curNeighbor list empty??? ({curNeighborList})")
        return 0, 0
    return df_neighborVox.iloc[0].voxID, df_neighborVox.iloc[0].availCnt

#--------------------------------------------------------------------------------
def recurTraceVox(df_skel2xList,voxID,fxPathLen,voxIDchain,voxSegHist,depth=0):
#--------------------------------------------------------------------------------
    # Recursive voxel chain trace.
    # print(f"recur voxID={voxID}")

    df_curVox = df_skel2xList[df_skel2xList['voxID']==voxID]
    curNeighborIDXList = df_curVox.iloc[0].neighborList
    i = 0
    i_max = -1
    neighborAvailCnt_max = -1
    while i < len(curNeighborIDXList):
        neighborVoxID,                    neighborAvailCnt_cur                  =
neighborAvailQuery(curNeighborIDXList[i],df_skel2xList)
        if     (neighborAvailCnt_cur     >     neighborAvailCnt_max)     and     not
voxSegUsed(voxID,neighborVoxID,voxSegHist):
            neighborAvailCnt_max = neighborAvailCnt_cur
            i_max = i
        i += 1
    if neighborAvailCnt_max > 0:
        [xidxNext,yidxNext,zidxNext] = curNeighborIDXList[i max]
        df_nextVox                                                              =
df_skel2xList[(df_skel2xList["xidx"]==xidxNext)&(df_skel2xList["yidx"]==yidxNext)&
(df_skel2xList["zidx"]==zidxNext)]
        nextVoxID = df_nextVox.iloc[0].voxID
        if       not       voxUsedinChain(nextVoxID,voxIDchain)       and       not
voxSegUsed(voxID,nextVoxID,voxSegHist):
            voxIDchain.append(nextVoxID)
            voxSegNew = sorted([voxID,nextVoxID])
            voxSegHist.append(voxSegNew)
            df_skel2xList.loc[df_curVox.index[0], 'availCnt'] -= 1
            df_skel2xList.loc[df_curVox.index[0], 'usedCnt'] += 1
            [x1,y1,z1]                                                          =
[df_curVox.iloc[0].x,df_curVox.iloc[0].y,df_curVox.iloc[0].z]
            [x2,y2,z2]                                                          =
[df_nextVox.iloc[0].x,df_nextVox.iloc[0].y,df_nextVox.iloc[0].z]
            segLen = np.sqrt((x2-x1)**2+(y2-y1)**2+(z2-z1)**2)
            fxPathLen += segLen
            df_skel2xList.loc[df_nextVox.index[0], 'availCnt'] -= 1
```

```
            df_skel2xList.loc[df_nextVox.index[0], 'usedCnt'] += 1
            if        (df_nextVox.iloc[0].neighborCnt        <        3)        &
(df_nextVox.iloc[0].neighborCnt > 0):
                depth += 1
                if depth < 999:
                    # Failsafe recursion depth (Python default 1,000).
                    #---------------------- recursion ---------------------------
-
                    df_skel2xList,fxPathLen,voxIDchain,voxSegHist \
                        = recurTraceVox(df_skel2xList,nextVoxID,fxPathLen,
                                        voxIDchain,voxSegHist,depth)
                    #---------------------- recursion ---------------------------
-
    return df_skel2xList,fxPathLen,voxIDchain,voxSegHist

#-------------------------------------------------------------------------------
def calcVoxelDims(df_fx, voxelRatio, IS_VALIDATE=False):
#-------------------------------------------------------------------------------
    # Calculate the voxel cube dimensions and world coordinate x/y/z mins/maxs
    # and voxelDimAdj


    if not IS_VALIDATE:
        mean_fxVol = np.mean(df_fx["vol"])
        mean_fxRad = (0.75*mean_fxVol/pi)**0.3333
        mean_fxRad2 = np.mean(df_fx["rad2"])
        mean_fxDiam = 2*mean_fxRad
        voxelDimRaw = mean_fxDiam/1000     # mm to m
        # mean_fxDiam2 = 2*mean_fxRad2
        # voxelDimRaw = mean_fxDiam2          # already in m
        # voxelDimAdj = voxelDimRaw/0.9   # increasing scale increases ydim
        # voxelDimAdj = voxelDimRaw/0.75   # increasing scale increases ydim
        voxelDimAdj = voxelDimRaw/voxelRatio

        buffer = 2 * voxelDimAdj
        xmin = min(df_fx["x"]) - buffer
        xmax = max(df_fx["x"]) + buffer
        xspan = xmax-xmin            # spanning distance (m) world coord units plus
margin buffers
        ymin = min(df_fx["y"]) - buffer
        ymax = max(df_fx["y"]) + buffer
        yspan = ymax-ymin            # spanning distance (m) world coord units plus
margin buffers
        zmin = min(df_fx["z"]) - buffer
        zmax = max(df_fx["z"]) + buffer
        zspan = zmax-zmin            # spanning distance (m) world coord units plus
margin buffers

        xdim = int(np.ceil(xspan/voxelDimAdj))
        ydim = int(np.ceil(yspan/voxelDimAdj))
        zdim = int(np.ceil(zspan/voxelDimAdj))
    else:
        mean fxVol = 1e-9
        # mean_fxD = 2*(0.75*mean_fxVol/pi)**0.3333
        voxelDimRaw = 1/1000
        voxelDimAdj = voxelDimRaw/0.9   # increasing scale increases ydim

        mastDim = 104 # 104 for 1mm cubes

        xmin = 0
        xmax = mastDim
        ymin = 0
        ymax = mastDim
        zmin = 0
        zmax = mastDim

        xdim = mastDim
        ydim = mastDim
        zdim = mastDim

    return xdim, ydim, zdim, xmin, xmax, ymin, ymax, zmin, zmax, voxelDimAdj
```

```
#-------------------------------------------------------------------------------
def VALIDATE_setVox_Analytic (C):
#-------------------------------------------------------------------------------
    # Validation routine for calculation length of a half circle

    progBarTtl = "Set FX 1x Vox"
    xcenter = 52
    xyrad2 = 2500
    subsamp = 200
    for ix in tqdm(range(xSpanBuffer*subsamp-1,(xSpanRaw+xSpanBuffer)*subsamp+1),\
                   desc="Validation Arc",ncols=TQDM_COLS,disable=isQuiet):
        x = ix/subsamp
        ang1 = pi/4
        rlen = x - xcenter

        x2 = rlen * sin(ang1) + xcenter
        x3 = -rlen * sin(ang1) + xcenter

        analyticLen = []

        ysquare =  max(0,xyrad2 - rlen**2)
        y = sqrt(ysquare) + xcenter
        y1 = -sqrt(ysquare) + xcenter
        analyticLen.append(rlen*pi)
        C.setvox(1,x,y,xcenter,elid=x)     # upper semi-circle 1
        analyticLen.append(rlen*pi)
        C.setvox(1,xcenter,y,x,elid=x)     # upper semi-circle 2
        analyticLen.append(rlen*pi)
        C.setvox(1,x2,y1,x2,elid=x)        # upper semi-circle 4 @ ang1
        analyticLen.append(rlen*pi)
        C.setvox(1,x,y1,xcenter,elid=x)    # lower semi-circle 1
        analyticLen.append(rlen*pi)
        C.setvox(1,xcenter,y1,x,elid=x)    # lower semi-circle 2
        analyticLen.append(rlen*2)
        C.setvox(1,x,xcenter,xcenter,elid=x) # horz line 1

        analyticLen.append(rlen*2)
        C.setvox(1,x2,xcenter+8,x3,elid=x) # horz line 2

        ang2 = pi*(1/4+1/16)
        x4 = rlen * cos(ang1)*cos(ang2)+xcenter
        z4 = rlen * cos(ang1)*sin(ang2)+xcenter
        y4 = rlen * sin(ang1)+xcenter
        analyticLen.append(rlen*2)
        C.setvox(1,x4,y4+4,z4,elid=x)         # diagonal line

        suma = sum(analyticLen)

    print(f"Validation                          Analytic                          Len:
{analyticLen}\n\ttotal={sum(analyticLen):0.2f}")

    # Optional test plot for debugging
    if False:
        fig = plt.figure(2)
        ax = fig.add_subplot(111, projection='3d')
        indices = np.where(C.voxels == 1)
        ax.scatter(indices[0],indices[1],indices[2],c='r')
        #
ax.scatter(array_3d[:,:,0].flatten(),array_3d[:,:,1].flatten(),array_3d[:,:,2].fla
tten(),c='r')
        ax.set_title("C.voxels")
        ax.set_xlabel('X-axis')
        ax.set_ylabel('Y-axis')
        ax.set_zlabel('Z-axis')
        ax.set_xlim(0, 104)
        ax.set_ylim(0, 104)
        ax.set_zlim(0, 104)
        plt.show()

        C.plotvox(sf=2)
```

```
    return C, suma

#-------------------------------------------------------------------------------
def map_1X_to_2X(C, C2, xdim, ydim, zdim):
#-------------------------------------------------------------------------------
    xs1 = []
    ys1 = []
    zs1 = []
    xs1s = []
    ys1s = []
    zs1s = []
    xs2 = []
    ys2 = []
    zs2 = []

    xs2s = []
    ys2s = []
    zs2s = []

    progBarTtl = "Expand 1x skeleton to 2x"
    for i in tqdm(range(xdim),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):
        for j in range(ydim):
            for k in range(zdim):
                vval = C.voxels[i,j,k]
                # Populate native fracture traces
                if vval > 0:
                    xs1.append(i+int(xdim/2))
                    ys1.append(j+int(5*ydim/2))
                    zs1.append(k+int(3*zdim/2))
                # Populate skeletonized native fracture traces
                if skel3d[i,j,k] > 0:
                    xs1s.append(i+int(xdim/2))
                    ys1s.append(j+int(5*ydim/2))
                    zs1s.append(k+int(5*zdim/2))
                for ii in range(2):
                    for jj in range(2):
                        for kk in range(2):
                            # C2.setvox(vval,2*i+ii,2*j+jj,2*k+kk)
                            C2.voxels[2*i+ii,2*j+jj,2*k+kk] = vval
                            # Populate 2x native fracture traces
                            if vval > 0:
                                xs2.append(2*i+ii)
                                ys2.append(2*j+jj)
                                zs2.append(2*k+kk)
    return xs2s, ys2s, zs2s

#-------------------------------------------------------------------------------
def plot_C2_skel3d2x(C2,skel3d2x):
#-------------------------------------------------------------------------------
    #%%% Validate
    fig = plt.figure(12)
    ax = fig.add_subplot(111, projection='3d')
    indices = np.where(C2.voxels == 1)
    ax.scatter(indices[0],indices[1],indices[2],c='r',)
    ax.set_title("C2.voxels")
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
    ax.set_zlabel('Z-axis')
    ax.set_xlim(0, 208)
    ax.set_ylim(0, 208)
    ax.set_zlim(0, 208)
    plt.show()

    fig = plt.figure(13)
    ax = fig.add_subplot(111, projection='3d')
    indices = np.where(skel3d2x > 0.999)
    ax.scatter(indices[0],indices[1],indices[2],c='r')
    ax.set_title("skel3d2x")
    ax.set_xlabel('X-axis')
    ax.set_ylabel('Y-axis')
```

```
    ax.set_zlabel('Z-axis')
    ax.set_xlim(0, 208)
    ax.set_ylim(0, 208)
    ax.set_zlim(0, 208)
    plt.show()

    # fig = plt.figure(14)
    # surf = ax.plot_surface(indices[0],indices[1],indices[2], cmap='viridis')
    # # Add a color bar
    # fig.colorbar(surf, shrink=0.5, aspect=5)
    # plt.show()

    # fig = plt.figure(15)
    # ax = fig.add_subplot(111, projection='3d')
    # indices = np.where(skel3d2x == 1)
    # surf = ax.plot_surface(indices[0],indices[1],indices[2], cmap='viridis')
    # # Add a color bar
    # fig.colorbar(surf, shrink=0.5, aspect=5)
    # plt.show()

#-------------------------------------------------------------------------------
def build_2x_fxList(skelel2,x2s2,y2s2,z2s2):
#-------------------------------------------------------------------------------
    progBarTtl = "Build 2x skeleton fracture element list"
    skel2xList:  list[skelel2] = []
    nbuf = 1 # number of voxel layers from outer surface voxel volume to omit from
             # analysis, largely to avoid invalid array index accesses.
    mc2s = []
    voxID = 0
    for                 i                 in                tqdm(range(nbuf,2*xdim-
nbuf),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):
        for j in range(nbuf,2*ydim-nbuf):
            for k in range(nbuf,2*zdim-nbuf):
                # Populate skeletonized 2x native fracture traces
                if skel3d2x[i,j,k] > 0:
                    xs2s.append(i)
                    ys2s.append(j)
                    zs2s.append(k)
                    neighborCnt = 0
                    neighborList = []
                    for ii in range(-nbuf,nbuf+1):
                        for jj in range(-nbuf,nbuf+1):
                            for kk in range(-nbuf,nbuf+1):
                                if skel3d2x[i+ii,j+jj,k+kk] > 0:
                                    if (ii != 0) | (jj != 0) | (kk != 0):
                                        neighborCnt += 1
                                        neighborList.append([i+ii,j+jj,k+kk])
                    [x,y,z] = C2.IDX2xyz(i, j, k)
                    voxID += 1

skel2xList.append(skelel2(i,j,k,voxID,x,y,z,neighborCnt,neighborList))
                    mc2s.append(neighborCnt)

    df_skel2xList = pd.DataFrame([vars(s) for s in skel2xList])

    return df_skel2xList, mc2s

#-------------------------------------------------------------------------------
def
populateFXtraceLabels(listOfvoxIDchains,df_skel2xList,lopl,SKIP_LT_2_mm,UNIT_CONV)
:
#-------------------------------------------------------------------------------
    # determine labels for fracture trace plots

    fl = []
    ml = []
    ll = []
    voxIDfirstList = []
    voxIDlastList = []
    # ---- Deterimine start-, mid-, and end-trace voxels for label placement
    for i in range(len(listOfvoxIDchains)):
```

```
        voxlist = listOfvoxIDchains[i]
        voxlistLen = len(voxlist)
        voxFirst = voxlist[0]
        voxMidid = voxlist[int(len(voxlist)/2)]
        voxLast = voxlist[-1]
        ### print(f"i={i}   f={voxFirst}  m={voxMidid}  l={voxLast}")

        voxIDfirstList.append(voxFirst)
        voxIDlastList.append(voxLast)

        vsl = df_skel2xList[df_skel2xList["voxID"] == voxFirst]
        fl.append([vsl.iloc[0].xidx,vsl.iloc[0].yidx,vsl.iloc[0].zidx])
        vsl = df_skel2xList[df_skel2xList["voxID"] == voxMidid]
        ml.append([vsl.iloc[0].xidx,vsl.iloc[0].yidx,vsl.iloc[0].zidx])
        vsl = df_skel2xList[df_skel2xList["voxID"] == voxLast]
        ll.append([vsl.iloc[0].xidx,vsl.iloc[0].yidx,vsl.iloc[0].zidx])

    listOfFxPathLen = []
    firstList = []
    midList = []
    lastList = []

    # ---- Calculate total fracture trace length and build plot label list
    for i in range(len(lopl)):
        if not SKIP_LT_2_mm or (SKIP_LT_2_mm and lopl[i] >= 2/UNIT_CONV):
            listOfFxPathLen.append(lopl[i])
            firstList.append(fl[i])
            midList.append(ml[i])
            lastList.append(ll[i])

    return   listOfFxPathLen,   firstList,   midList,   lastList,   voxIDfirstList,
voxIDlastList

#--------------------------------------------------------------------------
def plot3dInteractive(xs2s, ys2s, zs2s, mc2s, sf, firstList, midList, lastList,
                      voxIDfirstList, voxIDlastList, fxtot, listOfFxPathLen,
                      pltTitle, isSave3dPlot=False,pngName="none.png"):
#--------------------------------------------------------------------------
    # ---- define color mapping for the plots
    lut = np.zeros((5, 4))
    lut[0,:] = [255,0,0,255]
    lut[1,:] = [0,255,0,255]
    lut[2,:] = [0,0,255,255]
    lut[3,:] = [0,255,255,255]
    lut[4,:] = [255,255,0,255]
    lut[:,-1] = 255 # No transparency (aka [:,4])

    # Plot the points, update its lookup table
    #p3d = mlab.points3d(x, y, z, s, scale_mode='none')
    #p3d.module_manager.scalar_lut_manager.lut.number_of_colors = len(s)
    #p3d.module_manager.scalar_lut_manager.lut.table = lut

    # ---- Plot fracture traces
    if True:
        # Color
        sf = .75
        voxplot   =   mlab.points3d(xs2s,   ys2s,   zs2s,   mc2s,   scale_factor=sf,
mode='cube')  # works, all white cubes
        voxplot.module_manager.scalar_lut_manager.lut.number_of_colors = 5
        voxplot.module_manager.scalar_lut_manager.lut.table = lut
    else:
        # All White
        sf = 1
        voxplot = mlab.points3d(xs2s, ys2s, zs2s, scale_factor=sf, mode='cube')  #
works, all white cubes
        #voxplot.glyph.scale_mode = 'scale_by_vector'
        #voxplot.mlab_source.dataset.point_data.scalars = mc

    # mlab.text3d(50, 50, 50, '{:0.1f}'.format(listOfFxPathLen[0]*1000), scale=3,
color=textColor)
    # for i in len(listOfFxPathLen):
```

```
    textColor = (1,0,0)
    firstColor = (0,0,1)
    lastColor = (0,1,0)

    voxDetail = 0

    # ---- Label plots
    for i in range(len(midList)):
        if voxDetail == 0:
            mlab.text3d(midList[i][0],midList[i][1],midList[i][2],'        {:0.0f}
mm'.format(listOfFxPathLen[i]*UNIT_CONV),scale=2,orient_to_camera=True,color=textC
olor)

mlab.text3d(firstList[i][0],firstList[i][1],firstList[i][2],'s',scale=1,orient_to_
camera=True,color=firstColor)
            mlab.text3d(lastList[i][0],lastList[i][1],lastList[i][2],'
e',scale=1,orient_to_camera=True,color=lastColor)
        elif voxDetail == 1:
            mlab.text3d(midList[i][0],midList[i][1],midList[i][2],'        {:0.0f}
mm'.format(listOfFxPathLen[i]*UNIT_CONV),scale=2,orient_to_camera=True,color=textC
olor)
            mlab.text3d(firstList[i][0],firstList[i][1],firstList[i][2],'s{:d}
s'.format(i),scale=1,orient_to_camera=True,color=(0,0,1))
            mlab.text3d(lastList[i][0],lastList[i][1],lastList[i][2],'e
{:d}e'.format(i),scale=1,orient_to_camera=True,color=(0,1,0))
        elif voxDetail == 2:
            mlab.text3d(midxList[i],midyList[i],midzList[i],'
{:0.0f}mm'.format(listOfFxPathLen[i]*UNIT_CONV),scale=2,orient_to_camera=True,colo
r=textColor)
            mlab.text3d(firstx[i],firsty[i],firstz[i],'sss                    {:d}
sss'.format(firstElID[i]),scale=1,orient_to_camera=True,color=(0,0,1))
            mlab.text3d(lastx[i],lasty[i],lastz[i],'eee                    {:d}
eee'.format(lastElID[i]),scale=1,orient_to_camera=True,color=(0,1,0))
        elif voxDetail == 3:
            mlab.text3d(midList[i][0],midList[i][1],midList[i][2],'
{:0.0f}mm'.format(listOfFxPathLen[i]*UNIT_CONV),scale=2,orient_to_camera=True,colo
r=textColor)
            mlab.text3d(firstList[i][0],firstList[i][1],firstList[i][2],'s{:d}
{:d}                                {:d}
s'.format(i,voxIDfirstList[i],i),scale=1,orient_to_camera=True,color=(0,0,1))
            mlab.text3d(lastList[i][0],lastList[i][1],lastList[i][2],'e  {:d} {:d}
{:d}e'.format(i,voxIDlastList[i],i),scale=1,orient_to_camera=True,color=(0,1,0))

    ttlMeas = "{:0,.0f} mm".format(fxtot)
    mlab.title(pltTitle+"
("+ttlMeas+")",color=(0,0,0),height=.85,opacity=0.5,size=0.5)

    if isSave3dPlot:
        mlab.savefig(pngName)
        mlab.close()
    else:
        mlab.show()

##############################################################################
#%% MAIN
##############################################################################
# \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/ \/
#%%% Initialize variables
##############################################################################
t0 = time.time()

#%% >Define program parameters (command line and predefined)
##############################################################################
#---- Pre-defined parameters
##############################################################################

if os.name == "nt" and any('SPYDER' in name for name in os.environ):
    # Windows Environment
    pathSep = "\\"
    elemPath = "%HOMEPATH%\\Documents\\script_repo\\python\\"
```

```
    plotPath = "%HOMEPATH%\\Documents\\script_repo\\python\\"
    d3plotName = "d3plot"
else:
    # Linux Environment
    pathSep = "/"
    elemPath = "/p/app/projects/bioshare/scripts/python/"

    plotPath = "."
    d3plotName = "d3plot"

fullPlot = plotPath + pathSep + d3plotName

elemFilePrefix  = "measElem_"
elemFilePostfix = ".txt"
dictFileName    = "measDict.txt"

defPartStr = "35"
voxelRatioListStr = "1.00-1.20"
################################################################################
#---- Command line parameters
################################################################################
parser = argparse.ArgumentParser(description ='Automated Fracture Trace Measuring',

formatter_class=argparse.RawDescriptionHelpFormatter,
                                 epilog= "Dictionary file: "+dictFileName+"\n" \
                                    +    "e.g.,\n" \
                                    +    "      i,inner\n" \
                                    +    "      o,outer\n\n" \
                                    +"references list of element files:\n" \
                                    +                                          "
"+elemFilePrefix+"i"+elemFilePostfix+"\n" \
                                    +                                          "
"+elemFilePrefix+"o"+elemFilePostfix+"\n" \
                                    +"The   second   dictionary   arguments   are
descriptions.")
parser.add_argument("-p", "--parts", type = str, nargs = 1,
                    metavar = "<partListStr>", dest = "partList",
                    default = defPartStr,
                    help = "Part number(s) (concat multiple with '+' " + \
                           "NO spaces, default="+defPartStr+", whole model).")
parser.add_argument("-r", "--rootPath", type = str, nargs = 1,
                    metavar = "<d3plotDirectory>", dest = "rootPath",
                    default = plotPath,
                    help = "d3plot directory path. (default="+plotPath+").")
parser.add_argument("-d", "--d3plot", type = str, nargs = 1,
                    metavar = "<fullPath>", dest = "d3plotName",
                    default = "d3plotName",
                    help = "<d3plotName> file name. (default="+d3plotName+").")
parser.add_argument("-e", "--elemListChar", type = str, nargs = 1,
                    metavar="<abc..xyz012..9>", dest= "layers",
                    default = '@',
                    help = "Element list file infix character(s), no spaces,\
                        for file(s) 'measElem_#.txt'")
parser.add_argument("-hv", "--histvar", type = str, nargs = 1,
                    metavar = "<hv#>", dest = "metric",
                    default = "hv1",
                    help = "Fracture metric. (default=hv1).")
parser.add_argument("-l", "--log", type = str, nargs = 1,
                    metavar = "<logFile>", dest = "logFile",
                    default = "",
                    help = "Output to log file. (default=none).")
parser.add_argument("-v", "--voxelRatioList", type = str, nargs = 1,
                    metavar = "<voxRatioList>", dest = "voxelRatioListStr",
                    default = voxelRatioListStr,
                    help = "Voxel ratio, larger shrinks voxels size. \
                            Format #.## or #.##-#.## (e.g., 2.89 or \
                            2.81-2.95). (default="+voxelRatioListStr+").")
parser.add_argument('-b',  '--batch',   dest='isBatch',   action='store_true',
                    help ='Batch mode, suppress 3D inter-active window (default: no
batch)')
```

```
parser.add_argument('-S',         '--Save3dPlot',             dest='isSave3dPlot',
action='store_true',
                    help ='Save 3D plot. Suitable for running analyses \
                           in background. (default: no saved plot)')
parser.add_argument('-s',       '--saveInt3dPlot',          dest='isIntSave3dPlot',
action='store_true',
                    help ='Save all interim vox ratio 3D plots. Suitable for running
analyses \
                           in background. (default: no save plots)')
parser.add_argument('-P',        '--ProgressBars',           dest='isShowProgress',
action='store_true',
                    help  ='Display  progress  output.  Not  suitable  for  running
analyses \
                           in background. (default: no Progress Bars)')
parser.add_argument('-V',  '--verbose', dest='isVerbose',  action='store_true',
                    help ='Verbose output (default: false)')

a = parser.parse_args()

partList = [int(a.partList[0])] # NOTE: d3plot part number, which includes the
                            # elements (e.g., [i]nner, [m]iddle, ...).

listParts = False
metricList = [a.metric]
ihv = int(a.metric[0].replace("hv",'')) - 1        # zero-based, e.i., hist var-1
if isinstance(a.logFile,list):
    logFile = a.logFile[0]
else:
    logFile = a.logFile

layers = a.layers[0]

rootPath = a.rootPath[0]
fullPlot = rootPath + pathSep + a.d3plotName[0]

if isinstance(a.voxelRatioListStr,list):
    voxelRatioListStr = a.voxelRatioListStr[0]

if len(voxelRatioListStr.split('-')) == 1:
    voxelRatioList = [float(voxelRatioListStr)]
elif len(voxelRatioListStr.split('-')) == 2:
    voxelRatioListStr = voxelRatioListStr.split('-')
    voxelRatioList = []
    for                                  voxRat                                  in
range(int(float(voxelRatioListStr[0])*100),int(float(voxelRatioListStr[1])*100)+1)
:
        voxelRatioList.append(voxRat/100)
else:
    sys.exit("Invalid -v parameter {}, exiting...".format(voxelRatioListStr))

isBatch = a.isBatch
isSave3dPlot = a.isSave3dPlot
isIntSave3dPlot = a.isIntSave3dPlot
if not isBatch or isSave3dPlot or isIntSave3dPlot:
    import mayavi.mlab as mlab       # 3D interactive Plotting
showProgress = a.isShowProgress
isQuiet = not showProgress
isVerbose = a.isVerbose

elemDict = []
dictFileFullPath = elemPath + dictFileName
try:
    with open(dictFileFullPath, 'r') as file:
        for line in file:
            if not line.startswith('#'):
                elemDict.append(line.strip().split(','))
except (FileNotFoundError):
    print(f"*** Warning Missing {dictFileFullPath} dictionary file."\
          + f" Using whole Part {partList}.")
    layers = '@'     # Force whole part, regardless of command line
```

```
pltTitle = ""
for i in range(len(elemDict)):
    if elemDict[i][0] in layers:
        pltTitle = ' '.join([pltTitle,elemDict[i][1]]).strip()

if isVerbose:
    print(f"Analyzing the {pltTitle} from d3plot file: {fullPlot}")

################################################################################
#%% >Read last state of simulation (d3plot)
################################################################################

p,dr,numStates,numNodes,node_ids,node_pos,numParts,part_ids,partType, \
    partNumElements,fullMod_element_ids,fullMod_element_conn,d3p_element_stress, \
    d3p_element_1princ_stress,                          d3p_element_2princ_stress,
d3p_element_3princ_stress,\
    d3p_element_tresca_stress, d3p_element_vm_stress, d3p_element_signvm_stress,\
    d3p_element_strain,          d3p_element_eps,          d3p_element_vm_strain,
d3p_element_signvm_strain,\
    d3p_element_maxprinc_strain,                        d3p_element_2princ_strain,
d3p_element_minprinc_strain,\
    d3p_element_tresca_strain, d3p_element_plast_strain, d3p_element_hist_var,\
    d3p_element_inf_strain, d3p_element_green_strain, d3p_element_almansi_strain,\
    d3p_element_rate_strain, d3p_element_deletion \
    = initD3plot(fullPlot,partList,listParts,metricList,ihv,isVerbose=isVerbose)

################################################################################
# ---- Read History variable for the last state
################################################################################
tcyc = time.time()
p.ist = numStates-1
p.ihv = ihv
fullMod_HistVar = dr.get_data(d3p_element_hist_var,p)

################################################################################
#%% >Build universe of elements (UOE) subset defined in options (default whole part)
################################################################################
elList = []
if layers != '@':
    for i in range(len(layers)):
        elemFile = elemFilePrefix + layers[i] + elemFilePostfix
        fullElem = elemPath+elemFile

        f = open(fullElem)
        lines = f.readlines()
        f.close()
        elList = elList + [int(x) for x in lines]
else:
    elList = fullMod_element_ids
elList = sorted(elList)

################################################################################
#%% >Link UOE to their fracture outcome and connected nodes and isolate elements of
interest (EOI) (e.g., fractured elements)
################################################################################
element_ids = []
element_conn:  list[solid_vector] = []  # solid_vector lsreader class
histVars = []

fx_element_ids = []
fx_element_conn:  list[solid_vector] = []  # solid_vector lsreader class
fx_histVars = []

t1 = time.time()
idx = 0
fx_idx = 0

# Test if fullMod_element_ids array is sorted, if so, use fast binary search
if np.all(fullMod_element_ids[:-1] <= fullMod_element_ids[1:]):
    # is sorted, use binary search
    lookupFunc = lambda x,y : bisect_left(x, y)
```

```
else:
    lookupFunc = lambda x,y : np.where(x == y)[0][0]

# if lists are the same length, the elements may line up, in order for a very fast
search.
if len(elList) == len(fullMod_element_ids):
    tryFast = True
    progBarTtl = "isolating {:,d} layer elements (fast 1:1)".format(len(elList))
else:
    tryFast = False
    progBarTtl = "isolating {:,d} layer elements".format(len(elList))

#%%% Loop through element list, matching connections and outcomes
for i in tqdm(range(len(elList)),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):
    if tryFast:
        idxLookup = i
        if fullMod_element_ids[idxLookup] != elList[i]:
            tryFast = False
            idxLookup = lookupFunc(fullMod_element_ids,elList[i])
            progBarTtl = "isolating {:,d} layer elements: ".format(len(elList))
    else:
        try:
            idxLookup = lookupFunc(fullMod_element_ids,elList[i])
        except (IndexError):
            idxLookup == len(fullMod_element_ids)

    if idxLookup == len(fullMod_element_ids):
        if isVerbose:
            print(f"***  WARNING  ***  Element  {elList[i]}  missing  from  part(s)
{partList}")
    else:
        if fullMod_element_ids[idxLookup] == elList[i]:
            hVar = fullMod_HistVar[idxLookup]
            element_ids.append(fullMod_element_ids[idxLookup])
            element_conn.append(fullMod_element_conn[int(idxLookup)])
            histVars.append(hVar)
            idx += 1

            # ---- Generate list of fractured elements
            if hVar < 1:
                fx_element_ids.append(fullMod_element_ids[idxLookup])
                fx_element_conn.append(fullMod_element_conn[int(idxLookup)])
                fx_histVars.append(hVar)
                fx_idx += 1

if isVerbose:
    print(f"isolating time: {time.time()-t1:0.1f} seconds.")

if len(fx_histVars) == 0:
    if logFile != "":
        try:
            with open(logFile,"a") as f:
                print("fx_count = 0 fx_CST = 0 0 0 tot_fx_len = 0.0 mm   voxelRatio
= {} 0.0 {} {}"\
                      .format(voxelRatioList[0],layers,rootPath),file=f)
        except:
            print(f"open(logFile,\"a\") as f: case: {rootPath1}")

    sys.exit("No fractures detected, exiting...")

partNumElements = idx
fx_partNumElements = fx_idx


#############################################################################
#%% >Generate list of nodes that belong to EOI and list of node and element IDs
#############################################################################
fx_elNode = []
progBarTtl = "Identifying fractured element NODES"
for                                    i                                    in
tqdm(range(len(element_ids)),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):
```

```
    if histVars[i] != 1:
        fx_nodeList = []
        j = 0
        while element_conn[i].Node(j) != 0:
            try:
                fx_nodeList.index(element_conn[i].Node(j))
            except (ValueError, IndexError):
                fx_nodeList.append(element_conn[i].Node(j))
                fx_elNode.append([element_conn[i].Node(j),element_ids[i]])
            j += 1

fx_elNodeSrt = sorted(fx_elNode)
df_fx_elNode = pd.DataFrame(data=fx_elNodeSrt)
df_fx_elNode.columns = ["NodeID","ElemID"]

#############################################################################
#%% >Generate list of CONNECTED EOI by matching SHARED nodes
#############################################################################
if showProgress:
    progBarTtl = "Generating list of connected fractured elements"
    printProgressBar(0,  len(fx_elNodeSrt)-1,  prefix  =  progBarTtl,  suffix  =
'Complete', length = 50)
fx_connectedElems = []
i = 0
while i < len(fx_elNodeSrt)-1:
    df_list = df_fx_elNode[df_fx_elNode["NodeID"] == fx_elNodeSrt[i][0]]
    dfLen = len(df_list)
    if dfLen > 1:
        # elConnList = []
        for k in range(dfLen-1):
            for j in range(k+1,dfLen):

fx_connectedElems.append([df_list.iloc[k].ElemID,df_list.iloc[j].ElemID])
    i += dfLen
    if showProgress:
        printProgressBar(i,  len(fx_elNodeSrt)-1,  prefix  =  progBarTtl,  suffix  =
'Complete', length = 50)
if showProgress:
    print("")

fxCnt = 0
progBarTtl = "Counting fractured elements"

# Generate an array for the range of hv (max absolute value) to use
# hv as an index.
hvMax = int(max(abs(min(histVars)),abs(max(histVars))))+1
hvCnts = np.zeros([hvMax])
for i in tqdm(range(partNumElements),desc=progBarTtl,disable=isQuiet):
    hv = histVars[i]
    if hv != 1:
        fxCnt += 1
        hvCnts[int(abs(hv))] += 1

#        sys.exit("Fracture        Counts        S-39={},        C-57={},        T-
75={}".format(hvCnts[39],hvCnts[57],hvCnts[75]))

#############################################################################
#%% >Generate list of consolidated EOI centroid coordinates, estimated radius,
fracture outcome, and element IDs.
#############################################################################

fullarray \
 = calcCentroid(dr,p,partNumElements,histVars,element_ids,element_conn,
                node_ids,node_pos,partType,isVerbose=isVerbose)
df_fullarray = pd.DataFrame([vars(x) for x in fullarray])
df_fx = df_fullarray.loc[(df_fullarray["hv"] != 1)]



#%% Analyze one or more voxelRatios
processed_median = False
```

```
while not processed_median:
    voxRatArray = np.zeros([len(voxelRatioList),3])
    voxRatIdx = 0
    for voxelRatio in voxelRatioList:

##########################################################################
        #%%% >Generate voxel cubes -  low resolution (1X) and high resolution (2X)

##########################################################################
        # ---- Define model coordinate ranges


        xdim, ydim, zdim, xmin, xmax, ymin, ymax, zmin, zmax, voxdelDimAdj \
            = calcVoxelDims(df_fx, voxelRatio)

        precSave = np.get_printoptions()['precision']
        np.set_printoptions(precision=5,suppress=True)
        segCol = []

        C                                                                    =
voxelUniverse(xdim=xdim,ydim=ydim,zdim=zdim,llf=[xmin,ymin,zmin],urb=[xmax,ymax,zm
ax])
        C2                                                                   =
voxelUniverse(xdim=2*xdim,ydim=2*ydim,zdim=2*zdim,llf=[xmin,ymin,zmin],urb=[xmax,y
max,zmax])

        if not IS_VALIDATE:

##########################################################################
            #%% >Populate 1X voxel cubes with fractured elements

##########################################################################
            progBarTtl = "Set FX 1x Vox"
            for                                  i                                in
tqdm(range(len(df_fx)),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):

C.setvox(1,df_fx.iloc[i].x,df_fx.iloc[i].y,df_fx.iloc[i].z,elid=df_fx.iloc[i].elID
)

            progBarTtl = "Join FX Vox"
            lenElems = len(fx_connectedElems)
            for                                  i                                in
tqdm(range(lenElems),desc=progBarTtl,ncols=TQDM_COLS,disable=isQuiet):

C.joinElems(df_fx,fx_connectedElems[i][0],fx_connectedElems[i][1],1,useSize=False)

            # Uncomment for debugging
            # print(f"C.xx/yy/zz count = {len(C.xx)}")
            # C.plotvox(sf=2)
        else:

##########################################################################
            #%% >If validation analysis, overwrite voxel cubes with analytic data
(e.g., circles, lines, etc.)

##########################################################################
            UNIT_CONV = 1    # mm to mm
            C, suma = VALIDATE_setVox_Analytic (C)


##########################################################################
        #%% >Skeletonize 1X voxel cubes to contract wide fractures to single single-
element-wide

##########################################################################
        #
https://docs.w3cub.com/scikit_image/api/skimage.morphology.html#skimage.morphology
.skeletonize_3d
        skel3d = skimage.morphology.skeletonize_3d(C.voxels)
```

```
################################################################################
        #%% >Map skeletonized 1X voxel cubes into 2X voxel cubes and skeletonize 2X
voxel cubes

################################################################################
        xs2s, ys2s, zs2s =  map_1X_to_2X(C, C2, xdim, ydim, zdim)

        skel3d2x = skimage.morphology.skeletonize_3d(C2.voxels)

        # ---- Optional Validation plots for debugging
        if IS_VALIDATE and False:
            plot_C2_skel3d2x(C2,skel3d2x)


################################################################################
        #%% >Generate list of fractured 2x voxel cube metrics (including include
voxel ID, centroid, fractured neighbor list and counts)

################################################################################
        # ---- metrics include voxel ID, centroid, fractured neighbor list and
counts
        df_skel2xList, mc2s = build_2x_fxList(skelel2,xs2s,ys2s,zs2s)


################################################################################
        #%% >Recursive analysis

################################################################################

        if len(df_skel2xList) == 0:
            fxtot = fxCnt * voxelDimAdj/2 * 1000
            if logFile == "":
                print("fx_count = {} fx_CST = {} {} {} tot_fx_len = {:0,.1f} mm
voxelRatio = {} {:0.1f} {} {} Estimated".format(fxCnt,hvCnts[57],hvCnts[39],\
                                hvCnts[75],fxtot,voxelRatio,fxtot,layers,rootPath))
            else:
                # Output to logFile. Helpful for batch-mode.
                # USER EDIT for customized output
                with open(logFile,"a") as f:
                    print("fx_count = {} fx_CST = {} {} {} tot_fx_len = {:0,.1f} mm
voxelRatio = {} {:0.1f} {} {} Estimated".format(fxCnt,hvCnts[57],hvCnts[39],\
                        hvCnts[75],fxtot,voxelRatio,fxtot,layers,rootPath),file=f)
            sys.exit("Fracture Chains not detected {}, Estimated length {:0,.1f}
mm, exiting...".format(fxCnt,fxtot))

        voxSegHist = []
        listOfvoxIDchains = []
        listOfFxPathLen = []
        cntr = 0
        if showProgress:
            progBarTtl = "recursive fx skel 2x analysis:"
            availCntTotal = sum(df_skel2xList["availCnt"])
            printProgressBar(0,  availCntTotal-1,  prefix  =  progBarTtl,  suffix  =
'Complete', length = 50)

        #%%% >While list of fractured 2x voxel cubes have untraversed neighbors
        while len(df_skel2xList[df_skel2xList['availCnt']!=0]) > 1:
            #%%%% >Sort list by untraversed neighbor count (smallest-to-largest)
            df_skel2xList = df_skel2xList.sort_values(by=["availCnt"])
            availCnt1 =len (df_skel2xList[df_skel2xList["availCnt"] == 1])
            if availCnt1 > 0:
                #%%%%  >If  available,  choose  voxel  cubes  with  one  neighbor
(indicating end-of-chain)
                ilocStart = len(df_skel2xList[df_skel2xList["availCnt"] == 0])
                ilocEnd = ilocStart + availCnt1
            else:
                #%%%% >Otherwise choose voxel with the most neighbors (if > 2,
indicates a junction)
                # ---- Sort the list highest-to-lowest and select the largest (first
voxel)
```

```
                df_skel2xList                                                    =
df_skel2xList.sort_values(by=["availCnt"],ascending=False)
                ilocStart = 0
                ilocEnd     =     len(df_skel2xList[df_skel2xList["availCnt"]     ==
df_skel2xList.iloc[0].availCnt])
                ilocEnd = 1
            #%%%% >For each chosen voxel cube
            for iloc in range(ilocStart,ilocEnd):
                cntr += 1

##########################################################################
                # Recursively scan terminal element chains

##########################################################################
                #%%%%% >Start a new fracture trace
                fxPathLen = 0
                voxID = df_skel2xList.iloc[iloc].voxID
                voxIDchain =[voxID]
                #%%%%%  >Build  fracture  trace  by  recursively  traversing  next
untraversed neighbor until chain terminates at a junction or end of chain
                #--------------------------- recursion ----------------------------
-----
                df_skel2xList,fxPathLen,voxIDchain,voxSegHist                    =
recurTraceVox(df_skel2xList,voxID,fxPathLen,voxIDchain,voxSegHist)
                #--------------------------- recursion ----------------------------
-----
                listOfvoxIDchains.append(voxIDchain)
                listOfFxPathLen.append(fxPathLen)
                #if     iloc     ==     34     and     ilocEnd     ==     36     and
len(df_skel2xList[df_skel2xList['availCnt']!=0]) == 2:
                if cntr == 5000:

print("iloc=",iloc,"ilocEnd=",ilocEnd,"lenAvailCnt=",len(df_skel2xList[df_skel2xLi
st['availCnt']!=0]),'\n')
                if showProgress:
                    availCntCur = max(1,sum(df_skel2xList["availCnt"]))
                    printProgressBar(availCntTotal-availCntCur,    availCntTotal-1,
prefix = progBarTtl, suffix = 'Complete', length = 50)
        if showProgress:
            print("")

        lopl = deepcopy(listOfFxPathLen)
        fxtot = 0
        fxtraceCnt = len(lopl)
        for i in range(fxtraceCnt):
            if not SKIP_LT_2_mm or (SKIP_LT_2_mm and lopl[i] >= 2/UNIT_CONV):
                fxtot += lopl[i] * UNIT_CONV

        # voxRatArray constaints to define column indices
        FXTRACECNT = 0; FXTOT = 1; VOXELRATIO = 2
        voxRatArray[voxRatIdx,FXTRACECNT] = fxtraceCnt
        voxRatArray[voxRatIdx,FXTOT] = fxtot
        voxRatArray[voxRatIdx,VOXELRATIO] = voxelRatio
        voxRatIdx += 1
        if isVerbose > 0:
            print(f"voxelRatio = {voxelRatio}   fxtraceCnt = {fxtraceCnt}  fxtot =
{fxtot}")

        if isIntSave3dPlot:
            listOfFxPathLen,   firstList,   midList,   lastList,   voxIDfirstList,
voxIDlastList \
                =
populateFXtraceLabels(listOfvoxIDchains,df_skel2xList,lopl,SKIP_LT_2_mm,UNIT_CONV)
            pngName = rootPath + pathSep + "IntPlot_"+str(voxelRatio)+"vox.png"
            plot3dInteractive(xs2s,  ys2s,  zs2s,  mc2s,  2,  firstList,  midList,
lastList,
                              voxIDfirstList,        voxIDlastList,        fxtot,
listOfFxPathLen,
                              pltTitle, isIntSave3dPlot, pngName=pngName)
```

```
    # Sort voxel ratios by fracture trace length
    voxRatArray = voxRatArray[np.lexsort((voxRatArray[:,FXTRACECNT],))]

    # Determine index of median
    voxMedian = int(np.ceil(voxRatArray.shape[0]/2)) - 1
    voxSelect = voxMedian
    voxSelectArray = voxRatArray

    # sort voxel ratios first by frace trace counts (largest-to-smallest),
    # then by fracture trace lengths (shortest-to-longest).
    voxMinCntMaxLenArray      =      voxRatArray[np.lexsort((voxRatArray[:,FXTOT],-
voxRatArray[:,FXTRACECNT]))]
    # The voxel ratio with the least number of fracture trace segments
    # and the longest trace length is in the last row of the sorted array
    voxMinCntMaxLen = -1

    # Override the median voxel ratio selection.
    voxSelect = voxMinCntMaxLen
    voxSelectArray = voxMinCntMaxLenArray

    if voxSelectArray[voxSelect,2] == voxelRatio:
        processed_median = True
    else:
        voxelRatioList = [voxSelectArray[voxSelect,2]]

    if isVerbose > 0:
        print(f"voxRatArray = {voxRatArray}  voxMinCntMaxLen = {voxMinCntMaxLen}
voxMedian = {voxMedian}")
        print(f"voxelRatioList = {voxelRatioList}")

##########################################################################
#%% >Generate interactive fracture trace plot
##########################################################################
listOfFxPathLen, firstList, midList, lastList, voxIDfirstList, voxIDlastList \
    =
populateFXtraceLabels(listOfvoxIDchains,df_skel2xList,lopl,SKIP_LT_2_mm,UNIT_CONV)


# ---- debugging plot
if False:
    fig = plt.figure(2)
    ax = fig.add_subplot(111, projection='3d')
    sf = 2
    mlab.points3d(xs2s, ys2s, zs2s, scale_factor=sf, mode='cube',color=mc2s)   #
works, all white cubes
    ax.set_xlabel('X Label')
    ax.set_ylabel('Y Label')
    ax.set_zlabel('Z Label')
    mlab.show()

if showProgress:
    print(f"Analysis time: {time.time()-t0:0.1f} seconds.")

if logFile == "":
    print("fx_count = {} fx_CST = {} {} {} tot_fx_len = {:0,.1f} mm  voxelRatio =
{} {:0.1f} {} {}"\

.format(fxCnt,hvCnts[57],hvCnts[39],hvCnts[75],fxtot,voxelRatio,fxtot,layers,rootP
ath))
else:
    keepTrying = True
    kTcntr = 0
    while keepTrying and kTcntr < 1000:
        # retry to overcome file permission errors
        try:
            with open(logFile,"a") as f:
                print("fx_count = {} fx_CST = {} {} {} tot_fx_len = {:0,.1f} mm
voxelRatio = {} {:0.1f} {} {}"\

.format(fxCnt,hvCnts[57],hvCnts[39],hvCnts[75],fxtot,voxelRatio,fxtot,layers,rootP
ath),file=f)
```

```
        except PermissionError:
            kTcntr += 1
            time.sleep(1)
            continue
        keepTrying = False

if IS_VALIDATE:
    print(f"Analytic={suma:0.3f}                              Calculated={fxtot:0.3f}
Ratio={fxtot/suma:0.3f}")

if not isBatch or isSave3dPlot or isIntSave3dPlot:
    pngName = rootPath + pathSep + "PrimaryPlot_"+str(voxelRatio)+"vox.png"
    plot3dInteractive(xs2s, ys2s, zs2s, mc2s, 2, firstList, midList, lastList,
                      voxIDfirstList, voxIDlastList, fxtot, listOfFxPathLen,
                      pltTitle, isSave3dPlot,pngName=pngName)
```

List of Symbols, Abbreviations, and Acronyms

1×	low resolution
2×	high resolution
3D	three-dimensional
AFTM	Automated Fracture Trace Measuring
API	application programming interface
ARL	U.S. Army Combat Capabilities Development Command Army Research Laboratory
BHBT	behind-helmet blunt trauma
DoD	Department of Defense
EI	elements of interest
FE	finite element
FEM	finite element model
ID	identification
UE	universe of elements
v	version

1 (PDF)	DEFENSE TECHNICAL INFORMATION CTR DTIC OCA
1 (PDF)	DEVCOM ARL FCDD RLB CI TECH LIB