Graphical User Interface for Data Visualization and Analysis of Material Mechanics

Colin Kang

September 2020

Abstract

In recent years, nanoindentation techniques have emerged as critical methods for measuring material properties. However, the automated analysis of raw data is not centralized, leading to a difficulty in efficiently determining material mechanics due to the volatility of having to run different scripts. In this paper, the vectorization capabilities of python libraries are used to provide insight into the mechanical properties of materials through the development and optimization of the following data calculations: engineering and true stress-strain; Young's modulus using either slope or contact stiffness measurements with Sneddon's correction; energy dissipated; stress-strain at material failure; stress-strain burst behavior. The data calculation algorithms are shown to be computationally efficient for processing 26 separate data sets simultaneously each of at least 250,000 points since they linearly scale or have constant speed in addition to linearly scaling. The paper also outlines the development of an intuitive graphical user interface that visualizes the data calculations of individual or multiple raw data sets simultaneously through multiple plot functionality. The development of the graphical user interface improves upon the current efficacy in processing raw data sets to determine mechanical properties through the centralization of all its data calculation algorithms.

1 Introduction

In the realm of materials science, particularly in analyzing the mechanical properties of materials, scientific instruments produce raw data with thousands to millions of distinct points which measure a variety of parameters including load on sample (mN), displacement into surface (nm), time on sample (s), harmonic contact stiffness (N/m), and hardness (GPa). Often times, quantifying material properties require additional analysis of these measurements. For instance, both the instantaneous load and displacement measurements of a sample are dependent on material specifications such as the sample's height and cross-sectional area, whereas a stress and strain provide better insight as they are not geometry dependent properties. When we manufacture materials, we may add defects or impurities which change properties such as toughness or hardness. The shifts in structure can create emergent effects as seen with work on hierarchical architectures [5]. Yet to create these new material functionalities, we must not rely solely on theory but also generate concrete, insightful data including the strain size of burst events and variations of Young's modulus which can only be known through analysis of the raw data. Thus, we must be able to efficiently and effectively process raw data from scientific instruments into useful outputs in the form of data calculations and graphical visualization that fully encapsulate a material's mechanical properties.

Common techniques for measuring mechanical properties of materials include tensile testing, nanoindentation, in-situ nanoindentation [7], in-situ microcompression testing [8], and in-situ microtensile testing [6]. All of these techniques can be used to investigate fundamental properties of materials such as Young's modulus, which identifies the intrinsic linear-elastic stiffness of materials such as amorphous compounds of Si and Li in compatible atomic ratios as shown in Liontas and Greer [10]. Nanoindentation devices can measure the instantaneous stiffness of a material, which can be used in conjunction with the specimen's initial cross-sectional area and height to extrapolate the Young's modulus value. Some can even be used in a quasi-static way to decrease noise in the data for a more accurate analysis of the material [11]. These stiffness readings of a material can be useful when automated into many different calculations, such as determining where hardening occurs in the material through strain rate. In the hardening behavior of materials, flat-punch nanoindentation like those carried out on the G200 nanoindentor by Zhang et al. provides specific insight and can illuminate the effects of loading rate on mechanical response [12].

One emerging area of research where these techniques are increasingly critical is additive manufacturing on the nano-scale [9]. As the diversity of materials synthesized through this process increases, efficient mechanical analysis becomes critically important for identifying both desirable properties and flaws in newly synthesized materials. Close analysis of these nano-architected materials with techniques such as InSEM enable unique mechanical testing conditions which provides insight into mechanical mechanisms and material physics through data analysis [4].

This project enables highly versatile calculations of the raw data which provides a tool for quantification of the mechanical properties of materials. Thus, we transformed traditional processing methods into an accurate, fast, and intuitive software using algorithmic components of shift, rolls, and vectorization in python. The objective of the graphical user interface was to allow for optimal efficiency in calculating processes and values such as energy dissipation, engineering and true stress-strain, stiffness and Young's modulus via both a slope and contact stiffness measurement (CSM) method, Sneddon's correction, parsing cyclic data, burst behavior, failure points, saving calculated information into exportable SVGs, and singular and multiple graphical output formats. With the creation of the Materials Data Visualization and Calculations Graphical User Interface (MDVaC GUI), we become able to effortlessly investigate the mechanical behavior of materials.

2 Methods

2.1 Language, Library Versions, and Formal Definitions

For the creation of the Materials Data Visualization and Calculations Graphical User Interface, the language used was Python 3.8.3. The project was split into two bulk sections: the mechanics data calculations and the graphical visualization output. For the creation of the algorithms outlined in section 3.1, the primary Python libraries used were pandas 1.1.0, numpy 1.19.1, and scipy 1.5.2. For the creation of the graphical visualization tools outlined in section 3.2, the main Python libraries used were matplotlib 3.3.1, tkinter 3.8.3, garbage collector (gc), and miscellaneous operating system interfaces (oc). A comprehensive list including more trivial Python libraries can be found in Appendix A. The integrated development environments (IDEs) used were Jupyter Notebook later translated to PyCharm for accessibility in creating and implementing the actual graphical user interface.

The pandas library's data table classes were used as the foundation for the calculations to be vectorized and thus optimized without long iteration. The term vectorization refers to simplifying an algorithm from N iterations to N/M iterations where M is the amount of data elements manipulated simultaneously. The vectorization capabilities of the pandas' data table class made it the best choice for dealing with data ranging from 50,000 individual data points to 5,000,000 individual data points. Also, the pandas library allowed for easy transfer of the initial raw data from the csv files into manipulable tabular data.

For each subsequent algorithm implemented for automation of data calculations (Young's modulus, Sneddon's correction, burst events, etc...), the algorithms were made in Jupyter notebooks to be individual modules so that testing of the algorithms with real data could be categorized. Each algorithm was built in such a way for smooth transition into the larger framework of the graphical user interface in the PyCharm IDE.

2.2 General Code Structure of the Graphical User Interface

The structure of the graphical user interface is distributed among three classes: the main application class, the csv interface class, and the data calculation and single graphing software class.

The main application class initializes the parameters of the tkinter frame and instantiates the csv interface class. It is this class that allows the graphical user interface to be executed in so few commands.

The csv interface class deals with the raw data in the form of csv files and organizes the raw data to be manipulated in the data calculation and single graphing software class. The csv interface, which we go into depth in section 3.4, is shown in Figure 7. In the csv interface class lies the ability to simultaneously upload up to 26 alphabetized data files of arbitrary size and also parse such data files if the data is cyclic. The class is further broken down to deal with cyclic data in the form of the load controlled, displacement controlled, and arbitrary load or displacement peaks. We define load controlled cyclic data to be data where we apply an increasing normal load up until a maximum non-failing load before unloading and repeating. Displacement controlled cyclic data is similarly defined to load controlled cyclic data except we increase the nanoindentation depth rather than the load. Lastly, we define arbitrary load or displacement peaks cyclic data to be data where each load-displacement curve reaches a different maximum and minimum value per cycle. The main functionality of the class comes in organizing all the data into a cohesive manner to easily reference all files while doing data calculations form the raw data. The class also has a section to access the data calculation and single graphing software class to update the interface for the ability to plot multiple graphs on a single figure before exporting the figure.

The data calculation and single graphing software class deals with each individually uploaded raw data file. It is broken down into sections for each algorithm: engineering stress and strain, true stress and strain, Young's modulus (slope method), Young's modulus (contact stiffness measurement), energy dissipated, stress and strain at ultimate failure of the material, and burst events. Each calculation is stored as an additional column of the data table for the particular data file or outputted in a visual manner in the graphical user interface. Corresponding graphs can be plotted of both the raw data (such as load-displacement curves) and calculated data (such as stress-strain curves) and bounded by domain and range. The data calculation and single graphing interface is shown in Figure 1, which I will go into depth on for each involved algorithm.

2.3 Data Types and Structures Compatible with the Software

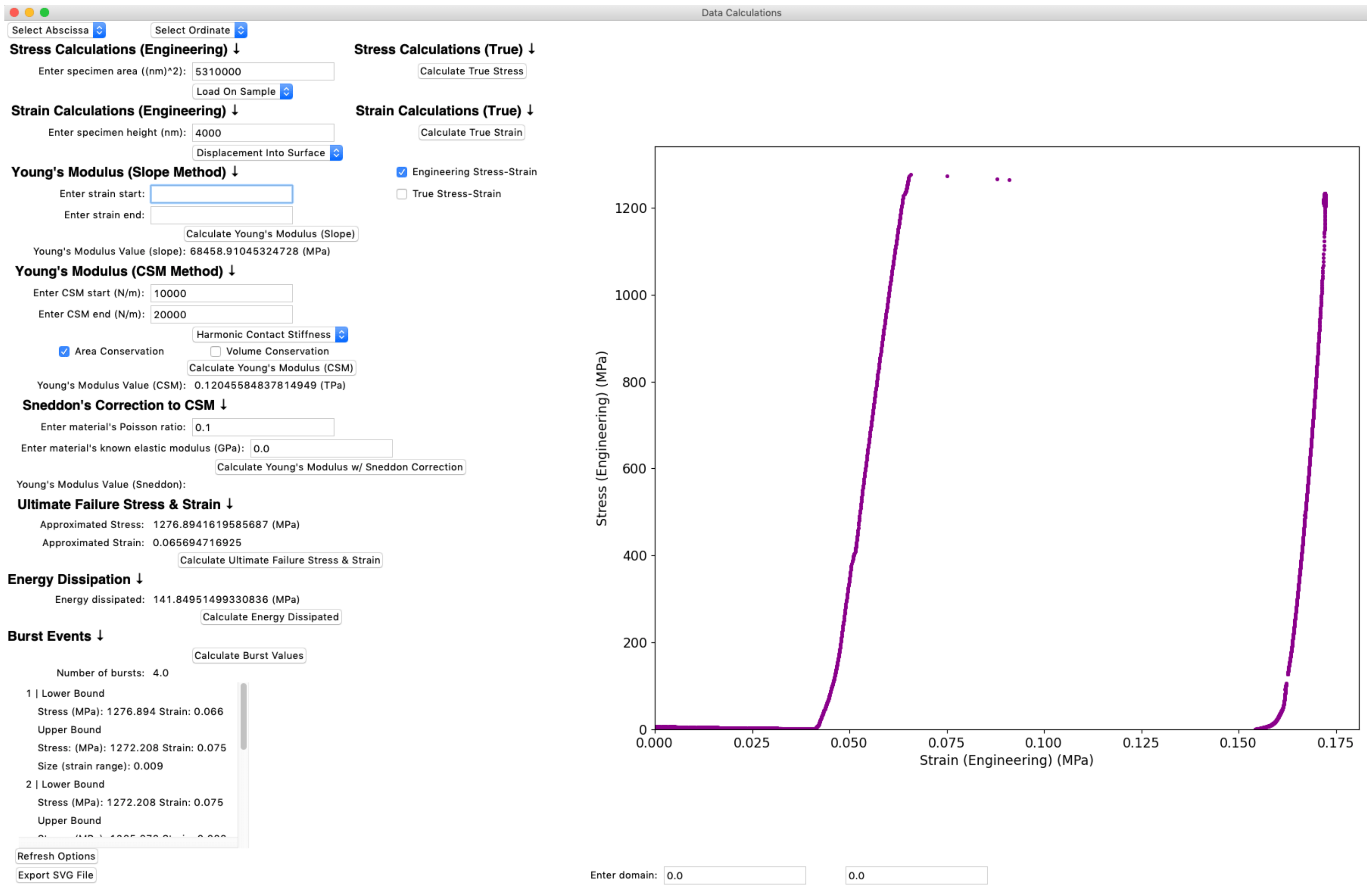
For processing of all algorithms and data visualization capabilities of the graphical user interface, real experimental data was used as input. The experimental data outlined in this paper used samples tested in an in-situ nanoindentor (SEMentor) [2]. The particular samples analyzed are cylindrical zinc oxide (ZnO) structures which have been labeled as ZBE24.01 samples 101, 102, 105 and ZBE24.02 samples 114 and 115. The respective diameters of the ZBE24.01 samples were 2.6 μm, 2.5 μm, and 2.3 μm. The respective diameters of the ZBE24.02 samples were 4.344 μm and 1.4 μm. For all 5 ZnO samples, the height of each sample was 4 μm. The program assumes the following units for data categories within the raw csv file: displacement into surface (nm), dynamic damping (N*s/m), dynamic displacement (nm), dynamic frequency (Hz), dynamic phase (degrees), extension (nm), force (mN), load on sample (mN), spring stiffness (N/m) or harmonic contact stiffness (N/m), time (s), dynamic stiffness (N/m), hardness (GPa), and modulus (GPa). The graphical user interface optimally processes csv files of the following format for each column: row 1 (0th index): data category; row 2 (1st index): unit; row 3-∞ (2nd index): numerical data (float64 or float128). Each raw csv data file inputted should also not have any extra rows or columns. That is, if there are n unique data categories, the data file should have no more than n columns or an overflow error would occur. Similarly, if there are n distinct data points for a particular category, the data file should have no more than $n+1$ rows. We permit $n+1$ rows because the category name counts as a data point. If the raw csv data file has extra rows or columns, the data file can be manipulated by uploading it into a pandas DataTable, removing excess rows or columns by changing the dimensions of the DataTable, and then re-exporting it as a csv file. Before processing the csv files into the data calculation algorithms, the user can plot raw data individually in the graphical user interface as seen in Figure 2.

3 Results and Discussion

3.1 Data Calculation Algorithms

3.1.1 Engineering Stress and Strain

The algorithm that calculates the engineering stress of the material is found in lines 1-31 of Appendix B. Using the pandas DataTable generated from the raw csv file, the algorithm creates an additional column through vectorization. This particular vectorization is from the load column that is measured in mN from the nanoindentation instrument divided by the specimen's cross-sectional area as given by user input and then subsequently multiplied by a factor of 10^{15}. All calculations occur simultaneously rather than in iteration, grouped by each individual data point from the load data. The reason for multiplying by the 10^{15} factor is to simplify the engineering stress values into the basic Pascal unit. From all of the calculated engineering


Select Abscissa

Select Ordinate

Stress Calculations (Engineering) ↓

Enter specimen area ((nm)^2): 5310000

Load On Sample

Strain Calculations (Engineering) ↓

Enter specimen height (nm): 4000

Displacement Into Surface

Young's Modulus (Slope Method) ↓

Enter strain start:

Enter strain end:

Calculate Young's Modulus (Slope)

Young's Modulus Value (slope): 68458.91045324728 (MPa)

Young's Modulus (CSM Method) ↓

Enter CSM start (N/m): 10000

Enter CSM end (N/m): 20000

Harmonic Contact Stiffness

☑ Area Conservation ☐ Volume Conservation

Calculate Young's Modulus (CSM)

Young's Modulus Value (CSM): 0.12045584837814949 (TPa)

Sneddon's Correction to CSM ↓

Enter material's Poisson ratio: 0.1

Enter material's known elastic modulus (GPa): 0.0

Calculate Young's Modulus w/ Sneddon Correction

Young's Modulus Value (Sneddon):

Ultimate Failure Stress & Strain ↓

Approximated Stress: 1276.8941619585687 (MPa)

Approximated Strain: 0.065694716925

Calculate Ultimate Failure Stress & Strain

Energy Dissipation ↓

Energy dissipated: 141.84951499330836 (MPa)

Calculate Energy Dissipated

Burst Events ↓

Calculate Burst Values

Number of bursts: 4.0

1 | Lower Bound

Stress (MPa): 1276.894 Strain: 0.066

Upper Bound

Stress: (MPa): 1272.208 Strain: 0.075

Size (strain range): 0.009

2 | Lower Bound

Stress (MPa): 1272.208 Strain: 0.075

Upper Bound

Stress Calculations (True) ↓

Calculate True Stress

Strain Calculations (True) ↓

Calculate True Strain

☑ Engineering Stress-Strain

☐ True Stress-Strain

Enter domain: 0.0 0.0

Enter range: 0.0 0.0

Figure 1: Data Calculation Sheet Demonstration with ZnO Sample 101

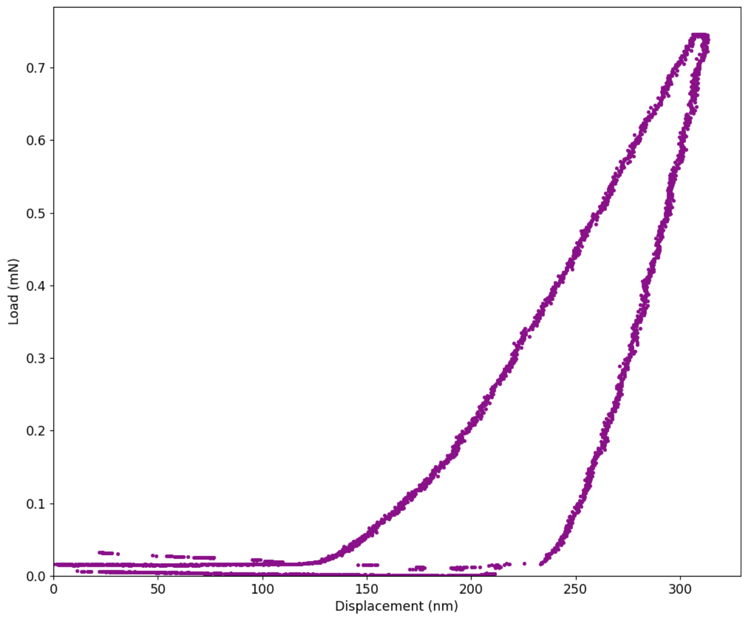


Figure 2: Sample Singular Load Displacement Curve from Raw CSV File

stress values, the average is taken and then logarithmically manipulated to determine the optimal prefix for the data that is outputted through the graphical user interface. The potential units range from Pa to PPa, although most values from experimental data that we observed in the ZnO samples fall in the MPa to GPa range. However, had we looked at polymers, the values would have been in the kPa range. The calculated unit value is then updated in the universal array that stores the units for each of the imported csv files. We will define the efficiency for this algorithm and the ones to follow as the wall-to-wall time it takes for each function to run using python's time import function time(). Any calculation under a tenth of a second is computationally inexpensive with the vectorization of pandas. The three experimental samples that were used to measure the efficiency of the algorithms were ZBE24.01 101, 105, and 102. The data for sample 101 has dimensions 41,162 rows by 6 columns so it has 246,972 distinct data points. The data for sample 105 has dimensions 45551 rows by 6 columns so it has 273,306 distinct data points. The data for sample 102 has dimensions 54,222 rows by 6 columns so it has 325,332 distinct data points. For the calculation of all instantaneous engineering stress data points in samples 101, 105, and 102 the wall-to-wall times of completion were 0.002481 seconds, 0.003041 seconds, and 0.003171 seconds respectively so the engineering stress algorithm is sufficiently fast for processing large quantities of data.

The algorithm that calculates the engineering strain of the material is found in lines 33-38 of Appendix B. Similarly to the engineering stress algorithm, the algorithm creates an additional column for the pandas DataTable. The engineering strain value is calculated from a vectorization of the displacement column that is measured in nm divided by the specimen's initial height as given by user input. Since the engineering strain values are a ratio of the displacements over the initial height, there was no need for an automation of the unit as the unit stored is none. The same three experimental samples of ZBE24.01 were used to measure the efficiency of the engineering strain calculation. For the calculation of all instantaneous engineering strain data points in samples 101, 105, and 102 the wall-to-wall times of completion were 0.001119 seconds, 0.001251 seconds, and 0.001404 seconds respectively. The execution times for the engineering strain values relative to the engineering stress values were approximately 2-3 times faster. Thus, the algorithm is computationally inexpensive and easily scales for much larger data sets. In Figure 3, we see the graphical output of the engineering stress-strain curve in the graphical user interface for the ZnO sample 101.

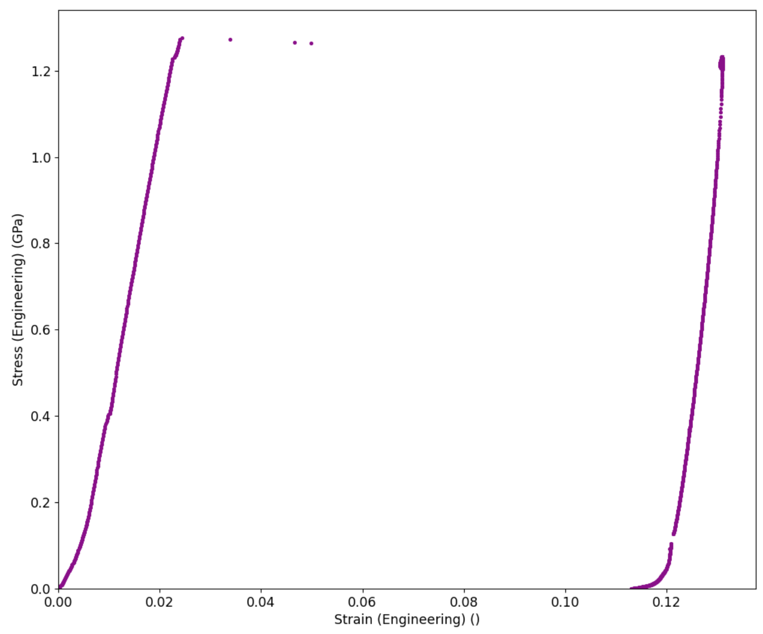


Figure 3: Engineering Stress-Strain Curve for ZBE24.01 Sample 101

3.1.2 True Stress and Strain

The algorithm that calculates the true stress of the material is found in lines 1-4 of Appendix C. The algorithm creates an additional column to the pandas DataTable, and each datapoint in that column are the vectorization of the previously calculated engineering stress multiplied by an extension ratio. The extension ratio is given to be the following:

$$\lambda = \frac{instantaneous\ specimen\ length\ (nm)}{initial\ specimen\ height\ (nm)} \tag{1}$$

where

$$instantaneous\ specimen\ length\ (nm) =$$
$$initial\ specimen\ height\ (nm) + instantaneous\ displacement\ (nm)$$

The unit stored in the universal storage array for the vectorized values is the same as the unit calculated in the engineering stress algorithm. For the calculation of all instantaneous true stress data points in samples 101, 105, and 102 the wall-to-wall times of completion were 0.001275 seconds, 0.001437 seconds, and 0.001685 seconds respectively so the algorithm is computationally fast for these large data sets.

The algorithm that calculates the true strain of the material is found in lines 6-9 of Appendix C. Similarly to the engineering stress algorithm, the algorithm creates an additional column for the pandas DataTable. The true strain value is calculated as the the following: $\ln(\lambda)$ along the vectorization of each individual displacement (nm) value. As the values take the same unit as the engineering strain calculations, the universal array stores the unit as none for this column. For the calculation of all instantaneous true strain data points in samples 101, 105, and 102 the wall-to-wall times of completion were 0.001563 seconds, 0.001792 seconds, and 0.002384 seconds respectively. The execution times for the true strain values relative to the true stress values were oscillating at approximately 1 as the ratio. Thus, the algorithm has high processing speeds because it has the same execution time as the true stress algorithm. In Figure 4, we see the graphical output of the true stress-strain curve in the graphical user interface for the ZnO sample 101.

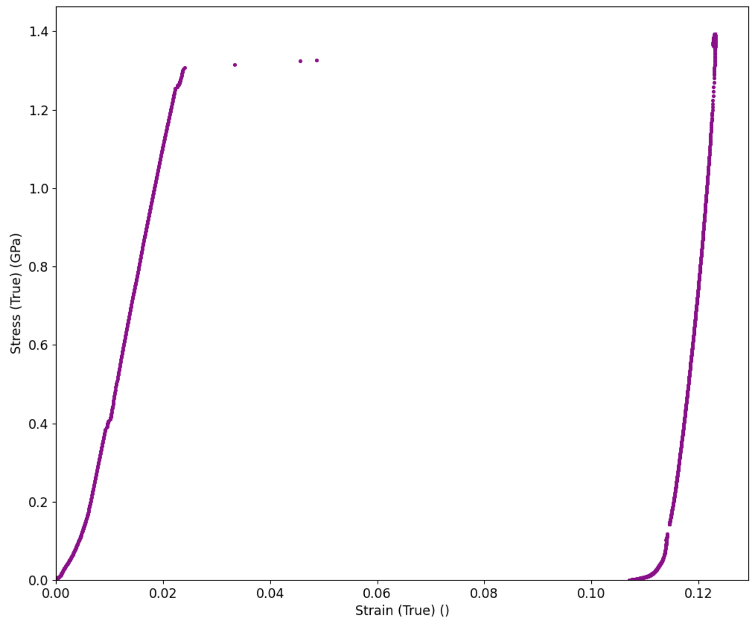


Figure 4: True Stress-Strain Curve for ZBE24.01 Sample 101

3.1.3 Young's Modulus (Slope)

The algorithm for calculating the Young's modulus of the material using the slope method is found in Appendix D. It is bounded by the user-input strain start value and end value. We use the strain range rather than the stress range because the strain has less varying noise throughout the entire load-displacement cycle. This is because we looked at displacement controlled experiments. Had we looked at load controlled experiments, we would have used the stress range over the strain range because the measured variable would have been the displacement. From the strain range input, we use lambda functions to precisely store each strain value. It follows that the Young's modulus value E is given by this equation:

$$E = \frac{stress\ @\ strain\ end\ -\ stress\ @\ strain\ start}{strain\ end\ -\ strain\ start} \tag{2}$$

For the calculation of each individual Young's modulus using this method in samples 101, 105, and 102 the wall-to-wall times of completion were 0.03393 seconds, 0.03779 seconds, and 0.04548 seconds. The correlation between these times shows that each additional 100,000 distinct data points results in 0.01 seconds longer execution time. The algorithm is optimized since the strain values are pinpointed in a vectorized manner using min().

3.1.4 Young's Modulus (Contact Stiffness Measurement)

The algorithm that calculates the Young's modulus of the material using the contact stiffness measurement is found in Appendix E, and works with either two experimental assumptions: volume conservation or area conservation. We define the relationship between the contact stiffness measurement and the Young's modulus to be

$$k = \frac{AE}{L} \tag{3}$$

In the case of volume conservation

$$\frac{L}{L_o} = \frac{A}{A_o} \tag{4}$$

so we get a Young's modulus expression

$$E = \frac{kL^2}{A_o L_o} \tag{5}$$

where L is the instantaneous length of the specimen, L_o is the initial specimen height, A is the instantaneous cross-sectional area of the specimen, and A_o is the initial cross-sectional area of the specimen. Both calculations use the vectorized columns and are multiplied by 10^9 so that the calculated Young's modulus value is initial in raw Pascals, so a similar logarithmic manipulation can occur for automation of the optimal unit assignment. The wall-to-wall execution times in calculating the Young's modulus using this CSM method for samples 101, 105, and 102 were 0.01631 seconds, 0.01881 seconds, and 0.03210 seconds respectively. The algorithm is vectorized to calculate each instantaneous Young's modulus using the CSM data column, but must be computationally longer due to iterating whether data points fit within the range, as well as calculating the mean of the values to output a single value.

3.1.5 Sneddon's Correction to Young's Modulus with Contact Stiffness Measurement

The algorithm for applying Sneddon's correction to the contact stiffness measurement of Young's modulus is found in Appendix F. The version of Sneddon's correction to Young's Modulus uses either volume conservation or area conservation depending on the contact stiffness measurement. We first calculate the material's compliance in order to remove the stiffness associated with the material acting as a flat-punch indenter to the base film [1]. In both cases, the compliance C is generally calculated as the following using the user-input poisson value v and literature Young's modulus value E of the material being tested using the graphical user interface:

$$C = \frac{\sqrt{\pi}(1 - v^2)}{2E\sqrt{A}} \tag{6}$$

Specifically, the volume conservation method uses the relationship outlined in equation 4 to calculate the instantaneous cross-sectional area, whereas the area conservation simply uses the initial cross-sectional area of the specimen. After calculating the compliance, the algorithm vectorizes all of the original CSM values to make new stiffness values including Sneddon's correction:

$$k_{Sneddon} = \frac{1}{\frac{1}{CSM} - C} \tag{7}$$

The algorithm then solves for the Young's modulus value using the same method as the Young's Modulus (Contact Stiffness Measurement) but with the new stiffness values. For this Sneddon's correction algorithm, the computational speed for samples 101, 105, and 102 are 0.0229 seconds, 0.03185 seconds, and 0.05609 seconds respectively. Since the core of the algorithm uses the Young's modulus contact stiffness measurement, the computational speed is similar but is approximately 1.69 times slower due to the calculation for compliance.

3.1.6 Energy Dissipated

The algorithm for calculating the energy dissipated by the material along it's load and unload sequence is found in Appendix G. We defined the energy dissipation to be the area under the stress-strain curve calculated by the graphical user interface. As seen in the function, there are two types of stress-strain curves available to use for the calculation: engineering stress-strain versus true stress-strain. This is indicated by $self.stress\_type$ and $self.strain\_type$. The algorithm simply uses $trapz()$ from the numpy library, which is the trapezoidal approximation of the integral. Since our data typically has at least 50,000 points, this approximation is very precise since our Δx, the strain differential, is extremely small. The wall-to-wall execution times in calculating the energy dissipated from the material for samples 101, 105, and 102 were 0.0004251, 0.0004301, and 0.0008050 seconds respectively. Since the algorithm only calls on $trapz()$, which is already vectorized, the processing speed we see is fast enough for increasingly large data sets.

3.1.7 Stress and Strain at Ultimate Failure

The algorithms for precisely determining the material's stress and strain at failure (e.g. fracture) are found in Appendix H. Both functions in Appendix H are essentially the same, with the only difference being the stress or strain assignment, so we can just explain the logic of one. Firstly, when a material fails, the nanoindentation device will still determine and output the instantaneous load and displacement occurring on the specimen, so we need to determine what values correspond to the actual material and the material past failure. Thus, the basic principle the algorithm is built on is the differential between strain values in each adjacent data point. In $n-1$ strain differences for n data points, the differences between adjacent data points are relatively small, except the drastically larger differences in the region of material failure. As a result, the goal is to find the two adjacent data points where the largest strain difference occur, and pinpoint the index of the left side data point between the two. We first imported and assigned the column of calculated strain values (either engineering or true) to an arbitrary array z. We then created a function called $shift(a)$ that creates two versions of z: one with values shifted left an index, the other with values shifted right an index. The function then creates a 2-D array with the two shifted versions of z. After, we found the difference between the 2-D array of shifted indices and a 2-D array of the original array and stored it as a new array *diff*. Lastly, we searched for the largest values in *diff* and outputted the corresponding indices, subtracting by 2 to get the leftmost values. Then, we stored the set of indices into the general storage for each individual csv file to be accessed in the graphical user interface. The wall-to-wall execution times for the algorithm determining failure stress in samples 101, 105, and 102 were 0.002348 seconds, 0.002470 seconds, and 0.003016 seconds respectively. Furthermore, the wall-to-wall execution times for the algorithm determining failure strain in samples 101, 105, and 102 were 0.001125 seconds, 0.001384 seconds, and 0.001715 seconds respectively. In our definition of an algorithm being computationally inexpensive, both algorithms are optimized because they only search through one list (column) of data without redundancy since each value is only iterated only once before finding the stress and strain at failure in the material.

3.1.8 Burst Events

The algorithm for determining the behavior of burst events in the material is found in Appendix I. First, let us define a burst event. Bursts in a material are phenomena where the material experiences a jolting change in both stress and strain due to many potential factors including specimen orientation. However, the material maintains shape and does not fail during or right after the burst event occurs. The primary way to differentiate between a burst event and failure in the material is the slope of the stress-strain curve following after either event occurs. In a burst event, after there is a noticeable difference in strain, the stress-strain curve retains the same upward slope from before the burst event occurs. Also, the larger change in strain is mitigated relative to a failure event in the material. In material failure, the slope of the stress-strain curve is drastically different following the failure event than before the failure event. Also, after testing with sufficient raw data, it became obvious that the difference in strain between two data points during a burst event is noticeably smaller than if material failure had otherwise occurred. For this algorithm, we first created an array *all_diff* that is composed of differences between adjacent calculated strain data points. After, we found a logarithmic factor where when subsequently subtracted by 3 to shift the size and make in an exponential factor. The logarithmic factor is the exponent for a ten that was then multiplied by 2. In order to come up with this value *mean_changer* as the comparison for valid burst events, we ran through 10 samples of burst data with manually indicated bursts and fine-tuned *mean_changer* until it was precisely accurate at reading the size of burst events to check for validity. That is, if two data values in *all_diff* were greater than *mean_changer* times the mean of *all_diff*, the strain difference was large enough to be considered a burst event. In lines 11-34 of Appendix I, the algorithm simply stores the stress and strain values for the two data points comprising the burst event, the size of the burst event in terms of strain difference, and the overall number of bursts in the data that is later outputted in the graphical user interface. The computational speeds of the samples 101, 105, and 102 in the burst algorithm were 0.002708 seconds, 0.002913 seconds, and 0.003050 seconds respectively. The computational speed for this algorithm is within the same decile as the stress and strain at ultimate failure algorithm, so it is sufficiently fast at identifying burst behavior for processing increasingly large data sets.

3.2 Insights in Overall Algorithmic Efficiency

The three experimental samples that were used to measure the efficiency of the algorithms (ZBE24.01 101, 105, and 102) were graphically analyzed to find a relationship in the scaleability of each algorithm relative to other samples (in terms of data size). As iterated before, sample 101 has 246,972 distinct data points, sample 105 has 273,306 distinct data points, and sample 102 has 325,332 distinct data points respectively. In Figure 5, the execution times for engineering stress and strain, true stress and strain, failure stress and strain, energy dissipated, and burst event algorithms all take less than two-hundredths of a second, with the slight variations in time mainly coming from noise in the data. Another noticeable trend in the aforementioned algorithms is the linear relationship in time relative to the respectively increasing size of the data file. Even though the algorithms are vectorized, the larger data file size requires more computing power, hence the ever slightly increase in execution time. Moreover, as seen in Figure 5, the times of completion for the two Young's Modulus algorithms and Sneddon's correction algorithm are a single magnitude longer relative to the other algorithms. This is due to the areas of the algorithms that are not fully vectorized. That is, for the slope method, the algorithm iterates over values to find the values that most closely match the user-input strain start and end before exiting iteration because it cannot simultaneously look for matching values. In

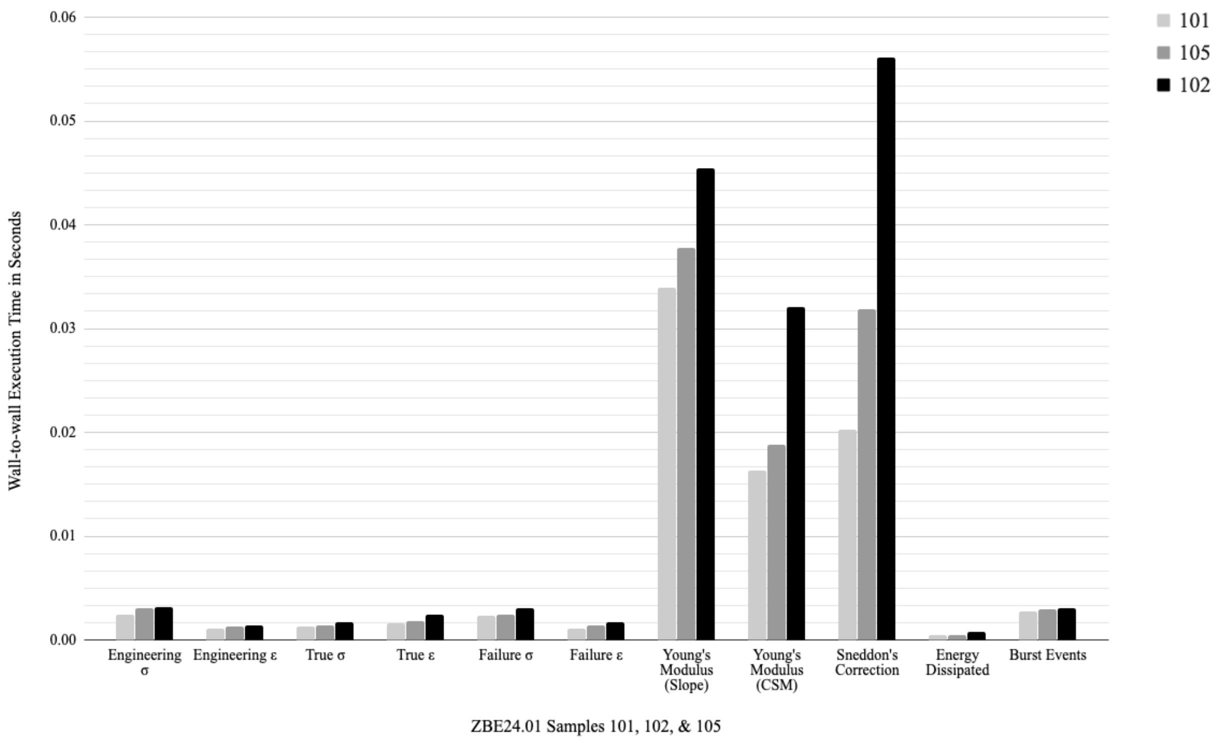


Figure 5: Execution Times of Data Calculation Algorithms in ZBE24.01 Samples 101, 102, 105

the CSM method, the algorithm individually checks whether each CSM data points falls within the CSM range, but then vectorizes the Young's modulus value at each instant. In order to output a single value, we found the mean of all Young's modulus values within the range, which adds to the computation speed given the size of the data. The Sneddon's correction algorithm follows the same principle as the CSM method, but is computationally longer due to the brute calculation of the compliance that is only partially vectorized. There is also a discrepancy between the slope method algorithm and the CSM or Sneddon's correction ones. For the slope method algorithm, the computational speed from sample 101 to sample 105 is 1.862x slower, but only 1.011x slower from sample 105 to sample 102. As the size of the data increases, the algorithms speed decreases at smaller intervals. However, the CSM method and Sneddon's correction algorithms see a different outcome. For the CSM method, the computational speed for sample 101 is 1.042x slower than sample 105, and 1.434x slower from sample 105 to 102. Similarly, for the Sneddon's correction algorithm, the computational speed from sample 101 to sample 105 is 1.418x slower, and 1.479 slower from sample 105 to sample 102. Thus, as the size of the data increases, these two algorithms become, on average, approximately 1.210x slower for each additional 40,000 distinct data points.

3.3 Parsing Cyclic Data

3.3.1 Load and Displacement Controlled

The algorithm for parsing load controlled cyclic data is found in Appendix J. The algorithms for both the load controlled and displacement controlled parsing are the same except displacement is substituted for load in the latter. Hence, we will only analyze the load controlled algorithm for parsing cyclic data. In a load controlled sample, both the load and unload have similar displacement and identical load readings on the way to max load and back down. Since the load reading will typically start at 1 mN, we found the maximum load controlled value (the load at the peak of the load-displacement curve) and multiplied it by 2 then subtracted by 1 to get the exact number of rows in a single load-unload cycle. After, we grouped the data by the number of rows for each cycle. For example, if there are 110 data points (for the sake of simplicity), and a cycle consists of 11 data points, the grouping would result in 10 distinct groups of data. Each group of data is a newly created individual pandas DataTable that essentially acts as its own csv raw data file. Following the grouping of data into n distinct DataTables, we created and stored n raw data files in the graphical interface with the same data calculation and graphical visualization capabilities as a single imported raw data file. Lines 14-62 of Appendix J shows the code involved in executing and storing all of the distinct csv data files created from the parsing of the single large, cyclic data file.

3.3.2 Arbitrary Load and Displacement Peaks and Minima

The algorithm for parsing cyclic data with arbitrary displacement peaks and minima is found in Appendix K. We see an example of such cyclic data in Figure 6. The algorithms for both the arbitrary load and displacement peaks and minima are the same with the substitute of load in the displacement algorithm for the former. Thus, we will only analyze the arbitrary displacement peaks and minima algorithm for parsing cyclic data. In random cyclic data, the size of each group differs significantly because the peak and minima of the displacement values are different for each group. However, in analyzing the random cyclic data, we observed that while although each peak is completely unpredictable, adjacent cycles share the same minima value for the right side of the former cycle and the left side of the latter cycle. We built on this observation by finding the shared local minima between adjacent cycles in order to differentiate the cycles into groups. The minima found are from the left side of each cycle, so we took the *cumsum*() of the array of minima to get all values between adjacent cycles' minima and then created the groups from these. This enables the raw cyclic data to be parsed in such a way that regardless of the size of each cycle, the minima pinpoint where the cycle begins and starts without having to look at the peak of the displacement. After the grouping is complete, lines 15-63 of Appendix K function similarly to lines 14-62 of Appendix J in terms of storing each new group as its own data file to be processed through the graphical user interface.

3.4 Multiple Plot Functionality in the Graphical User Interface

The motivation for this particular algorithm was to simultaneously view the graphical results of multiple imported data files. For any particular sample, one may want to determine how annealing or prestraining affects the behavior of the specimen [3]. Thus, one could upload many raw csv data files of a single type of material, with variations being the specimen's initial cross-sectional area, geometry, and height. Thus, the calculated engineering or true stress-strain curves, or the Young's modulus values for instance can be viewed on a single graph to differentiate elastic properties given such parameterizations of a material. The algorithm enabling the ability to plot multiple figures on a single graph is found in Appendix L. The algorithm builds on the universal storage throughout the graphical user interface in the forms of dictionaries. There is a new dictionary created for each new csv file, where all of the data and subsequent calculations are stored in a categorical manner. The multiple plot function accesses all dictionaries and checks whether the user's abscissa and ordinate selections match for each dictionary. If and only if the abscissa and ordinate selections match, the algorithm will iterate a new scatter plot consisting of all data points for each csv data file onto the single graph. The results of this function are outlined in Figure 7, where we see distinct plots on the

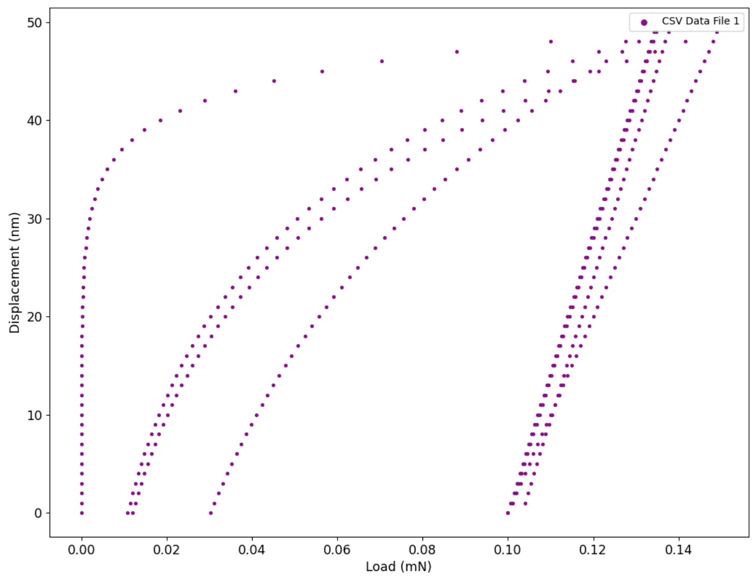


Figure 6: Sample Cyclic Data with Random Minima and Maxima

single graph corresponding to manipulations of a single specimen type.

4 Conclusion

In this project, we created the Materials Data Visualization and Calculation Graphical User Interface (MDVaC GUI) by utilizing the computational power of python pandas and numpy to execute theoretical calculations necessary for efficient analysis of the mechanics of materials. This analysis enables further understanding of the underlying mechanicsms and principle behaviors of the material. We effectively address the problem of manual extrapolation from generated raw data by using dictionaries and DataTables to vectorize the developed data calculation algorithms in an additive manner where each calculated data point is uniquely stored and assigned to an updated data file. Using the ZnO samples, we demonstrated the computational efficiency of all 11 algorithms, and the easy scaleability of all algorithms regardless of a sufficient increase in the number of processed data points since the Young's modulus (slope and CSM) algorithms and Sneddon's correction algorithms have a constant initial processing speed. Thus, the MDVaC GUI has the potential to significantly improve upon the efficacy of processing raw data sets into usable insights towards a material's mechanical properties.

5 Acknowledgements

I would like to thank Rebecca Gallivan, who was my mentor for this project. She taught me how to be relentlessly resourceful, ask difficult questions, and shape my approaches to the many obstacles I faced. I would also like to thank Dr. Julia Greer and the Greer Group at Caltech for giving me the opportunity to work with Rebecca after a cold email I sent out in January of 2020. I am grateful to have attended meetings, presentations, and conferences which introduced me to the beautiful world of materials science.

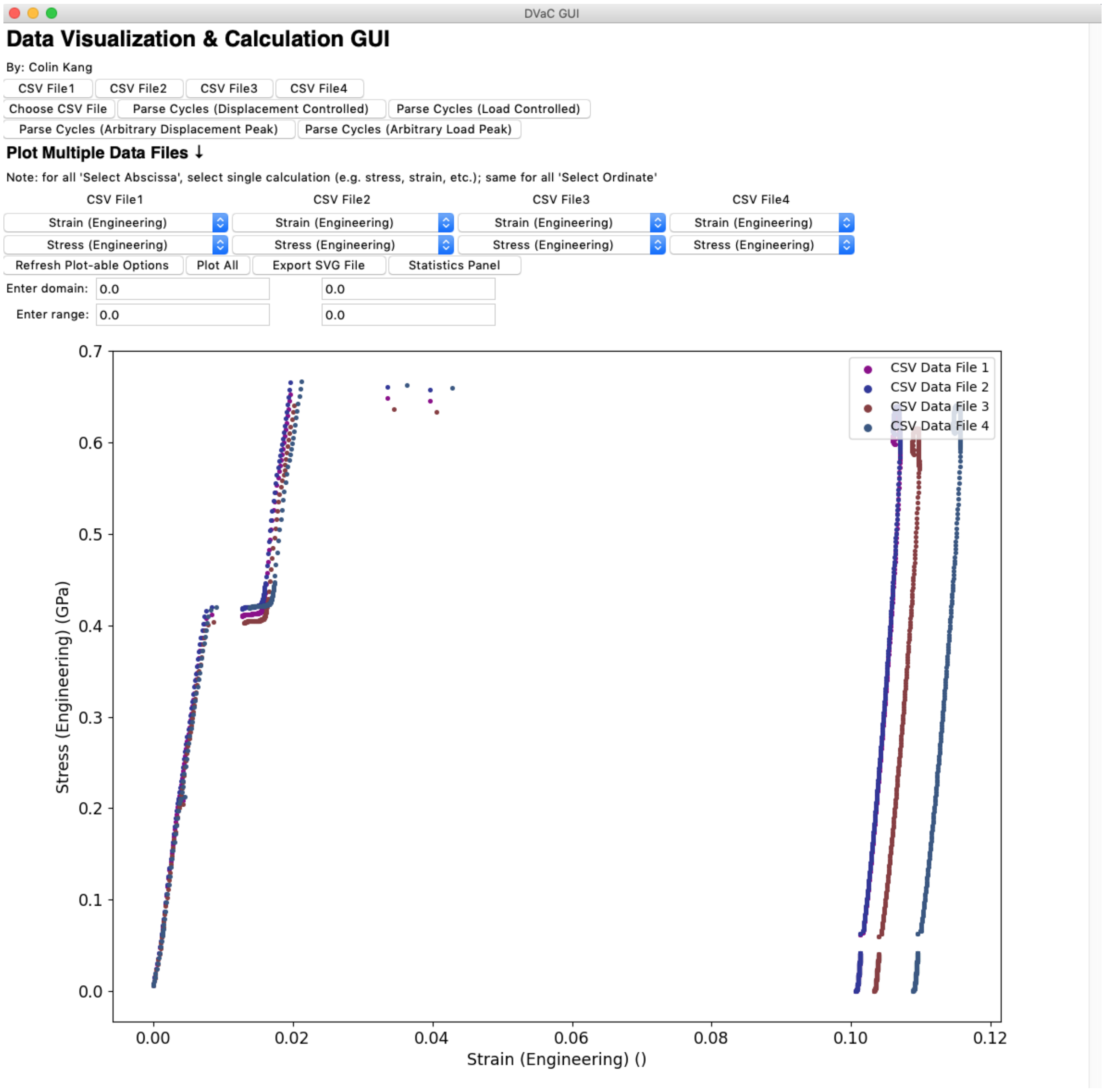


Figure 7: Multiplot Demonstration with Regular, Annealed, Pre-strained, and Re-milled ZnO Samples

References

[1] Julia R. Greer and W.D. Nix. "Size Dependence in Mechanical Properties of Golda t the Micron Scale in the Absence of Strain Gradients". In: *Applied Physics A* 90.1 (2008).

[2] Ju-Young Kim, Michael J. Burek, and Julia R. Greer. "The In-Situ Mechanical Testing of Nanoscale Single-Crystalline Nanopillars". In: *JOM: The Journal of the Minerals, Metals, and Materials Society* 61.12 (2009).

[3] Seok-Woo Lee, Seung Min Jane Han, and W.D. Nix. "Uniaxial Compression of fcc Au Nanopillars on an MgO Substrate: The Effects of Prestraining and Annealing". In: *Acta Materialia* 57.15 (2009).

[4] Rachel Liontas and Julia R. Greer. "3D Nano-architected Metallic Glass: Size Effect Suppresses Catastrophic Failure". In: *Acta Materialia* 133 (2017).

[5] Widianto P. Moestopo et al. "Pushing and Pulling on Ropes: Hierarchical Woven Materials". In: *Advanced Science* 7 (2020).

[6] Femto Tools$^{\text{TM}}$. *In-Situ SEM Material Tensile Testing.* URL: `https://www.femtotools.com/applications/in-situ-sem-material-testing/micro-tensile-testing`. accessed: 08.01.2020).

[7] Femto Tools$^{\text{TM}}$. *In-Situ SEM Material Testing.* URL: `https://www.femtotools.com/applications/in-situ-sem-material-testing/mechanical-testing-of-metamaterials`. (accessed: 08.01.2020).

[8] Femto Tools$^{\text{TM}}$. *In-Situ SEM Material Testing.* URL: `https://www.femtotools.com/applications/in-situ-sem-material-testing/micropillar-compression-testing`. (accessed: 08.01.2020).

[9] Andrey Vyatskikh et al. "Additive Manufacturing of 3D Nano-architected Metals". In: *Nature Communications* 9.593 (2018).

[10] Xiaoxing Xia et al. "Electrochemically Reconfigurable Architected Materials". In: *Nature* 573 (2019), pp. 205–213.

[11] Daryl W. Yee et al. "Additive Manufacturing of 3D-Architected Multifunctional Metal Oxides". In: *Advanced Materials* 31.33 (2019).

[12] Xuan Zhang et al. "Lightweight, Flaw-tolerant, and Ultrastrong Nanoarchitected Carbon". In: *Proceedings of the National Academy of Sciences* 116.14 (2019).

6 Appendix A

Table 1: Python packages used in creation of MDVaC GUI with version used and most recent version compatible.

Package	Version	Latest Version
Pillow	7.2.0	—
altgraph	0.17	—
certifi	2020.6.20	—
cycler	0.10.0	—
joblib	0.16.0	—
kiwisolver	1.2.0	—
macholib	1.14	—
matplotlib	3.3.1	3.3.2
modulegraph	0.18	—
numpy	1.19.1	1.19.2
pandas	1.1.0	1.1.2
pip	20.2.2	20.2.3
pyparsing	2.4.7	—
python-dateutil	2.8.1	—
pytz	2020.1	—
scipy	1.5.2	—
setuptools	49.6.0	50.3.0
six	1.15.0	—
threadpoolctl	2.1.0	—

7 Appendix B

```
def engineering_stress(self, data: {}):
    data['Stress (Engineering)'] = data[self.load_column] / self.specimen_area *
    1000000000000000
    data['Stress (Engineering)'].astype('float64')
    n = data['Stress (Engineering)'].mean()
    if n > 0.0:
        log_value = int(math.log10(n))
    elif n == 0.0:
        log_value = 0.0
    else:
        log_value = int(math.log10(-n)) + 1
    if 0 <= log_value <= 1:
        self.stress_pascal_unit = 'Pa'
    elif 2 <= log_value <= 4:
        data['Stress (Engineering)'] = data['Stress (Engineering)'] / 1000
        self.stress_pascal_unit = 'kPa'
    elif 5 <= log_value <= 7:
        data['Stress (Engineering)'] = data['Stress (Engineering)'] / 1000000
        self.stress_pascal_unit = 'MPa'
    elif 8 <= log_value <= 10:
        data['Stress (Engineering)'] = data['Stress (Engineering)'] / 1000000000
        self.stress_pascal_unit = 'GPa'
    elif 11 <= log_value <= 13:
        data['Stress (Engineering)'] = data['Stress (Engineering)'] / 1000000000000
        self.stress_pascal_unit = 'TPa'
    else:
        data['Stress (Engineering)'] = data['Stress (Engineering)'] / 1000000000000000
        self.stress_pascal_unit = 'PPa'
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) -
    1)] = self.stress_pascal_unit
    return data

def engineering_strain(self, data: {}):
    data['Strain (Engineering)'] = data[self.displacement_column] / self.specimen_height
    data['Strain (Engineering)'].astype('float64')
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) - 1)]
    = ''
    return data
```

Listing 1: Engineering Stress and Strain Calculations

8 Appendix C

```
def true_stress(self, data: {}):
    data['Stress (True)'] = data['Stress (Engineering)'] * ((self.specimen_height + data[
    self.displacement_column]) / self.specimen_height)
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) - 1)] =
    self.stress_pascal_unit
    return data

def true_strain(self, data: {}):
    data['Strain (True)'] = np.log((self.specimen_height + data[self.displacement_column]) /
     self.specimen_height)
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) - 1)] = ''
    return data
```

Listing 2: True Stress and Strain Calculations

9 Appendix D

```
def yms(self, data: {}):
    s1i = min(data[self.strain_type], key=lambda x:abs(x-self.strain_start))
    s2i = min(data[self.strain_type], key=lambda x:abs(x-self.strain_end))
    s1 = data[self.strain_type][data[self.strain_type] == s1i].index
    s2 = data[self.strain_type][data[self.strain_type] == s2i].index
    self.youngs_modulus_value_slope = (data[self.stress_type][s2[0]] - data[self.stress_type
    ][s1[0]]) / (data[self.strain_type][s2[0]] - data[self.strain_type][s1[0]])
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) - 1)] =
    self.stress_pascal_unit
    return data
```

Listing 3: Young's Modulus Slope Algorithm

10 Appendix E

```
def ymcsm(self, data: {}):
    if self.volume_conservation is True:
        data.loc[(data[self.csm_column]).between(self.csm_start, self.csm_end,
        inclusive=True), "Young's Modulus (CSM)"] = data[self.csm_column] *
        (self.specimen_height + data[self.displacement_column]) ** 2) /
        (self.specimen_area * self.specimen_height) * 1000000000
    elif self.area_conservation is True:
        data.loc[(data[self.csm_column]).between(self.csm_start, self.csm_end,
        inclusive=True), "Young's Modulus (CSM)"] = data[self.csm_column] *
        (self.specimen_height + data[self.displacement_column]) / self.specimen_area *
        1000000000
    first_temp = data[data["Young's Modulus (CSM)"].notna()]
    c = first_temp["Young's Modulus (CSM)"].mean()
    if math.isnan(c) is True:
        c = 0.0
    if c > 0.0:
        log_value_csm = int(math.log10(c))
    elif c == 0.0:
        log_value_csm = 0.0
    else:
        log_value_csm = int(math.log10(-c)) + 1
    if 0 <= log_value_csm <= 1:
        self.csm_pascal_unit = 'Pa'
    elif 2 <= log_value_csm <= 4:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"] / 1000
        self.csm_pascal_unit = 'kPa'
    elif 5 <= log_value_csm <= 7:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"] / 1000000
        self.csm_pascal_unit = 'MPa'
    elif 8 <= log_value_csm <= 10:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"] / 1000000000
        self.csm_pascal_unit = 'GPa'
    elif 11 <= log_value_csm <= 13:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"] / 1000000000000
        self.csm_pascal_unit = 'TPa'
    else:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"] / 1000000000000000
        self.csm_pascal_unit = 'PPa'
    last_temp = data[data["Young's Modulus (CSM)"].notna()]
    self.youngs_modulus_value_csm = last_temp["Young's Modulus (CSM)"].mean()
    try:
        data["Young's Modulus (CSM)"] = data["Young's Modulus (CSM)"].astype('float64')
    except ValueError:
        data["Young's Modulus (CSM"] = None
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) - 1)] =
    self.csm_pascal_unit
    return data
```

Listing 4: Young's Modulus Contact Stiffness Measurement Algorithm

11 Appendix F

```
def sneddon(self, dadta: {}):
    if self.volume_conservation is True:
        partial_compliance_sneddon = (math.sqrt(math.pi) * (1 - (self.poisson_ratio **
        2))) / (2 * self.known_elastic_modulus)
        data["Sneddon's correction to CSM (volume conservation)"] =
        np.reciprocal(np.reciprocal(data[self.csm_column]) - (partial_compliance_sneddon
        / np.sqrt((self.specimen_area * self.specimen_height) / (self.specimen_height +
        data[self.displacement_column]))))
        self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) -
        1)] = 'N/m'
        data.loc[(data[self.csm_column]).between(self.csm_start, self.csm_end,
        inclusive=True), "Young's Modulus (CSM w/ Sneddon's correction)"] = \
        data["Sneddon's correction to CSM (volume conservation)"] *
        ((self.specimen_height + data[self.displacement_column]) ** 2) /
        (self.specimen_area * self.specimen_height) * 1000000000
    elif self.area_conservation is True:
        area = self.specimen_area
        compliance_sneddon = (math.sqrt(math.pi) * (1 - (self.poisson_ratio ** 2))) / (2
        * self.known_elastic_modulus * math.sqrt(area))
        data["Sneddon's correction to CSM (area conservation)"] =
        np.reciprocal(np.reciprocal(data[self.csm_column]) - compliance_sneddon)
        self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) -
        1)] = 'N/m'
        data.loc[(data[self.csm_column]).between(self.csm_start, self.csm_end,
        inclusive=True), "Young's Modulus (CSM w/ Sneddon's correction)"] = \
        data["Sneddon's correction to CSM (area conservation)"] * (self.specimen_height +
        data[self.displacement_column]) / self.specimen_area * 1000000000
    first_sneddon_temp = data[data["Young's Modulus (CSM w/ Sneddon's
    correction)"].notna()]
    c2 = first_sneddon_temp["Young's Modulus (CSM w/ Sneddon's correction)"].mean()
    if math.isnan(c2) is True:
        c2 = 0.0
    if c2 > 0.0:
        log_value_sneddon = int(math.log10(c2))
    elif c2 == 0.0:
        log_value_sneddon = 0.0
    else:
        log_value_sneddon = int(math.log10(-c2)) + 1
    if 0 <= log_value_sneddon <= 1:
        self.sneddon_pascal_unit = 'Pa'
    elif 2 <= log_value_sneddon <= 4:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus
        (CSM w/ Sneddon's correction)"] / 1000
        self.sneddon_pascal_unit = 'kPa'
    elif 5 <= log_value_sneddon <= 7:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus
        (CSM w/ Sneddon's correction)"] / 1000000
        self.sneddon_pascal_unit = 'MPa'
    elif 8 <= log_value_sneddon <= 10:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus
        (CSM w/ Sneddon's correction)"] / 1000000000
        self.sneddon_pascal_unit = 'GPa'
    elif 11 <= log_value_sneddon <= 13:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus
        (CSM w/ Sneddon's correction)"] / 1000000000000
        self.sneddon_pascal_unit = 'TPa'
    else:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus (CSM w
    / Sneddon's correction)"] / 1000000000000000
        self.sneddon_pascal_unit = 'PPa'
    last_sneddon_temp = data[data["Young's Modulus (CSM w/ Sneddon's
    correction)"].notna()]
    self.youngs_modulus_value_sneddon = last_sneddon_temp["Young's Modulus (CSM w/
    Sneddon's correction)"].mean()
    try:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = data["Young's Modulus
        (CSM w/ Sneddon's correction)"].astype('float64')
    except ValueError:
        data["Young's Modulus (CSM w/ Sneddon's correction)"] = None
    self.units_list[data.columns[len(data.columns) - 1].format(len(data.columns) -
    1)] = self.sneddon_pascal_unit
    return data
```

Listing 5: Sneddon's Correction to Young's Modulus Contact Stiffness Measurement Algorithm

12 Appendix G

```
def energy_dissipated(self, data:{}):
    wrk = np.trapz(data[self.stress_type], data[self.strain_type])
    self.energy_dissipated_value = wrk
    return data
```

Listing 6: Energy Dissipated Calculation

13 Appendix H

```
def ultimate_stress(self, data: {}):
    def shift(a):
        a_r = np.roll(a, 1)
        a_l = np.roll(a, -1)
        return np.stack([a_l, a_r], axis=1)
    z = np.array(data[self.strain_type])
    diff = abs(shift(z) - z.reshape(-1, 1))
    diff = diff[1:-1]
    indices = diff.argmax(axis=0) - 2
    self.ultimate_stress_value = data[self.stress_type][indices[0]]
    return data

def ultimate_strain(self, data: {}):
    def shift(a):
        a_r = np.roll(a, 1)=
        a_l = np.roll(a, -1)=
        return np.stack([a_l, a_r], axis=1)
    z = np.array(data[self.strain_type])
    diff = abs(shift(z) - z.reshape(-1, 1))
    diff = diff[1:-1]
    indices = diff.argmax(axis=0) - 2
    self.ultimate_strain_value = data[self.strain_type][indices[0]]
    return data
```

Listing 7: Ultimate Stress and Strain at Failure Detection Algorithm

14 Appendix I

```
def bursts(self, data: {}):
    all_diff = data[self.strain_type].diff()
    if all_diff.mean() >= 0.0:
        log_value = math.log10(all_diff.mean())
    else:
        log_value = math.log10(-all_diff.mean()) + 1
    factor = abs(log_value) - 3
    mean_changer = 2 * (10 ** factor)
    large_diff_upper = all_diff.index[all_diff >= (mean_changer * all_diff.mean())]
    large_diff_lower = large_diff_upper - 1
    if self.toggle_bursts is True:
        for i in range(0, len(large_diff_lower), 2):
            self.large_diff[i] = large_diff_lower[int(i / 2)]
            self.large_diff[i + 1] = large_diff_upper[int(i / 2)]
        for j in range(0, len(self.large_diff) * 2, 2):
            self.burst_stress_strain[j] = data[self.stress_type][self.large_diff[int(j /
            2)]]
            self.burst_stress_strain[j + 1] =
            data[self.strain_type][self.large_diff[int(j / 2)]]
        for k in range(0, len(self.large_diff), 2):
            self.burst_size[int(k/2)] = data[self.strain_type][self.large_diff[k + 1]] -
            data[self.strain_type][self.large_diff[k]]
    else:
        for x in range(0, len(large_diff_lower)):
            self.large_diff.append(large_diff_lower[x])
            self.large_diff.append(large_diff_upper[x])
        for w in range(0, len(self.large_diff)):
            self.burst_stress_strain.append(data[self.stress_type][self.large_diff[w]])
            self.burst_stress_strain.append(data[self.strain_type][self.large_diff[w]])
        for z in range(0, len(self.large_diff), 2):
            self.burst_size.append(data[self.strain_type][self.large_diff[z + 1]] -
            data[self.strain_type][self.large_diff[z]])
        self.toggle_bursts = True
    self.num_bursts_value = len(self.large_diff) / 2
    return data
```

Listing 8: Burst Events Detection Algorithm

15 Appendix J

```
def parse_load_cycles(self, r):
    global index
    global csv_list
    temp = index
    for i in data_frames[r].columns:
        if 'Load' in i:
            load_column = i
            break
    self.unit_storage(data_frames[r])
    temp_df = self.reindex(data_frames[r])
    val = int(temp_df[load_column].max() * 2)
    nrows = val - 1
    groups = temp_df.groupby(temp_df.index // nrows)
    self.csv_calculation[r].destroy()
    self.abscissa_drops[r].destroy()
    self.ordinate_drops[r].destroy()
    self.csv_label[r].destroy()
    self.a_completed[r] = True
    self.o_completed[r] = True
    del data_frames[r]
    gc.collect()
    for (frameno, frame) in groups:
        csv_list[index] = 'parse'
        csv_calculation = tk.Button(self.parent, text='CSV File ' + str(temp) +
        alphabet[frameno], width=10, command=partial(self.calc_window, index))
        csv_calculation.grid(row=2, column=(index * 4), columnspan=4, sticky=tk.EW)
        csv_identities.append(alphabet[index])
        data_frames[index] = frame
        data_frames[index] = data_frames[index].append(pd.Series(), ignore_index=True)
        data_frames[index] = data_frames[index].shift(1)
        data_frames[index].index = range(val)
        for i in data_frames[index].columns:
            data_frames[index].loc[0, i] = self.units_list[i]
        index += 1
        csv_label = tk.Label(self.parent, text='CSV File ' + str(temp) +
        alphabet[frameno], width=25)
        csv_label.grid(row=7, column=(index-1) * 10, columnspan=10, sticky=tk.EW)
        column_names = []
        for col_name in data_frames[index-1].columns:
            column_names.append(col_name)
        self.a_completed[index-1] = False
        self.abscissa_buttons[index-1] = tk.StringVar(self.parent)
        self.abscissa_buttons[index-1].set('Select Abscissa')
        self.abscissa_drops[index-1] = tk.OptionMenu(self.parent,
        self.abscissa_buttons[index-1], 'Select Abscissa', command=lambda *args: None)
        for i in column_names[:]:
            self.abscissa_drops[index - 1]['menu'].add_command(label=i + ' (' +
            self.units_list[i] + ')', command=tk._setit(self.abscissa_buttons[index - 1],
            partial(self.select_adata, index - 1)))
        self.abscissa_drops[index-1].grid(row=8, column=(index-1) * 10, columnspan=10,
        sticky=tk.EW)
        self.o_completed[index-1] = False
        self.ordinate_buttons[index-1] = tk.StringVar(self.parent)
        self.ordinate_buttons[index-1].set('Select Ordinate')
        self.ordinate_drops[index-1] = tk.OptionMenu(self.parent,
        self.ordinate_buttons[index-1], 'Select Ordinate', command=lambda *args: None)
        for i in column_names[:]:
            self.ordinate_drops[index - 1]['menu'].add_command(label=i + ' (' +
            self.units_list[i] + ')', command=tk._setit(self.ordinate_buttons[index - 1],
            i, partial(self.select_odata, index - 1)))
        self.ordinate_drops[index - 1].grid(row=9, column=(index - 1) * 10,
        columnspan=10, sticky=tk.EW)
```

Listing 9: Parse Load-Controlled Data Files Algorithm

16 Appendix K

```
def parse_arbitrary_displacement_cycles(self, r):
    global index
    global csv_list
    temp = index
    for i in data_frames[r].columns:
        if 'Displacement' in i:
            displacement_column = i
            break
    self.unit_storage(data_frames[r])
    temp_df = self.reindex(data_frames[r])
    col = temp_df[displacement_column]
    minima = (col <= col.shift()) & (col < col.shift(-1))
    g = minima.cumsum()
    groups = temp_df.groupby(g)
    self.csv_calculation[r].destroy()
    self.abscissa_drops[r].destroy()
    self.ordinate_drops[r].destroy()
    self.csv_label[r].destroy()
    self.a_completed[r] = True
    self.o_completed[r] = True
    del data_frames[r]
    gc.collect()
    for (frameno, frame) in groups:
        csv_list[index] = 'parse'
        csv_calculation = tk.Button(self.parent, text='CSV File ' + str(temp) +
        alphabet[frameno], width=10, command=partial(self.calc_window, index))
        csv_calculation.grid(row=2, column=(index * 4), columnspan=4, sticky=tk.EW)
        csv_identities.append(alphabet[index])
        data_frames[index] = frame
        data_frames[index] = data_frames[index].append(pd.Series(), ignore_index=True)
        data_frames[index] = data_frames[index].shift(1)
        data_frames[index].index = range(int(frame.size / 2) + 1)
        for i in data_frames[index].columns:
            data_frames[index].loc[0, i] = self.units_list[i]
        index += 1
        csv_label = tk.Label(self.parent, text='CSV File ' + str(temp) +
        alphabet[frameno], width=25)
        csv_label.grid(row=7, column=(index - 1) * 10, columnspan=10, sticky=tk.EW)
        column_names = []
        for col_name in data_frames[index - 1].columns:
            column_names.append(col_name)
        self.a_completed[index - 1] = False
        self.abscissa_buttons[index - 1] = tk.StringVar(self.parent)
        self.abscissa_buttons[index - 1].set('Select Abscissa')
        self.abscissa_drops[index - 1] = tk.OptionMenu(self.parent,
        self.abscissa_buttons[index - 1], 'Select Abscissa', command=lambda *args: None)
        for i in column_names[:]:
            self.abscissa_drops[index - 1]['menu'].add_command(label=i + ' (' +
            self.units_list[i] + ')', command=tk._setit(self.abscissa_buttons[index - 1],
            i, partial(self.select_adata, index - 1)))
        self.abscissa_drops[index - 1].grid(row=8, column=(index - 1) * 10, columnspan=10,
        sticky=tk.EW)
        self.o_completed[index - 1] = False
        self.ordinate_buttons[index - 1] = tk.StringVar(self.parent)
        self.ordinate_buttons[index - 1].set('Select Ordinate')
        self.ordinate_drops[index - 1] = tk.OptionMenu(self.parent,
        self.ordinate_buttons[index - 1], 'Select Ordinate', command=lambda *args: None)
        for i in column_names[:]:
            self.ordinate_drops[index - 1]['menu'].add_command(label=i + ' (' +
            self.units_list[i] + ')', command=tk._setit(self.ordinate_buttons[index - 1],
            i, partial(self.select_odata, index - 1)))
        self.ordinate_drops[index - 1].grid(row=9, column=(index - 1) * 10, columnspan=10,
        sticky=tk.EW)
```

Listing 10: Parse Arbitrarily Displacement Peaking and Dipping Data Files Algorithm

17 Appendix L

```
def plot_all(self):
    for w in range(0, len(self.a_completed)):
        if self.a_completed[w] is True:
            self.a_ready = True
        else:
            self.a_ready = False
    for x in range(0, len(self.o_completed)):
        if self.o_completed[x] is True:
            self.o_ready = True
        else:
            self.o_ready = False
    if self.a_ready is True and self.o_ready is True:
        self.fig = Figure(figsize=(10, 10), dpi=100)
        try:
            for i in self.abscissa_values:
                pa = csv_identities[0].units_list[self.abscissa_values[i]]
                po = csv_identities[0].units_list[self.ordinate_values[i]]
                break
        except AttributeError:
            for j in self.abscissa_values:
                pa = self.units_list[self.abscissa_values[j]]
                po = self.units_list[self.ordinate_values[j]]
                break
        for z in self.abscissa_values:
            xl = self.abscissa_values[z]
            yl = self.ordinate_values[z]
            break
        self.fig.subplots_adjust(left=0.1, bottom=0.25, right=0.9, top=0.975)
        self.graph = self.fig.add_subplot(111, xlabel=xl + ' (' + pa + ')', ylabel=yl + '
        (' + po + ')')
        for i in data_frames:
            df = data_frames[i]
            df = self.reindex(df)
            df = self.re_zero(df)
            self.graph.scatter(df[self.abscissa_values[i]], df[self.ordinate_values[i]],
            s=0.5, label='CSV Data File ' + str(i+1))
        ax = self.fig.gca()
        ax.xaxis.label.set_size(12.5)
        ax.yaxis.label.set_size(12.5)
        ax.tick_params(axis='x', labelsize=12.5)
        ax.tick_params(axis='y', labelsize=12.5)
        if self.xmin_exists is True and self.xmax_exists is True and self.ymin_exists is
        True and self.ymax_exists is True:
            ax.set(xlim=(self.xmin_value, self.xmax_value), ylim=(self.ymin_value,
            self.ymax_value))
        elif self.xmin_exists is True and self.xmax_exists is True:
            ax.set(xlim=(self.xmin_value, self.xmax_value))
        elif self.ymin_exists is True and self.ymax_exists is True:
            ax.set(ylim=(self.ymin_value, self.ymax_value))
        self.graph.legend(loc="upper right", markerscale=10)
        canvas = FigureCanvasTkAgg(self.fig, master=self.parent)
        canvas.draw()
        canvas.get_tk_widget().grid(row=13, rowspan=36, column=0, columnspan=80,
        sticky=tk.NW)
        self.can_export_multiplot = True
    else:
        self.create_pop_up("Must have all 'Select Abscissa' and all 'Select Ordinate'
        selected w/ same calculation for abscissa and ordinate respectively.")
```
Listing 11: Multiple Figure Plotting of Raw Data and Calculations Functionality on Single Graph