



AFRL-AFOSR-VA-TR-2023-0333

Numerically Predicting High Explosive Violent Response (HEVR) in Air Force Explosives

Maestas, Joseph
APPLIED RESEARCH ASSOCIATES, INC
4300 SAN MATEO BLVD NE STE A220
ALBUQUERQUE, NM, 87110-1295
US

04/04/2023
Final Technical Report

DISTRIBUTION A: Distribution approved for public release.

Air Force Research Laboratory
Air Force Office of Scientific Research
Arlington, Virginia 22203
Air Force Materiel Command

REPORT DOCUMENTATION PAGE

PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ORGANIZATION.

1. REPORT DATE 20230404		2. REPORT TYPE Final		3. DATES COVERED	
				START DATE 20180501	END DATE 20221231
4. TITLE AND SUBTITLE Numerically Predicting High Explosive Violent Response (HEVR) in Air Force Explosives					
5a. CONTRACT NUMBER FA9550-18-C-0008		5b. GRANT NUMBER		5c. PROGRAM ELEMENT NUMBER 61102F	
5d. PROJECT NUMBER		5e. TASK NUMBER		5f. WORK UNIT NUMBER	
6. AUTHOR(S) Joseph Maestas					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) APPLIED RESEARCH ASSOCIATES, INC 4300 SAN MATEO BLVD NE STE A220 ALBUQUERQUE, NM 87110-1295 US					8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research 875 N. Randolph St. Room 3112 Arlington, VA 22203				10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/AFOSR RTA1	11. SPONSOR/MONITOR'S REPORT NUMBER(S) AFRL-AFOSR-VA-TR-2023-0333
12. DISTRIBUTION/AVAILABILITY STATEMENT A Distribution Unlimited: PB Public Release					
13. SUPPLEMENTARY NOTES					
14. ABSTRACT The extreme conditions experienced by Air Force energetics in operational settings can lead to failed initiation, sub-detonative reactions, and unintended violent response. This effort investigated methods for simulating and predicting this behavior using continuum and mesoscale models and methods for calibrating and validating these models. Artificial intelligence/machine learning methods were leveraged for calibrating highly non-linear models for various aspects of the problem and these were, in turn, generalized into frameworks to be used with future energetic materials. The main result of this research is an understanding of the data and models needed to simulate mechanical initiation in high explosives and a process, or workflow, for calibrating these models.					
15. SUBJECT TERMS					
16. SECURITY CLASSIFICATION OF:				17. LIMITATION OF ABSTRACT	
a. REPORT U	b. ABSTRACT U	c. THIS PAGE U	UU		18. NUMBER OF PAGES 201
19a. NAME OF RESPONSIBLE PERSON CHIPING LI					19b. PHONE NUMBER (Include area code) 426-8574

Standard Form 298 (Rev. 5/2020)
Prescribed by ANSI Std. Z39.18

Final Performance Report
for
Numerically Predicting High Explosive Violent Response
(HEVR) in Air Force Explosives
Contract No. FA9550-18-C-0008

Reporting Period: 1 May 2018 – 31 December 2021

Principal Investigator

Dr. Joey Maestas

Applied Research Associates, Inc.
956 W. John Sims Pkwy
Niceville, FL 32579
850-678-5222
jmaestas@ara.com

1. Objectives

There have been no changes to the research objectives or statement of work.

2. Status of Effort

Please see attached final technical report.

3. Accomplishments/New Findings

- Developed an initial HERMES model for the material of interest
- Determined model enhancements needed for the HERMES model
- Generated a statistically relevant set of representative volume elements (RVEs) for the material of interest. Can be shared with the community.
- Developed a workflow and frameworks for calibrating the HERMES model
- Developed machine learning methods for creating complete equations of state
- Developed a sensor for measuring fill displacement in penetrating weapons

4. Personnel Supported

<i>Person</i>	<i>Role</i>	<i>Tasks this Period</i>
Joey Maestas	Principal Investigator	• Overview research • Complete equations of state development • Constituent strength and burn models • ML/optimization methods
Erik Spencer	Technical Intern	• Fill displacement sensor • Adhesive strength model • HTPB strength model
Daniel Cano	Staff Engineer	• RVE mesh development • HERMES (continuum) strength/damage models
Matthew VandenBoom	Technical Intern	• Python scripts for optimization/parameterization

5. Publications

No publications submitted during this project's period of performance. We are in the process of writing a paper for the fill displacement sensor and have started a paper for AP EOS calibration. Once submitted, they will acknowledge this program.

6. Interactions/Transitions

i. Dissemination

- a) Discussed work and project status with AFRL/RW colleagues.
- b) Presented status at tri-service reviews.
- c) Presented work to ARA colleagues.
- d) Presented work to interested parties at AFLCMC/EB.

ii. Collaboration

- a) We worked with Dr. Alain Beliveau (ARA) in developing the fill displacement sensor.
- b) We worked with Marc Cawkwell (LANL) to acquire updated EOS's for RDX and HMX.
- c) We worked with Ericka Amborn (ARA) to find test data for validating deflagration models.
- d) We worked with John Clayton and Richard Becker of the Army Research Laboratory (ARL) to implement crystal strength models they developed for RDX.
- e) We received information and guidance on the HERMES model from Brad White and Jack Reaugh (LLNL).

iii. Transitions

- a) Shared product EOS models developed under this work with NSWC/Dahlgren.
- b) Shared product EOS models and deflagration models developed under this work with AFLCMC/EBD.
- c) We are working with the computational energetics team at AFRL/RWMA to transition input decks and scripts developed under this project.
- d) We are creating a report outlining the methods needed for calibrating the HERMES model. This will be delivered to AFRL/RWMA.

7. New Outcomes

None during this reporting period.

8. Honors/Awards

None during this reporting period.



Final Technical Report

Numerically Predicting High Explosive Violent Response (HEVR) in Air Force Explosives

Joseph T. Maestas, PhD

Applied Research Associates, Inc.
Emerald Coast Division
956 W. John Sims Pkwy
Niceville, FL 32578

FA9550-18-C-0008

February 12, 2022

Distribution Statement A. Approved for public release; distribution is unlimited. AP- PROVAL PENDING

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY)		2. REPORT TYPE		3. DATES COVERED (From — To)		
02-12-2022		Final Performance		1 May 2018 — 31 December 2021		
4. TITLE AND SUBTITLE Numerically Predicting High Explosive Violent Response (HEVR) in Air Force Explosives				5a. CONTRACT NUMBER		
				FA9550-18-C-0008		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Maestas, Joseph, T.				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Applied Research Associates, Inc. Emerald Coast Division 956 W. John Sims Pkwy Niceville, FL 32578				8. PERFORMING ORGANIZATION REPORT NUMBER		
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Air Force Office of Scientific Research 875 North Randolph Street, Suite 375 Arlington, VS 22203				10. SPONSOR/MONITOR'S ACRONYM(S)		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION / AVAILABILITY STATEMENT Approval for public release; distribution is unlimited.						
13. SUPPLEMENTARY NOTES The views, opinions and/or findings contained in this report are those of the authors and should not be construed as an official U.S. Government position, policy or decision, unless so designated by other documentation.						
14. ABSTRACT The extreme conditions experienced by Air Force energetics in operational settings can lead to failed initiation, sub-detonative reactions, and unintended violent response. This effort investigated methods for simulating and predicting this behavior using continuum and mesoscale models and methods for calibrating and validating these models. Artificial intelligence/machine learning methods were leveraged for calibrating highly non-linear models for various aspects of the problem and these were, in turn, generalized into frameworks to be used with future energetic materials. The main result of this research is an understanding of the data and models needed to simulate mechanical initiation in high explosives and a process, or workflow, for calibrating these models.						
15. SUBJECT TERMS Non-ideal energetics; mesoscale modeling; complete equations of state; deflagration; machine learning; fill displacement						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON	
a. REPORT	b. ABSTRACT	c. THIS PAGE			Maestas, Joseph, T.	
U	U	U	UU	195	19b. TELEPHONE NUMBER (include area code) (505) 850-2925	

This page was intentionally left blank

Contents

1	Introduction	1
2	Continuum Model Development	2
2.1	Model Selection	2
2.2	Material of Interest	3
2.3	Submodel Calibration	3
2.3.1	Unreacted Equation of State	3
2.3.2	Product Equation of State	4
2.3.3	Constitutive Model	6
2.3.4	Fragmentation Model	8
2.3.5	Ignition Model	10
2.3.6	Burn Model	10
2.3.7	Detonation Model	10
3	Mesoscale and Constituent Studies	11
3.1	Constituent Models	13
3.1.1	RDX	13
3.1.2	Aluminum	13
3.1.3	Ammonium Perchlorate	13
3.1.4	HTPB	15
3.2	Mesoscale Studies	16
3.2.1	Sample Generation	16
3.2.2	Adiabatic Compression	19
4	Fill Displacement Diagnostic	21
4.1	Background	21
4.2	Sensor Selection	22
4.3	Prototype Development	22
4.4	Bench Testing	23
4.5	Subscale Penetration Testing	24
5	Impact and Future Work	27
	References	28
A	Code	31
A.1	Unreacted EOS Optimization	31
A.2	Cheetah tool	40
A.3	Strength Model Optimization	50
A.3.1	Fitness.py	50
A.3.2	CustomPool.py	58
A.3.3	MultiSim_Inds.py	60
A.3.4	ExpClasses.py	65

A.3.5	SM_GA_Multi.py	68
A.4	CREST Lite Optimization	79
A.5	Ammonium Perchlorate EOS Optimization	98
A.5.1	EOS_Optimize	98
A.5.2	OptimizationMain	114
A.5.3	EOS_Functions	123
A.5.4	EOS_Plots	136
A.5.5	EOS_Evaluation	143
A.5.6	ProcessQueue	161
A.5.7	bestingen	163
A.6	Calculating RVEs	164
A.7	Generating RVEs	173
A.8	Converting RVE to STL	175
A.9	Mesoscale Adiabatic Compression	178

List of Figures

1	JWL unreacted equation of state fit to shock data.	4
2	Polynomial EOS fit to JWL model. Solid lines represent principal adiabat pressure and dashed lines represent the energy term. Note that energy terms ($\Gamma \cdot \rho$) overlay each other.	5
3	Maxwell constructions for the product equation of state.	6
4	Yield strength as a function of strain rate.	7
5	Optimized strength model response (black solid line) compared to data (blue dots) for unconfined compression.	8
6	Simulated plastic strain within the sample after impact.	9
7	Calculated plastic strain and S/V fits versus impact velocity.	9
8	Linear burn rate as a function of pressure.	11
9	Run distance calculation using bi-linear fits.	12
10	Optimized model Pop Plot fit.	12
11	Isothermal compression comparison between data and model fit.	15
12	Hugoniot comparison between data and model fit.	16
13	Isobaric compression comparison between data and model fit.	17
14	Isothermal compressibility comparison between data and model fit.	18
15	Specific heat comparison between data and model fit.	18
16	Isothermal compression of HTPB at various temperatures (in Kelvin).	19
17	RVE sample generation. Left image shows full shaped geometries, middle image shows same geometry but without the binder, and right image shows RVE after conversion to FEA mesh.	19
18	CTH adiabatic compression calculation. Surrounding water is not illustrated.	20
19	State variables over time. Solid line represents average.	20
20	EOS model fit to computed states.	21
21	AMR sensor operation (Honeywell HMC1501).	22
22	HMC1501 voltage response plot.	23
23	AMR sensor array.	24
24	Single sensor circuit layout.	24
25	Prototype bench testing.	25
26	Effects of ferrous metal presence on sensor signal outputs.	25
27	Completed prototype prior to mounting.	26
28	An early version of the sensor placed in mounting hardware (prior to epoxy fill).	26

List of Tables

1 Introduction

Air Force weapon systems are constantly being pushed to survive extreme loading environments such as seen in hard and deeply buried target (HDBT) attacks. During these penetration events, the explosive payload undergoes significant acceleration, pressure, and shear loading that can damage the material and possibly cause it to react with violent response, referred to as HEVR (high explosive violent response). The underlying physics coupling energetic material strength, damage, deflagration, and detonation responses are not well understood, and therefore almost no modeling, simulation, and analysis (MS&A) goes into predicting HEVR in order to guide weapon design or operational weaponeering. Despite these difficulties, new models are currently being developed to describe energetic response to non-shock mechanical stimulus, but such models have only been applied to a handful of Department of Energy (DoE) explosives. This effort aims to advance the understanding of the underlying physics, specifically the interplay of material damage and reaction mechanisms, so as to apply HEVR models to HDBT scenarios. Addressing this requires a multi-directional approach of experimentation, theory, and simulation both at the continuum and meso- scales. The current state-of-the-art is at a unique position where new experiments, analysis techniques, and advancements in mesoscale simulation can be used to inform these emerging models. Generating such models will allow weapon developers to optimize their designs and increase the Air Force's predictive capability for determining the survivability of a weapon.

The objective of the proposed research was to investigate the damage and ignition mechanisms of non-ideal, aluminized explosives, such that sufficient information can be gathered to parameterize a numerical HEVR model. The first step was to determine which HEVR model is best suited for describing energetic response resulting from loading typically experienced in penetration events, and what data are required to calibrate that model. We determined that the HERMES (High Explosive Response to MEchanical Stimulus) model offered the best balance between accuracy and calibration complexity.^[24] HERMES uses submodels to describe strength, damage from cracking and fragmentation, ignition, ignition front propagation, and burning after ignition. A significant amount of work went into calibrating these submodels using existing data and investigating the needs for any additional data that either mesoscale simulations or experimentation might provide.

On the mesoscale front, describing all the necessary physics required developing robust models for equation of state, strength, burning, and damage for each constituent of the material of interest. A number of new models were developed as part of this effort as well as optimization frameworks used to calibrate these models. However, the sheer number of models needed ended up being too large to complete during the period of performance. In addition to material model development, mesoscale studies require accurate grain structures, so a number of scripts were written to generate grain geometries in a statistically-relevant manner for different solution methods.

Finally, a major aspect missing in this problem is knowledge of the actual environment the

explosive experiences during a penetration event. There existed no data showing the actual fill response during impact, so prior material models had been calibrated to subscale and laboratory experiments whose pressures and strain rates are far below real life events. As part of this effort, a new experimental device and method were developed to measure fill displacement in full scale tests.

The full project as described in the original proposal has not been completed, since the large scope of the required effort was not fully appreciated. This final report provides an overview of the project to date. [Section 2](#) describes model development for the material of interest at the continuum scale, [Section 3](#) provides an overview of mesoscale studies and constitutive material model development, [Section 4](#) describes the development of a fill displacement sensor, and finally, [Section 5](#) outlines potential future work on this effort.

2 Continuum Model Development

The overall goal of this effort is to provide the Air Force with continuum-scale models for evaluating weapon designs and engagement scenarios. To that end, we investigated models that are available within government codes (DoE or DoD). Of special interest are codes that AFRL uses such as CTH (Sandia National Laboratories hydrocode), Sierra (SNL finite element analysis code), ALE3D (Lawrence Livermore National Laboratory multiphysics code), and EPIC (DoD FEA code). The following sections describe model selection and the needs for calibrating that model.

2.1 Model Selection

Within the codes mentioned above, there are only a few models capable of describing energetic reaction from mechanical insult (all from either CTH or ALE3D). These include:

- Coupled Damage and Reaction, CDAR (CTH). This model, based on Baer-Nunziato multiphase reactive theory, couples constitutive mechanical, damage, and burn models in a thermodynamically consistent manner and was originally developed to assess propellant reactions to fragment impact.[\[18, 19\]](#)
- Damage Initiation Reaction, DMGIR (CTH). This is a computational model developed to capture the energetic response due to non-shock mechanical insults.[\[30\]](#) This is a complementary model to the History Variable Reactive Burn[\[13\]](#) (HVRB) model, which is typically used to predict prompt shock-initiated reaction. DMGIR uses a Johnson-Cook strength and damage model[\[11\]](#) and couples the damage into the HVRB model as a second history variable. The damage is coupled to the reaction rate, so mechanical loading can initiate reaction.
- Propellant Energetic Response to Mechanical Stimuli, PERMS (ALE3D). This is a reactive flow style model that is used to model the reaction of ammonium perchlo-

rate (AP)-aluminum-hydroxyl-terminated polybutadiene (HTPB) propellants.[17] The model uses two reactions to describe the AP-HTPB burning and the subsequent aluminum burning.

- High Explosive Response to Mechanical Stimulus, HERMES (ALE3D). This model uses a set of submodels to describe strength, damage from cracking and fragmentation, ignition, ignition front propagation, and burning after ignition.[27, 23, 25, 26, 24] The model is implemented in ALE3D and requires setting 34 parameters. HERMES is capable of describing mechanical initiation, damage evolution, slow burning, deflagration, and detonation, but at the cost of increased complexity.

After investigating the strengths and limitations of these models, it was determined that the HERMES model offered the greatest accuracy and flexibility for problems of interest. CDAR, another viable model, did not provide a robust detonation model consistent with reactive flow model calibration techniques being developed at AFRL. DMGRIR is simple and easy to calibrate (with the correct data), but the HVRB model it is based on is not able to accurately capture deflagration or slow burn. Finally, PERMS does not accurately represent the strength of a multiphase matrix of gas and solid species.

2.2 Material of Interest

Non-ideal explosives with large critical diameters offer increased survivability characteristics needed for certain Air Force applications. These explosives typically comprise of a primary energetic material (i.e., RDX, HMX), an oxidizer such as ammonium perchlorate, aluminum for increased blast impulse, and plastic/polymer binder. For this work, the explosive of interest is comprised of: RDX, ammonium perchlorate (AP), powdered aluminum, and a hydroxyl-terminated polybutadiene (HTPB) binder.

2.3 Submodel Calibration

The HERMES model consists of six submodels: unreacted and product equations of state, constitutive (strength), subsonic burn, supersonic/detonation, and damage. These submodels were calibrated to existing data and are described below.

2.3.1 Unreacted Equation of State

This model describes the pressure-energy-volume relationship of the unreacted material. It is important for capturing shock and hydrostatic response. HERMES uses Grüneisen model with a quadratic form of the principal adiabat:

$$P_s = A_1\mu + A_2\mu^2 + A_3\mu^3 \quad (1a)$$

$$\frac{\Gamma}{V} = \rho_0(B_0 + B_1\mu + B_2\mu^2) \quad (1b)$$

$$P = P_s + \frac{\Gamma}{V}(e - e_s), \quad (1c)$$

where V is specific volume and $\mu = \frac{\rho}{\rho_0} - 1$. Ideally, the principal adiabat would be calibrated from room temperature compression data in order to better capture the non-shock behavior. However, for this initial calibration, the Hugoniot was used as the reference curve.

A Jones-Wilkins-Lee (JWL) model was initially calibrated to shock data derived from an AFRL wedge test series (as part of another effort). The plot in [Figure 1](#) below shows the fit.

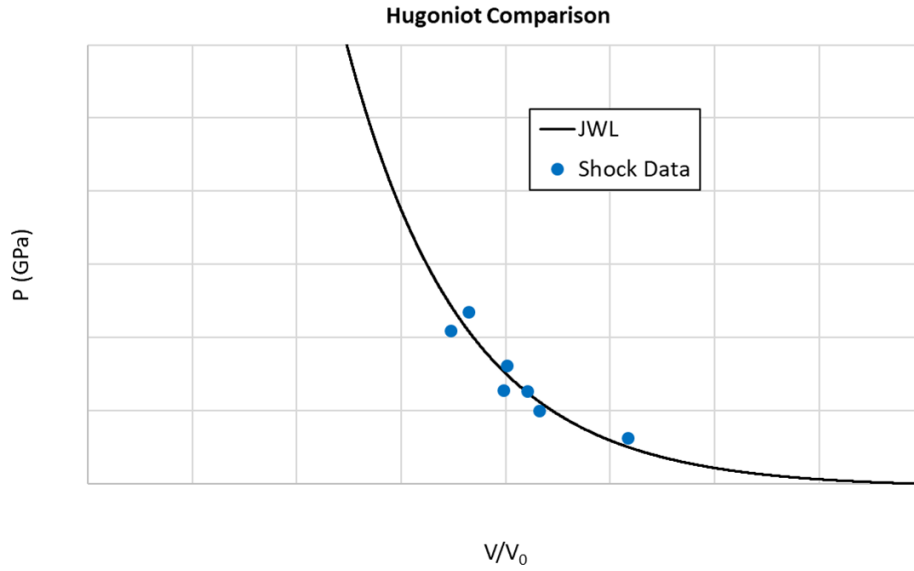


Figure 1: JWL unreacted equation of state fit to shock data.

An optimization code was written to calibrate the HERMES model, [Eq. 1](#), to the existing JWL model using Python. The code (included in [subsection A.1](#)) is based on a genetic algorithm using the Python package DEAP [6]. The resulting fit is given in [Figure 2](#) below.

2.3.2 Product Equation of State

The product EOS is used to describe the pressure-energy-volume relationship of the reacted product gases. These expanding gases are primarily responsible for the overall energy of the burn, detonation, and blast so require accurate description. The HERMES model uses a tabular form of the EOS with four columns for specific energy density (cal g^{-1}), pressure (atm), mass density (g cm^{-3}), and temperature. The table is arranged in isotherms

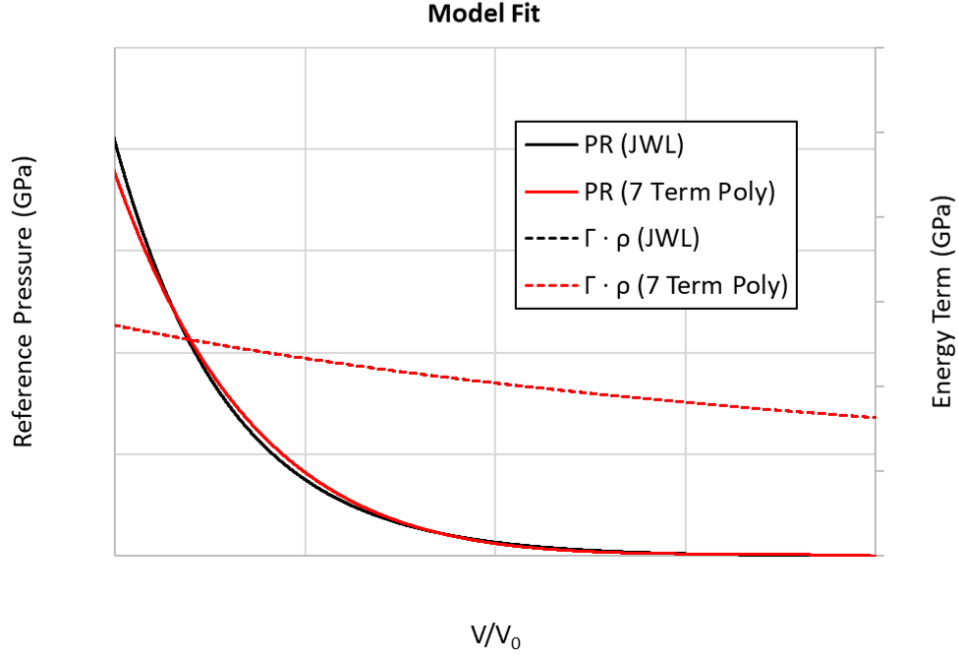


Figure 2: Polynomial EOS fit to JWL model. Solid lines represent principal adiabat pressure and dashed lines represent the energy term. Note that energy terms ($\Gamma \cdot \rho$) overlay each other.

with an initial temperature at or near ambient and density increasing from lowest to highest.

A table of EOS data was created using Cheetah, a thermochemical code from Lawrence Livermore National Laboratory [2], and consisted of 25 temperatures from 298K to 10,000K and 250 densities from $1e-5$ to 5 g/cm^3 . However, after plotting a number of isotherms, it was discovered that the model contained van der Waal loops at low temperatures in expansion. van der Waal loops are areas with non-monotonic pressure-volume relationships that indicate a vapor-liquid coexistence region. While physical, these tend to lead to code crashes as the pressure cannot converge. To rectify these areas, corrections known as Maxwell constructions were applied based on the method described in the Panda II manual [15]. In this method, points of mechanical equilibrium are computed on either side of the local maximum/minimum (known as spinodals) which have the same pressure and Gibb's free energy. Figure 3 illustrates the constructions for the product EOS.

The Maxwell constructions were applied using a Python script which is included in subsection A.2. That script outputs data in a format that HERMES can use and also a format that BCAT (Sandia EOS tool) can import for making SESAME tables.

Maxwell Constructions

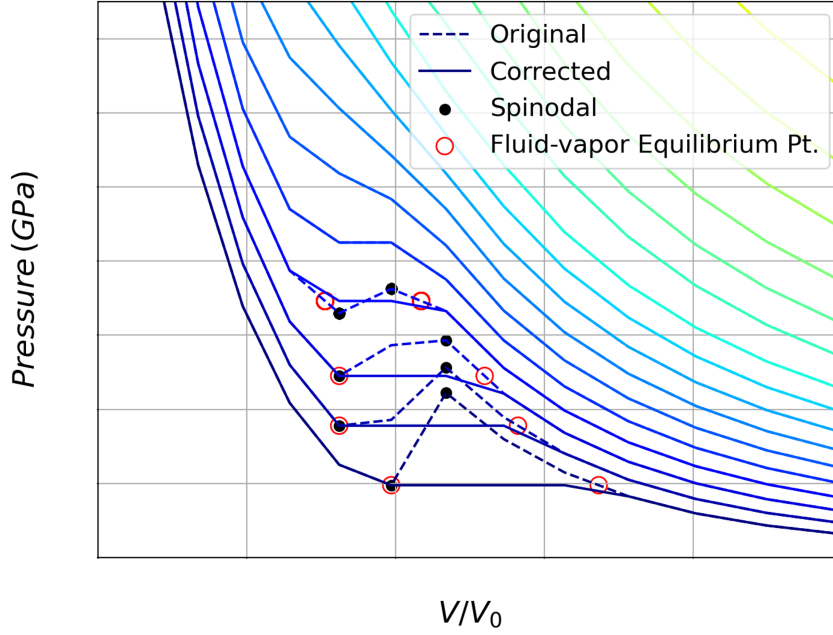


Figure 3: Maxwell constructions for the product equation of state.

2.3.3 Constitutive Model

HERMES uses a pressure and strain-dependent strength model that degrades with accumulated damage:

$$Y_e = Y_i f_e (1 - \Omega) + Y_r \Omega \quad (2)$$

where Ω is the damage measure ($0 =$ no damage, $1 =$ fully damaged), f_e is the strain hardening term, Y_r is the strength of the fully damage material and Y_i is the strength of the intact material. Strain hardening is computed with

$$f_e = \frac{\epsilon_p}{\epsilon_p + \epsilon_h} + C \left(1 - \frac{\epsilon_p}{\epsilon_p + \epsilon_h} \right) \quad (3)$$

where C and ϵ_h are parameters. The fully damaged strength is given by

$$Y_r = R Y_0 \left(\frac{A_r}{Y_0} + B_r \frac{P_m}{R Y_0} \right) \quad (4)$$

where A_r is the cohesion (pressure-independent) aspect, B_r is the pressure-dependent aspect, P_m is the pressure of the porous structure, and Y_0 is the characteristic yield strength. The parameter R is the strain rate factor which is calculated with

$$R = \left(1 + \frac{\dot{\epsilon}_p}{\dot{\epsilon}_0} \right)^r \quad (5)$$

where $\dot{\epsilon}_p$ is the plastic strain rate and r and $\dot{\epsilon}_0$ are parameters.

The intact strength is given by

$$Y_i = \begin{cases} RY_0 \left[\sqrt{s_0 + m_0 \frac{P_m}{RY_0} (m/6)^2} - (m/6) \right] & , P_m \geq 0 \\ Y_i(0) + \frac{3[Y_i(0) - Y_t]}{Y_t} P_m & , P_m < 0 \end{cases} \quad (6)$$

where $Y_t = b_{ten} Y_0$ and the input parameter b_{ten} is the ratio of uniaxial tensile strength to uniaxial compressive strength.

For this effort, the fully damaged residual strength model was not calibrated due to lack of data. The strain rate factor, Eq. 5, was fit to uniaxial compression data at four different rates and is shown in Figure 4.

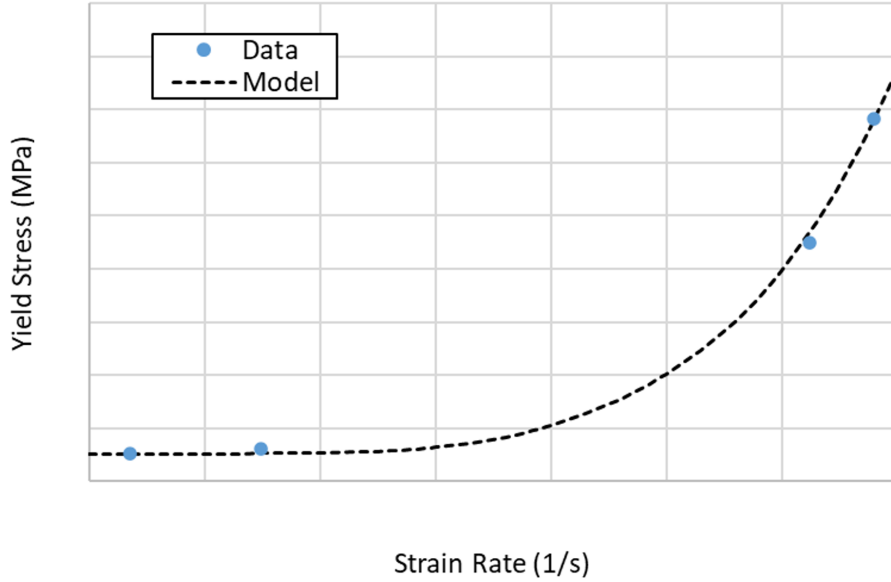


Figure 4: Yield strength as a function of strain rate.

The intact yield strength and strain hardening, $Y_i f_e$, was calibrated to triaxial compression data using single element compression calculations and a genetic algorithm. ALE3D was used to compress a single element at a reference rate of 1 s^{-1} that was initially preloaded to the prescribed hydrostatic pressure. Ideally, the simulations would comprise of a multi-element mesh that matched the geometry of the samples in the experiment. However, HERMES is an explicit model so loading at a rate of 1 s^{-1} results in long calculation times, which makes optimization difficult. The optimization used 60 children per generation and ran for 100 generations. Results of the optimal parameter set for the unconfined compression test is given below in Figure 5.

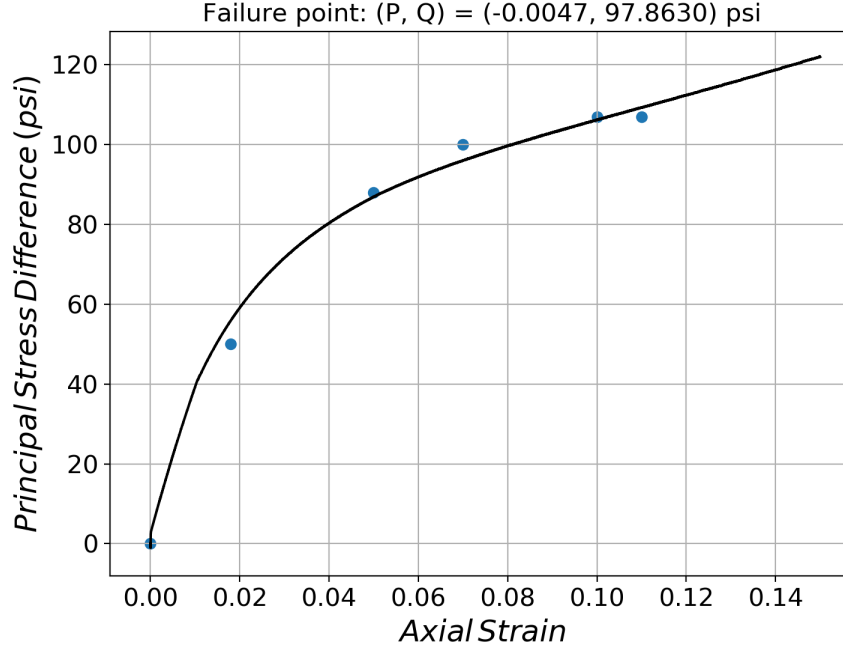


Figure 5: Optimized strength model response (black solid line) compared to data (blue dots) for unconfined compression.

The optimizer scripts leverage the Python `Multiprocessing` package to run multiple simulations in parallel and the `Pickle` package for saving, logging, and restarting. All files associated with this genetic algorithm optimizer are given in [subsection A.3](#).

2.3.4 Fragmentation Model

Fragmentation is described using a model that relates equivalent plastic strain to a ratio of surface area to volume:

$$S/V = A(\max[0, \epsilon_p - \epsilon_0])^n, \quad (7)$$

where A , ϵ_0 , and n are parameters. Once material has exceeded an initial plastic strain, ϵ_0 , the fragmentation damage increases with increasing strain. Fitting this model is done using shotgun friability test data. In this test, cylindrical samples of explosive are shot against a solid wall with increasing velocities and the fragments collected. Determining S/V is typically done indirectly, by burning the resulting fragments in a bomb calorimeter and measuring the burn rate (increased surface area directly correlates to increased burn rate). The fragmentation model was calibrated by running a series of ALE3D calculations to simulate shotgun tests. Each specimen was 2 cm long by 1.73 cm in diameter with an applied initial velocity, and the simulations were run in 2D axisymmetric space using a mesh resolution of 50 elements per centimeter. [Figure 6](#) shows an example of the sample after impact.

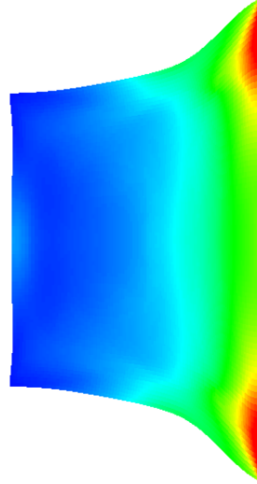


Figure 6: Simulated plastic strain within the sample after impact.

The average plastic strain (at the end of the simulation) was collected for each impact velocity and then (manually) fit to existing friability S/V data. The resulting calibration is shown in Figure 7. Note that the decrease in average plastic strain for the highest impact velocities is due to simulations crashing early from mesh entanglement. If run to completion, the average plastic strain would plateau around the peak value illustrated in the plot.

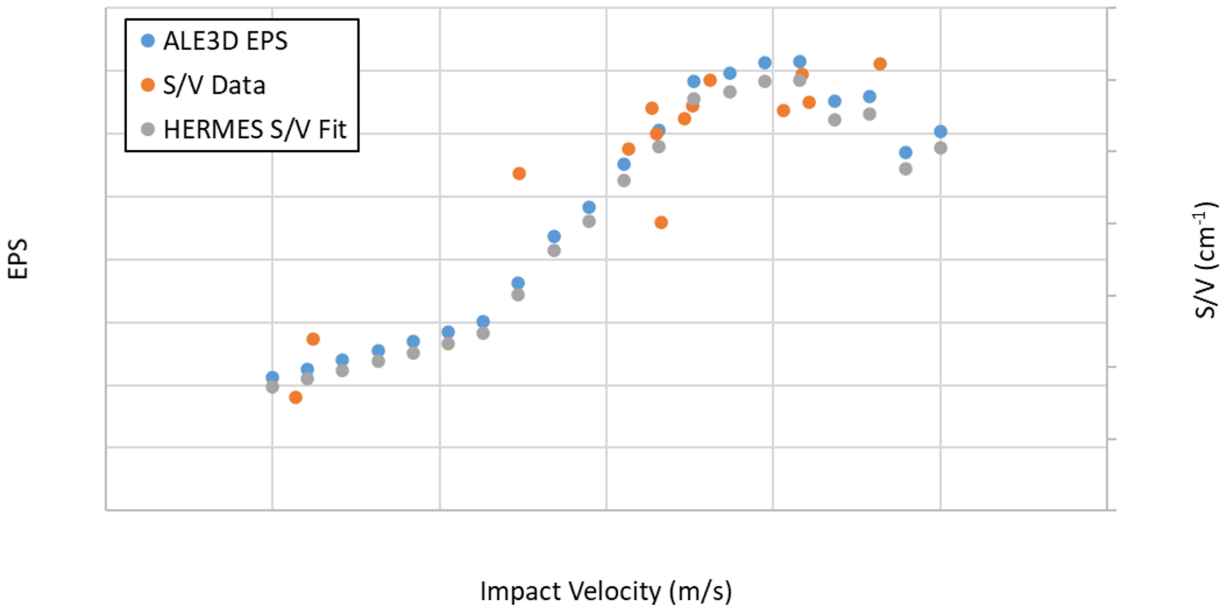


Figure 7: Calculated plastic strain and S/V fits versus impact velocity.

2.3.5 Ignition Model

Ignition is controlled with the following equation:

$$Ignit = \int_0^t \left(\frac{p - s_2/2}{\sigma_n} \right)^{pn} \left(2 - \frac{27|s_1 s_2 s_3|}{4Y^3} \right)^{ps} \dot{\epsilon} dt, \quad (8)$$

where p is pressure, s_i are the principal stress deviators, and Y is the yield stress. The parameters, σ_n , pn , and ps , control how ignition starts from pressure and shear loading and need to be calibrated to test data. This model was not calibrated for this effort due to lack of data. However, planned AFRL HEST tests and AFLCMC/EBD dynamic fill tests can be used for future calibration.

2.3.6 Burn Model

Subsonic burn is captured using a pressure-dependent laminar flame model. The model is described in terms of S/V and is given by

$$\dot{\lambda} = \frac{S}{V} v_b(P)(1 - \lambda)^{2/3} \quad (9)$$

where v_b is the pressure-dependent laminar burn speed, and the $2/3$ factor is meant to represent an inward burning of the fragment. The laminar burn speed is described with Saint-Robert's law:

$$v_b = v_0 p^n \quad (10)$$

where v_0 is the unit pressure burn speed and n is the pressure exponent. This rate was fit to closed bomb calculated values using a least squares method and is shown in [Figure 8](#). The data exhibits two distinct slopes but, unfortunately, the form of the model allows for only a single slope.

2.3.7 Detonation Model

HERMES uses a simplified version of the CREST reactive flow model, known as CREST Lite, to describe supersonic burn (i.e. detonation). The model uses a two-rate reaction based on Z , a function of entropy, and is given by

$$Z_s = \max(0, Z - c_{13}) \quad (11a)$$

$$b_1 = c_0 Z_s^{c_1} \quad (11b)$$

$$b_2 = c_2 Z_s^{c_3} \quad (11c)$$

$$m_2 = c_{10} Z_s^{c_{11}} (1 - \lambda) b_2^{-0.5} \quad (11d)$$

$$\dot{\lambda}_1 = (1 - \lambda_1) \sqrt{-2b_1 \ln(1 - \lambda_1)} \quad (11e)$$

$$\dot{\lambda}_2 = \lambda_2 (1 - \lambda_1) \sqrt{2b_2 [\lambda_1 b_2 / b_1 - \ln(1 - \lambda_1)]} \quad (11f)$$

$$\dot{\lambda} = m_2 \dot{\lambda}_2, \quad (11g)$$

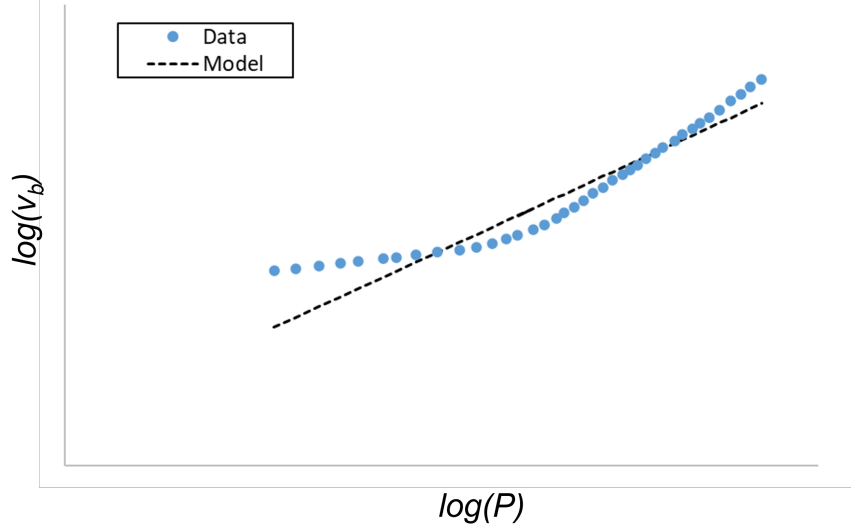


Figure 8: Linear burn rate as a function of pressure.

where c_i values are parameters. This model differs from the original CREST formulation [8] in that the final reaction rate, $\dot{\lambda}$, depends only on the slow reaction rate $\dot{\lambda}_2$, rather than both the slow and fast.

To calibrate this model, another genetic algorithm optimization approach was employed. Each set of c_i parameters were evaluated by running a 1D shock-to-detonation simulation using ALE3D. The simulated run distance was computed using a bi-linear fit method as illustrated in Figure 9 below.

The model was calibrated to Pop Plot data from an AFRL wedge test series using 20 generations with a population of 40 parameter sets per generation. The resulting fit is given in Figure 10.

The Python code for the genetic algorithm is provided in subsection A.4.

3 Mesoscale and Constituent Studies

Understanding mechanical initiation of energetics requires investigating behaviors at the mesoscale. Here, mesoscale is defined as the grain scale of an explosive. In this work, a number of models were compiled or developed for materials that make up the explosive of interest and then some initial studies were performed.

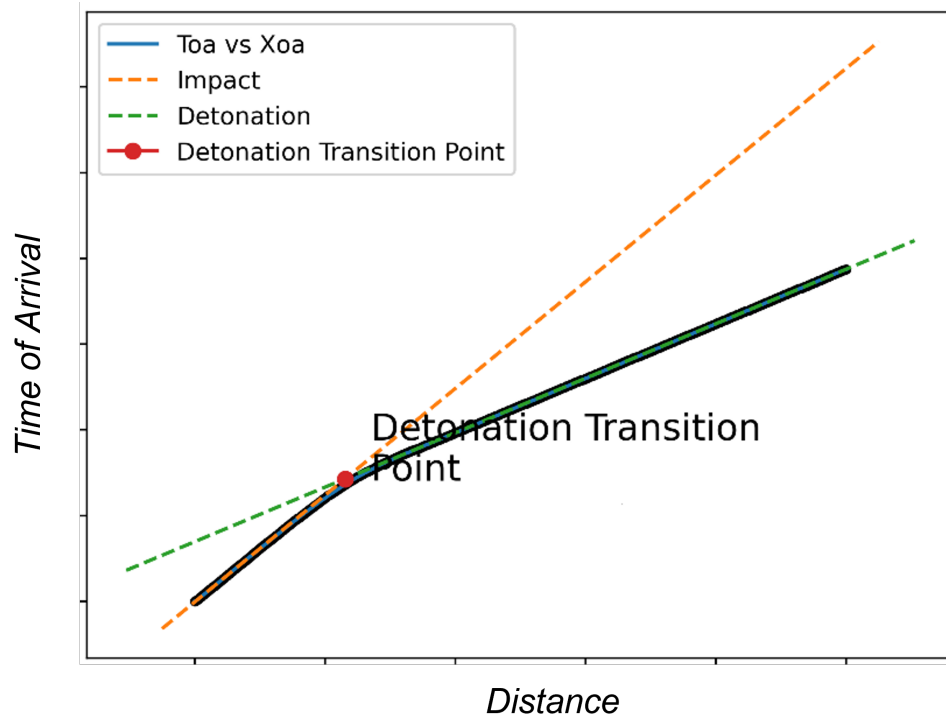


Figure 9: Run distance calculation using bi-linear fits.

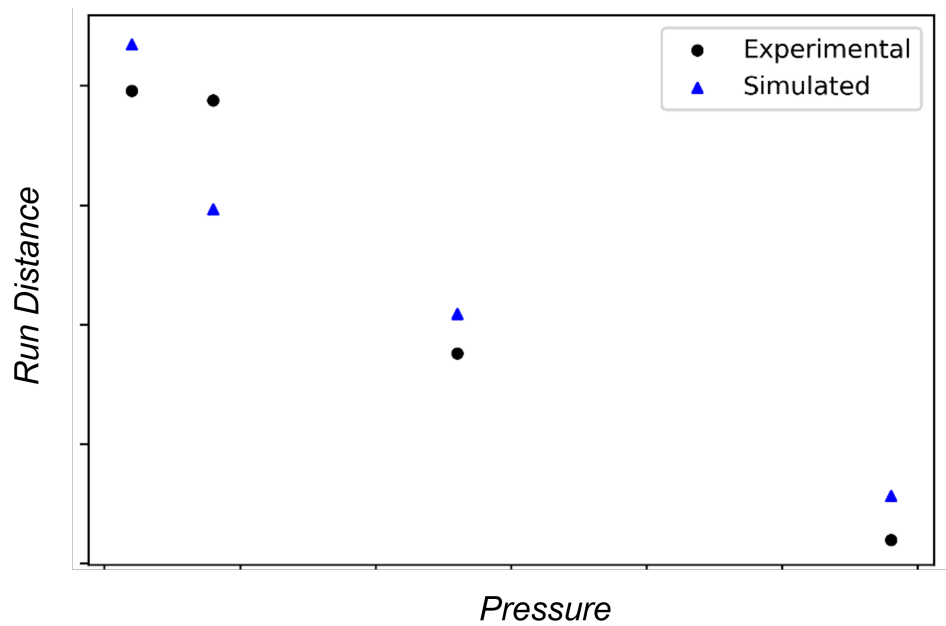


Figure 10: Optimized model Pop Plot fit.

3.1 Constituent Models

Various models are required to describe the behavior of the constituent materials for mesoscale calculations. Ideally, each constituent material would have models describing the continuum-scale behavior of interest: material state, ignition, burning, strength, fracture, etc. However, this turned out to be a monumental amount of work for the material of interest, so only a few models were developed in this effort. Others were leveraged from existing literature or other research projects.

3.1.1 RDX

The main energetic of the material of interest is RDX. Fortunately, adequate models exist for this material. Under this effort, an ARL strength model [5] was implemented in ALE3D and tested. A complete equation of state from LANL [3] was also implemented and tested. Both α - and γ -RDX models were acquired from LANL (in SESAME format), but only the α model was implemented, which is more appropriate for the low pressure, mechanical insults expected for this work. Ideally, a multi-tabular model would be used with a solid-solid phase transition.

3.1.2 Aluminum

Powdered aluminum comprises a significant portion of the material. Here, again, adequate models exist for this material. For equation of state, a tabular (SESAME format) model from G. I. Kerley [14] that is included in the CTH material database was used. For strength, a Steinberg-Guinan-Lund model for 1100-O alloy was employed [29]. Various burn models exist for aluminum, but these were not implemented due to time limitations.

3.1.3 Ammonium Perchlorate

Ammonium perchlorate (AP) is used as an oxidizer within the material of interest. Due to its extensive use in propellant, AP is well studied across a range of temperatures and pressures. A complete equation of state has been developed by researchers at Washington State University [32], but this model was fit to data at lower temperatures and pressures than is required for this project. As part of this work, an optimization framework was created to develop a complete equation of state for AP.

The EOS model follows an approach taken by Menikoff [20] that uses a generalized Hayes EOS. The Hayes EOS is based on Helmholtz free energy of the form

$$F(V, T) = - \int_{V_0}^V P_0(V) dV - \theta(V) \int_0^{T/\theta} \frac{1}{\tilde{T}} \left(\frac{T}{\theta(V)} - \tilde{T} \right) C_V(\tilde{T}) d\tilde{T} \quad (12)$$

where P_0 is the cold curve, C_V is temperature-dependent specific heat, and $\theta(V)$ is the Debye temperature. These three functions are the aspects that were optimized in this work. For

the cold curve, a Birch-Murnaghan form was assumed:

$$P_0(V) = \frac{3}{2}K_0 \left[(V/V_0)^{-7/3} - (V/V_0)^{-5/3} \right] \left[1 + \frac{3}{4}(K'_0 - 4)[(V/V_0)^{-2/3} - 1] \right], \quad (13)$$

where K_0 and K'_0 are parameters. The specific heat is assumed to have the form

$$C_V(\tilde{T}) = \frac{C\tilde{T}^3}{c_0 + c_1\tilde{T} + c_2\tilde{T}^2 + c_3\tilde{T}^3}, \quad (14)$$

where parameters C has units of specific heat and c_i are unitless. The Debye temperature was given a general rational function form to capture a wide range of behaviors:

$$\theta(V) = \theta_0 \frac{A_0 + A_1(V/V_0) + A_2(V/V_0)^2 + A_3(V/V_0)^3 + A_4(V/V_0)^4}{B_0 + B_1(V/V_0) + B_2(V/V_0)^2 + B_3(V/V_0)^3 + B_4(V/V_0)^4}, \quad (15)$$

where A_i and B_i are unitless and $\theta_0 = 1K$ for consistent units. The Helmholtz free energy is used to compute pressure and internal energy using thermodynamic relations:

$$P = -\frac{\partial F}{\partial V} \quad (16)$$

and

$$e = F - T \frac{\partial F}{\partial T} \quad (17)$$

Once all models are defined, a table of P , e , V , and S (entropy) values is created and converted into a SESAME binary file for evaluation. For the optimization, the model was evaluated against the following data:

- Isothermal compression data from [21] and [16]
- Shock Hugoniot data from [32]
- Ambient pressure isobar data from [22]
- Isothermal compressibility at ambient conditions from [9]
- Constant pressure heat capacity data from [31] and [4]

The EOS tool BCAT was used to read in the SESAME file and compute the necessary thermodynamic values. The process utilized some useful thermodynamic relations including the coefficient of thermal expansion

$$\alpha = \frac{1}{V} \left(\frac{\partial V}{\partial T} \right)_P, \quad (18)$$

isothermal compressibility

$$\beta_T = -\frac{1}{V} \left(\frac{\partial V}{\partial P} \right)_T, \quad (19)$$

and the relationship between specific heats

$$C_P - C_V = \frac{\alpha^2 T}{\rho \beta_T}. \quad (20)$$

The optimization was again implemented using DEAP and was run for 250 generations with 100 children per generation. The best fitting results are given in [Figure 11](#) - [Figure 15](#) below. As seen in the plots, there appears to be some errors associated with the thermal response at high temperatures. This is likely due to the functional form of the Debye temperature, [Eq. 15](#), which covers an extensive range of behaviors, most of which are nonphysical.

Future studies will adapt the Debye temperature to ensure that the Grüneisen parameter is continuous, smooth, and physical. All code used for the optimization is provided in [subsection A.5](#).

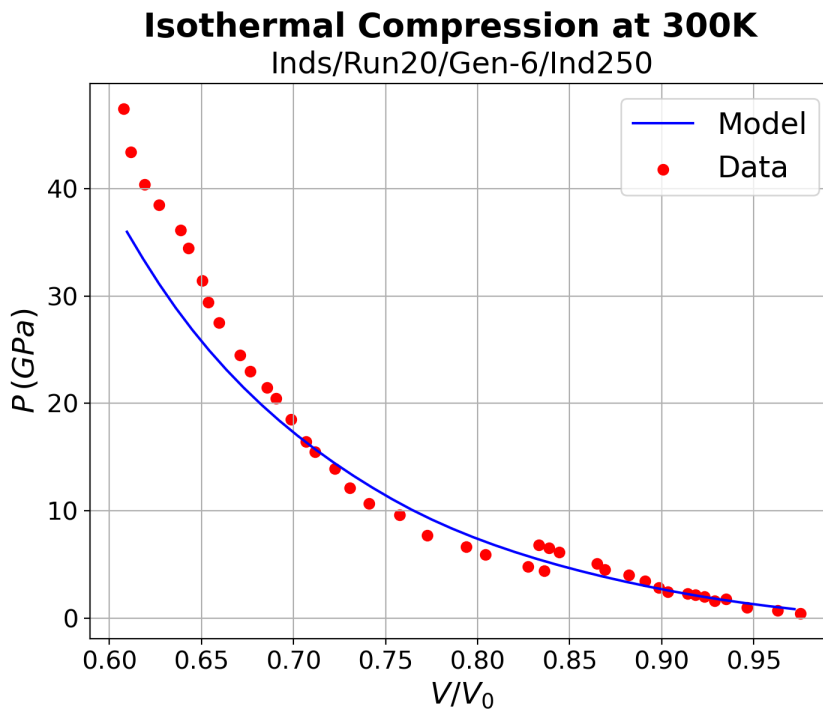


Figure 11: Isothermal compression comparison between data and model fit.

3.1.4 HTPB

The hydroxyl-terminated polybutadiene (HTPB) binder represents the largest uncertainty for the model since its behavior can change drastically with variations in formulation (namely, the amount of plasticizer) and age. Work was begun on a complete equation of state following the same procedure as was used for AP. Isothermal compression data was obtained from [\[1\]](#) (shown in [Figure 16](#) below). A limited dataset of specific heat versus temperature was

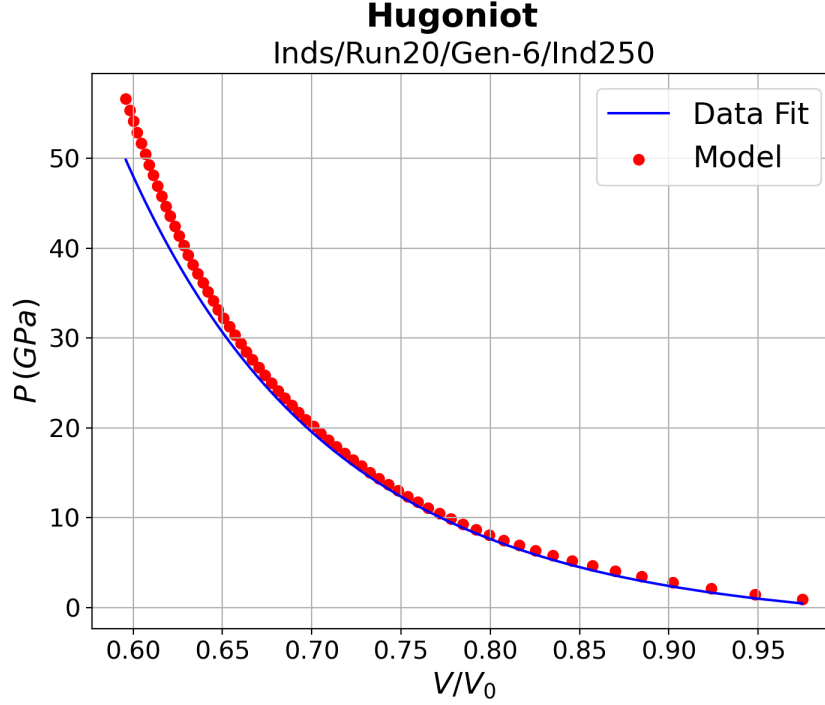


Figure 12: Hugoniot comparison between data and model fit.

obtained from [28]. Despite limited non-shock and thermal data, a large amount of high-pressure, shock data exists for this material. References [10], [7], and [12] provide shock data derived from experiments or molecular dynamic simulations.

3.2 Mesoscale Studies

Some initial mesoscale utilities were developed for planned complete equation of state development project. Aspects needed are representative volume elements (RVE) that capture the realistic grain structure of the material of interest, a method for investigating the EOS behavior of the mesoscale sample, and material models for the constituent materials. Not all material models were complete at the end of this YIP project, but methods were developed for creating RVEs and subjecting those samples to controlled loading. These methods are described in the sections below.

3.2.1 Sample Generation

Mesoscale simulations require representative volume elements (RVE) or numerical samples of the grain structure that accurately represent the material of interest. In this effort, scripts were created to automatically generate RVE samples using ParticlePack, a particle packing tool for Lawrence Livermore National Laboratory. The scripts were then used to create numerous samples for varying levels of porosity. The steps taken are described below.

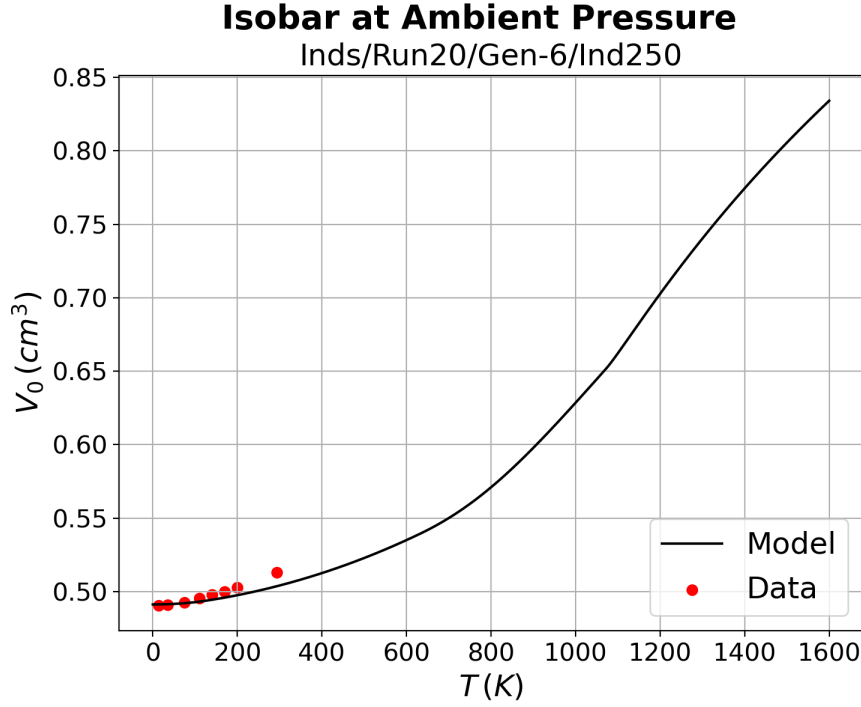


Figure 13: Isobaric compression comparison between data and model fit.

Creating initial RVE Generating a statistically representative set of numerical samples requires capturing the particle size distributions of each constituent material. In the script `rve_generate.py`, included in [subsection A.6](#), each grain (RDX and AP) and sphere (aluminum) have sizes sampled from a Gaussian distribution. Further, the volume fraction that each material encompasses is also sampled from a Gaussian distribution. Although many materials exhibit non-normal particle size distributions, this current method should capture the general trends in sizes and can be updated in the future.

Converting to STL Once all the ParticlePack calculations are complete, the actual geometries are generated using ALE3D. The script `run_gen3d.py`, included in [subsection A.7](#), is a useful utility for doing this automatically. ALE3D creates VisIt plot files which can then be converted to STLs. The script `Visit_STL_gen.py`, in [subsection A.8](#), is used for this.

Converting to Exodus Some mesoscale analyses require finite element analysis (FEA) for investigating low strain behavior, which requires a meshed geometry. To do so, a script was created that converts the STL geometries into ExodusII mesh format using the Sculpt tool within Cubit (SNL meshing tool). The resulting mesh, in ExodusII format, is structured with hexahedral elements, sidesets are defined for each boundary, and blocks are defined for each material (one per STL material). [Figure 17](#) gives an example of a mesh created with this process.

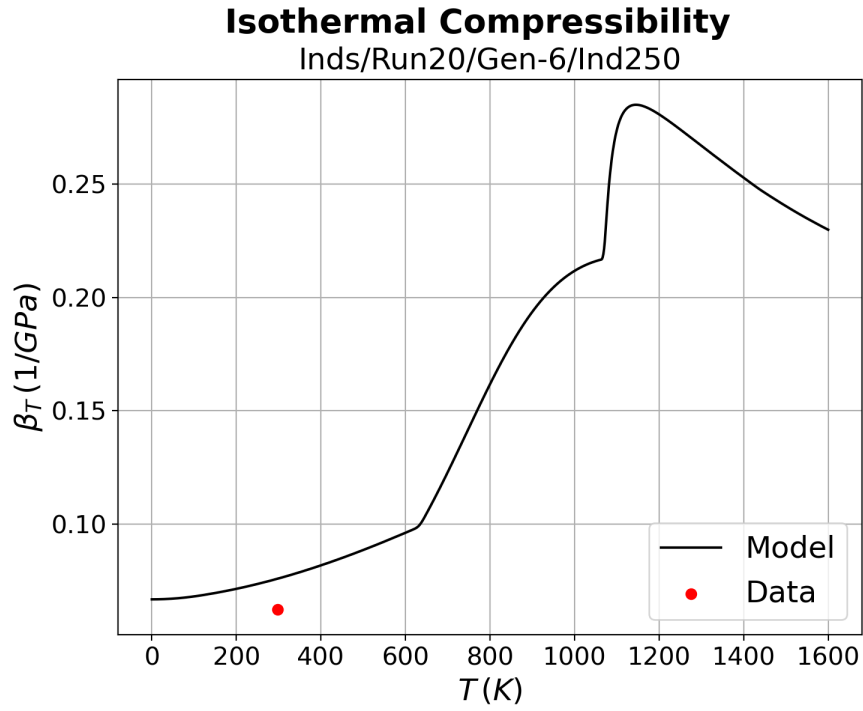


Figure 14: Isothermal compressibility comparison between data and model fit.

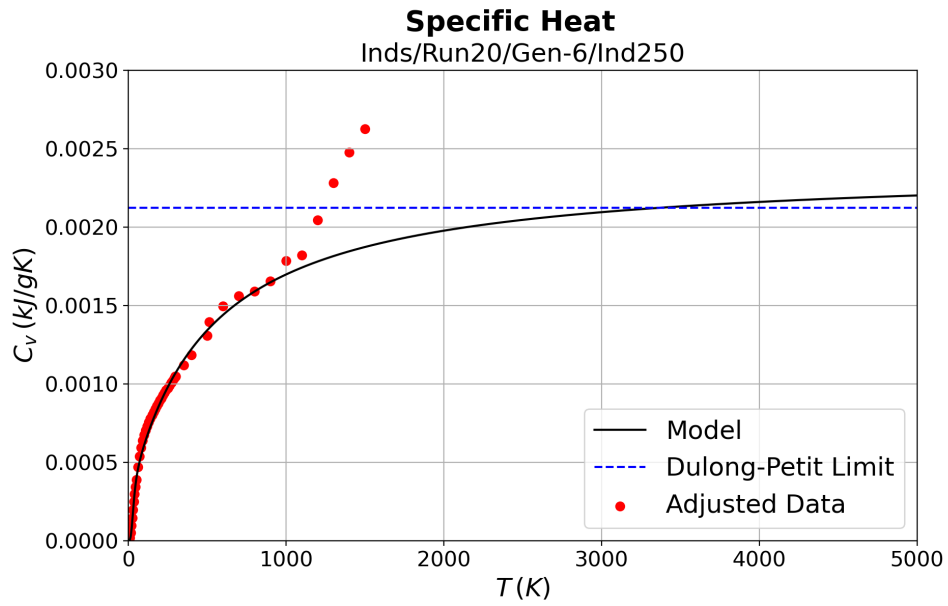


Figure 15: Specific heat comparison between data and model fit.

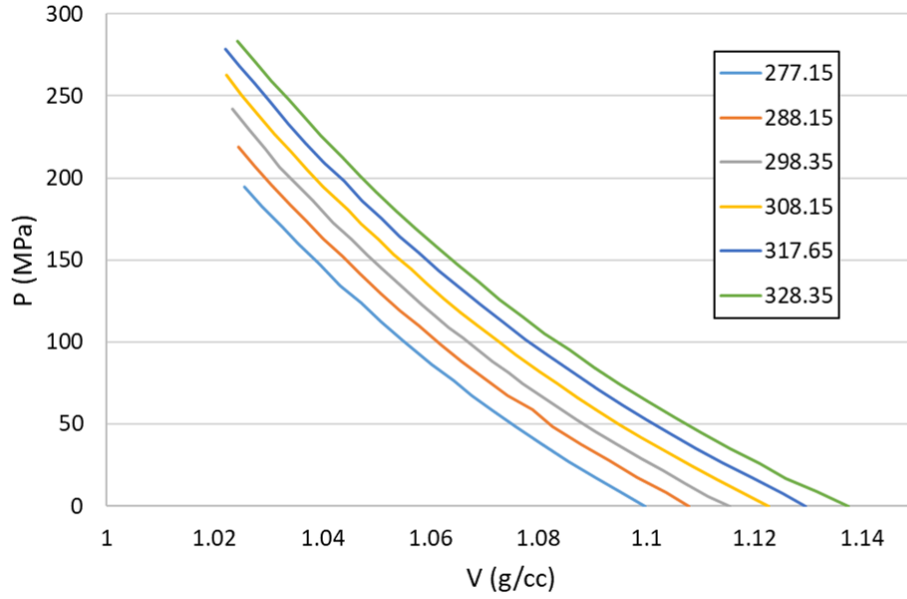


Figure 16: Isothermal compression of HTPB at various temperatures (in Kelvin).

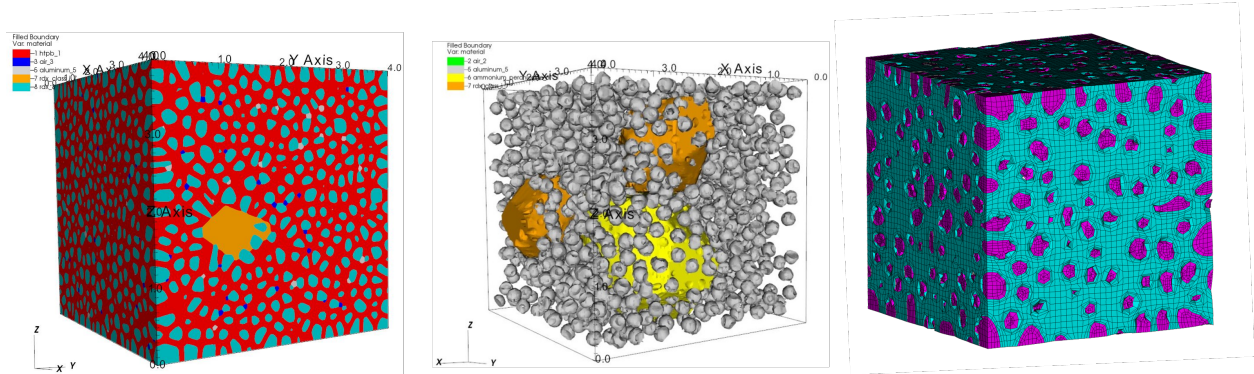


Figure 17: RVE sample generation. Left image shows full shaped geometries, middle image shows same geometry but without the binder, and right image shows RVE after conversion to FEA mesh.

3.2.2 Adiabatic Compression

The HERMES model requires an unreacted equation of state based on adiabatic compression data. In the absence of experimental data, mesoscale simulations can be used. Towards this end, a framework was created to adiabatically compress mesoscale samples, measure the average state histories, and create a fit to the quadratic Grüneisen model (Eq. 1). The tool uses STL particle samples created using the scripts above and CTH to simulate the compression.

Hydrodynamic compression is achieved by placing water around the RVE sample and heating it, which causes it to expand and squeeze the specimen. Heat conduction is not applied, so heat from the surrounding water does not affect the sample measurements. Symmetry boundary conditions are applied on all sides to prevent heat or mass loss. One hundred tracers are placed evenly throughout the sample to measure pressure, temperature, and density over time. These state data are averaged over all tracers and then further averaged over all samples in the set. Finally, the quadratic equation of state is fit to the resulting state data. The Grüneisen parameter is fit using the thermodynamic relation:

$$\frac{\Gamma}{V} = -\frac{1}{V} \frac{\partial \ln T}{\partial \ln V} = -\frac{\partial \ln T}{\partial V} = \rho_0(\mu + 1)^2 \frac{\partial \ln T}{\partial \mu}. \quad (21)$$

This process was applied to a test problem using a material similar to the material of interest but with different constituents. Two samples were run in CTH (as seen in [Figure 18](#), for example) and the resulting state histories ([Figure 19](#)) were then used to fit an EOS model ([Figure 20](#)). The post-processing script for fitting the unreacted EOS is given in [subsection A.9](#).

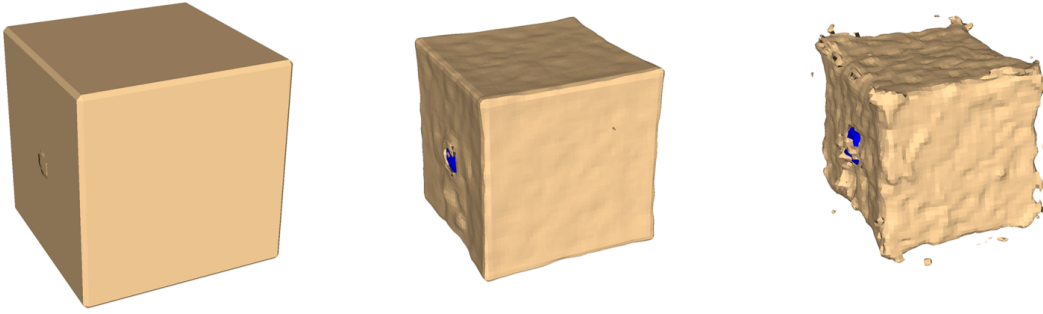


Figure 18: CTH adiabatic compression calculation. Surrounding water is not illustrated.

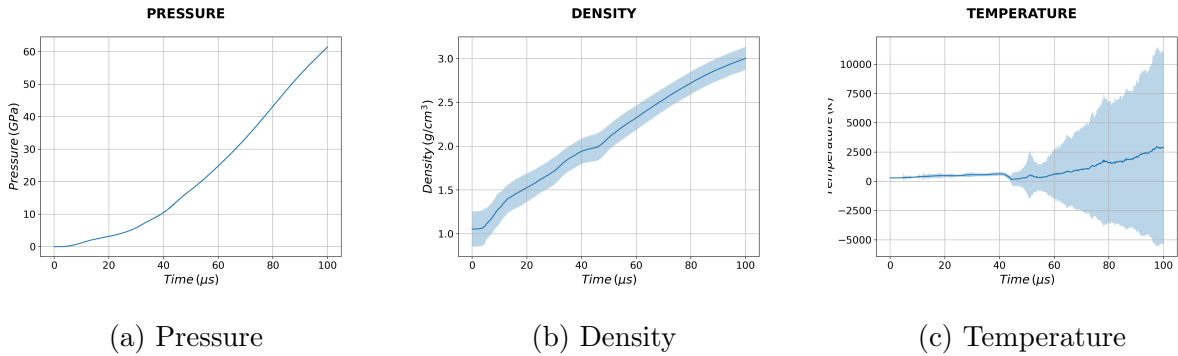


Figure 19: State variables over time. Solid line represents average.

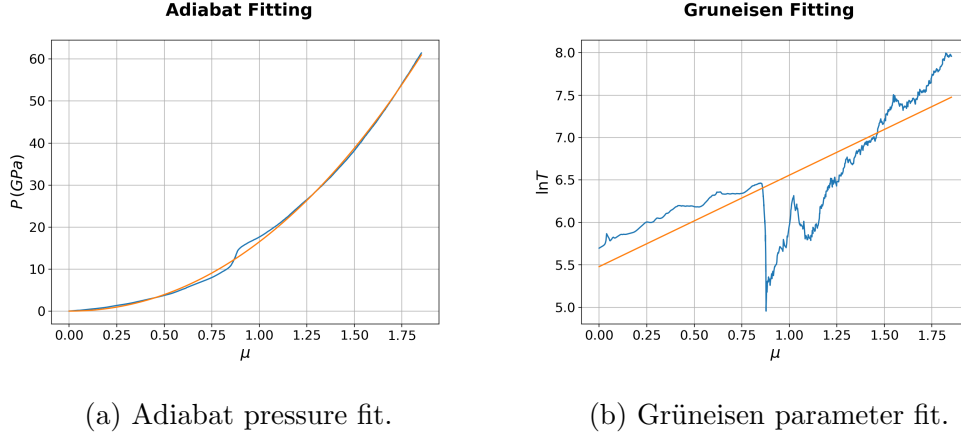


Figure 20: EOS model fit to computed states.

4 Fill Displacement Diagnostic

Predicting the behavior of explosive fill within a warhead during a penetration event not only requires accurate models, but also an understanding of the actual environment the fill experiences. The goal of this study was to create a high-speed sensor package to detect linear displacement within a penetrator's fill material. The sensor package should be capable of detecting linear motion between 0 and 1.5 inches, should have an update rate of less than 10 microseconds (100,000 samples per second), and able to survive the extreme pressures and shock expected during impact.

4.1 Background

Constitutive models for explosive fills are often calibrated to laboratory scale experiments at quasi-static conditions, i.e. triaxial compression, tension, and shear. Moderate strain rate effects can be estimated using split-Hopkinson bar tests, but these neglect compression (pressure) effects. The conditions in a penetration event exhibit large pressures (which have been measured), but the actual deformation within the fill is difficult to measure directly. Such data would greatly improve model accuracy for important behaviors such as fill sliding across internal features, void collapse, and fuzewell loading.

There are numerous methods for measuring linear displacement including potentiometers, Hall effect sensors, piezo-electric transduction, capacity sensors, inductive sensors, etc. For this application, the sensor is required to measure at high sampling rates (> 100 kHz), survive large shocks and pressures, and remain accurate over a potentially large range of linear movement.

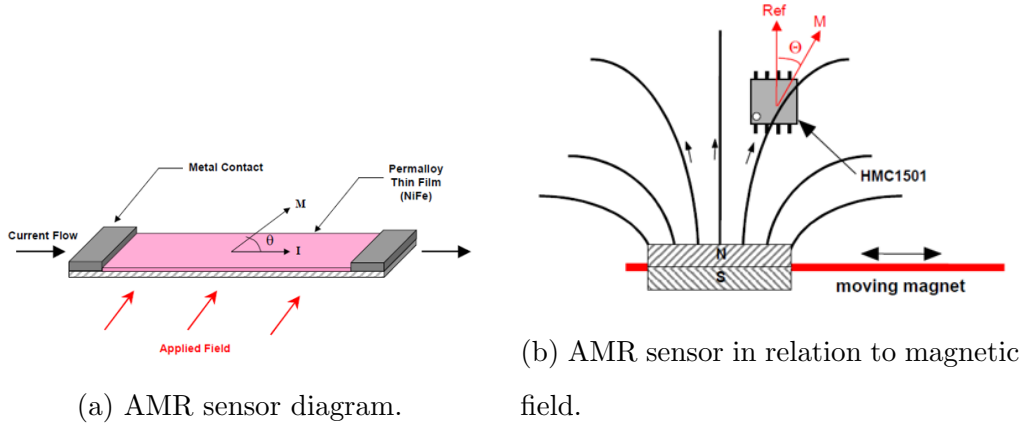


Figure 21: AMR sensor operation (Honeywell HMC1501).

4.2 Sensor Selection

After studying the different commercially available sensors, the team decided to pursue anisotropic magnetoresistive (AMR) sensors. These sensors rely on the anisotropic magnetoresistance property that exists in certain types of ferrous materials. The AMR effect is when a material's electrical resistance changes with respect to the angle between the direction of electric current and direction of magnetization. The approach of this method is to embed a magnet within the fill and track its linear movement using a series of AMR sensors. The Honeywell HMC1501 AMR sensor was chosen for this project as it met all requirements and was reasonably priced. Diagrams showing how the sensor works (and AMR effect in general) are given in [Figure 21](#) above.

The HMC1501 has a linear measurement range of -45° to 45° as shown in the voltage response plot in [Figure 22](#). It uses an integrated Wheatstone bridge to provide an angle resolution of 0.07° .

4.3 Prototype Development

A initial prototype sensor was created using a series of AMR sensors to extend the range of displacement measurement, as illustrated in [Figure 23a](#). The number of sensors was initially set at three for this prototype, but theoretically can be increased to capture larger displacements.

Reading the AMR outputs ([Figure 23b](#)) and translating to a displacement requires converting the analog voltage to digital data. This is done using an instrumentation amplifier to amplify the 120 mV signal prior to sending it to a microprocessor with an analog to digital converter (ADC) ([Figure 24](#)). The microcontroller chosen for this effort, the Teensy 3.5, has two ADC's and its own multiplexer for combining the separate channels into a single output.

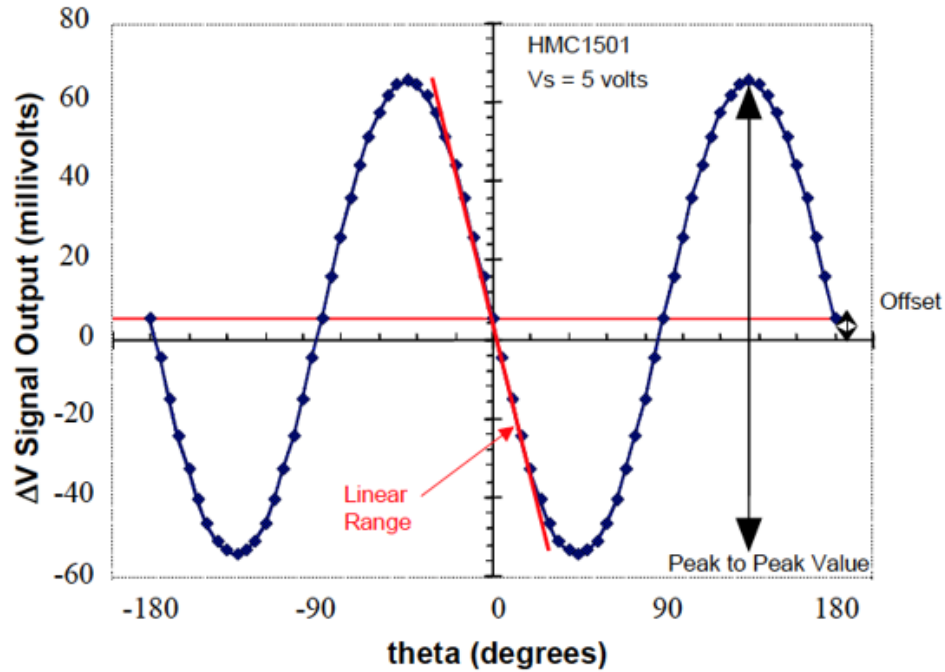


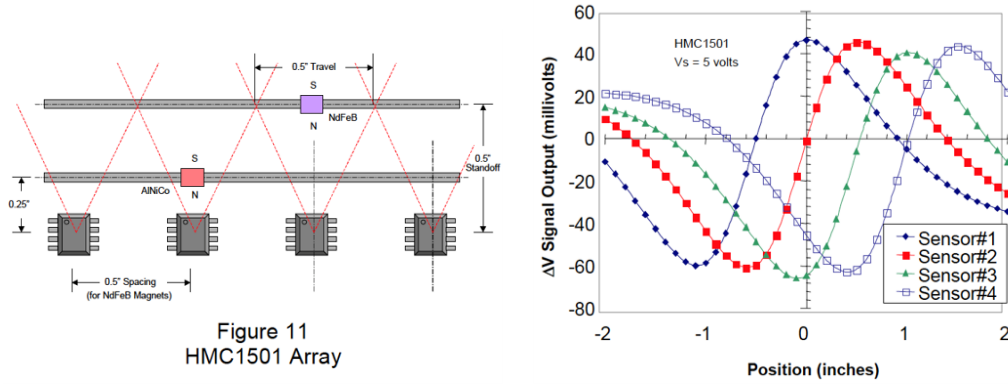
Figure 22: HMC1501 voltage response plot.

It is also inexpensive and has fast processing speeds.

The three sensor array can read sensor data, triangulate position, and output corresponding voltage every 5.7 to 6.7 microseconds. An algorithm was developed to estimate the linear position of the witness magnet using outputs from all three AMR sensors. When first implemented on the Teensy microcontroller, the algorithm took about 276.8 microseconds to get through the loop - much too slow for the application. After investigation, it was determined that this was due to slow library trigonometric functions within the Teensy and slow ADC functions for reading the pins. A library was found that optimized the use of ADCs, and using that, the loop time was reduced to 52 microseconds - still five times too slow. Finally, the trigonometric library functions were replaced with Taylor series approximations inline in the code which resulted in a loop time of about 7 microseconds, well within the target time.

4.4 Bench Testing

The prototype was first tested using a breadboard configuration and bench-level experiments. This included measuring known distances in static tests, comparing dynamic measurements to an existing Honeywell COTS sensor, and testing in the presence of ferrous metal (some images of testing are seen in [Figure 25](#)). Accurate measurements in the presence of ferrous metal is a requirement as the sensor will be mounted adjacent to a steel penetrator case. The results of the testing showed that the signals are slightly altered in the presence of ferrous



(a) Linear array of AMR sensors.

(b) Transfer functions for array.

Figure 23: AMR sensor array.

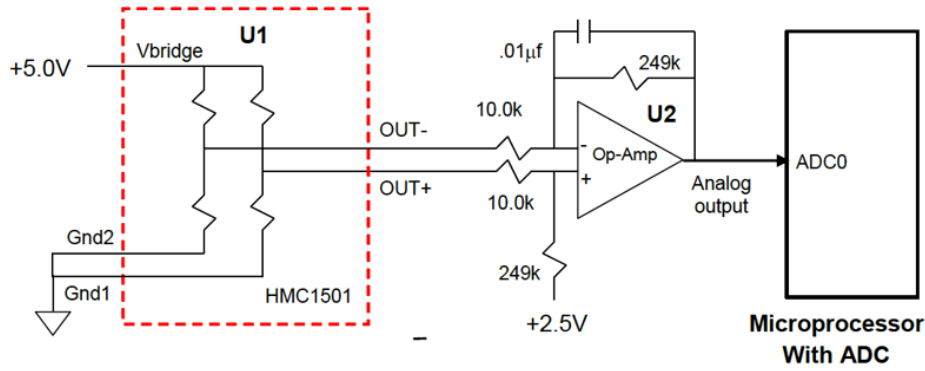


Figure 24: Single sensor circuit layout.

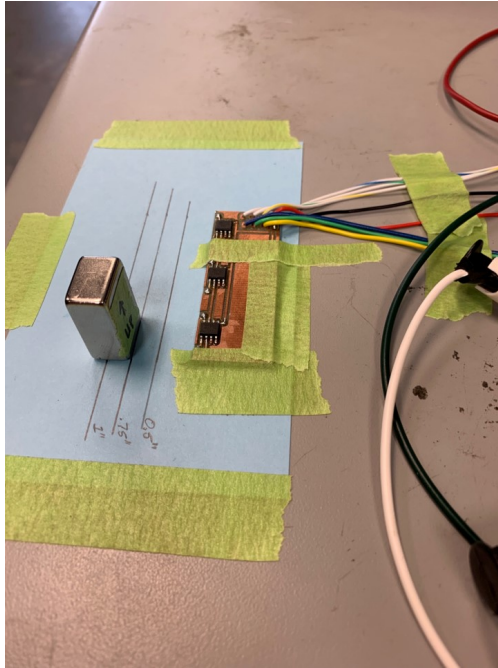
metal (as shown in Figure 26), but the resulting displacement measurement is unaffected.

4.5 Subscale Penetration Testing

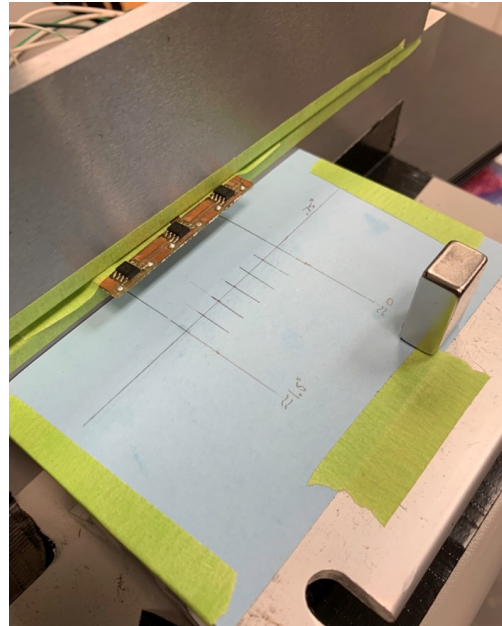
Once the schematic was complete from breadboard testing, Ultiboard was used to design a printed circuit board. The board was populated with components using soldering techniques common with penetrator fuze design. The final prototype is shown below in Figure 27.

Hardware was designed for mounting the sensor to a subscale penetrator (known as a SPEAR). It was originally fabricated out of G10 epoxy/fiberglass laminate (as seen in Figure 28), but an updated model was later fabricated from stainless steel. Not shown in the figure is the epoxy fill. Prior to testing, the entire cavity is filled with potting epoxy under vacuum for shock hardening.

Initially, a test series was identified for which this effort could “piggyback” with. The

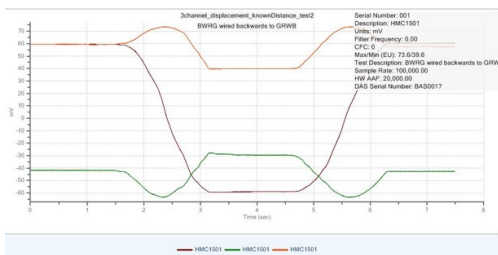


(a) Initial calibration.

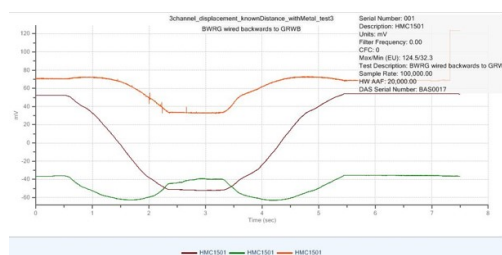


(b) Testing response near ferrous mass.

Figure 25: Prototype bench testing.



(a) Nominal conditions.



(b) In presence of ferrous metal.

Figure 26: Effects of ferrous metal presence on sensor signal outputs.

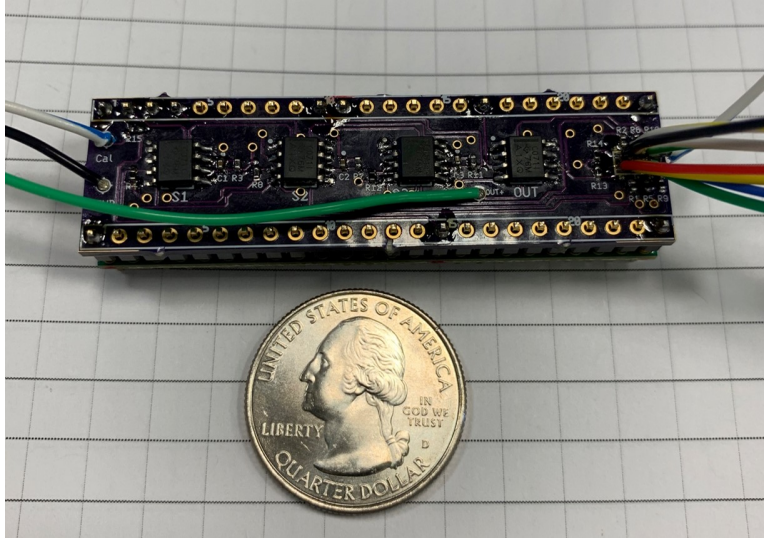


Figure 27: Completed prototype prior to mounting.

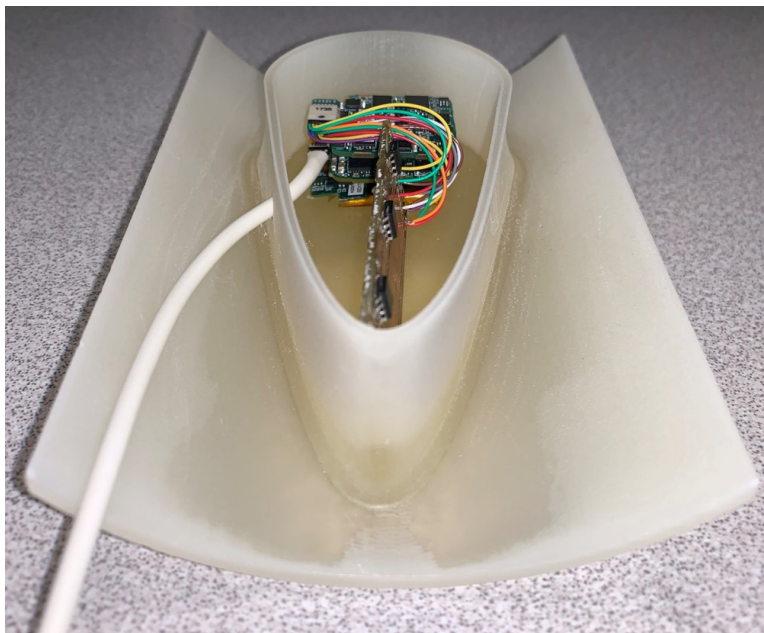


Figure 28: An early version of the sensor placed in mounting hardware (prior to epoxy fill).

program office gave permission as they were interested in pressure measurements in the nose and the displacement sensor would be aft of center offering little interference. Unfortunately, those tests, initially delayed due to Covid, were eventually canceled and carried out at a different site.

5 Impact and Future Work

The main accomplishment of this work is the development of workflow for calibrating HEVR models for non-ideal explosives. The method uses data obtained from standard Air Force explosive accreditation/safety tests and leverages information from mesoscale calculations to supplement needed data. While an initial model has been created for the material of interest, the methods and scripts have been generalized enough to be applied to any explosive fill. These models will provide DoD weapon designers a tool for evaluating conditions that may lead to fill failure, providing significant cost savings (less need for extensive testing) and improved operational success.

The fill displacement sensor is currently the only experimental method for measuring fill movement during penetration events. It has the potential to prove or disprove current theories regarding material behavior at the pressures and strain rates experienced in those extreme environments. It may also be useful for other applications that require fast distance measurements in high shock environments.

While much was accomplished during this project, a number of tasks were not completed due to lack of time, funding, or testing resources. These include:

1. Finalize complete equation of state for AP
2. Generate a complete equation of state for HTPB
3. Run mesoscale compression calculations ([subsubsection 3.2.2](#)) to determine continuum equation of state. Compare Hugoniot to existing shock experiments.
4. Update all HERMES submodels to include new unreacted EOS
5. Calibrate HERMES ignition model to AFRL HEST and AFLCMC/EBD dynamic fill tests
6. Validate fill displacement sensor with a cannon test

These may be addressed in future, related efforts.

References

- [1] J. W. Barlow. Measurement of the pvt behavior of cis-1,4-polybutadiene. *Polymer Engineering & Science*, 18(3):238–245, 1978.
- [2] S. Bastea and L. E. Fried. Cheetah 9.0 user’s manual. Report LLNL-SM-769619, Lawrence Livermore National Laboratory.
- [3] M. J. Cawkwell, D. J. Luscher, F. L. Addessio, and K. J. Ramos. Equations of state for the and polymorphs of cyclotrimethylene trinitramine. *Journal of Applied Physics*, 119(18):185106, 2016.
- [4] Jr. Chase, M.W. NIST-JANNAF thermochemical tables, fourth edition. *J. Phys. Chem. Ref. Data, Monograph 9*, 9:1–1951, 1998.
- [5] J. D. Clayton and R. Becker. Elastic-plastic behavior of byclotrimethylene trinitramine single crystals under spherical indentation: Modeling and simulation. Report ARL-RP-364, Army Research Laboratory.
- [6] Félix-Antoine Fortin, François-Michel De Rainville, Marc-André Gardner, Marc Parizeau, and Christian Gagné. DEAP: Evolutionary algorithms made easy. *Journal of Machine Learning Research*, 13:2171–2175, jul 2012.
- [7] Markus G. Fröhlich, Thomas D. Sewell, and Donald L. Thompson. Molecular dynamics simulations of shock waves in hydroxyl-terminated polybutadiene melts: Mechanical and structural responses. *The Journal of Chemical Physics*, 140(2):024902, 2014.
- [8] C. Handley, N. Whitworth, H. James, B. Lambourn, and M-A Maheswaran. The CREST reactive-burn model for explosives. *EPJ Web of Conferences*, 10, 01 2011.
- [9] S. Haussühl. Elastic and thermoelastic properties of isotypic KClO_4 , RbClO_4 , CsClO_4 , TlClO_4 , NH_4ClO_4 , TlBF_4 , NH_4BF_4 and BaSO_4 . *Zeitschrift für Kristallographie*, 192:137–145, 1990.
- [10] Justin B. Hooper, Dmitry Bedrov, Grant D. Smith, Ben Hanson, Oleg Borodin, Dana M. Dattelbaum, and Edward M. Kober. A molecular dynamics simulation study of the pressure-volume-temperature behavior of polymers under high pressure. *The Journal of Chemical Physics*, 130(14):144904, 2009.
- [11] G. R. Johnson and W. H. Cook. Fracture characteristics of three metals subjected to various strains, strain rates, temperatures, and pressures. 21:31–48.
- [12] J. J. Jordan, D. Montaigne, P. Gould, C. Neel, G. Sunny, and C. Molek. High strain rate and shock properties of hydroxyl-terminated polybutadiene (HTPB) and varying amounts of plasticizer. *J. Dynamic Behavior Mater.*, 2:91–100, 01 2016.
- [13] G. I. Kerley. CTH equation of state package: porosity and reactive burn models. Report SAND92-0553, Sandia National Laboratories.

- [14] G. I. Kerley. Theoretical equation of state for aluminum. 5:441–449.
- [15] G. I. Kerley. User’s manual for PANDA II: A computer code for calculating equations of state. Report SAND88-2291, Sandia National Laboratories.
- [16] Matthew P. Kroonblawd, Batikan Koroglu, Joseph M. Zaug, Philip F. Pagoria, Nir Goldman, Eran Greenberg, Vitali B. Prakapenka, Martin Kunz, Sorin Bastea, and Elissaios Stavrou. Effects of pressure on the structure and lattice dynamics of ammonium perchlorate: A combined experimental and theoretical study. *The Journal of Chemical Physics*, 149(3):034501, 2018.
- [17] J. L. Maienschein, E. L. Lee, J. E. Reaugh, C. I. Merrill, and R. R. Lambert. Modeling the impact response of booster propellants. 1997.
- [18] Ewing Matheson, Douglas Schaeffer Drumheller, and Melvin R. Baer. A coupled damage and reaction model for simulating energetic material response to impact hazards. 1999.
- [19] Ewing Matheson, John T. Rosenberg, and Lockheed Martin. The role of damage mode in delayed detonation of composite energetic materials. 2002.
- [20] R. Menikoff. Complete EOS for PBX 9502. Report LA-UR-09-06529, Los Alamos National Laboratory.
- [21] Suhithi M. Peiris, G. I. Pangilinan, and T. P. Russell. Structural properties of ammonium perchlorate compressed to 5.6 GPa. *The Journal of Physical Chemistry A*, 104(47):11188–11193, 2000.
- [22] H. J. Prask, C. S. Choi, N. J. Chesser, and G. J. Rosasco. Ammonium perchlorate structure and dynamics at low temperatures. *The Journal of Chemical Physics*, 88(8):5106–5122, 1988.
- [23] J. E. Reaugh. Calculating the dynamics of high explosive violent response (HEVR) after ignition. Report LLNL-TR-407915, Lawrence Livermore National Laboratory.
- [24] J. E. Reaugh. HERMES: A model to describe deformation, burning, explosion, and detonation. Report LLNL-TR-516119, Lawrence Livermore National Laboratory.
- [25] J. E. Reaugh. Implementation of strength and burn models for plastic-bonded explosives and propellants. Report LLNL-TR-412938, Lawrence Livermore National Laboratory.
- [26] J. E. Reaugh. Modifications and applications of the HERMES model: June-october 2010. Report LLNL-TR-462751, Lawrence Livermore National Laboratory.
- [27] J. E. Reaugh. Progress in model development to quantify high explosive violent response (hevr) to mechanical insult. Report LLNL-TR-405903, Lawrence Livermore National Laboratory.

- [28] Dr. Yogesh Sinha, B. Sridhar, and R. Kishnakumar. Study of thermo-mechanical properties of HTPB–paraffin solid fuel. *Arabian Journal for Science and Engineering*, 41, 06 2016.
- [29] Daniel Jeremy Steinberg, S. G. Cochran, and M. W. Guinan. A constitutive model for metals applicable at high-strain rate. *Journal of Applied Physics*, 51:1498–1504, 1980.
- [30] S. N. Todd and T. L. Caipen. Damage initiated reaction model: numerical results. 2009.
- [31] Edgar F. Westrum and Bruce H. Justice. Molecular freedom of the ammonium ion. heat capacity and thermodynamic properties of ammonium perchlorate from 5°–350°k. *The Journal of Chemical Physics*, 50(12):5083–5087, 1969.
- [32] J. M. Winey, Y. A. Gruzdkov, Z. A. Dreger, B. J. Jensen, and Y. M. Gupta. Thermo-mechanical model and temperature measurements for shocked ammonium perchlorate single crystals. *Journal of Applied Physics*, 91(9):5650–5656, 2002.

A Code

All code developed under this effort is listed below.

A.1 Unreacted EOS Optimization

Python code for optimizing HERMES unreacted equation of state.

```
# -*- coding: utf-8 -*-
"""
Created on Fri Jun  7 12:48:42 2019

@author: mvandenboom
"""

import random
from deap import base, creator, tools, algorithms
import math
import os
import matplotlib.pyplot as plt
import numpy as np
import threading

def CreateData(path, FitRange):

    V = np.arange(FitRange[0], FitRange[1], FitRange[2])
    f = open(path, 'r')
    JWLVars = f.readlines()
    A = float(JWLVars[0])
    R1 = float(JWLVars[2])
    R2 = float(JWLVars[3])
    W = float(JWLVars[4])
    f.close()
    splitpath = path.rsplit('/', 1)
    PV_Path = splitpath[0] + "/JWL_Data.txt"
```

```

#put data into a file so it can be read back into a single 2D array
f = open(PV_Path, "w+")
for point in range(len(V)):
    p =
    ↪ A*((1-W/(R1*V[point]))*np.exp(-R1*V[point])-(1-W/R1)*(1-W/R2)**(-1)*(1-W/(R2*V[point]))*np.
    f.write(str(V[point]) + " " + str(p))
    if point<(len(V)-1):
        f.write("\n")

f.close()

#read back into 2D array
return np.genfromtxt(PV_Path)

def SaveResults(path, individual, ngen, popsize, gen, LIMITS):

    f = open(path + "Results" + gen + ".txt", "w+")
    f.write("Population Size = " + str(popsize) + "\n")
    f.write("Generations = " + str(ngen) + "\n")
    f.write("A1 = " + str(individual[0]) + "\n")
    f.write("A2 = " + str(individual[1]) + "\n")
    f.write("A3 = " + str(individual[2]) + "\n")
    f.write("R-squared value = " + str(individual.fitness.values[0]) + '\n')
    f.write("Limits:\n")
    f.write("A1 LL: " + str(LIMITS[0][0]) + '\n')
    f.write("A1 UL: " + str(LIMITS[0][1]) + '\n')
    f.write("A2 LL: " + str(LIMITS[1][0]) + '\n')
    f.write("A2 UL: " + str(LIMITS[1][1]) + '\n')
    f.write("A3 LL: " + str(LIMITS[2][0]) + '\n')
    f.write("A3 UL: " + str(LIMITS[2][1]) + '\n')
    f.close()

#called during evaluation to ensure
def TestFunc(v, A1, A2, A3):
    result = ((A1 * ((1/v)-1))+(A2*(((1/v)-1)**2)) + (A3*(((1/v)-1)**3)))

```

```

return result

def evaluateparallel(individual, result, index, indnum, flags, data):
    P = data[:,1]
    V = data[:,0]

    #calculate total sum of squares
    Error = 0
    for point in range(len(V)):
        fx = P[point]
        hx = TestFunc(V[point], individual[0], individual[1], individual[2])
        Error = Error + ((fx - hx)**2)

    #you can not get the return of a function called as a thread so valuse are
    #returned by puting them into a predetermined location into an array
    result[index] = Error
    flags[index].set() #tell main thread that this evaluation is finished

def mutate(individual, indpb, LIMITS):

    #max distance that can be mutated at once
    #percent of range for a given value
    mutrangemax = .1

    for i in range (len(individual)):
        if random.random()< indpb:
            #allows for more "precision" in the mutations
            thismutrange = mutrangemax * random.uniform(0,1)

            #mutate the individual
            individual[i] =
            ↪ random.uniform(individual[i]-((LIMITS[i][1]-LIMITS[i][0])*thismutrange),
            ↪ individual[i]+((LIMITS[i][1]-LIMITS[i][0])*thismutrange))

```

```

        #make sure values are still within limits
        if individual[i] < LIMITS[i][0]:
            individual[i] = LIMITS[i][0]
        if individual[i] > LIMITS[i][1]:
            individual[i] = LIMITS[i][1]
    return individual,

def crossover(ind1, ind2, indpb):
    '''
    2 children are created where a is between the a's of the parents and b is between the
    ↪ b's of the parents
    '''
    ind1new = []
    ind2new = []

    for i in range(len(ind1)):
        if random.random() < indpb:
            if ind1[i] < ind2[i]:
                ind1new.append(random.uniform(ind1[i], ind2[i]))
                ind2new.append(random.uniform(ind1[i], ind2[i]))
            else:
                ind1new.append(random.uniform(ind2[i], ind1[i]))
                ind2new.append(random.uniform(ind2[i], ind1[i]))
        else:
            ind1new.append(ind1[i])
            ind2new.append(ind2[i])
    for i in range(len(ind1)):
        ind1[i] = ind1new[i]
        ind2[i] = ind2new[i]

    return ind1, ind2

def plot(best, data, path, FitRange):

```

```

fig, ax = plt.subplots(1,1)

P_JWL = data[:,1]
V_JWL = data[:,0]
ax.scatter(V_JWL, P_JWL, s=30, color='b', lw = 0) # plots the line being fit to as
    ↪ point so they can be seen against the fit results
v_EOS=np.arange(FitRange[0], FitRange[1], FitRange[2])
p_EOS = []
for x in v_EOS:
    p_EOS.append(TestFunc(x,best[0], best[1], best[2]))

ax.plot(v_EOS, p_EOS, '-k') #plot the results of the fit
ax.set_title('JWL FIT',fontsize=18)
ax.set_xlabel(r'$V/V_0$',fontsize=18)
ax.set_ylabel(r'$Pressure \, (GPa) \$',fontsize=18)
R_Squared = best.fitness.values[0]
ax.text(0.8, 60.0, r'$\sum (P_{JWL} - p_i)^2 = %.4f \, \$' % R_Squared,
    ↪ color='k',fontsize=18)
ax.set_xlim([FitRange[0], FitRange[1]])
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure=plt.gcf() # get current figure
figure.set_size_inches(8,6)

plt.savefig(path + "Plot.png",format='png',dpi=200)

def eaSimple(population, toolbox, cxpb, mutpb, ngen, Results_Path, LIMITS, stats=None,
    halloffame=None, verbose=__debug__):

    logbook = tools.Logbook()
    logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])

    # Evaluate the individuals with an invalid fitness
    threads= []

```

```

threadcount = 0
results = [None] * len(population)
invalid_inds = []
flags = [threading.Event()] * len(population)
for i in range(len(flags)):
    flags[i].clear()
j = 0 # current index in results array
for i in range(len(population)):
    if not population[i].fitness.valid:
        threads.append(threading.Thread(name = "Thread " + str(threadcount), target =
        ↪ toolbox.evaluateparallel, args = (population[i], results, j, i, flags)))
        threads[threadcount].start()
        invalid_inds.append(i)
        threadcount += 1
        j+=1
for t in range(j):
    threads[t].join()
for i in range(j):
    population[invalid_inds[i]].fitness.values = results[i],
del threads[:]
del results[:]
del flags[:]
del invalid_inds[:]

if halloffame is not None:
    halloffame.update(population)

record = stats.compile(population) if stats else {}
logbook.record(gen=0, nevals=threadcount, **record)
if verbose:
    print logbook.stream

# Begin the generational process
for gen in range(1, ngen + 1):
    # Select the next generation individuals

```

```

offspring = toolbox.select(population, int(math.floor(len(population)*.9)))

immigrants = toolbox.population(n = (len(population) -
↪ int(math.floor(len(population)*.9)) - 1))
# Vary the pool of individuals
for ind in immigrants:
    offspring.append(ind)

offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

prevBest = tools.selBest(population, 1)
offspring.append(prevBest[0])
#offspring.append(tools.selBest(population, 1))

# Evaluate the individuals with an invalid fitness
threads= []
threadcount = 0
results = [None] * len(offspring)
invalid_inds = []
flags = [threading.Event()] * len(offspring)
for i in range(len(flags)):
    flags[i].clear()
j = 0 # current index in results array
for i in range(len(offspring)):
    if not offspring[i].fitness.valid:
        threads.append(threading.Thread(name = "Thread " + str(threadcount),
↪ target = toolbox.evaluateparallel, args = (offspring[i], results, j,
↪ i, flags)))
        threads[threadcount].start()
        invalid_inds.append(i)
        threadcount += 1
        j+=1
for t in range(j):
    threads[t].join()
for i in range(j):

```

```

        offspring[invalid_inds[i]].fitness.values = results[i],
del threads[:]
del results[:]
del flags[:]
del invalid_inds[:]

# Update the hall of fame with the generated individuals
if halloffame is not None:
    halloffame.update(offspring)

# Replace the current population by the offspring
population[:] = offspring

# Append the current generation statistics to the logbook
record = stats.compile(population) if stats else {}
logbook.record(gen=gen, nevals=threadcount, **record)
if verbose:
    print logbook.stream

#Save the best individual every hundred generations
if gen % 100 == 0:
    SaveResults(Results_Path, halloffame[0], ngen, len(population), str(gen/100),
        ↳ LIMITS)
return population, logbook

def createindividual(LIMITS):
    creator.create("MinError", base.Fitness, weights=(-1.0,))
    creator.create("Individual", list, fitness=creator.MinError)

    newind = creator.Individual()
    for i in range(len(LIMITS)):
        newind.append(random.uniform(LIMITS[i][0], LIMITS[i][1]))

    return newind

```

```

def main (path):

    popsize = 100
    ngen = 200

    #range of values to test
    # 0 is min
    # 1 is max
    # 2 is the step length
    FitRange = [0.6, 1, .01]

    A1LL = 0          # A lower limit [GPa]
    A1UL = 100        # A upper limit [GPa]
    A2LL = -200       # R1 lower limit
    A2UL = 200        # R1 upper limit
    A3LL = 0          # R2 lower limit multiplier
    A3UL = 100000      # R2 upper limit multiplier
    LIMITS=( [A1LL,A1UL] , [A2LL,A2UL] , [A3LL,A3UL] )

    #put the results of the curve being fit to in an 2-D array that is used in
    #every evaluation to save on Read Right/Write and calculation
    RealData = CreateData(path, FitRange)

    toolbox = base.Toolbox()
    toolbox.register("individual", createindividual, LIMITS = LIMITS)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("evaluateparallel", evaluateparallel, data = RealData)
    toolbox.register("mate", crossover, indpb = .5)
    toolbox.register("mutate", mutate, indpb = .5, LIMITS = LIMITS)
    toolbox.register("select", tools.selTournament, tournsize = 3)

    #determine the path to and create the folder where the results will be stored
    #Allows the program to be run more than once and have the results be saved to a new
    → location each time
    splitpath = path.rsplit('/', 1)

```

```

ResultsFolderPath = splitpath[0] + "/Fit_Results"
if not os.path.isdir(ResultsFolderPath):
    os.mkdir(ResultsFolderPath)
ResultsNum = 1 + len(os.listdir(ResultsFolderPath))
Results_Path = splitpath[0] + "/Fit_Results/Results" + str(ResultsNum) + "/"
os.mkdir(Results_Path)

pop = toolbox.population(n=popsize)
hof = tools.HallOfFame(1)
stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
stats.register("median", np.median)

#genetic algorithm
pop, logbook = eaSimple(pop, toolbox, cxpb=.5, mutpb = 0.5, ngen = ngen, Results_Path
→ = Results_Path, LIMITS = LIMITS, stats = stats, halloffame = hof, verbose =
→ True,)

print("Best individual is: " + str(hof[0]) + "\nwith fitness: " + str(hof[0].fitness))

nevals = logbook.select("nevals")
print "Total Evaluations: " + str(sum(nevals))

SaveResults(Results_Path, hof[0], ngen, popsize, "Final", LIMITS)
plot(hof[0], RealData, Results_Path, FitRange)

main("C:/Users/mvandenboom/Documents/Shock_Compression_Fits/JWL.txt")

```

A.2 Cheetah tool

Python code for working with Cheetah product equation of state tables. Includes Maxwell constructions.

```

# -*- coding: utf-8 -*-
"""
Created on 17 February 2021.

@author: jmaestas
*****

CHEETAH TABLE TOOL

This script processes tabular equations of state generated with Cheetah. It
corrects van der Waals loops by generating "Maxwell constructions". The EOS
matrix is output as a table that BCAT can use named 'chtinp'. It also creates
an eos.txt file needed for HERMES v89c.

*****
"""

# Include the necessary libraries
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import matplotlib.pyplot as pl

# Define initial density
R0 = 1.80 # g/cc

# Read in Cheetah RHO table
data = pd.read_csv("cheetah.dat", header=0, names=['T', 'V', 'P', 'E', 'S'],
                  skiprows=[1], skipinitialspace=True)

# =====
# GATHER ISOTHERMS
# =====

# Get temperature and volume mesh

```

```

TMESH = sorted(set(data['T']))
VMESH = sorted(set(data['V']))

NT = len(TMESH)
NV = len(VMESH)

# Set up dictionary to store isotherms
grouped = data.groupby('T')
isotherms = {}
new_isotherms = {}

for T in TMESH:
    df_new = grouped.get_group(T)

    # make sure volume is ascending
    if df_new['V'].iloc[0] > df_new['V'].iloc[-1]:
        df_new = df_new[::-1]

    isotherms[T] = df_new
    new_isotherms[T] = isotherms[T].copy()

# =====
# PLOT ORIGINAL ISOTHERMS
# =====

# Initialize
colors = pl.cm.jet(np.linspace(0, 1, NT))
fig1, ax1 = plt.subplots(1, 1)
fig2, ax2 = plt.subplots(1, 1)
fig3, ax3 = plt.subplots(1, 1)
fig4, ax4 = plt.subplots(1, 1)

# Loop through isotherms and plot them
for T in TMESH:
    P = isotherms[T]['P'] / 9869.23266716 # atm to GPa

```

```

V = isotherms[T]['V'] * R0 # relative volume
E = isotherms[T]['E'] / 239.00574 # cal/g to GPa/g/cc
ax1.plot(V, P, color=colors[TMESH.index(T)])
ax2.plot(V, P, color=colors[TMESH.index(T)])
ax3.plot(V, E, color=colors[TMESH.index(T)])
ax4.plot(V, E, color=colors[TMESH.index(T)])

# Plot format options
fig1.suptitle('Isotherms (Original): Pressure', fontsize=20, fontweight='bold')
ax1.set_xlabel(r'$V/V_0$', fontsize=18)
ax1.set_ylabel(r'$Pressure \, (GPa)$', fontsize=18)
ax1.set_xlim([0.5, 2.])
ax1.set_ylim([-2., 100.])
ax1.grid(True)
# ax1.legend(loc='upper right', fontsize=15)
ax1.tick_params(axis='both', which='major', labelsize=15)
fig1.set_size_inches(8, 6)
fig1.savefig("original_isotherms_p.png", format='png', dpi=200)

fig2.suptitle('Isotherms (Original): Pressure', fontsize=20, fontweight='bold')
ax2.set_xlabel(r'$V/V_0$', fontsize=18)
ax2.set_ylabel(r'$Pressure \, (GPa)$', fontsize=18)
ax2.set_xlim([1.0, 2.])
ax2.set_ylim([0., 1.5])
ax2.grid(True)
# ax2.legend(loc='upper right', fontsize=15)
ax2.tick_params(axis='both', which='major', labelsize=15)
fig2.set_size_inches(8, 6)
fig2.savefig("original_isotherms_p_zoom.png", format='png', dpi=200)

fig3.suptitle('Isotherms (Original): Energy', fontsize=20, fontweight='bold')
ax3.set_xlabel(r'$V/V_0$', fontsize=18)
ax3.set_ylabel(r'$Energy \, (GPa/g/cc)$', fontsize=18)
ax3.set_xlim([0.5, 2.])
ax3.set_ylim([-10., 20.])

```

```

ax3.grid(True)
# ax3.legend(loc='upper right', fontsize=15)
ax3.tick_params(axis='both', which='major', labelsize=15)
fig3.set_size_inches(8, 6)
fig3.savefig("original_isotherms_e.png", format='png', dpi=200)

fig4.suptitle('Isotherms (Original): Energy', fontsize=20, fontweight='bold')
ax4.set_xlabel(r'$V/V_0$', fontsize=18)
ax4.set_ylabel(r'$Energy \, (GPa/g/cc)$', fontsize=18)
ax4.set_xlim([1.0, 2.])
ax4.set_ylim([-5., -3.])
ax4.grid(True)
# ax4.legend(loc='upper right', fontsize=15)
ax4.tick_params(axis='both', which='major', labelsize=15)
fig4.set_size_inches(8, 6)
fig4.savefig("original_isotherms_e_zoom.png", format='png', dpi=200)

# =====
# GENERATE MAXWELL CONSTRUCTIONS
# =====
# Maxwell constructions are based on method described in PANDA manual

# Initialize
colors = pl.cm.jet(np.linspace(0, 1, NT))
fig1, ax1 = plt.subplots(1, 1)
fig2, ax2 = plt.subplots(1, 1)

# Counter for plot legend
counter = 0

# Loop through the isotherms
for T in TMESH:
    P = isotherms[T]['P']
    V = isotherms[T]['V']
    E = isotherms[T]['E']

```

```

S = isotherms[T]['S']
G = E + P * V - T * S # Gibbs free energy

# Copy arrays for plotting prior to corrections
P_old = np.array(P)
V_old = np.array(V)
E_old = np.array(E)

# Compute derivative
dPdV = np.gradient(P, V)

# Check if there's any positive slopes. This indicates the presence of
# van der Waals loops or liquid-vapor coexistence regions.
if any(slope > 0.0 for slope in dPdV):

    # If so, find spinodals, C and D
    pi_lt = next(x[0] for x in enumerate(dPdV) if x[1] > 0.)
    pi_left = pi_lt - 1

    pi_rt = next(x[0] for x in enumerate(dPdV[::-1]) if x[1] > 0.)
    pi_right = NV - pi_rt + 1

    pi_C = list(P).index(min(P[pi_left:pi_right]))
    P_C = P.iloc[pi_C]

    pi_D = list(P).index(max(P[pi_left:pi_right]))
    P_D = P.iloc[pi_D]

    # Find number of points needed in construction
    num_maxwell_pts = pi_D - pi_C + 1

    # Next, find points A and B where pressure and Gibb's free energy are
    # equal.

    # Refine an area of pressure space where pressures are equal on either

```

```

# side of the spinoidals
P_range = np.linspace(P_C, P_D, 50)

# Find the corresponding Gibbs energy to the left and right of the
# spinoidals. Pressure needs to be increasing for the interpolation
# function to work properly.
G_left = np.interp(P_range, np.array(P.iloc[:pi_C + 1]))[:, :-1],
                    np.array(G.iloc[:pi_C + 1]))[:, :-1])
G_right = np.interp(P_range, np.array(P.iloc[pi_D:]))[:, :-1],
                    np.array(G.iloc[pi_D:]))[:, :-1])

# Find point where Gibb's free energy is equal
G_diff = 99999.
for i in range(len(G_left)):
    G_diff_check = abs(G_left[i] - G_right[i])
    if G_diff_check < G_diff:
        G_diff = G_diff_check
        G_star = G_left[i]

# P_A and P_B should be the same, but take average any way
P_A = np.interp(G_star, G_left, P_range)
P_B = np.interp(G_star, G_right, P_range)

P_AB = (P_A + P_B) / 2.

# Find corresponding volumes
V_A = np.interp(P_AB, np.array(P.iloc[:pi_C + 1]))[:, :-1],
                 np.array(V.iloc[:pi_C + 1]))[:, :-1])
V_B = np.interp(P_AB, np.array(P.iloc[pi_D:]))[:, :-1],
                 np.array(V.iloc[pi_D:]))[:, :-1])

# Find other variables at A and B
E_A = np.interp(P_AB, np.array(P.iloc[:pi_C + 1]))[:, :-1],
                 np.array(E.iloc[:pi_C + 1]))[:, :-1])
E_B = np.interp(P_AB, np.array(P.iloc[pi_D:]))[:, :-1],

```

```

np.array(E.iloc[pi_D:][::-1])

S_A = np.interp(P_AB, np.array(P.iloc[:pi_C + 1][::-1],
                                np.array(S.iloc[:pi_C + 1][::-1])
S_B = np.interp(P_AB, np.array(P.iloc[pi_D:][::-1],
                                np.array(S.iloc[pi_D:][::-1])

# Loop through volumes and make corrections if within the range of
# V_A and V_B
for i in range(NV):
    v = V.iloc[i]

    if v >= V_A and v <= V_B:
        # Compute fraction of vapor
        f_v = (v / V_B - 1.) / (V_A / V_B - 1)

        # Make corrections
        new_isotherms[T]['P'].iloc[i] = P_AB
        new_isotherms[T]['E'].iloc[i] = f_v * E_A + (1 - f_v) * E_B
        new_isotherms[T]['S'].iloc[i] = f_v * S_A + (1 - f_v) * S_B

# Plots
ax1.scatter(V.iloc[pi_D] * R0, P_D / 9869.23266716, color='k')
ax1.scatter(V_B * R0, P_AB / 9869.23266716, s=100, facecolors='none',
            edgecolors='r')

if counter == 0:
    ax1.scatter(V.iloc[pi_C] * R0, P_C / 9869.23266716, color='k',
                label='Spinodal')
    ax1.scatter(V_A * R0, P_AB / 9869.23266716, s=100, facecolors='none',
                edgecolors='r', label='Fluid-vapor Equilibrium Pt.')
    ax1.plot(V_old * R0, P_old / 9869.23266716, '--',
             color=colors[TMESH.index(T)], label='Original')
    ax1.plot(V * R0, new_isotherms[T]['P'] / 9869.23266716,
             color=colors[TMESH.index(T)], label='Corrected')

```

```

else:
    ax1.scatter(V.iloc[pi_C] * R0, P_C / 9869.23266716, color='k')

    ax1.scatter(V_A * R0, P_AB / 9869.23266716, s=100, facecolors='none',
                edgecolors='r')
    ax1.plot(V_old * R0, P_old / 9869.23266716, '--',
             color=colors[TMESH.index(T)])
    ax1.plot(V * R0, new_isotherms[T]['P'] / 9869.23266716,
             color=colors[TMESH.index(T)])

    ax2.plot(V_old * R0, E_old / 239.00574, '--', color=colors[TMESH.index(T)])
    ax2.plot(V * R0, new_isotherms[T]['E'] / 239.00574,
             color=colors[TMESH.index(T)])

    counter += 1

# Plot format options
fig1.suptitle('Maxwell Constructions', fontsize=20, fontweight='bold')
ax1.set_xlabel(r'$V/V_0$', fontsize=18)
ax1.set_ylabel(r'$Pressure \, (GPa)$$', fontsize=18)
ax1.set_xlim([1.0, 2.])
ax1.set_ylim([0., 1.5])
ax1.grid(True)
ax1.legend(loc='upper right', fontsize=15)
ax1.tick_params(axis='both', which='major', labelsize=15)
fig1.set_size_inches(8, 6)
fig1.savefig("maxwell_constructions_p.png", format='png', dpi=200)

fig2.suptitle('Maxwell Constructions', fontsize=20, fontweight='bold')
ax2.set_xlabel(r'$V/V_0$', fontsize=18)
ax2.set_ylabel(r'$Energy \, (GPa/g/cc)$$', fontsize=18)
ax2.set_xlim([1.0, 2.])
ax2.set_ylim([-5., -3.])
ax2.grid(True)
# ax2.legend(loc='upper right', fontsize=15)

```

```

ax2.tick_params(axis='both', which='major', labels=15)
fig2.set_size_inches(8, 6)
fig2.savefig("maxwell_construction_e.png", format='png', dpi=200)

# =====
# GENERATE TABLE FOR BCAT
# =====

# Initialize a file to write to and write out column labels
output = 'chtinp'
header_line = ['T (K)', 'V (cc/g)', 'P (atm)', 'E (cal/g)', 'S (cal/g-K)']

frames = []
i = 0
# Loop through isotherms
for T in TMESH[:, -1]:

    if (-1)**i > 0:
        frames.append(new_isotherms[T])
    else:
        frames.append(new_isotherms[T][::-1])

    i += 1

bcat_data = pd.concat(frames)

# Write out data
bcat_data.to_csv(output, header=header_line, index=False)

# =====
# GENERATE TABLE FOR HERMES
# =====

# Initialize a file to write to and write out column labels
output = 'eos.txt'

```

```

header_line = ['e', 'p', 'rho', 't']

frames = []
# Loop through isotherms
for T in TMESH:
    frames.append(new_isotherms[T][::-1])

hermes_data = pd.concat(frames)

# Adjust dataframe
hermes_data['RHO'] = 1. / hermes_data['V']
hermes_data.drop(['V', 'S'], axis=1)
hermes_data = hermes_data[['E', 'P', 'RHO', 'T']]

# Write out data
hermes_data.to_csv(output, header=header_line, index=False)

```

A.3 Strength Model Optimization

The genetic algorithm script for optimizing the intact strength model is comprised of six files described below.

A.3.1 Fitness.py

Contains different fitness (figure of merit) classes to be used by the GA.

```

# -*- coding: utf-8 -*-
"""
Created on Fri Jul 12 16:05:24 2019

@author: mvandenboom
"""

import sys
from collections import Sequence
from operator import mul, truediv

```

```

from deap import base

class WeightedFitness(base.Fitness):
    def __gt__(self, other):
        return not self.__le__(other)

    def __ge__(self, other):
        return not self.__lt__(other)

    def __le__(self, other):
        if self.wvalues == other.wvalues:
            return True

        differences = [(other.wvalues[i] - self.wvalues[i]) for i in
↪ range(len(self.wvalues))]
        #print sum(differences)
        if sum(differences) > 0:
            return True
        else:
            return False

    def __lt__(self, other):
        differences = [(other.wvalues[i] - self.wvalues[i]) for i in
↪ range(len(self.wvalues))]
        #print sum(differences)
        if sum(differences) > 0:
            return True
        else:
            return False

class IndependentFitness(object):
    """The fitness is a measure of quality of a solution. If *values* are
provided as a tuple, the fitness is initalized using those values,
otherwise it is empty (or invalid)."""

```

:param values: The initial values of the fitness as a tuple, optional.

Fitnesses may be compared using the ``>``, ``<``, ``>=``, ``<=``, ``==``, ``!=``. The comparison of those operators is made lexicographically.

Maximization and minimization are taken care off by a multiplication between the :attr:`weights` and the fitness :attr:`values`. The comparison can be made between fitnesses of different size, if the fitnesses are equal until the extra elements, the longer fitness will be superior to the shorter.

Different types of fitnesses are created in the :ref:`creating-types` tutorial.

.. note::

*When comparing fitness values that are **minimized**, ``a > b`` will return :data:`True` if **a** is **smaller** than **b**.*

"""

weights = (-1, 2,-1)

"""The weights are used in the fitness comparison. They are shared among all fitnesses of the same type. When subclassing :class:`Fitness`, the weights must be defined as a tuple where each element is associated to an objective. A negative weight element corresponds to the minimization of the associated objective and positive weight to the maximization.

.. note::

If weights is not defined during subclassing, the following error will occur at instantiation of a subclass fitness object:

``TypeError: Can't instantiate abstract <class Fitness[...]> with abstract attribute weights.``

"""

wvalues = [None]*len(weights)

"""Contains the weighted values of the fitness, the multiplication with the weights is made when the values are set via the property :attr:`values`. Multiplication is made on setting of the values for efficiency.

Generally it is unnecessary to manipulate wvalues as it is an internal attribute of the fitness used in the comparison operators.

"""

```
def __init__(self, values=[]):
    if self.weights is None:
        raise TypeError("Can't instantiate abstract %r with abstract "
                        "attribute weights." % (self.__class__))

    if not isinstance(self.weights, Sequence):
        raise TypeError("Attribute weights of %r must be a sequence."
                        % self.__class__)

    if len(values) > 0:
        self.values = values

def getValues(self):
    values = []
    for i in range(len(self.wvalues)):
        if self.wvalues[i] is None:
            values.append(self.wvalues[i])
        else:
            values.append(truediv(self.wvalues[i], self.weights[i]))
    return tuple(values)
```

#TODO finish remaking value properties

```
def setValues(self, values):
    """
    values is for setting all values in the set from a tuple
    index and value are for setting one value in fitness
```

```

"""

try:
    self.wvalues = list(map(mul, values, self.weights))
except TypeError:
    _, _, traceback = sys.exc_info()
    raise(TypeError, "Both weights and assigned values must be a " \
        "sequence of numbers when assigning to values of " \
        "%r. Currently assigning value(s) %r of %r to a " \
        "fitness with weights %s."
        % (self.__class__, values, type(values),
           self.weights), traceback)

def setVal(self, index, value):
    self.wvalues[index] = value * self.weights[index]

def delValues(self):
    self.wvalues = [None]*len(self.weights)

values = property(getValues, setValues, delValues,
    ("Fitness values. Use directly ``individual.fitness.values = "
     ↪ values`` "
     "in order to set the fitness and ``del individual.fitness.values``"
     ↪ "
     "in order to clear (invalidate) the fitness. The (unweighted)
     ↪ fitness "
     "can be directly accessed via ``individual.fitness.values``."))

def dominates(self, other, obj=slice(None)):
    """Return true if each objective of *self* is not strictly worse than
    the corresponding objective of *other* and at least one objective is
    strictly better.

    :param obj: Slice indicating on which objectives the domination is
        tested. The default value is `slice(None)`, representing
        every objectives.

```

```

"""

not_equal = False
for self_wvalue, other_wvalue in zip(self.wvalues[obj], other.wvalues[obj]):
    if self_wvalue > other_wvalue:
        not_equal = True
    elif self_wvalue < other_wvalue:
        return False
return not_equal

@property
def valid(self):
    """Assess if a fitness is valid or not."""
    return not None in self.wvalues

def __hash__(self):
    return hash(self.wvalues)

def __gt__(self, other):
    return not self.__le__(other)

def __ge__(self, other):
    return not self.__lt__(other)

def __le__(self, other):
    if self.wvalues == other.wvalues:
        return True

    differences = [(other.wvalues[i] - self.wvalues[i]) for i in
        ↪ range(len(self.wvalues))]
    if sum(differences) > 0:
        return True
    else:
        return False

def __lt__(self, other):

```

```

differences = [(other.wvalues[i] - self.wvalues[i]) for i in
↪ range(len(self.wvalues))]
if sum(differences) > 0:
    return True
else:
    return False

def __eq__(self, other):
    return self.wvalues == other.wvalues

def __ne__(self, other):
    return not self.__eq__(other)

def __deepcopy__(self, memo):
    """Replace the basic deepcopy function with a faster one.

    It assumes that the elements in the :attr:`values` tuple are
    immutable and the fitness does not contain any other object
    than :attr:`values` and :attr:`weights`.
    """
    copy_ = self.__class__()
    copy_.wvalues = self.wvalues
    return copy_

def __str__(self):
    """Return the values of the Fitness object."""
    return str(self.values if self.valid else tuple())

def __repr__(self):
    """Return the Python code to build a copy of the object."""
    return "%s.%s(%r)" % (self.__module__, self.__class__.__name__,
        self.values if self.valid else tuple())

class CustomFitness(object):

```

```

length = 3

def __init__(self):
    if self.length == 0:
        raise ValueError("length of fitness must be set")
    self.values = [None,]*self.length

@property
def total(self):
    if self.valid:
        return sum(self.values)
    else:
        raise ValueError("Fitness is not valid")

@property
def valid(self):
    return not None in self.values

#comparisons are reversed because this is for minimizing
def __eq__(self, other):
    return self.total == other.total
def __ne__(self, other):
    return not self.__eq__(other)

def __lt__(self, other):
    return self.total < other.total
def __le__(self, other):
    return not self.__gt__(other)

def __gt__(self, other):
    return self.total < other.total
def __ge__(self, other):
    return not self.__lt__(other)

```

```

def delval(self):
    self.values = [None]*self.length

def __str__(self):
    return str(self.total)

def __repr__(self):
    return str(self.values)

if __name__ == "__main__":
    test = CustomFitness()
    print(test.values)
    test.values[0] = 1
    test.values[2] = 4
    test.values[1] = 2
    print(repr(test))
    test2 = CustomFitness()
    test2.values[0] = .5
    test2.values[1] = 1
    test2.values[2] = 2
    print(test.total)
    print(test2.total)

    #should return false because test2 is better because it is lower
    print(test<test2)

```

A.3.2 CustomPool.py

Contains functions and classes for initializing multiprocessing pools.

```

# -*- coding: utf-8 -*-
"""
Created on Fri Nov 22 15:56:14 2019

```

```

@author: mvandenboom
"""

import multiprocessing
import sys
import random
import traceback

def workerfunc(queue, results):
    try:
        for func, funcID, args, kwargs in iter(queue.get, 'END'):
            result = func(*args, **kwargs)
            results.put((funcID, result))
    except:
        with open('./errors/file' + str(random.randint(0, 10000)) + '.txt', 'w+') as f:
            info = sys.exc_info()
            traceback.print_exception(info[0], info[1], info[2], file = f)

class Pool(object):
    def __init__(self, processCount = multiprocessing.cpu_count(), queue = None):
        self.processCount = processCount
        if queue is None:
            self.queue = multiprocessing.JoinableQueue()
        elif isinstance(queue, (type(multiprocessing.JoinableQueue()),
            ↪ type(multiprocessing.Queue()))):
            self.queue = queue
        else:
            print("Queue is not a Queue or JoinableQueue, creating new queue")
            self.queue = multiprocessing.JoinableQueue()

        self._pool = []
        self._results = multiprocessing.Queue()
        for i in range(self.processCount):

```

```

        self._pool.append(multiprocessing.Process(target = workerfunc, name =
        ↪ "process " + str(i), args = (self.queue, self._results)))
        self._pool[i].start()

self._RunCounter = 0

def put(self, func, *args, **kwargs):
    funcID = self._RunCounter
    self._RunCounter += 1
    callList = [func, funcID, args, kwargs]
    self.queue.put(callList)

    return funcID

def GetAResult(self,):
    return self._results.get()

def close(self):
    for i in range(self.processCount):
        self.queue.put("END")

```

A.3.3 MultiSim_Inds.py

Contains classes related to individual (child) parameter set descriptions.

```

# -*- coding: utf-8 -*-
"""
Created on Wed Aug 28 14:52:41 2019

@author: mvandenboom
"""
import math

```

```

class Parameter(dict):
    def __init__(self, value, name, LowerLimit=-float('1e100'),
        ↪ UpperLimit=float('1e100')):
        self['value'] = value
        self['name'] = name
        self['UL'] = UpperLimit
        self['LL'] = LowerLimit

    @property
    def value(self):
        """
        value is the actual value of the parameter. All other values in this dict are
        supporting information such as limits, parameter name, etc.
        """
        return self['value']

    @value.setter
    def value(self, value):
        self['value'] = value

    @property
    def name(self):
        return self['name']

    @property
    def UL(self):
        return self['UL']

    @property
    def LL(self):
        return self['LL']

    def __repr__(self):

```

```

basestr = self['name'] + ': ' + str(self['value']) + ' LL: ' + str(self['LL']) + '
↳ UL: ' + str(self['UL'])

defaults = ['name', 'value', 'LL', 'UL']
added = ''
for j in [i for i in self.keys() if i not in defaults]:
    added += str(j) + ': ' + str(self[j]) + ', '
if len(added) > 0:
    added = '{' + added
    added = added[:-2]
    added = added + '}'
    basestr = basestr + '\n' + added
return basestr

def __str__(self):
    return str(self.value)

#magic methods that make this behave like a float unless another value is
↳ specifically called

class Parameter2(float):
    def __new__(cls, value, *args, **kwargs):
        return float.__new__(cls, value)

    def __init__(self, value, name, LowerLimit=-float('1e100'),
↳ UpperLimit=float('1e100')):
        self.name = str(name)
        self.UL = float(UpperLimit)
        self.LL = float(LowerLimit)

    def copy(self, value = None):
        if value is None:
            value = self

```

```

        return self.__class__(value = value, name = self.name, LowerLimit = self.LL,
                               ↪ UpperLimit = self.UL)

def __repr__(self):
    return str(self.name) + ': ' + str(float(self)) + ' LL: ' + str(self.LL) + ' UL: '
    ↪ + str(self.UL)

def __str__(self):
    return float.__repr__(self)

def limits(self):
    if self > self.UL:
        return self.copy(self.UL)
    if self < self.LL:
        return self.copy(self.LL)
    return self

class Individuals(object):
    def __init__(self, *args):
        self.params = []

        for arg in args:
            self.params.append(Parameter2(*arg))

    @property
    def paramsdict(self):
        return {i.name: i for i in self.params}

    def __getitem__(self, key):
        if isinstance(key, int):
            return self.params[key]
        elif isinstance(key, str):
            return self.paramsdict[key]

```

```

def __setitem__(self, key, value):
    if isinstance(value, Parameter2):
        self.params[key] = value
    else:
        try:
            self.params[key] = self.params[key].copy(value)
        except ValueError:
            print('value cant be converted to number, nothing was changed')

def __str__(self):
    string = ''
    for params in self:
        string = string + str(params) + '\n'
    return string

def __repr__(self):
    string = ''
    for params in self:
        string = string + repr(params) + '\n'
    return string

def __len__(self):
    return len(self.params)

if __name__ == "__main__":
    # unit test
    ind = Individuals((.1, 'a', 0, 10), (.2, 'b', -1, 1))
    print(str(ind['a']))

    a = Parameter2(math.pi, 'a', 0, 10)

    b = float(a)
    print(repr(b))

```

A.3.4 ExpClasses.py

Contains classes for describing the experiment being calibrated to.

```
# -*- coding: utf-8 -*-
"""
Created on Wed Dec 11 15:44:29 2019

@author: mvandenboom
"""
import numpy as np

class experiment(object):
    def __init__(self, *args, **kwargs):
        for key, value in kwargs.items():
            self.__dict__[key] = value
            self.files = [
                './ALE3D_Files/hermes_udf.dll',
                './ALE3D_Files/eos.paramv86',
                './ALE3D_Files/eos.txt',
            ]

    @property
    def values(self):
        data = np.genfromtxt(self.data)
        return [data[i][0] for i in range(np.shape(data)[0])]
        #return list(np.split(np.genfromtxt(self.data), 2, 1)[0])

    @property
    def realresults(self):
        data = np.genfromtxt(self.data)
        return [data[i][1] for i in range(np.shape(data)[0])]

class indparams(object):
    # __slots__ = 'line', 'column', 'name'
    def __init__(self, line, column, name):
```

```

    self.line = line

    self.column = column

    self.name = name

def __str__(self):
    return str(self.name) + " Line: " + str(self.line) + " Column: " +
    ↪ str(self.column)

def Calculations(folder, p_conf, loading):
    # Read in the data
    x_file = 'x-nsleft-mean'
    loadx_file = 'loadx-nsright-sum'

    x_data = np.genfromtxt(folder + '/' + x_file)
    x_L = x_data[:,1]

    loadx_data = np.genfromtxt(folder + '/' + loadx_file)
    LOADx = loadx_data[:,1]

    # Original dimensions of 3mm cube
    L0 = 0.3

    # Compute strains
    STRAIN_xx = x_L/L0          ###Use This for Axial Strain## x-axis strain (Load is
    ↪ applied at x=0 plane, displacement measured from origin)

    # Compute stresses
    SIG_x = 1.0E5*loading*abs(LOADx)/(L0-x_L)**2          # x-axis normal stress and
    ↪ maximum principal stress [MPa]

    psi2Pa = 6894.757          # psi to Pa
    psi2MPa = psi2Pa/1.0e6      # psi to MPa
    p_confMPa = p_conf*psi2MPa

```

```

Q = SIG_x - p_confMPa

return (Q, STRAIN_xx)

#Triaxial compression. Confinement = 19000psi
TriaxConfine19000 = experiment(name = "TriaxConfine19000", )
TriaxConfine19000.indparams = {
    "y0" : indparams(145, 2, 'y0'),
    'em0' : indparams(145, 4, 'em0'),
    'epsh' : indparams(147, 0, 'epsh'),
    'capC' : indparams(147, 3, 'capC')
}

TriaxConfine19000.files.append('./ALE3D_Files/Triax/19000/HERMES_Single_Triaxial.ain')
TriaxConfine19000.data = './ALE3D_Files/Triax/19000/data.txt'
TriaxConfine19000.Calculations = Calculations

#Triaxial compression. Confinement = 7250psi
TriaxConfine7250 = experiment(name = "TriaxConfine7250", )
TriaxConfine7250.indparams = {
    "y0" : indparams(145, 2, 'y0'),
    'em0' : indparams(145, 4, 'em0'),
    'epsh' : indparams(147, 0, 'epsh'),
    'capC' : indparams(147, 3, 'capC')
}

TriaxConfine7250.files.append('./ALE3D_Files/Triax/7250/HERMES_Single_Triaxial.ain')
TriaxConfine7250.data = './ALE3D_Files/Triax/7250/data.txt'
TriaxConfine7250.Calculations = Calculations

#Triaxial compression. Confinement = 14500psi
TriaxConfine14500 = experiment(name = "TriaxConfine14500", )

```

```

TriaxConfine14500.indparams = {
    "y0" : indparams(145, 2, 'y0'),
    'em0' : indparams(145, 4, 'em0'),
    'epsh' : indparams(147, 0, 'epsh'),
    'capC' : indparams(147, 3, 'capC')
}

TriaxConfine14500.files.append('./ALE3D_Files/Triax/14500/HERMES_Single_Triaxial.ain')
TriaxConfine14500.data = './ALE3D_Files/Triax/14500/data.txt'
TriaxConfine14500.Calculations = Calculations

#Tension
Tension = experiment(name = "Tension")
#Tension.indparams =

```

A.3.5 SM_GA_Multi.py

Main code for running the genetic algorithm.

```

# -*- coding: utf-8 -*-
"""
Created on Wed Sep 25 14:49:53 2019

@author: mvandenboom
"""

import random
import os
import sys
import shutil
import time
import math
import numpy as np
import pickle

from deap import base, creator, tools
import CustomPool

```

```

from MultiSim_Inds import Individuals
from Fitness import CustomFitness as IndependentFitness
from ExpClasses import TriaxConfine19000, TriaxConfine14500, TriaxConfine7250

def Calculations(*args, **kwargs):
    """
    all calculator functions will have 1 parameter, the folder inside the
    indfolder where all the output files are
    they will return a two tuple that contains principle stress difference array and
    x-axis strain array
    """
    pass

# testing
#popsize = 2
#ngen = 2

popsize = 60
ngen = 100

#for the parameters 0 is the name, 1 is the value,
#2 is the zero indexed line number, 3 is the zero indexed column number
parameters = [('y0', .000025, 147, 2), ('em0', .786, 147, 4), ('epsh', .025, 149, 0),
↳ ('capC', .005, 149, 3)]

maxrange = .1

experiments = [TriaxConfine19000, TriaxConfine14500, TriaxConfine7250]

for i, exp in enumerate(experiments):
    exp.ExperimentNum = i

```

```

experiments2 = [('+', 'addition'), ('-', 'subtraction'), ('*', 'multiplication'),]

creator.create("MultiError", IndependentFitness, length=len(experiments),
↳ fitmap=map(lambda x: x.name, experiments))
creator.create("Individual", Individuals, fitness=creator.MultiError, )

def evaluate(ind, gen, testpath, experiment):
    #TODO remove this when finished testing
    def RunSim(ind, indpath, experiment):
        #copy files to the original location of the subfolder for the individual
        experimentpath = indpath + '/' + experiment.name
        os.mkdir(experimentpath)
        ALE3Dpath = None
        for exp_file in experiment.files:

            shutil.copy(exp_file, experimentpath)

            #get the path of the ALE3D input deck in this individual's folder
            if '.ain' in exp_file:
                ALE3DFileName = exp_file.rsplit('/', 1)[-1]
                ALE3Dpath = experimentpath + '/' + ALE3DFileName
        print(ALE3Dpath)

        #open ale3D file and get all text into list of lines 'doc'
        with open(ALE3Dpath, 'r') as f:
            doc = f.readlines()

            #find and change params in 'doc'
            for param in ind:
                indparam = experiment.indparams[param.name]

                changedline = doc[indparam.line].split()
                newline = ''
                for i in range(len(changedline)):

```

```

        if i == indparam.column:
            newline = newline + ' ' + str(param)
        else:
            newline = newline + ' ' + changedline[i]
    newline = newline + '\n'
    doc[indparam.line] = newline

#open ale3d file and write all text from 'doc'
with open(ALE3Dpath, 'w') as ALE3D:
    ALE3D.writelines(doc)
absALE3Dpath = os.path.abspath(ALE3Dpath)
absExperimentpath = os.path.abspath(experimentpath)
#run ale3d
#TODO update for linux
#add module load ale3d
gencommand = f'"C:/Program Files/LLNL Software/ALE3D/bin/gen3d.exe" -in
↳ {absALE3Dpath} -outputdir {absExperimentpath} -cd {absExperimentpath}'
runcommand = f'"C:/Program Files/LLNL Software/ALE3D/bin/ale3d.exe" -in
↳ {absALE3Dpath} -outputdir {absExperimentpath} -cd {absExperimentpath}'
os.system(gencommand)
os.system(runcommand)

for i in os.listdir(experimentpath):
    if os.path.isdir(experimentpath + '/' + i):
        return experimentpath + '/' + i

# TODO log errors by creating a file in the ind's folder that holds the error
def ClosestValue(array, value):
    index = (np.abs(array-value)).argmin()
    return index

genpath = testpath + '/Gen' + str(gen)
indpath = genpath + '/Individual' + str(ind.indnum)

```

```

'''
create the path for the individual, try/except is used over if/else becuae
the dir may be created by another process working on a different experiment
for the same individal in between the if and the os.mkdir'''
try:
    os.mkdir(indpath)
except OSError:
    pass

resultsfolder = RunSim(ind, indpath, experiment)

with open(indpath + '/' + experiment.name + '.txt', 'a+') as logs:
    logs.write("calculating results")
    results = experiment.Calculations(resultsfolder)

    logs.write('results calculated\n')

    values = experiment.values
    realresults = experiment.realresults #experimental values
    logs.write('values retrieved from experiment\n')

    STRAIN_xx = results[1]
    Q = results[0]

    indresults = [Q[i] for i in [ClosestValue(STRAIN_xx, j) for j in values]]
    logs.write('results calculated\n')
    error = sum([(t - r)**2 for t, r in zip (indresults, realresults)])
    logs.write(str(type(error)))
    logs.write('\n')
    logs.write(str(error))
return error

def eaMain(population, toolbox, cxpb, mutpb, startgen, ALE3Dpath, Results_Path, mainpath,

```

```

        stats = None, halloffame = None, logbook = None, verbose=__debug__):
#make sure logbook exists
    if logbook is None:
        logbook = tools.Logbook()
        logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])
    else:
        for i in logbook.header:
            sys.stdout.write( i.ljust(8))

# make sure path for individual tests exists
    if not os.path.exists(mainpath):
        os.mkdir(mainpath)
#del mainpath
    p = CustomPool.Pool(8)

    if startgen == 0:

        testpath = "./Individual_Tests/Gen0"
        if not os.path.exists(testpath):
            os.mkdir(testpath)

    indctr = 0
    for ind in population:
        if not ind.fitness.valid:
            ind.indnum = indctr
            indctr += 1

            ind.funcIDs = []
            for experiment in experiments:
                ind.funcIDs.append(p.put(evaluate, ind, 0, mainpath, experiment))
    for i in range((indctr)*len(experiments)):
        result = p.GetAResult()
        result[0] #funcID
        for ind in population:

```

```

        if hasattr(ind, "indnum"):
            if result[0] in ind.funcIDs:
                ind.fitness.values[ind.funcIDs.index(result[0])] = result[1]#
                ↪ setVal(ind.funcIDs.index(result[0]), result[1])

#TODO got here correctly
    if halloffame is not None:
        halloffame.update(population)

    SaveGen(Results_Path, halloffame[0], 0, popsize, ngen)

    record = stats.compile(population) if stats else {}
    logbook.record(gen=0, nevals=indctr, **record)
    startgen = 1
    if verbose:
        print(logbook.stream)

    cp = dict(population = population, generation = 1, halloffame = halloffame,
        ↪ logbook = logbook, rndstate = random.getstate())

    pickle.dump(cp, open(Results_Path + 'Cp.pkl', "wb"))

#start generational process
    for gen in range(startgen, ngen + startgen):
        print("started gen " + str(gen))
        testpath = "./Individual_Tests/Gen" + str(gen)
        if not os.path.exists(testpath):
            os.mkdir(testpath)

        print('made gen path')
        offspring = toolbox.select(population, int(math.floor(popsiz*.9)))
        print('copied offspring')
        immigrants = toolbox.population(n = (popsiz - int(math.floor(popsiz*.9)) - 1))

```

```

for ind in immigrants:
    offspring.append(ind)
print('new immigrants')
# Vary the pool of individuals
offspring = varAnd(offspring, toolbox, cxpb, mutpb)
print('varied individuals')
prevBest = tools.selBest(population, 1)
offspring.append(prevBest[0])
print('added prevBest')

indctr = 0
for ind in offspring:
    if hasattr(ind, "indnum"):
        del ind.indnum
    if hasattr(ind, "funcIDs"):
        del ind.funcIDs
print('wiped indnums and funcIDs')
for ind in offspring:
    if not ind.fitness.valid:
        ind.indnum = indctr
        indctr += 1

        ind.funcIDs = []
        for experiment in experiments:
            ind.funcIDs.append(p.put(evaluate, ind, gen, mainpath, experiment))

print('started experiments')
for i in range((indctr)*len(experiments)):
    result = p.GetAResult()
    result[0] #funcID
    for ind in offspring:
        if hasattr(ind, "indnum"):
            if result[0] in ind.funcIDs:
                ind.fitness.values[ind.funcIDs.index(result[0])] = result[1]
print('got results')

```

```

    if halloffame is not None:
        halloffame.update(offspring)

    # Replace the current population by the offspring
    population[:] = offspring

    SaveGen(Results_Path, halloffame[0], gen, popsize, ngen)

    record = stats.compile(population) if stats else {}
    logbook.record(gen=gen, nevals=indctr, **record)
    if verbose:
        print(logbook.stream)

    cp = dict(population = population, generation = gen + 1, halloffame = halloffame,
        ↪ logbook = logbook, rndstate = random.getstate())

    pickle.dump(cp, open(Results_Path + 'Cp.pkl', "wb"))

def mutate(ind):
    #TODO create this
    for i in range(len(ind)):
        if random.random() < .25:
            ind[i] = ind[i].copy(random.triangular(ind[i].LL, ind[i].UL, ind[i]))
    return ind,

def crossover(ind1, ind2):
    #TODO create this
    for i in range(len(ind1)):
        if random.random() < .25:
            a = ind1[i]
            b = ind2[i]
            ind1[i] = b.copy()

```

```

        ind2[i] = a.copy()
    return (ind1, ind2)

def varAnd(population, toolbox, cxpb, mutpb):
    offspring = [toolbox.clone(ind) for ind in population]

    # Apply crossover and mutation on the offspring
    for i in range(1, len(offspring), 2):
        if random.random() < cxpb:
            offspring[i - 1], offspring[i] = toolbox.mate(offspring[i - 1],
                                                            offspring[i])

            offspring[i - 1].fitness.delval()
            offspring[i].fitness.delval()

    for i in range(len(offspring)):
        if random.random() < mutpb:
            offspring[i], = toolbox.mutate(offspring[i])
            offspring[i].fitness.delval()

    return offspring

def SaveGen(Results_Path, ind, gen, popsize, ngen, testpath = None, StartTime = None):
    New_Results_Path = Results_Path + '/Gen' + str(gen)
    os.mkdir(New_Results_Path)
    f = open(New_Results_Path + "/Results.txt", 'w+')
    f.write("Population Size = " + str(popsize) + "\n")
    f.write("Generations = " + str(ngen) + "\n")
    if StartTime is not None:
        f.write("Gen " + str(gen) + " took " + str(time.time() - StartTime) + "
        ↪ seconds\n")
    f.write("Fitness Values: " + str(ind.fitness.values) + '\n')
    f.write("Fitness Total: " + str(ind.fitness.total) + '\n')
    f.write("Individual:\n")

```

```

f.write(repr(ind) + '\n')
f.close()

def GenIndividual(cls = Individuals):
    indparams = []
    for param in parameters:
        param_LL = param[1]-(param[1]*maxrange)
        param_UL = param[1]+(param[1]*maxrange)
        indparams.append((random.uniform(param_LL, param_UL), param[0], param_LL,
        ↪ param_UL))
    return cls(*indparams)

def main(path, checkpoint = None):
    toolbox = base.Toolbox()
    toolbox.register("individual", GenIndividual, cls = creator.Individual)
    toolbox.register("population", tools.initRepeat, list, toolbox.individual)
    toolbox.register("evaluate", evaluate, ALE3Dpath = path)
    toolbox.register("select", tools.selTournament, tournsize = 3)
    toolbox.register("save", SaveGen)
    toolbox.register("mate", crossover)
    toolbox.register("mutate", mutate)

    #Set up results folder
    ResultsFolderPath = "./Fit_Results"
    if not os.path.isdir(ResultsFolderPath):
        os.mkdir(ResultsFolderPath)
    ResultsNum = 1 + len(os.listdir(ResultsFolderPath))
    Results_Path = "./Fit_Results/Results" + str(ResultsNum) + "/"

    if checkpoint:
        with open(checkpoint, 'r') as cp_file:
            cp = pickle.load(cp_file)

```

```

pop = cp["population"]
startgen = cp["generation"]
hof = cp["halloffame"]
logbook = cp["logbook"]
random.setstate(cp["rndstate"])
Results_Path = checkpoint.rsplitleft('/', 1)[0] + '/'
else:
    pop = toolbox.population(n=popsizes)
    hof = tools.HallOfFame(1)
    logbook = None
    startgen = 0
    os.mkdir(Results_Path)

stats = tools.Statistics(lambda ind: ind.fitness.total)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
stats.register("median", np.median)

eaMain(pop, toolbox, .5, .5, startgen, path, Results_Path, mainpath =
↳  "./Individual_Tests", stats = stats,
    halloffame = hof, logbook = logbook, verbose = True)

if __name__ == '__main__':
    path = './'
    main(path,)
```

A.4 CREST Lite Optimization

Genetic algorithm script for optimizing the CREST lite reactive flow model to Pop Plot data.

```

# -*- coding: utf-8 -*-
"""
```

Created on Mon Jun 24 10:02:52 2019

@author: mvandenboom

"""

```
import random
import math
import os
import matplotlib.pyplot as plt
import numpy as np
import shutil
import multiprocessing
import time as pytime #import under different name as time was being used as a variable
↳ elsewhere
import pickle
import sys
```

```
from deap import base, creator, tools, algorithms
```

```
creator.create("MinError", base.Fitness, weights=(-1.0,))
```

```
creator.create("Individual", list, fitness=creator.MinError)
```

```
def slopeinterceptline(slope, intercept, label):
```

```
    axes = plt.gca()
    x_vals = np.array(axes.get_xlim())
    y_vals = intercept + slope * x_vals
    plt.plot(x_vals, y_vals, '--', label = label)
```

```
def findlimits(xoa, toa):
```

```
    maxlen = int(math.floor(len(xoa)/3))
    firstchk = int(math.ceil(len(xoa)/20))
    if firstchk < 2:
        firstchk = 2
    firstendchk = -firstchk-1
    if firstendchk > len(xoa)-2:
```

```

        firstendchk = len(xoa)-2
i_ll = 0
o_ul = len(xoa) - 1

im_initial = (toa[firstchk] -toa[0])/(xoa[firstchk]-xoa[0])# get initial input slope

for i in range(firstchk+1, maxlen + 1):
    #if the percent change in the slope is greater than 1% from the original use the
    ↪ previous as the limit
    if math.fabs((((toa[i] - toa[0])/(xoa[i] - xoa[0]) - im_initial)/im_initial)*100)
    ↪ > 1 :
        i_ul = i-1
        break
    i_ul = i

om_initial = (toa[firstendchk] -toa[-1])/(xoa[firstendchk]-xoa[-1])

for i in range(firstendchk-1, -maxlen - 1, -1):
    if math.fabs((((toa[i]-toa[-1])/(xoa[i]-xoa[-1]) - om_initial)/om_initial)*100) >
    ↪ 1:
        o_ll = i + 1
        break
    o_ll = i

return i_ll, i_ul, o_ll, o_ul

#           2 2 4 3
def findMB(x1,y1,x2,y2):
    x1 = float(x1)
    x2 = float(x2)
    y1 = float(y1)
    y2 = float(y2)
    m = ((y2 - y1) / (x2 - x1))
    b = m*-x1 + y1
    return m,b

```

```

def error(x1,y1,x2,y2):
    x1 = float(x1)
    x2 = float(x2)
    y1 = float(y1)
    y2 = float(y2)
    return math.sqrt(((x2-x1)**2) + ((y2-y1)**2))

def RunSim(path, basepath, numTracers, individual, velocity):
    shutil.copy(basepath, path)
    shutil.copy(basepath.rsplit('/', 1)[0] + '/ale_udf.dll', path)
    shutil.copy(basepath.rsplit('/', 1)[0] + '/eos.paramv86', path)
    shutil.copy(basepath.rsplit('/', 1)[0] + '/eos.txt', path)
    filename = basepath.rsplit('/', 1)[-1]
    newpath = path + '/' + filename
    paramfile = path + '/' + 'eos.paramv86'
    f = open(paramfile, 'r')
    doc = f.readlines()
    f.close()

    ainfile = open(newpath, 'r')
    doc2 = ainfile.readlines()
    ainfile.close()
    linenum = 42
    changedpart = 2
    changedline = doc2[linenum-1]
    changedline = changedline.split()
    newline = ''
    for j in range(len(changedline)):
        if j == changedpart:
            newline = newline + " " + str(velocity)
        else:
            newline = newline + " " + changedline[j]

```

```

newline = newline + '\n'
doc2[linenum-1] = newline
ainfile = open(newpath, 'w')
ainfile.writelines(doc2)
ainfile.close()

for i in range(len(individual)):
    linenum = individual[i][1]
    changedpart = individual[i][2]
    changedline = doc[linenum-1]
    changedline = changedline.split()
    newline = ''
    for j in range(len(changedline)):
        if j == changedpart:
            if i == 0 or i == 2 or i == 5:
                newline = newline + " " + str(10**individual[i][0])
            else:
                newline = newline + " " + str(individual[i][0])
        else:
            newline = newline + " " + changedline[j]
    newline = newline + '\n'
    doc[linenum-1] = newline
f = open(paramfile, 'w')
f.writelines(doc)
f.close()

gencommand = ''' + 'C:/Program Files/LLNL/ALE3D 4.30.29/bin/ALE3D.exe' + ''' + '
↳ --gen -in ' + newpath + ' -outdir ' + path + ' -cd ' + path
simcommand = ''' + 'C:/Program Files/LLNL/ALE3D 4.30.29/bin/ALE3D.exe' + ''' + '
↳ --sim -in ' + newpath + ' -outdir ' + path + ' -cd ' + path
os.system(gencommand)
os.system(simcommand)

tracerpath = path + "/timehist.hermes_shockInit_1d_steel.001/"
if os.path.isdir(tracerpath):
    if len(os.listdir(tracerpath))/2 == numTracers:

```

```

        return True
    else:
        return RunSim(path, basepath, numTracers, individual, velocity)
else:
    return RunSim(path, basepath, numTracers, individual, velocity)

def evaluateparallel(args, velocity, ALE3DFinished = False):#individual, indnum,
↪ basepath, goal
    individual = args[0]
    indnum = args[1]
    basepath = args[2]
    gen = args[3]
    numTracers = 501
    path = "C:/Users/mvandenboom/Documents/Hermes_Test/Individual_Tests/Gen" + str(gen) +
↪ "/Individual" + str(indnum) + '/Velocity' + str(velocity[0]).translate(None, '.')
    if not os.path.isdir(path):
        os.mkdir(path)

    if not ALE3DFinished:
        ALE3DFinished = RunSim(path, basepath, numTracers, individual, velocity[0])

    tracerpath = path + "/timehist.hermes_shockInit_1d_steel.001/"
    try:
        tracers = os.listdir(tracerpath)

        toa = []
        xoa = []

        for i in range(numTracers):
            tracerp = np.genfromtxt(tracerpath + "p-tracer-markers." + str('{:03d}'.
↪ format(i)), skip_header = 1, delimiter = "\t")
            tracerx = np.genfromtxt(tracerpath + "x-tracer-markers." + str('{:03d}'.
↪ format(i)), skip_header = 1, delimiter = "\t")
            time = tracerp[:,0]

```

```

pressureMbar = tracerp[:,1]
pressure = [x*100 for x in pressureMbar]# convert pressure from mbar to gpa
distancecm = tracerx[:,1]
distance = [x* 10 for x in distancecm]# convert distance from cm to mm
tracerdistime = np.vstack((time, distance, pressure)).T # creates 2d numpy
↳ array

for j in tracerdistime:
    if j[2] > .1:

        toa.append(j[0])
        xoa.append(j[1])
        break

xoa = xoa - xoa[0]
toa = toa - toa[0]

i_ll, i_ul, o_ll, o_ul = findlimits(xoa, toa)

try:
    mi,bi = np.polyfit(xoa[i_ll:i_ul],toa[i_ll:i_ul],1)
    mo,bo = np.polyfit(xoa[o_ll:o_ul],toa[o_ll:o_ul],1)
except:
    mi,bi = findMB(xoa[i_ll],toa[i_ll], xoa[i_ul],toa[i_ul])
    mo,bo = findMB(xoa[o_ll],toa[o_ll], xoa[o_ul],toa[o_ul])
plt.clf()
plt.plot(xoa, toa, label = "Toa vs Xoa")
slopeinterceptline(mi,bi, "Impact")
slopeinterceptline(mo,bo, "Detonation")

fit_i_1D = np.poly1d([mi,bi])
fit_o_1D = np.poly1d([mo,bo])

rd_1D = (bo-bi)/(mi-mo)
rt_1D = fit_i_1D(rd_1D)

```

```

plt.plot(rd_1D, rt_1D, marker = 'o', label = 'Detonation Transition Point')
plt.legend()

plt.text(rd_1D + 2 ,rt_1D - 1, "Detonation Transition\nPoint:\n(" +
↪ str('{:05.3f}'.format(rd_1D)) + ", " + str('{:05.3f}'.format(rt_1D)) + ")",
↪ color = 'k', fontsize = 15)

plt.xlabel(r'$Distance \, (mm)$',fontsize=18)
plt.ylabel(r'$Time \, of \, Arrival \, (\mu s) $',fontsize=18)

#plt.scatter(xoa,toa,s=10, color = 'k')

plt.savefig(path+'/Run Distance Plot.png', bbox_inches='tight', dpi = 400)

error = math.fabs(velocity[1]-rd_1D)**2

return error

except:
    return evaluateparallel(args, velocity, ALE3DFinished)

def evaluatemain(args):
    path = "C:/Users/mvandenboom/Documents/Hermes_Test/Individual_Tests/Gen" +
↪ str(args[3]) + "/Individual" + str(args[1])
    try:
        os.mkdir(path)
    except:
        pass
    individual = args[0]
    indnum = args[1]
    basepath = args[2]
    gen = args[3]

```

```

velocity = args[4]
newargs = (individual, indnum, basepath, gen)
error = evaluateparallel(newargs, velocity)
return (error, indnum)

def mutate(individual, indpb, LIMITS):

    #max distance that can be mutated at once
    #percent of range for a given value
    mutrangemax = .1

    for i in range (len(individual)):
        if random.random() < indpb:
            #allows for more "precision" in the mutations
            thismutrange = mutrangemax * random.uniform(0,1)

            #mutate the individual
            individual[i][0] =
            ↪ random.uniform(individual[i][0] - ((LIMITS[i][1] - LIMITS[i][0]) * thismutrange),
            ↪ individual[i][0] + ((LIMITS[i][1] - LIMITS[i][0]) * thismutrange))

            #make sure values are still within limits
            if individual[i][0] < LIMITS[i][0]:
                individual[i][0] = LIMITS[i][0]
            if individual[i][0] > LIMITS[i][1]:
                individual[i][0] = LIMITS[i][1]
    return individual,

def crossover(ind1, ind2, indpb):
    '''
    2 children are created where each value is between the equivilant value of the
    ↪ parents
    '''

```

```

ind1new = []
ind2new = []

for i in range(len(ind1)):
    if random.random() < indpb:
        if ind1[i] < ind2[i]:
            ind1new.append(random.uniform(ind1[i][0], ind2[i][0]))
            ind2new.append(random.uniform(ind1[i][0], ind2[i][0]))
        else:
            ind1new.append(random.uniform(ind2[i][0], ind1[i][0]))
            ind2new.append(random.uniform(ind2[i][0], ind1[i][0]))
    else:
        ind1new.append(ind1[i][0])
        ind2new.append(ind2[i][0])
for i in range(len(ind1)):
    ind1[i][0] = ind1new[i]
    ind2[i][0] = ind2new[i]

return ind1, ind2

def plotfromfile(path, savepath, prevgen):#indnum, gen, path):
    #experimental pressure (gPa) vs run distance (mm)
    try:
        experimentalpvsvx = np.array([[5.8, 18.8], [4.9, 29.4], [4.6, 29.8], [7.4, 11.0]])
        expP = experimentalpvsvx[:,0]
        expX = experimentalpvsvx[:,1]

        plt.clf()
        plt.scatter(expP, expX,s=15, color = 'k', label = "Experimental")

        #experimental velocity (mm/us) vs run distance (mm)
        simX = [None] * 4
        experimentalvvsvx = np.array([(0.0896,18.8), (0.0829,29.4), (0.0751,29.8), (0.1088,
↵ 11.0)])

```

```

for i in enumerate(experimentalvvsx):

    tracerpath = path + '/Velocity' + str(i[1][0]).translate(None, '.') +
    ↪ "/timehist.hermes_shockinit_1d_steel.001/"
    tracers = os.listdir(tracerpath)
    numTracers = len(tracers)/2

    toa = []
    xoa = []

    for k in range(numTracers):
        tracerp = np.genfromtxt(tracerpath + "p-tracer-markers." + str('{:03d}'.
        ↪ format(k)), skip_header = 1, delimiter = "\t")
        tracerx = np.genfromtxt(tracerpath + "x-tracer-markers." + str('{:03d}'.
        ↪ format(k)), skip_header = 1, delimiter = "\t")
        time = tracerp[:,0]
        pressureMbar = tracerp[:,1]
        pressure = [x*100 for x in pressureMbar]
        distancecm = tracerx[:,1]
        distance = [x* 10 for x in distancecm]
        tracerdistime = np.vstack((time, distance, pressure)).T # creates 2d
        ↪ numpy array

        for j in tracerdistime:
            if j[2] > .1:

                toa.append(j[0])
                xoa.append(j[1])
                break
    xoa = xoa - xoa[0]
    toa = toa - toa[0]

    i_ll, i_ul, o_ll, o_ul = findlimits(xoa, toa)

    try:

```

```

        mi,bi = np.polyfit(xoa[i_ll:i_ul],toa[i_ll:i_ul],1)
        mo,bo = np.polyfit(xoa[o_ll:o_ul],toa[o_ll:o_ul],1)
    except:
        mi,bi = findMB(xoa[i_ll],toa[i_ll], xoa[i_ul],toa[i_ul])
        mo,bo = findMB(xoa[o_ll],toa[o_ll], xoa[o_ul],toa[o_ul])

    fit_i_1D = np.poly1d([mi,bi])
    fit_o_1D = np.poly1d([mo,bo])

    rd_1D = (bo-bi)/(mi-mo)
    rt_1D = fit_i_1D(rd_1D)

    simX[i[0]] = rd_1D
    plt.ylabel(r'$Distance \, (mm)$',fontsize=18)
    plt.xlabel(r'$Pressure \, (gPa) $',fontsize=18)

    plt.scatter(expP, simX, s = 15, c = 'b', marker = '^', label = "Simulated")
    plt.legend()
    plt.savefig(savepath + '/plot.png', bbox_inches='tight', dpi = 400)
    return
except:
    try:
        prevpath = savepath.rsplit('/', 1)[0] + '/Gen' + str(prevgen) + '/plot.png'
        shutil.copy(prevpath, savepath)
    except:
        f = open(savepath + "/Failed_To_Plot.txt", 'w+')
        f.close()

def SaveGen(path, testspath, individual, indnum, ngen, popsize, gen, LIMITS, StartTime =
↳ None):
    newpath = path + "/Gen" + gen
    if gen == "Final":
        newpath = path + "/Final"
    os.mkdir(newpath)

```

```

f = open(newpath + "/Results.txt", 'w+')
f.write("Population Size = " + str(popsiz) + "\n")
f.write("Generations = " + str(ngen) + "\n")
f.write("CC0 = " + str(10**individual[0][0]) + "\n")
f.write("CC1 = " + str(individual[1][0]) + "\n")
f.write("CC2 = " + str(10**individual[2][0]) + "\n")
f.write("CC3 = " + str(individual[3][0]) + "\n")
f.write("CC9 = " + str(individual[4][0]) + "\n")
f.write("CC10 = " + str(10**individual[5][0]) + "\n")
f.write("CC11 = " + str(individual[6][0]) + "\n")
f.write("CC13 = " + str(individual[7][0]) + "\n")
f.write("Error = " + str(individual.fitness.values) + '\n')
if StartTime is not None:
    f.write("Gen " + str(gen) + " took " + str(pytime.time() - StartTime) + "
    ↪ seconds\n")
f.write("Limits:\n")
f.write("CC0 LL = " + str(10**LIMITS[0][0]) + "\n")
f.write("CC0 UL = " + str(10**LIMITS[0][1]) + "\n")
f.write("CC1 LL = " + str(LIMITS[1][0]) + "\n")
f.write("CC1 UL = " + str(LIMITS[1][1]) + "\n")
f.write("CC2 LL = " + str(10**LIMITS[2][0]) + "\n")
f.write("CC2 UL = " + str(10**LIMITS[2][1]) + "\n")
f.write("CC3 LL = " + str(LIMITS[3][0]) + "\n")
f.write("CC3 UL = " + str(LIMITS[3][1]) + "\n")
f.write("CC9 LL = " + str(LIMITS[4][0]) + "\n")
f.write("CC9 UL = " + str(LIMITS[4][1]) + "\n")
f.write("CC10 LL = " + str(10**LIMITS[5][0]) + "\n")
f.write("CC10 UL = " + str(10**LIMITS[5][1]) + "\n")
f.write("CC11 LL = " + str(LIMITS[6][0]) + "\n")
f.write("CC11 UL = " + str(LIMITS[6][1]) + "\n")
f.write("CC13 LL = " + str(LIMITS[7][0]) + "\n")
f.write("CC13 UL = " + str(LIMITS[7][1]) + "\n")
f.close()
plotpath = testspath + '/Individual' + str(indnum)
prevgen = 0

```

```

if gen == "Final":
    prevgen = ngen
else:
    prevgen = int(gen) - 1
plotfromfile(plotpath, newpath, prevgen)

def eaSimple(population, toolbox, cxpb, mutpb, startgen, ngen, popsize, Results_Path,
↳ path, LIMITS, stats=None,
        halloffame=None, logbook = None, verbose=__debug__):
    p = multiprocessing.Pool(8, maxtasksperchild = 3)
    if logbook is None:
        logbook = tools.Logbook()
        logbook.header = ['gen', 'nevals'] + (stats.fields if stats else [])
    else:
        for i in logbook.header:
            sys.stdout.write( i.ljust(8))
        print '\n'
    StartTime = pytime.time()
    experimental = [(0.0896,18.8), (0.0829,29.4), (0.0751,29.8), (0.1088, 11.0)]

    if startgen == 0:
        # Evaluate the individuals with an invalid fitness
        testpath = "C:/Users/mvandenboom/Documents/Hermes_Test/Individual_Tests/Gen0"
        os.mkdir(testpath)

        '''
        invalidargs = []
        results = []
        for i in range(len(population)):
            if not population[i].fitness.valid:
                invalidargs.append((population[i], i, path, 0))
        for i in invalidargs:
            results.append(evaluatemain(i))
        for i in range(len(results)):

```

```

        population[results[i][1]].fitness.values = results[i][0]

'''
invalidargs = []
resultsdict = {}
for i in range(len(population)):
    if not population[i].fitness.valid:
        for j in experimental:
            invalidargs.append((population[i], i, path, 0, j))
results = p.map(evaluatemain, invalidargs)
for i in results:
    if resultsdict.has_key(i[1]):
        resultsdict[i[1]] = resultsdict[i[1]] + i[0]
    else:
        resultsdict[i[1]] = i[0]
for i in resultsdict:
    population[i].fitness.values = (resultsdict[i],)
resultsdict.clear()

if halloffame is not None:
    halloffame.update(population)

#Save the best individual every generation
bestind = 0
for i in range(len(population)):
    if population[i].fitness.values[0] < population[bestind].fitness.values[0]:
        bestind = i

SaveGen(Results_Path, testpath, population[bestind], bestind, ngen,
↪ len(population), '0', LIMITS, StartTime)

record = stats.compile(population) if stats else {}
logbook.record(gen=0, nevals=len(invalidargs), **record)
startgen = 1

```

```

if verbose:
    print logbook.stream
cp = dict(population = population, generation = 1, halloffame = halloffame,
↪ logbook = logbook, rndstate = random.getstate())

pickle.dump(cp, open(Results_Path + 'Cp.pkl', "wb"))
else:
    pass
# Begin the generational process
for gen in range(startgen, ngen + startgen):
    # Select the next generation individuals
    StartTime = pytime.time()

    testpath = "C:/Users/mvandenboom/Documents/Hermes_Test/Individual_Tests/Gen" +
    ↪ str(gen)
    os.mkdir(testpath)
    offspring = toolbox.select(population, int(math.floor(popsize*.9)))

    immigrants = toolbox.population(n = (popsize - int(math.floor(popsize*.9)) - 1))
    # Vary the pool of individuals
    for ind in immigrants:
        offspring.append(ind)

    offspring = algorithms.varAnd(offspring, toolbox, cxpb, mutpb)

    prevBest = tools.selBest(population, 1)
    offspring.append(prevBest[0])
    #offspring.append(tools.selBest(population, 1))

    # Evaluate the individuals with an invalid fitness
    invalidargs = []
    resultsdict = {}
    for i in range(len(offspring)):
        if not offspring[i].fitness.valid:

```

```

        for j in experimental:
            invalidargs.append((offspring[i], i, path, gen, j))
results = p.map(evaluatemain, invalidargs)
for i in results:
    if resultsdict.has_key(i[1]):
        resultsdict[i[1]] = resultsdict[i[1]] + i[0]
    else:
        resultsdict[i[1]] = i[0]
for i in resultsdict:
    offspring[i].fitness.values = (resultsdict[i],)
resultsdict.clear()

# Update the hall of fame with the generated individuals
if halloffame is not None:
    halloffame.update(offspring)

# Replace the current population by the offspring
population[:] = offspring

# Append the current generation statistics to the logbook
record = stats.compile(population) if stats else {}
logbook.record(gen=gen, nevals=len(invalidargs), **record)
if verbose:
    print logbook.stream

#Save the best individual every generation
bestind = 0
for i in range(len(population)):
    if population[i].fitness.values[0] < population[bestind].fitness.values[0]:
        bestind = i
SaveGen(Results_Path, testpath, population[bestind], bestind, ngen,
↪ len(population), str(gen), LIMITS, StartTime)

cp = dict(population = population, generation = gen + 1, halloffame = halloffame,
↪ logbook = logbook, rndstate = random.getstate())

```

```

        pickle.dump(cp, open(Results_Path + 'Cp.pkl', "wb"))

    return population, logbook

def createindividual(LIMITS):
    newind = creator.Individual()
    for i in range(len(LIMITS)):
        newind.append([random.uniform(LIMITS[i][0], LIMITS[i][1]), LIMITS[i][2],
        ↪ LIMITS[i][3]])
    return newind

def main(path, checkpoint = None):

    popsize = 40
    ngen = 20

    maxrange = 5

    CC0 = math.log10(float("4.1e5"))
    CC1 = 2.5
    CC2 = math.log10(float("4.3e5"))
    CC3 = 3.0
    CC9 = 1.0
    CC10 = math.log10(float("6.0e6"))
    CC11 = 0.7
    CC13 = float("3.3e-5")
    LIMITS=([CC0 - (CC0/maxrange),CC0 + (CC0/maxrange), 6, 5],
            [CC1 - (CC1/maxrange),CC1 + (CC1/maxrange), 6, 6],
            [CC2 - (CC2/maxrange),CC2 + (CC2/maxrange), 7, 0],
            [CC3 - (CC3/maxrange),CC3 + (CC3/maxrange), 7, 1],
            [CC9 - (CC9/(maxrange/2)),CC9 + (CC9/(maxrange/2)), 7, 2],
            [CC10 - (CC10/maxrange),CC10 + (CC10/maxrange), 7, 3],

```

```

[CC11 - (CC11/maxrange),CC11 + (CC11/maxrange), 7, 4],
[CC13 - (CC13/(maxrange/2)),CC13 + (CC13/(maxrange/2)), 7, 5], )

toolbox = base.Toolbox()
toolbox.register("individual", createindividual, LIMITS = LIMITS)
toolbox.register("population", tools.initRepeat, list, toolbox.individual)
toolbox.register("evaluateparallel", evaluateparallel)
toolbox.register("mate", crossover, indpb = .5)
toolbox.register("mutate", mutate, indpb = .7, LIMITS = LIMITS)
toolbox.register("select", tools.selTournament, tournsize = 3)

#determine the path to and create the folder where the results will be stored
#Allows the program to be run more than once and have the results be saved to a new
→ location each time
splitpath = path.split('/', 1)
ResultsFolderPath = splitpath[0] + "/Fit_Results"
if not os.path.isdir(ResultsFolderPath):
    os.mkdir(ResultsFolderPath)
ResultsNum = 1 + len(os.listdir(ResultsFolderPath))
Results_Path = splitpath[0] + "/Fit_Results/Results" + str(ResultsNum) + "/"

if checkpoint:
    with open(checkpoint, 'r') as cp_file:
        cp = pickle.load(cp_file)
        pop = cp["population"]
        startgen = cp["generation"]
        hof = cp["halloffame"]
        logbook = cp["logbook"]
        random.setstate(cp["rndstate"])
        Results_Path = checkpoint.split('/', 1)[0] + '/'
else:
    pop = toolbox.population(n=popsizes)
    hof = tools.HallOfFame(1)
    logbook = None
    startgen = 0

```

```

os.mkdir(Results_Path)

stats = tools.Statistics(lambda ind: ind.fitness.values)
stats.register("avg", np.mean)
stats.register("min", np.min)
stats.register("max", np.max)
stats.register("median", np.median)

#genetic algorithm
pop, logbook = eaSimple(pop, toolbox, cxpb=.5, mutpb = 0.5, startgen = startgen, ngen
↪ = ngen, popsize = popsize, Results_Path = Results_Path, path = path, LIMITS =
↪ LIMITS, stats = stats, halloffame = hof, logbook = logbook, verbose = True,)
print("Best individual is: " + str(hof[0]) + "\nwith fitness: " + str(hof[0].fitness))

nevals = logbook.select("nevals")
print "Total Evaluations: " + str(sum(nevals))

if __name__ == '__main__':
    main("path/to/input/deck", "path/to/restart/location")

```

A.5 Ammonium Perchlorate EOS Optimization

The genetic algorithm script for optimizing a complete equation of state is comprised of eight files described below.

A.5.1 EOS_Optimize

Main code for running the genetic algorithm.

```

# -*- coding: utf-8 -*-
"""
Created on Mon Jun 7 09:44:53 2021
@author: mvandenboom
"""

```

```

import pickle
import multiprocessing
import random
import os
import shutil
import traceback
import math

import numpy as np
import deap.base
import deap.creator
import deap.tools
import scipy.optimize

import eos_functions as eos
import eos_plots as plot
import eos_evaluation as evaluation

import OptimizationMain
import ProcessQueue

# Parameters

# Optimization Parameters

bounds = [
    (15, 4, 1),  # K0
    (6, 3, 1),   # K1
     #(4, 2, 1),  # theta0
    (0, 3, 1),   # A0
    (0, 3, 1),   # A1
    (0, 3, 1),   # A2
    (0, 3, 1),   # A3
    (0, 3, 1),   # A4

```

```

(0, 3, 1), # B0
(0, 3, 1), # B1
(0, 3, 1), # B2
(0, 3, 1), # B3
(0, 3, 1), # B4
(200000, 500, 1), # c0
(7500, 100, -1), # c1
(175, 10, 1), # c2
(.4710226, .1, 1) # c3
]

popsize = 100

seenIndsBase = None
seenIndsFile = "C:/users/mvandenboom/documents/Current_2021-06-04/seenInds.pkl"

problemCPFile = "/home/matthew/Documents/SpecificHeatFit/Inds/Run20/Gen220/cp.pkl"

#problemCPFile = None

seenIndsLock = multiprocessing.Lock()

icxpb = .25
imutpb = .2
scpb = .6 # chance for each mutation to be based on the original value rather than being
↳ random

# base path for individuals
BaseBasePath = "/home/matthew/Documents/SpecificHeatFit/Inds"

# Model Parameters

R0 = 2.036759 # 1.95 # g/cc, initial density
PREF = 0.000101325 # GPa, reference pressure
TREF = 298.15 # K, reference temperature

```

```

VO = 1. / RO  # cc/g, initial specific volume

# Specific heat
Cv = 0.001  # kJ/(g*K)
# c0 = 200000  # --
# c1 = -7500  # --
# c2 = 175  # --
# c3 = 0.4710226  # --

# SESAME parameters
ESFT = 0  # energy shift
FZ = 60  # formula number (number of electrons in molecule)
FW = 73.45  # formula weight (average molecular weight)
RHOREF = RO  # g/cc, reference density guess

# MatID number and name for SESAME file
MATID = '1'
bin_filename = 'ap_bin'
asc_filename = 'ap_asc'

# Define phase space mesh over which to generate the equation of state.

# Mesh ranges
RVMIN = 0.01  # --, minimum relative volume
RVMAX = 100.0  # --, maximum relative volume
NRV = 50  # number of points

TMIN = 0.0  # K
TMAX = 10000.0  # K
NT = 50  # number of points

# For density, make mesh resolution very fine near the small relative volumes
# (high pressures) and coarser near the large relative volumes where pressure
# varies less
RVMESH = np.logspace(np.log10(RVMIN), np.log10(RVMAX), NRV)

```

```

# For temperature, make mesh resolution finer near the low temperatures and
# coarser at high temperatures where CV varies less
# Temperatures must be in descending order!
TMESH = np.flipud(np.append([TMIN], np.logspace(-2, 0, NT - 1) * TMAX))

deap.creator.create("FitnessMin", deap.base.Fitness, weights=(-1.0,))

class Individual:
    """Individual class for evaluation."""

    def __init__(self, params):
        self.params = params
        self.fitness = deap.creator.FitnessMin()

    @property
    def Vs(self):
        """Return the Vs value for this individual."""
        V_test = V0 * np.linspace(RVMIN, RVMAX, 10000)
        Pc_test = eos.Pc(V_test, V0, self.K0, self.K1)
        val, idx = min((val, idx) for (idx, val) in enumerate(Pc_test))
        Vs = V_test[idx]
        return Vs

    def mesh(self, path):
        """Generate the mesh file based on this individual."""
        return eos.generate_eos_table(path, RVMESH, TMESH, R0, self.Vs,
                                     self.K0, self.K1, self.theta0, self.A0,
                                     self.A1, self.A2, self.A3, self.A4,
                                     self.B0, self.B1, self.B2, self.B3,
                                     self.B4, Cv, self.c0, self.c1, self.c2,
                                     self.c3)

```

```

def __str__(self):
    """Convert the individual to a string."""
    string = ""
    for i in self.params:
        string = string + str(i) + ","
    return string

def __hash__(self):
    """Return the hash of this individual."""
    return hash(tuple(self.params))

def __getitem__(self, key):
    try:
        return self.params[key]
    except (TypeError, IndexError, KeyError) as e:
        raise e

def __setitem__(self, key, value):
    self.params[key] = value

def __len__(self):
    return len(self.params)

# These functions are needed so that the variables can be accessed by name
# while still being accessible as a list. These functions could be
# generated at runtime to make the class definition clearer but that would
# prevent the individuals from being pickled

@property
def K0(self):
    return self.params[0]

@property
def K1(self):
    return self.params[1]

```

```

# this was changed from an optimization parameter to a constant of 1
# this was left as a property in order to not break other things

@property
def theta0(self):
    return self.params[2]

@property
def A0(self):
    return self.params[2]

@property
def A1(self):
    return self.params[3]

@property
def A2(self):
    return self.params[4]

@property
def A3(self):
    return self.params[5]

@property
def A4(self):
    return self.params[6]

@property
def B0(self):
    return self.params[7]

@property
def B1(self):
    return self.params[8]

```

```

@property
def B2(self):
    return self.params[9]

@property
def B3(self):
    return self.params[10]

@property
def B4(self):
    return self.params[11]

@property
def c0(self):
    return self.params[12]

@property
def c1(self):
    return self.params[13]

@property
def c2(self):
    return self.params[14]

@property
def c3(self):
    return self.params[15]

def __eq__(self, other):
    if isinstance(other, type(self)):
        for s, o in zip(self, other):
            if s != o:
                return False
        return True
    else:

```

```

        return NotImplemented

numruns = 0
def evalWrapper(x):
    global numruns
    r = evaluate(Individual(x), 0, numruns)
    numruns += 1
    return r[0][0]

def evaluate(ind, gen, indNum, showplot=False): # basePath=".."):
    """Evaluate the individual."""
    try:
        originalIndNum = indNum
        global seenInds
        if hash(ind) in seenInds:
            print("seen ind " + str(indNum) + " before, returning")
            total_error = seenInds[hash(ind)][0]
            gen = seenInds[hash(ind)][1]
            indNum = seenInds[hash(ind)][2]
            path = seenInds[hash(ind)][3]
            return (total_error,)
        try:
            os.mkdir(basePath + "/Gen" + str(gen))
        except FileExistsError:
            pass

        path = basePath + "/Gen" + str(gen) + "/Ind" + str(indNum)
        os.mkdir(path)

        ↪ shutil.copy("/home/matthew/Documents/SpecificHeatFit/Current_2021-07-19/src/matinp",
        ↪ path + "/matinp")
        #print("Generating mesh " + str(os.getpid()))
        #with warnings.catch_warnings():

```

```

PRES, ENRG = ind.mesh(path)

#print("Plotting " + str(os.getpid()))

os.mkdir(path + "/Plots")

if showplot:

    plot.plot_cold_curve(path, RVMESH, V0, ind.Vs, ind.K0, ind.K1)

    plot.plot_debye(path, RVMESH, V0, ind.theta0, ind.A0, ind.A1, ind.A2, ind.A3,
        ↪ ind.A4, ind.B0, ind.B1, ind.B2, ind.B3, ind.B4)

    plot.plot_gruneisen(path, RVMESH, V0, ind.theta0, ind.A0, ind.A1, ind.A2,
        ↪ ind.A3, ind.A4, ind.B0, ind.B1, ind.B2, ind.B3, ind.B4)

    plot.plot_isotherms(path, RVMESH, TMESH, R0, PRES, ENRG)

#print("Gen Sesame " + str(os.getpid()))

bcat.gen_sesame(path, "chtinp", NRV, NT, ESFT, FZ, FW, PREF, TREF, RHOREF,
    MATID, bin_filename, asc_filename)

#print("starting comparisons " + str(os.getpid()))

compression_error = evaluation.compare_compression(path, R0, ind.K0, ind.K1,
    ↪ path[40:])

#print("Compression error = " + str(compression_error))

#print(repr(ind.K0) + " " + str(ind.K1))

#raise RuntimeError

hugoniot_error = evaluation.compare_hugoniot(path, R0, PREF, TREF, path[40:])

errors = evaluation.thermal_comparison(path,
    Cv, ind.c0, ind.c1, ind.c2,
    ind.c3, PREF, R0, path[40:])

V0_error, Bt_error, CV_error, alpha_error = errors

# print("done " + str(os.getpid()))

evaluation.check_isotherms(path, R0)

weights = (
    5000,    # V0
    5,       # Bt
    6,       # CV
    100,     # compression
    100,     # hugoniot
    50,      # alpha

```

```

)

total_error = V0_error * weights[0] + Bt_error * weights[1] + CV_error *
↳ weights[2] + compression_error * weights[3] + hugoniot_error * weights[4] +
↳ alpha_error * weights[5]
#total_error = compression_error + hugoniot_error * 10

seenIndsLock.acquire()
seenInds[hash(ind)] = (total_error, gen, indNum, path)
seenIndsLock.release()

global allErrors
allErrors = (compression_error, hugoniot_error, V0_error, Bt_error, CV_error,
↳ alpha_error)

with open(path + "/results.txt", "w+") as f:
    f.write(str(ind) + "\n")
    f.write(str(total_error) + "\n")
    f.write(f"Compression Error {compression_error} * {weights[3]} =
↳ {compression_error * weights[3]} \n")
    f.write(f"Hugoniot Error {hugoniot_error} * {weights[4]} = {hugoniot_error *
↳ weights[4]} \n")
    f.write(f"V0 Error {V0_error} * {weights[0]} = {V0_error * weights[0]} \n")
    f.write(f"Bt Error {Bt_error} * {weights[1]} = {Bt_error * weights[1]} \n")
    f.write(f"CV Error {CV_error} * {weights[2]} = {CV_error * weights[2]} \n")
    f.write(f"Alpha Error {alpha_error} * {weights[5]} = {alpha_error *
↳ weights[5]} \n")

except Exception as e:
    if "errWriteQueue" in globals():
        global errWriteQueue
        errWriteQueue.put("Gen " + str(gen) + " Ind " + str(indNum) + " " +
↳ str(os.getpid()) + "\n" + str(ind) + " " + str(e) + "\n" +
↳ traceback.format_exc())
    # raise e
total_error = np.inf

```

```

finally:
    print("ind " + str(originalIndNum) + " done. pid " + str(os.getpid()) +
          " Total error = " + str(total_error))
    # return total_error
    return ((total_error,), gen, indNum, path)

def CreateInd(bounds=bounds):
    """Create a new Individual."""
    args = [None] * len(bounds)

    for i, bound in enumerate(bounds):
        args[i] = abs(random.normalvariate(bound[0], bound[1])) * bound[2]

    args[2] = abs(args[2])

    ind = Individual(args)
    return ind

def Crossover(ind1, ind2, icxpb=icxpb):
    for i in range(len(ind1)):
        if random.random() < icxpb:
            ind1[i], ind2[i] = ind2[i], ind1[i]
    return ind1, ind2

def Mutate(ind, bounds=bounds, imutpb=imutpb, scpb=scpb):
    for i in range(len(ind)):
        if random.random() < imutpb:
            if random.random() < scpb:
                ind[i] = random.triangular(bounds[i][0] - 2 * bounds[i][1], bounds[i][0]
                    + 2 * bounds[i][1], ind[i])
            else:
                ind[i] = abs(random.normalvariate(bounds[i][0], bounds[i][1]))

```

```

    return ind

def init(seen, base, queue):
    """Function for initializing pool workers."""
    # print("calling pool init")
    global seenInds
    seenInds = seen

    global basePath
    basePath = base

    global errWriteQueue
    errWriteQueue = queue

def _fitnessGetter(ind):
    return ind.fitness.values

def _bettermean(data):
    cleanData = [x for x in data if not math.isinf(x[0])]
    return np.mean(cleanData)

def main():
    """Call this function."""
    global problem

    gaParams = {
        "ngen": 1,
        "cxpb": .6,
        "mutpb": .75,
        "popsize": 50
    }

```

```

stats = deap.tools.Statistics(_fitnessGetter)
stats.register("avg", _bettermean)
stats.register("min", np.min)
stats.register("max", np.max)

hof = deap.tools.HallOfFame(3)

problem = OptimizationMain.Problem(CreateInd, evaluate, Crossover, Mutate,
                                   gaParams, poolinit=init,
                                   poolinitargs=(seenIndsProxy, BasePath),
                                   stats=stats, halloffame=hof,
                                   verbose=True)

# try:
#     problem.optimise()
# except Exception as e:
#     print(e)
# print("done")
problem.pop.append(Individual([25.54093587069407, 3.9870467796057154, 1,
→ 7.217472521577932, 3.1361469805728, 0.6206369036588963, 1.1108183680178492,
→ 0.7802044469350626, 1.8438201089936035, -0.7733702322562603, 6.521304027609043,
→ -0.14823563076884838, 3.728106115114798, 199410.08451303848, -7392.780519167716,
→ 174.3828688004568, 0.44222810391450684]))
with ProcessQueue.ProcessQueueManager(ProcessQueue.WriteFunc, "STOP", BasePath +
→ "/IndErrors.txt") as q:
    problem.poolinitargs = problem.poolinitargs + (q,)
    problem.optimise()

return problem

def mainCP(pb):
    global problem
    problem = pb

```

```

with ProcessQueue.ProcessQueueManager(ProcessQueue.WriteFunc, "STOP", BasePath +
↳  "/IndErrors.txt") as q:
    problem.poolinitargs = problem.poolinitargs + (q,)
    problem.moreGens(250)

return problem

if __name__ == "_n_main_":
    BasePath = BaseBasePath + "/Run" + str(len(os.listdir(BaseBasePath)))
    os.mkdir(BasePath)
    if os.path.isfile(seenIndsFile):
        with open(seenIndsFile, "rb") as f:
            seenIndsBase = pickle.load(f)
    else:
        seenIndsBase = {}

    manager = multiprocessing.Manager()
    seenIndsProxy = manager.dict(seenIndsBase)
    init(seenIndsProxy, BasePath)
    if problemCPFile is None:
        main()
    else:
        with open(problemCPFile, "rb") as f:
            pb = pickle.load(f)
        mainCP(pb)
    ind = Individual([15.16, 6.4, 1.0, 3.0, 0.0, 2.0, -2.0, 2., 2.0, 3.0, 2.0, 2.0, 1.8])

if __name__ == "__main__":
    if problemCPFile is None:
        BasePath = BaseBasePath + "/Run" + str(len(os.listdir(BaseBasePath)))
        os.mkdir(BasePath)
        if os.path.isfile(seenIndsFile):
            with open(seenIndsFile, "rb") as f:
                seenIndsBase = pickle.load(f)

```

```

else:
    seenIndsBase = {}
    manager = multiprocessing.Manager()
    seenIndsProxy = manager.dict(seenIndsBase)
    init(seenIndsProxy, BasePath, None)
    main()

else:
    with open(problemCPFile, "rb") as f:
        pb = pickle.load(f)
        BasePath = pb.poolinitargs[1]
        manager = multiprocessing.Manager()
        seenIndsBase = {} # pb.poolinitargs[0]
        seenIndsProxy = manager.dict(seenIndsBase)
        init(seenIndsProxy, BasePath, None)
        pb.poolinitargs = (seenIndsProxy, pb.poolinitargs[1])
        pb.gaParams["cxpb"] = .6
        pb.gaParams["mutpb"] = .75
        pb.popsizes = 250
        if not hasattr(pb, "evalArgs"):
            pb.evalArgs = ()
        if True:
            with ProcessQueue.ProcessQueueManager(ProcessQueue.WriteFunc, "STOP",
                ↪ BasePath + "/IndErrors.txt") as q:
                pb.poolinitargs = pb.poolinitargs + (q,)
                pb.reEval()
                pb.poolinitargs = pb.poolinitargs[0:2]
        if True:
            mainCP(pb)

if __name__ == "_n_main_":
    BasePath = BaseBasePath + "/Min2"
    basePath = BasePath
    os.mkdir(BasePath)
    seenInds = {}

```

```

result = scipy.optimize.minimize(evalWrapper,
    ↪ np.array([25.54093587069407,3.9870467796057154,5.921812555628668,7.217472521577932,5.3264818058
    ↪ method="BFGS")

```

A.5.2 OptimizationMain

Classes and functions for initializing the genetic algorithm for DEAP.

```

# -*- coding: utf-8 -*-

```

```

"""

```

```

Created on Mon Jun 7 16:10:36 2021

```

```

@author: mvandenboom

```

```

"""

```

```

import math

```

```

import operator

```

```

import random

```

```

import multiprocessing

```

```

import deap.tools

```

```

import copy

```

```

import pickle

```

```

import datetime

```

```

class Problem:

```

```

    def __init__(self, indCreator, evaluate, crossover, mutation, gaParams, evalArgs=(),
        stats=None, halloffame=None, endGen=None, verbose=__debug__, **kwargs):

```

```

        """

```

```

        Create the problem object.

```

```

        Parameters

```

```

        -----

```

```

        indCreator : callable

```

```

            function that takes no arguments and returns a new individual.

```

```

        evaluate : callable

```

```

            function that takes in a single individual, the generation, and

```

```

            the number of the individual and returns a tuple of results, the

```

first of which is the fitness of the individual as a tuple, and everything else will be stored in the `otherResults` attribute of the individual. Only the fitness will be used in the evolution, the rest are stored for other use.

Ex return: ((fitness,), path_to_plots, time_taken)

`crossover` : callable

function that crosses over two individuals. The only parameters are the two individuals.

`mutation` : callable

function that mutates a single individual. The only parameter is the individual.

`stats` : `deap.tools.Statistics` object, optional

Comple statistics on the population. The default is `None`.

`halloffame` : `deap.tools.HallOfFame`, optional

keep track of the best individuals across the entire optimization. The default is `None`.

`endGen` : callable, optional

if present will be called at the end of every generation with the parameters in `endGenArgs` dict, if present. It will be the last call during the generation, after logging. The default is `None`.

`verbose` : bool, optional

Whether to print out the logs as they are taken. The default is `__debug__`.

`gaParams` : Dictionary

Dictionary containing parameters for the genetic algorithm. Currently required keys:

- `popsize` : int
 - how many individuals are in the population.
- `ngen` : int
 - how many generations should the genetic algorithm be run.
- `cxpb` : float [0.0, 1.0]
 - chance for two individuals being mated
- `mutpb` : float [0.0, 1.0]
 - chance for an individual to be mutated

Returns

None.

"""

```
self.pop = [indCreator() for i in range(gaParams["popsize"])]
self.popsize = gaParams["popsize"]
self.evaluate = evaluate
self.crossover = crossover
self.mutation = mutation
self.ngen = gaParams["ngen"]
self.stats = stats
self.hof = halloffame
self.verbose = verbose
self.currGen = 0
self.indCreator = indCreator
# self.endGen = endGen
self.gaParams = gaParams
self.kwargs = kwargs
self.evalArgs = evalArgs
if "poolinit" in kwargs:
    self.poolinit = kwargs["poolinit"]
else:
    self.poolinit = lambda *args: None

if "poolinitargs" in kwargs:
    self.poolinitargs = kwargs["poolinitargs"]
else:
    self.poolinitargs = tuple()

self.prevGens = {}

self.fulllogbook = ""
```

```

self.reEvalNum = -1

def evalWrapper(self, args):
    """Wrap around the evaluate function to expand zipped args."""
    return self.evaluate(*args)

def _genArgs(self, inds, genNum, evalArgs):
    args = []
    for i, ind in enumerate(inds):
        args.append((ind, genNum, i) + evalArgs)
    return args

def evalPop(self):
    """Evaluate the entire population.

The end result of this function is every individual in the population
has a valid fitness.

Returns: total number of evaluations completed.

This function can be overridden in order to get runtime speed
improvements in certain problems.
    """
    invalid_ind = [ind for ind in self.pop if not ind.fitness.valid]
    with multiprocessing.Pool(20, initializer=self.poolinit,
        ↪ initargs=self.poolinitargs) as p:
        results = p.map(self.evalWrapper, self._genArgs(invalid_ind, self.currGen,
        ↪ self.evalArgs))
    # print("finished evaling, setting fitnesses")
    for ind, res in zip(invalid_ind, results):
        ind.fitness.values = res[0]
        ind.otherResults = res[1:]
    # print("set fitnesses, returning")
    return len(invalid_ind)

def updateHOF(self):

```

```

        """Update the Hall of Fame with the current population."""
        if self.hof is not None:
            self.hof.update(self.pop)

def logStats(self, gen, nevals):
    """Log the statistics from a generation of individuals."""
    record = self.stats.compile(self.pop) if self.stats else {}
    self.logbook.record(gen=gen, nevals=nevals, **record)
    self.fulllogbook += "\n" + self.logbook.stream
    if self.verbose:
        print(self.fulllogbook)

    self.prevGens[gen] = copy.deepcopy(self.pop)

def _tournament(self, k, tournsize):
    allowable = [ind for ind in self.pop if not math.isinf(
        ind.fitness.values[0])]
    if len(allowable) == 0:
        return []
    chosen = []
    for i in range(k):
        aspirants = deap.tools.selRandom(allowable, tournsize)
        chosen.append(max(aspirants, key=operator.attrgetter("fitness")))
    return chosen

def selectPop(self):
    """Generate the population for the next generation."""
    offspring = self._tournament(int(self.popsiz * .8), 3)
    # print("offspring is " + str(len(offspring)) + " long")

    if self.hof is not None:
        offspring.append(self.hof[0])
        offspring.append(self.hof[0])

    offspring += [self.indCreator() for i in range(

```

```

        self.popsizes - len(offspring))]]

try:
    assert(len(self.pop) == len(offspring))

    self.pop[:] = offspring

except AssertionError:
    assert(len(offspring) == self.popsizes)
    del self.pop
    self.pop = offspring

random.shuffle(self.pop)

def varyPop(self):
    """Modify the population through crossover and mutation."""
    offspring = [copy.deepcopy(ind) for ind in self.pop]
    for i in range(1, len(offspring), 2):
        if random.random() < self.gaParams["cxpb"]:
            offspring[i - 1], offspring[i] = self.crossover(
                offspring[i - 1], offspring[i])
            del offspring[i - 1].fitness.values
            del offspring[i].fitness.values

    for i in range(len(offspring)):
        if random.random() < self.gaParams["mutpb"]:
            offspring[i] = self.mutation(offspring[i])
            del offspring[i].fitness.values

    self.pop = offspring

def optimise(self):
    """Run the genetic algorithm to optimize the population."""
    self.currGen = 0
    self.logbook = deap.tools.Logbook()

```

```

self.logbook.header = ['gen', 'nevals'] + \
    (self.stats.fields if self.stats else [])
print("evaling gen 0")
nevals = self.evalPop()
# print("updating hof gen 0")
self.updateHOF()
# print("logging stats gen 0")
self.logStats(0, nevals)

if self.endGen is not None:
    if "endGenArgs" in self.kwargs:
        self.endGen(**self.kwargs["endGenArgs"])
    else:
        self.endGen()

for gen in range(1, self.ngen + 1):
    self.currGen = gen
    # print("selecting pop for gen " + str(gen))
    self.selectPop()
    # print("varying the pop gen " + str(gen))
    self.varyPop()
    print("evaling gen " + str(gen))
    nevals = self.evalPop()
    # print("done evaling gen " + str(gen))

    self.updateHOF()

    self.logStats(gen, nevals)

if self.endGen is not None:
    if "endGenArgs" in self.kwargs:
        self.endGen(**self.kwargs["endGenArgs"])
    else:
        self.endGen()

```

```

def moreGens(self, num=1):
    """
    Run more generations of the evolution.

    This should be called after optimize() has been called and finished.

    This must be run from within a script, rather than the interactive
    interpreter because it uses multiprocessing.Pool which doesn't work in
    the interpreter.

    If any changes have been made since calling optimize() or a previous
    call of this function that would result in invalid fitnesses, call
    reEval() first.

        These changes would include adding a new individual to the
        population that doesn't have a valid fitness or changing the
        evaluation function so that fitnesses from before the change can't
        be directly compared.

    Parameters
    -----
    num : int, optional
        Number of generations to run. The default is 1.

    Returns
    -----
    None.

    """
    for gen in range(num):
        self.currGen += 1
        # print("selecting pop for gen " + str(gen))
        self.selectPop()
        # print("varying the pop gen " + str(gen))
        self.varyPop()
        print("evaling gen " + str(self.currGen))

```

```

nevals = self.evalPop()
# print("done evaling gen " + str(gen))

self.updateHOF()

self.logStats(self.currGen, nevals)

if self.endGen is not None:
    if "endGenArgs" in self.kwargs:
        self.endGen(**self.kwargs["endGenArgs"])
    else:
        self.endGen()

def pickleProblem(self, path=None):
    if path is None:
        path = self.poolinitargs[1] + "/Gen" + str(self.currGen) + "/cp.pkl"
    other = copy.deepcopy(self)
    other.poolinitargs = (dict(other.poolinitargs[0]), other.poolinitargs[1])
    try:
        del other.kwargs["poolinitargs"]
    except KeyError:
        pass
    with open(path, "wb+") as f:
        pickle.dump(other, f)

def endGen(self, *args, **kwargs):
    try:
        with open(self.poolinitargs[1] + "/Gen" + str(self.currGen) + "/endTime.txt",
            ↪ "w+") as f:
            f.write(str(datetime.datetime.now()))
    except:
        pass
    self.pickleProblem()

```

```

def reEval(self):
    """Re-evaluate all of the individuals in pop and the hall of fame"""
    realGen = self.currGen
    for i in self.hof:
        self.pop.append(i)
    self.currGen = self.reEvalNum
    for i in range(len(self.pop)):
        del self.pop[i].fitness.values
    self.evalPop()
    self.reEvalNum -= 1
    self.currGen = realGen
    if self.hof is not None:
        self.hof.clear()
        self.updateHOF()

```

A.5.3 EOS_Functions

Functions for computing necessary equation of state parameters.

```

# -*- coding: utf-8 -*-
"""
Created on Thu May 27 15:01:16 2021.

@author: jmaestas
*****
Equation of State Functions

Sets of functions for investigating complete equations of state.

*****
"""

import numpy as np
import os
import scipy.integrate as integrate

```

```

def Pc(V, V0, K0, K1):
    """Cold curve description. Using a Birch-Murnaghan model.

    Inputs
    -----
    V [float] - volume
    V0 [float] - reference volume
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure

    Outputs
    -----
    pressure [float] - pressure at volume

    """
    pressure = (
        1.5 * K0 * ((V / V0) ** (-7. / 3.) - (V / V0) ** (-5. / 3.))
        * (1.0 + 0.75 * (K1 - 4.0) * ((V / V0) ** (-2. / 3.) - 1.0))
    )
    return pressure

```

```

def P0(V, V0, Vs, K0, K1):
    """Modify cold curve pressure to treat expansion region.

    Inputs
    -----
    V [float] - volume
    V0 [float] - reference volume
    Vs [float] - volume at inflection point
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure

    Outputs

```

```

-----
pressure [float] - pressure at volume

"""
if V <= Vs:
    pressure = Pc(V, V0, K0, K1)
else:
    pressure = Pc(Vs, V0, K0, K1)
return pressure

def CV(TS, Cv, c0, c1, c2, c3):
    """Generate specific heat as a function of scaled temperature.

    Inputs
    -----
    TS [float] - scaled temperature (dimensionless)
    Cv [float] - scaling parameter
    c0 [float] - multiplier
    c1 [float] - multiplier
    c2 [float] - multiplier
    c3 [float] - multiplier

    Outputs
    -----
    specific_heat [float] - specific heat

    """
    specific_heat = (
        (Cv * TS ** 3.) /
        (c0 + c1 * TS + c2 * TS ** 2. + c3 * TS ** 3.)
    )
    return specific_heat

```

```

def THETA(V, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4):
    """Generate Debye temperature for scaling CV.

    Inputs
    -----
    V [float] - volume
    V0 [float] - reference volume
    theta0 [float] - scaling parameter
    A0 [float] - numerator multiplier
    A1 [float] - numerator multiplier
    A2 [float] - numerator multiplier
    A3 [float] - numerator multiplier
    A4 [float] - numerator multiplier
    B0 [float] - denominator multiplier
    B1 [float] - denominator multiplier
    B2 [float] - denominator multiplier
    B3 [float] - denominator multiplier
    B4 [float] - denominator multiplier

    Outputs
    -----
    theta [float] - Debye temperature

    """

    x = V / V0

    result = ((A0 + A1 * x + A2 * x**2 + A3 * x**3 + A4 * x**4) /
              (B0 + B1 * x + B2 * x**2 + B3 * x**3 + B4 * x**4))

    return result


def gruneisen(V, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4):
    """Numerically compute Gruneisen parameter as a function of volume.

```

Inputs

V [float] - volume

V0 [float] - reference volume

theta0 [float] - scaling parameter

A0 [float] - Debye Temp Model: numerator multiplier

A1 [float] - Debye Temp Model: numerator multiplier

A2 [float] - Debye Temp Model: numerator multiplier

A3 [float] - Debye Temp Model: numerator multiplier

A4 [float] - Debye Temp Model: numerator multiplier

B0 [float] - Debye Temp Model: denominator multiplier

B1 [float] - Debye Temp Model: denominator multiplier

B2 [float] - Debye Temp Model: denominator multiplier

B3 [float] - Debye Temp Model: denominator multiplier

B4 [float] - Debye Temp Model: denominator multiplier

Outputs

gamma [float] - Gruneisen parameter

"""

Compute Debye temperature at V

theta = THETA(V, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

Compute derivative of Debye temperature wrt volume using numerical

differentiation. Using 5 point stencil.

h = 0.01 # relative step size

*theta_Vp1 = THETA(V * (1. + 1. * h), V0, theta0,*
A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

*theta_Vp2 = THETA(V * (1. + 2. * h), V0, theta0,*
A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

*theta_Vm2 = THETA(V * (1. - 2. * h), V0, theta0,*
A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

*theta_Vm1 = THETA(V * (1. - 1. * h), V0, theta0,*

```

        A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

dtheta_dV = (-theta_Vp2 + 8. * theta_Vp1 - 8. * theta_Vm1 +
            theta_Vm2) / (12. * h * V)

gamma = (-V / theta) * dtheta_dV
return gamma

def F(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4,
    Cv, c0, c1, c2, c3):
    """Create Hayes model of the Helmholtz free energy.

Inputs
    -----
    V [float] - volume
    T [float] - temperature
    V0 [float] - reference volume
    Vs [float] - volume at inflection point
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure
    theta0 [float] - scaling parameter
    A0 [float] - Debye Temp Model: numerator multiplier
    A1 [float] - Debye Temp Model: numerator multiplier
    A2 [float] - Debye Temp Model: numerator multiplier
    A3 [float] - Debye Temp Model: numerator multiplier
    A4 [float] - Debye Temp Model: numerator multiplier
    B0 [float] - Debye Temp Model: denominator multiplier
    B1 [float] - Debye Temp Model: denominator multiplier
    B2 [float] - Debye Temp Model: denominator multiplier
    B3 [float] - Debye Temp Model: denominator multiplier
    B4 [float] - Debye Temp Model: denominator multiplier
    TS [float] - scaled temperature (dimensionless)
    Cv [float] - Specific Heat Model: scaling parameter
    c0 [float] - Specific Heat Model: multiplier

```

```

c1 [float] - Specific Heat Model: multiplier
c2 [float] - Specific Heat Model: multiplier
c3 [float] - Specific Heat Model: multiplier

Outputs
-----

helmholtz_energy [float] - Helmholtz free energy

"""

# Calculated Debye temperature
theta = THETA(V, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)

# compute cold curve integration
if V == V0:
    P0dV = 0.
else:
    P0dV, P0dV_err = integrate.quad(lambda x: P0(x, V0, Vs, K0, K1), V0, V)
    # check (relative) numerical integration error
    if (P0dV_err * 100.0 / P0dV > 1.):
        raise ValueError('Numerical integration error: cold curve')

# compute temperature integration
if T == 0.0:
    TdT = 0.
else:
    TdT, TdT_err = integrate.quad(
        lambda x: ((1. / x) * (T - x) * CV(x / theta, Cv, c0, c1, c2, c3)),
        0, T)
    # check (relative) numerical integration error
    if (TdT_err * 100.0 / TdT > 1.):
        raise ValueError('Numerical integration error: temperature')

# compute Helmholtz free energy
helmholtz_energy = -P0dV - TdT
return helmholtz_energy

```

```

def dFdV(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4,
        Cv, c0, c1, c2, c3, h=0.01):
    """Numerically compute derivative of F wrt volume.

    Inputs
    -----
    V [float] - volume
    T [float] - temperature
    V0 [float] - reference volume
    Vs [float] - volume at inflection point
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure
    theta0 [float] - scaling parameter
    A0 [float] - Debye Temp Model: numerator multiplier
    A1 [float] - Debye Temp Model: numerator multiplier
    A2 [float] - Debye Temp Model: numerator multiplier
    A3 [float] - Debye Temp Model: numerator multiplier
    A4 [float] - Debye Temp Model: numerator multiplier
    B0 [float] - Debye Temp Model: denominator multiplier
    B1 [float] - Debye Temp Model: denominator multiplier
    B2 [float] - Debye Temp Model: denominator multiplier
    B3 [float] - Debye Temp Model: denominator multiplier
    B4 [float] - Debye Temp Model: denominator multiplier
    TS [float] - scaled temperature (dimensionless)
    Cv [float] - Specific Heat Model: scaling parameter
    c0 [float] - Specific Heat Model: multiplier
    c1 [float] - Specific Heat Model: multiplier
    c2 [float] - Specific Heat Model: multiplier
    c3 [float] - Specific Heat Model: multiplier
    h [float] - relative step size for finite difference

    Outputs
    -----

```

```

    result [float] - dFdV

    """

    # Compute dFdV. Using 5-point stencil
    F_Vp1 = F(V * (1. + 1. * h), T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
              B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
    F_Vp2 = F(V * (1. + 2. * h), T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
              B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
    F_Vm2 = F(V * (1. - 2. * h), T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
              B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
    F_Vm1 = F(V * (1. - 1. * h), T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
              B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)

    result = (-F_Vp2 + 8. * F_Vp1 - 8. * F_Vm1 + F_Vm2) / (12. * h * V)

    return result

def dFdT(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4,
        Cv, c0, c1, c2, c3, h=0.01):
    """Numerically compute derivative of F wrt temperature.

    Inputs
    -----
    V [float] - volume
    T [float] - temperature
    V0 [float] - reference volume
    Vs [float] - volume at inflection point
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure
    theta0 [float] - scaling parameter
    A0 [float] - Debye Temp Model: numerator multiplier
    A1 [float] - Debye Temp Model: numerator multiplier
    A2 [float] - Debye Temp Model: numerator multiplier
    A3 [float] - Debye Temp Model: numerator multiplier

```

A4 [float] - Debye Temp Model: numerator multiplier
B0 [float] - Debye Temp Model: denominator multiplier
B1 [float] - Debye Temp Model: denominator multiplier
B2 [float] - Debye Temp Model: denominator multiplier
B3 [float] - Debye Temp Model: denominator multiplier
B4 [float] - Debye Temp Model: denominator multiplier
TS [float] - scaled temperature (dimensionless)
Cv [float] - Specific Heat Model: scaling parameter
c0 [float] - Specific Heat Model: multiplier
c1 [float] - Specific Heat Model: multiplier
c2 [float] - Specific Heat Model: multiplier
c3 [float] - Specific Heat Model: multiplier
h [float] - relative step size for finite difference

Outputs

result [float] - dFdT

"""

Compute dFdT using a 5-point stencil. When T = 0, use a basic 2-point
stencil.

if T == 0.0:

F_c = F(V, 0.0, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
F_Tp1 = F(V, 1.0, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
result = F_Tp1 - F_c

else:

*F_Tp1 = F(V, T * (1. + 1. * h), V0, Vs, K0, K1, theta0, A0, A1, A2, A3,*
A4, B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
*F_Tp2 = F(V, T * (1. + 2. * h), V0, Vs, K0, K1, theta0, A0, A1, A2, A3,*
A4, B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
*F_Tm2 = F(V, T * (1. - 2. * h), V0, Vs, K0, K1, theta0, A0, A1, A2, A3,*
A4, B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
*F_Tm1 = F(V, T * (1. - 1. * h), V0, Vs, K0, K1, theta0, A0, A1, A2, A3,*

```

        A4, B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)

    result = (-F_Tp2 + 8. * F_Tp1 - 8. * F_Tm1 + F_Tm2) / (12. * h * T)

    return result

def generate_eos_table(BasePath, RVMESH, TMESH, R0, Vs, K0, K1, theta0, A0, A1, A2, A3,
        A4, B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3):
    """Generate the EOS table in Cheetah format.

    Inputs
    -----
    RVMESH [float] - relative volume array
    TMESH [float] - temperature array
    R0 [float] - reference density
    Vs [float] - volume at inflection point
    K0 [float] - bulk modulus
    K1 [float] - derivative of bulk modulus wrt pressure
    theta0 [float] - scaling parameter
    A0 [float] - Debye Temp Model: numerator multiplier
    A1 [float] - Debye Temp Model: numerator multiplier
    A2 [float] - Debye Temp Model: numerator multiplier
    A3 [float] - Debye Temp Model: numerator multiplier
    A4 [float] - Debye Temp Model: numerator multiplier
    B0 [float] - Debye Temp Model: denominator multiplier
    B1 [float] - Debye Temp Model: denominator multiplier
    B2 [float] - Debye Temp Model: denominator multiplier
    B3 [float] - Debye Temp Model: denominator multiplier
    B4 [float] - Debye Temp Model: denominator multiplier
    Cv [float] - Specific Heat Model: scaling parameter
    c0 [float] - Specific Heat Model: multiplier
    c1 [float] - Specific Heat Model: multiplier
    c2 [float] - Specific Heat Model: multiplier
    c3 [float] - Specific Heat Model: multiplier

```

Outputs

PRES [array] - matrix of pressures

ENRG [array] - matrix of energies

"""

VO = 1. / R0 # cc/g, initial specific volume

Compute preliminary density array

RHOMESH = R0 / RVMESH

Write pressure and energy to arrays for plotting and debugging

NRV = len(RVMESH)

NT = len(TMESH)

PRES = np.zeros([NT, NRV])

ENRG = np.zeros([NT, NRV])

Initialize a Cheetah file to write to and write out column labels

output = BasePath + '/chtinp'

Remove any old file

if os.path.exists(output):

os.remove(output)

fout = open(output, 'w')

fout.write('T (K), V (cc/g), P (atm), E (cal/g), S (cal/g-K) \n')

Loop through densities and temperatures

i = 0

for T in TMESH:

j = 0

*if (-1) ** i > 0:*

RMESH = RHOMESH

else:

```

    RMESH = np.flipud(RHOMESH)
for R in RMESH:
    # Compute necessary terms
    V = 1.0 / R

    # Compute center point
    F_p = F(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
            B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)

    # Compute derivatives of F
    dFdV_p = dFdV(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
                   B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)
    dFdT_p = dFdT(V, T, V0, Vs, K0, K1, theta0, A0, A1, A2, A3, A4,
                   B0, B1, B2, B3, B4, Cv, c0, c1, c2, c3)

    # Compute pressure, energy, and entropy
    P = -dFdV_p
    E = F_p - T * dFdT_p
    S = -dFdT_p

    PRES[i, j] = P
    ENRG[i, j] = E

    # Convert units
    P *= 9869.23266716 # GPa to atm
    E *= 239.00574 # GPa/g/cc to cal/g
    S *= 239.00574 # GPa/gK/cc to cal/g-K

    # Write data to the file
    line = (f'{T} {V} {P} {E} {S}\n')
    fout.write(line)

    j += 1
i += 1

```

```
# Close the file
fout.close()
```

```
return PRES, ENRG
```

A.5.4 EOS_Plots

Functions for plotting equation of state comparisons and checks.

```
# -*- coding: utf-8 -*-
```

```
"""
```

```
Created on Thu May 27 16:31:57 .
```

```
@author: jmaestas
```

```
*****
```

```
Equation of State Plots
```

```
Plot out various parameters.
```

```
*****
```

```
"""
```

```
import eos_functions as eos
```

```
import numpy as np
```

```
import matplotlib.pyplot as plt
```

```
import os
```

```
saveDPI = 500
```

```
def plot_cold_curve(basePath, RVMESH, V0, Vs, K0, K1):
```

```
    """"Cold curve plot.
```

```
    Inputs
```

```
    -----
```

```
    RVMESH [array] - relative volume array
```

```
    V0 [float] - reference volume
```

```
    Vs [float] - volume at inflection point
```

K0 [float] - bulk modulus

K1 [float] - derivative of bulk modulus wrt pressure

Outputs

"""

VMESH = RVMESH * V0

P0_plot = [eos.P0(v, V0, Vs, K0, K1) for v in VMESH]

fig, ax = plt.subplots(1, 1)

ax.plot(RVMESH, P0_plot)

Figure properties

fig.suptitle('Cold Curve', fontsize=20, fontweight='bold')

ax.set_title('AP Model', fontsize=18)

ax.set_xlabel(r'\$V/V_0\$', fontsize=18)

ax.set_ylabel(r'\$P_0 \backslash, (GPa)\$', fontsize=18)

ax.set_xlim([0.4, 2.])

ax.set_ylim([-5, 150.])

ax.grid(True)

plt.tick_params(axis='both', which='major', labelsize=15)

figure = plt.gcf() *# get current figure*

figure.set_size_inches(8, 6)

if not os.path.exists(basePath + "/Plots"):

os.mkdir(basePath + "/Plots")

plt.savefig(basePath + "/Plots/cold_curve.png", format='png', dpi=saveDPI)

def plot_specific_heat(basePath, TMESH, Cv, c0, c1, c2, c3):

"""Specific heat plot.

Inputs

```

    TMESH [array] - temperature array
    Cv [float] - scaling parameter
    c0 [float] - multiplier
    c1 [float] - multiplier
    c2 [float] - multiplier
    c3 [float] - multiplier

    Outputs
    -----

    """

fig, ax = plt.subplots(1, 1)
ax.plot(TMESH, 1000 * eos.CV(TMESH, Cv, c0, c1, c2, c3))

# Figure properties
fig.suptitle('Specific Heat', fontsize=20, fontweight='bold')
ax.set_title('AP Model', fontsize=18)
ax.set_xlabel(r'$T \backslash, (K)$', fontsize=18)
ax.set_ylabel(r'$C_v \backslash, (J/g-K)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
if not os.path.exists(basePath + "/Plots"):
    os.mkdir(basePath + "/Plots")
plt.savefig(basePath + "/Plots/specific_heat.png", format='png', dpi=saveDPI)

def plot_debye(basePath, RVMESH, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4):
    """Debye temperature plot.

    Inputs
    -----

    RVMESH [array] - relative volume array
    V0 [float] - reference volume

```

```

theta0 [float] - scaling parameter
A1 [float] - Einstein oscillator multiplier
A2 [float] - Einstein oscillator multiplier
A3 [float] - Einstein oscillator multiplier
B1 [float] - Einstein oscillator scaler
B2 [float] - Einstein oscillator scaler
B3 [float] - Einstein oscillator scaler
C1 [float] - Einstein oscillator shifter
C2 [float] - Einstein oscillator shifter
C3 [float] - Einstein oscillator shifter

Outputs
-----

"""
VMESH = RVMESH * V0
DEBYE = [eos.THETA(v, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4)
          for v in VMESH]

fig, ax = plt.subplots(1, 1)
ax.plot(RVMESH, DEBYE)

# Figure properties
fig.suptitle('Debye Temperature', fontsize=20, fontweight='bold')
ax.set_title('AP Model', fontsize=18)
ax.set_xlabel(r'$V/V_0$', fontsize=18)
ax.set_ylabel(r'$\theta \backslash, (K)$', fontsize=18)
ax.set_xlim([0., 10.])
# ax.set_ylim([-5, 150.])
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
if not os.path.exists(basePath + "/Plots"):
    os.mkdir(basePath + "/Plots")

```

```

plt.savefig(basePath + "/Plots/Debye_temperature.png", format='png', dpi=saveDPI)

def plot_gruneisen(basePath, RVMESH, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2, B3, B4):
    """Gruneisen parameter plot.

    Inputs
    -----
    RVMESH [array] - relative volume array
    V0 [float] - reference volume
    theta0 [float] - scaling parameter
    A1 [float] - Einstein oscillator multiplier
    A2 [float] - Einstein oscillator multiplier
    A3 [float] - Einstein oscillator multiplier
    B1 [float] - Einstein oscillator scaler
    B2 [float] - Einstein oscillator scaler
    B3 [float] - Einstein oscillator scaler
    C1 [float] - Einstein oscillator shifter
    C2 [float] - Einstein oscillator shifter
    C3 [float] - Einstein oscillator shifter

    Outputs
    -----

    """

    VMESH = RVMESH * V0
    GRUNEISEN = [eos.gruneisen(v, V0, theta0, A0, A1, A2, A3, A4, B0, B1, B2,
                               B3, B4) for v in VMESH]

    fig, ax = plt.subplots(1, 1)
    ax.plot(RVMESH, GRUNEISEN)

    # Figure properties
    fig.suptitle('Gruneisen Parameter', fontsize=20, fontweight='bold')
    ax.set_title('AP Model', fontsize=18)

```

```

ax.set_xlabel(r'$V/V_0$', fontsize=18)
ax.set_ylabel(r'$\Gamma$', fontsize=18)
ax.set_xlim([0., 10.])
# ax.set_ylim([-5, 150.])
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(10, 6)
if not os.path.exists(basePath + "/Plots"):
    os.mkdir(basePath + "/Plots")
plt.savefig(basePath + "/Plots/Gruneisen_parameter.png", format='png', dpi=saveDPI)

def plot_isotherms(basePath, RVMESH, TMESH, R0, PRES, ENRG):
    """Isotherm plots.

    Inputs
    -----
    RVMESH [array] - relative volume array
    TMESH [array] - temperature array
    PRES [array] - pressure matrix
    ENRG [array] - energy matrix

    Outputs
    -----

    """
    RHOMESH = R0 / RVMESH

    # ----- Plot isotherms (pressure) -----
    fig, ax = plt.subplots(1, 1)

    i = 0
    for T in TMESH:
        if (-1) ** i > 0:

```

```

    RMESH = RHOMESH

else:
    RMESH = np.flipud(RHOMESH)

RV = R0 / RMESH
ax.plot(RV, PRES[i, :])
i += 1

# Figure properties
fig.suptitle('Isotherms - Pressure', fontsize=20, fontweight='bold')
ax.set_title('AP Model', fontsize=18)
ax.set_xlabel(r'$V/V_0$', fontsize=18)
ax.set_ylabel(r'$P \backslash, (GPa)$', fontsize=18)
ax.set_xlim([0.4, 2.])
ax.set_ylim([-5, 50.])
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
if not os.path.exists(basePath + "/Plots"):
    os.mkdir(basePath + "/Plots")
plt.savefig(basePath + "/Plots/pressure_isotherms.png", format='png', dpi=saveDPI)

# ----- Plot isotherms (energy) -----
fig, ax = plt.subplots(1, 1)

i = 0
for T in TMESH:
    if (-1) ** i > 0:
        RMESH = RHOMESH
    else:
        RMESH = np.flipud(RHOMESH)

    RV = R0 / RMESH
    ax.plot(RV, ENRG[i, :])

```

```

        i += 1

    # Figure properties
    fig.suptitle('Isotherms - Energy', fontsize=20, fontweight='bold')
    ax.set_title('AP Model', fontsize=18)
    ax.set_xlabel(r'$V/V_0$', fontsize=18)
    ax.set_ylabel(r'$E \text{ \textbackslash, (GPa/g/cm}^3\text{)}$', fontsize=18)
    ax.set_xlim([0.4, 2.])
    ax.set_ylim([-5, 50.])
    ax.grid(True)
    plt.tick_params(axis='both', which='major', labelsize=15)
    figure = plt.gcf() # get current figure
    figure.set_size_inches(9, 6)
    plt.savefig(basePath + "/Plots/energy_isotherms.png", format='png', dpi=saveDPI)

```

A.5.5 EOS_Evaluation

Functions for comparing experimental data to calculated values and generating figures of merit.

```

# -*- coding: utf-8 -*-
"""
Created on Sun May 30 08:30:32 2021.

@author: jmaestas
*****

EOS Evaluation

This is a set of evaluation scripts to compare the generated SESAME
EOS with data.

*****
"""
import eos_functions as eos

import numpy as np
import matplotlib.pyplot as plt

```

```

from scipy.signal import savgol_filter

plt.rcParams.update({"figure.max_open_warning" : 0})

# -----
# DATA FITS
# -----

def isothermal_compression(V, V0, K0, K1):
    """Generate isothermal compression curve in pressure.

    Isothermal compression curve (at room temperature) modeled as a third
    order Birch-Murnaghan. Used for comparing to data.

    """
    # Parameters
    # K0 = 15.16 # GPa
    # K1 = 6.4 # --

    P = (1.5 * K0 * ((V/V0) ** (-7./3.) - (V/V0) ** (-5./3.)) * (1.0 + 0.75 *
        (K1 - 4.0) * ((V/V0) ** (-2./3.) - 1.0)))

    return P

def hugoniot(V, V0):
    """Generate Hugoniot curve in pressure.

    Hugoniot curve modeled with a cubic polynomial. Used for comparing to data.

    """
    mu = V0 / V - 1. # compression
    P = 16.08 * mu + 41.70 * mu ** 2. + 63.28 * mu ** 3.

```

```

return P

# -----
# COMPARISONS
# -----

def compare_compression(basePath, R0, K0, K1, plotTitle="AP Model"):
    """Compare isothermal compression at room temperature.

Inputs
    -----
    R0 [float] - reference density
    BCAT input (eos description and sesame file) must be in local directory

Outputs
    -----
    error [float] - RMS error

    """
    new_raw_data = np.array([
        [243.4313725, 47.43553962],
        [244.9411765, 43.4150596],
        [247.9607843, 40.38233679],
        [251.1176471, 38.46981297],
        [255.7843137, 36.16119133],
        [257.4313725, 34.44707174],
        [260.4509804, 31.41434894],
        [261.8235294, 29.43676713],
        [264.1568627, 27.52463094],
        [268.6862745, 24.49119747],
        [270.8823529, 22.97451303],
        [274.5882353, 21.45711794],
        [276.5098039, 20.46774558],
        [279.8039216, 18.4892593],
    ])

```

```

[283.0980392,16.44487515],
[285.0196078,15.52140065],
[289.2745098,13.93784927],
[292.5686275,12.15705656],
[296.8235294,10.7053009],
[303.4117647,9.647834092],
[309.4509804,7.733953548],
[317.9607843,6.675582259],
[322.0784314,5.948767645],
[331.4117647,4.824110863],
[333.745098,6.799948315],
[334.8431373,4.42710857],
[335.9411765,6.53532319],
[338.1372549,6.138902348],
[346.372549,5.080660271],
[348.0196078,4.552702135],
[353.2352941,4.02306425],
[356.8039216,3.494201635],
[359.8235294,2.833801725],
[361.745098,2.437510095],
[366.1372549,2.303646994],
[367.7843137,2.171076009],
[369.7058824,2.038375812],
[371.9019608,1.64195497],
[374.372549,1.772587783],
[379.0392157,0.979616888],
[385.627451,0.712924379],
[390.5686275,0.447007139],
])

new_data = new_raw_data / np.array([400.50, 1])
V0 = 1. / R0 # cc/g, initial specific volume

# ----- ISOTHERM at 300K, 0.5 < V/V0 < 1.0 -----
# Gather SESAME data using BCAT
ISOT, hdr = bcats.isotherm(basePath, 'matinp', 1, R0 / 1., R0 / 0.5, 50, 2, 300.,

```

```

0, 1, 1)

Pt_ses = ISOT[1:, 2] # model ys
Rt_ses = ISOT[1:, 0]

RVt = RO / Rt_ses # x points
Vt = 1 / Rt_ses
model = np.column_stack((RVt, Pt_ses))

allowablepoints = (Pt_ses < 50) & (RVt > new_data[0, 0]) & (RVt < new_data[-1, 0])
assert allowablepoints.shape == RVt.shape

filtered_model = model[allowablepoints]
# Generate data curve

Pt_data = np.interp(filtered_model[:, 0], new_data[:, 0], new_data[:, 1])
#TODO change error function to use filtered model
"""with np.printoptions(threshold=np.inf):
    print("Pt_ses shape = " + str(Pt_ses.shape))
    print(Pt_ses)
    print("Rt_ses shape = " + str(Rt_ses.shape))
    print(Rt_ses)
    print(Pt_data)
    print(((Pt_data - Pt_ses) / Pt_data) ** 2)"""
# Model error
#error = np.sqrt(np.sum(((Pt_data - Pt_ses) / Pt_data) ** 2) / len(Pt_data))
error = np.sqrt(np.sum( ((Pt_data - filtered_model[:, 1]) / Pt_data) ** 2) /
    ↪ Pt_data.shape[0])
# Plot comparison
fig, ax = plt.subplots(1, 1)
ax.scatter(new_data[:, 0], new_data[:, 1], c='r', label='Data')
ax.plot(filtered_model[:, 0], filtered_model[:, 1], '-b', label='Model')

# Figure properties
fig.suptitle('Isothermal Compression at 300K', fontsize=20,

```

```

        fontweight='bold')
ax.set_title(plotTitle, fontsize=18)
ax.set_xlabel(r'$V/V_0$', fontsize=18)
ax.set_ylabel(r'$P \ , \ (GPa)$', fontsize=18)
#ax.set_xlim([0.5, 1.])
#ax.set_ylim([0.0, 80.])
ax.grid(True)
ax.legend(loc='upper right', fontsize=18)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig(basePath + "/Plots/isotherm_comparison.png", format='png', dpi=200)

return error

def compare_hugoniot(basePath, R0, PREF, TREF, plotTitle="AP Model"):
    """Compare Hugoniot.

    Inputs
    -----
    R0 [float] - reference density
    PREF [float] - reference pressure
    TREF [float] - reference temperature
    BCAT input (eos description and sesame file) must be in local directory

    Outputs
    -----
    error [float] - RMS error

    """

    V0 = 1. / R0 # cc/g, initial specific volume

    # ----- HUGONIOT at TREF < T < 5000K -----

```

```

# Gather SESAME data using BCAT
HUG, hdr = bcat.hugoniot(basePath, 'matinp', 1, R0, PREF, TREF, TREF, 5000., 100)
try:
    Ph_ses = HUG[:, 2]
    Rh_ses = HUG[:, 0]
    RVh = R0 / Rh_ses
    Vh = 1 / Rh_ses

    # Generate data curve
    Ph_data = np.array([hugoniot(i, V0) for i in Vh])

    allowablepoints = Ph_data < 50
    aps = allowablepoints

    # Model error
    #TODO add cutoff values to the hugoniot model
    base_error = np.sqrt(np.sum((np.absolute((Ph_data[aps] - Ph_ses[aps]) /
↪ Ph_data[aps]) * 100) **2) / len(Ph_data[aps])))

    flux_error = (abs(np.absolute(sum((Ph_data - Ph_ses)))) -
↪ sum(np.absolute((Ph_data - Ph_ses)))) * .01) ** 1.5
    error = base_error# + flux_error

    # Plot comparison
    fig, ax = plt.subplots(1, 1)
    ax.plot(RVh[aps], Ph_data[aps], '-b', label='Data Fit')
    ax.scatter(RVh[aps], Ph_ses[aps], c='r', label='Model')

    # Figure properties
    fig.suptitle('Hugoniot', fontsize=20, fontweight='bold')
    ax.set_title(plotTitle, fontsize=18)
    ax.set_xlabel(r'$V/V_0$', fontsize=18)
    ax.set_ylabel(r'$P \backslash, (GPa)^\$', fontsize=18)
    #ax.set_xlim([0.5, 1.])
    #ax.set_ylim([0.0, 80.0])
    ax.grid(True)
    ax.legend(loc='upper right', fontsize=18)

```

```

plt.tick_params(axis='both', which='major', labels=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig(basePath + "/Plots/hugoniot_comparison.png", format='png', dpi=200)
with open(basePath + "/hugoniotComp.txt", "w+") as f,
↪ np.printoptions(threshold=np.inf):
    f.write(f"Total hugoniot Error = {error}\n")
    f.write(f"Base Error = {base_error}\n")
    f.write(f"Flux Error = {flux_error}\n")
    f.write(repr(Ph_ses[aps]) + "\n")
    f.write(repr(Rh_ses[aps]) + "\n")
    f.write(repr(Ph_data[aps]) + "\n")
    f.write(repr((Ph_data[aps] - Ph_ses[aps]) / Ph_data[aps]) + "\n")
    f.write(repr(Vh) + "\n")
except Exception:
    error = np.inf

return error

def thermal_comparison(basePath, Cv, c0, c1, c2, c3, PREF, R0, plotTitle="AP Model"):
    """Compare specific heat.

    Inputs
    -----
    Cv [float] - scaling parameter
    c0 [float] - multiplier
    c1 [float] - multiplier
    c2 [float] - multiplier
    c3 [float] - multiplier

    Outputs
    -----
    VO_error [float] - VO RMS error

```

```

"""
V0 = 1. / R0 # cc/g, initial specific volume

# V0 vs T data from Prask (J. Chem. Phys., 1988)
# Units are [K, cm3/g]
vt_data = np.array([
    [13, 0.49052676],
    [35, 0.490747378],
    [75, 0.492462962],
    [110, 0.495268429],
    [140, 0.498087453],
    [170, 0.500020791],
    [200, 0.502980196],
    [293, 0.512963926]
])

# Isothermal compressibility data point (ambient conditions) from
# Haussuhl (Z. Kristallographie, 1990)
# Units are [K, 1/GPa]
compressibility_data = np.array([298.0, 0.06220])

# Cp vs Temperature Data from NIST and Westrum (J. Chem. Phys. 50, 1969).
# Units are [K, J/g-K]
cp_data = np.array([
    [5, 0.002183006],
    [10, 0.018126429],
    [15, 0.050426372],
    [20, 0.096116368],
    [25, 0.147717931],
    [30, 0.199283882],
    [35, 0.249496583],
    [40, 0.298320421],
    [45, 0.345328056],
    [50, 0.389949697],
    [60, 0.471144703],

```

[70, 0.539875563],
[80, 0.595786159],
[90, 0.641013201],
[100, 0.677693401],
[110, 0.709031824],
[120, 0.736809063],
[130, 0.762449591],
[140, 0.787021764],
[150, 0.810525581],
[160, 0.832961043],
[170, 0.85432815],
[180, 0.874270783],
[190, 0.893857297],
[200, 0.912731575],
[210, 0.930893616],
[220, 0.949411775],
[230, 0.967929934],
[240, 0.986448093],
[250, 1.004610134],
[260, 1.022772174],
[270, 1.040221978],
[273.15, 1.045919873],
[280, 1.0580279],
[290, 1.075477704],
[298, 1.089463694],
[298.15, 1.090078561],
[300, 1.093283627],
[300, 1.092868269],
[350, 1.17946429],
[400, 1.265650401],
[500, 1.437581391],
[513, 1.531207177],
[600, 1.681859578],
[700, 1.852939424],
[800, 2.025721557],

```

        [900, 2.20105712],
        [1000, 2.376392684],
        [1100, 2.544919099],
        [1200, 2.703231792],
        [1300, 2.842819328],
        [1400, 2.958574845],
        [1500, 3.042838053]
    ])

# Dulong-Petit limit (derived from crystal chemistry values)
dp_limit = 0.00212304 # kJ/g*K

# ----- Coefficient of Thermal Expansion -----

# First, compute the isobar at ambient pressure
ISOB, hdr = bcat.isobar(basePath, 'matinp', 1, PREF, 0, 1600, 1000, R0)
V = ISOB[:, 6]
T = ISOB[:, 1]
DENS = ISOB[:, 0]

# Compute V0 error
v_model = np.interp(vt_data[:, 0], T, V)
V0_error = np.sqrt(np.sum(((vt_data[:, 1] - v_model) / vt_data[:, 1])**2) /
                    len(vt_data))

V0_invert = -40000 * min(0, v_model[-1] - v_model[0])
V0_error += V0_invert

# Plot isobar
fig, ax = plt.subplots(1, 1)
ax.scatter(vt_data[:, 0], vt_data[:, 1], c='r', label='Data')
ax.plot(T, V, '-k', label='Model')

# Figure properties
fig.suptitle('Isobar at Ambient Pressure', fontsize=20, fontweight='bold')
ax.set_title(plotTitle, fontsize=18)

```

```

ax.set_xlabel(r'$T \backslash, (K)$', fontsize=18)
ax.set_ylabel(r'$V_0 \backslash, (cm^3)$', fontsize=18)
# ax.set_xlim([0.5, 1.])
# ax.set_ylim([0.0, 80.0])
ax.grid(True)
ax.legend(loc='lower right', fontsize=18)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig(basePath + "/Plots/ambient_isobar.png", format='png', dpi=200)

# Check if all data was generated (BCAT will cut off data that doesn't
# exist)
if max(T) < max(cp_data[:, 0]) or min(T) > min(cp_data[:, 0]):
    raise ValueError('BAD SET')

# Compute derivative, dV/dT
dVdT = np.gradient(V, T)
dVdT_data = np.gradient(vt_data[:, 1], vt_data[:, 0])

# Smooth it
dVdT = savgol_filter(dVdT, 31, 5)

# Compute coefficient of thermal expansion as a function of temperature
alpha = dVdT / V
alpha_data = dVdT_data / vt_data[:, 1]

# print(alpha.shape)
# print(alpha_data.shape)
# print(T.shape)
# print(vt_data.shape)
total_error = 0
for i in range(len(alpha_data)):
    xData = vt_data[i, 0]
    xModel = min(range(len(T)), key=lambda j: abs(xData - T[j]))

```

```

total_error += ((alpha_data[i] - alpha[xModel]) / alpha_data[i]) ** 2
# find the point in T with the closest x position and the compare them

alpha_inverted = -100000 * min(0, alpha[-1] - alpha[0])
# print(alpha[-1] - alpha[0])
# print(alpha_inverted)
# print(type(alpha_inverted))
alpha_error = np.sqrt(total_error / len(alpha_data))

alpha_error += alpha_inverted
fig, ax = plt.subplots(1, 1)
ax.scatter(vt_data[:, 0], alpha_data, c='r', label='Data')
ax.plot(T, alpha, '-k', label='Model')

# Figure properties
fig.suptitle('Coefficient of Thermal Expansion', fontsize=20,
            fontweight='bold')
ax.set_title(plotTitle, fontsize=18)
ax.set_xlabel(r'$T \, , \, (K)$', fontsize=18)
ax.set_ylabel(r'$\alpha \, , \, (1/K)$', fontsize=18)
ax.set_xlim([0., 1500.])
# ax.set_ylim([0, 0.002])
ax.grid(True)
ax.legend(loc='lower right', fontsize=18)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(10, 6)
plt.savefig(basePath + "/Plots/thermal_expansion.png", format='png', dpi=200)

# ----- Isothermal Compressibility -----
beta = []

# Loop through each temperature
i = 0
for t in T:

```

```

# Set density limits
R1 = DENS[i] * 0.9
R2 = DENS[i] * 1.1

# Compute isotherm at T
ISOT, hdr = bcat.isotherm(basePath, 'matinp', 1, R1, R2, 100, 1, t, 0, 1, 1)
V = ISOT[:, 7]
P = ISOT[:, 2]

# Remove any points that have been cutoff
# ind_cutoff = next(i for i, x in enumerate(P) if x != 0.0)
# V = V[ind_cutoff:]
# P = P[ind_cutoff:]

# Compute derivative, dV/dP
dVdP = np.gradient(V, P)

# Compute isothermal compressibility at VO
compressibility = -dVdP / V

# Gather value at PREF
beta.append(np.interp(PREF, P, compressibility))

# Increment
i += 1

# Compute Bt error
Bt_model = np.interp(compressibility_data[0], T, beta)
Bt_error = np.sqrt(np.sum(((compressibility_data[1] - Bt_model) /
                           compressibility_data[1]) ** 2))

fig, ax = plt.subplots(1, 1)
ax.scatter(compressibility_data[0], compressibility_data[1], c='r',
           label='Data')

```

```

ax.plot(T, beta, '-k', label='Model')

# Figure properties
fig.suptitle('Isothermal Compressibility', fontsize=20,
            fontweight='bold')
ax.set_title(plotTitle, fontsize=18)
ax.set_xlabel(r'$T \, , (K)$', fontsize=18)
ax.set_ylabel(r'$\beta_T \, , (1/GPa)$', fontsize=18)
# ax.set_xlim([0., 1500.])
# ax.set_ylim([0, 0.5])
ax.grid(True)
ax.legend(loc='lower right', fontsize=18)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig(basePath + "/Plots/isothermal_compressibility.png", format='png',
            dpi=200)

# ----- Specific Heat -----
ALPHA = np.interp(cp_data[:, 0], T, alpha)
BETA = np.interp(cp_data[:, 0], T, beta)

CV_model = [eos.CV(t, Cv, c0, c1, c2, c3) for t in np.arange(0, 5000)]
CV_data = cp_data[:, 1] * 0.001 - (ALPHA ** 2 / BETA) * V0 * cp_data[:, 0]

# Compute Cv error
cv_model = np.interp(cp_data[:, 0], np.arange(0, 5000), CV_model)
# CV_error = np.sqrt(np.sum(((CV_data - cv_model) / CV_data) ** 2) / len(CV_data))
CV_error = 0
CV_point_density = len(cp_data) / (cp_data[-1, 0] - cp_data[0, 0])
CV_multipliers = [None] * cp_data.shape[0]
for i in range(cp_data.shape[0]):
    if i == 0:
        CV_multipliers[i] = CV_point_density / \
            (2 / (cp_data[i + 1, 0] - cp_data[i, 0]))

```

```

elif i == cp_data.shape[0] - 1:
    CV_multipliers[i] = CV_point_density / \
        (2 / (cp_data[i, 0] - cp_data[i - 1, 0]))

else:
    CV_multipliers[i] = CV_point_density / \
        (3 / (cp_data[i + 1, 0] - cp_data[i - 1, 0]))
if cp_data[i, 0] > 350:
    CV_multipliers[i] *= 10

global multipliers
multipliers = CV_multipliers
CV_error_base = (np.sum((
    np.absolute((CV_data - cv_model) / CV_data)) * CV_multipliers) ** 2) * .8

# adds error if the data makes large swings above and below the model
CV_error_fluxuation = (abs(np.absolute(sum((CV_data - cv_model))) -
    ↪ sum(np.absolute((CV_data - cv_model)))) * 5000) ** 1.5

# force the adjusted data to be below the dulong-petit limit
CV_error_data_dp = ((np.sum(np.maximum(CV_data - dp_limit, np.zeros(CV_data.shape)))
    ↪ * 1000) ** 2) * .5

# force the model to be below the dulong-petit limit
CV_error_model_dp = (((np.sum(np.maximum(np.array(CV_model) - dp_limit,
    ↪ np.zeros((len(CV_model),)) )) / len(CV_model)) * 1000000) ** 2) * .5

#CV_error_deriv
CV_error_derivs = np.sum((np.minimum(CV_data[1:] - CV_data[:-1],
    ↪ np.zeros((CV_data.shape[0] - 1,))) * 10000) ** 2) * 1000

fig, ax = plt.subplots(1, 1)
ax.scatter(cp_data[:, 0], CV_data, c='r', label='Adjusted Data')
ax.plot(np.arange(0, 5000), CV_model, '-k', label='Model')

```

```

ax.plot([0, 5000], [dp_limit, dp_limit], '--b', label='Dulong-Petit Limit')

# Figure properties
fig.suptitle('Specific Heat', fontsize=20,
            fontweight='bold')
ax.set_title(plotTitle, fontsize=18)
ax.set_xlabel(r'$T \, , (K)$', fontsize=18)
ax.set_ylabel(r'$C_v \, , (kJ / g K)$', fontsize=18)
ax.set_xlim([0., 5000.])
ax.set_ylim([0, 0.003])
ax.grid(True)
ax.legend(loc='lower right', fontsize=18)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(10, 6)
plt.savefig(basePath + "/Plots/specific_heat_comparison.png", format='png', dpi=200)

CV_error = sum((CV_error_base, CV_error_fluxuation, CV_error_data_dp,
    ↪ CV_error_model_dp, CV_error_derivs))
with open(basePath + "/thermalComp.txt", "w+") as f,
    ↪ np.printoptions(threshold=np.inf):
    f.write(f"Total CV Error = {CV_error}\n")
    f.write(f"Base CV Error = {CV_error_base}\n")
    f.write(f"Flux CV Error = {CV_error_fluxuation}\n")
    f.write(f>Data DP CV Error = {CV_error_data_dp}\n")
    f.write(f"Model DP CV Error = {CV_error_model_dp}\n")
    f.write(f"Deriv CP Error = {CV_error_derivs}\n")

return V0_error, Bt_error, CV_error, alpha_error

def check_isotherms(basePath, R0):
    """Plot the SESAME isotherms to check for problems.

Inputs

```

```

-----
RO [float] - reference density
PREF [float] - reference pressure
TREF [float] - reference temperature
BCAT input (eos description and sesame file) must be in local directory

Outputs
-----

"""
# Generate a series of isotherms to check if there are any problem areas.
temps = [0, 100, 300, 500, 1000, 2000, 4000]

# Initialize plot
fig, ax = plt.subplots(1, 1)

for T in temps:
    ISOT, hdr = bcat.isotherm(basePath, 'matinp', 1, RO / 2., RO / 0.5, 100, 2,
                              T, 0, 1, 1)

    Pt = ISOT[:, 2]
    R = ISOT[:, 0]
    RV = RO / R

    # Plot
    ax.plot(RV, Pt, label=f'{T}K')

# Figure properties
fig.suptitle('SESAME Isotherms', fontsize=20, fontweight='bold')
ax.set_title('AP Model', fontsize=18)
ax.set_xlabel(r'$V/V_0$', fontsize=18)
ax.set_ylabel(r'$P \backslash, (GPa)$', fontsize=18)
# ax.set_xlim([0.5, 2.])
# ax.set_ylim([-5.0, 50.0])
ax.grid(True)
ax.legend(loc='upper right', fontsize=16)

```

```

plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig(basePath + "/Plots/sesame_isotherms.png", format='png', dpi=200)

# TODO: add Gruneisen verification
# TODO: add shock temperature comparison

```

A.5.6 ProcessQueue

Classes and functions for parallel computing.

```

# -*- coding: utf-8 -*-
"""
Created on Tue Jul 20 14:55:27 2021

@author: mvandenboom
"""
import multiprocessing

STOP_VAL = "STOP"
def WriteFunc(q, fileloc):
    with open(fileloc, "a+") as f:
        while True:
            v = q.get()
            if v == STOP_VAL:
                break
            f.write(str(v) + "\n")
            f.flush()

# while True:
#     v = q.get()
#     if v == STOP_VAL:
#         break
#     print(v)

```

```

class ProcessQueueManager:
    """Context manager

    Parameters
    -----
    target : TYPE
        DESCRIPTION.
    STOP_VAL : TYPE
        DESCRIPTION.
    *args : TYPE
        DESCRIPTION.
    **kwargs : TYPE
        DESCRIPTION.

    Returns
    -----
    None.

    """
    class PrivateQueueInterface:
        def __init__(self, queue):
            self.__Q = queue

        def __deepcopy__(self, *args, **kwargs):
            return self
        def put(self, val):
            self.__Q.put(val)
    def __init__(self, target, STOP_VAL, *args, **kwargs):
        self.target = target
        self.STOP_VAL = STOP_VAL
        self.args = args
        self.kwargs=kwargs
    def __enter__(self):

```

```

self.manager = multiprocessing.Manager()
self.queue = self.manager.Queue()
self.p = multiprocessing.Process(target=self.target, name="testname",
    ↪ args=(self.queue, ) + self.args, kwargs=self.kwargs)
self.p.start()
self.QInterface = self.PrivateQueueInterface(self.queue)
return self.QInterface

def __exit__(self, exc_type, exc_val, traceback):
    self.queue.put(self.STOP_VAL)
    self.p.join(10)
    # self.queue.close()

def test(*args):
    print(args)

def _genArgs(inds, genNum, evalArgs):
    args = []
    for i, ind in enumerate(inds):
        print((ind, genNum, i) + evalArgs)
    return args

if __name__ == "__main__":
    # queue = multiprocessing.Queue()
    # p = multiprocessing.Process(target=WriteFunc, name="testname", args=(queue,
    ↪  "./testwritefile.txt"))

    for i in _genArgs(range(5), 0, ("a", "b")):
        print(i)
    # with ProcessQueueManager(WriteFunc, STOP_VAL, "./testwritefile.txt") as q:
    #     q.put("test val 1")
    #     q.put("test val 2")

```

A.5.7 bestingen

Function for finding the best parameter set within a given generation.

```

import os

import numpy as np

def pathToResults(path):
    try:
        with open(path, "r") as f:
            return float(f.readlines()[1])
    except FileNotFoundError:
        return np.inf

def bestInGen(path):
    results = []
    for i, item in enumerate(os.listdir(path)):
        #if int(item[3:]) > 329 and int(item[3:]) < 370:
        if "Ind" in item:
            results.append((item, pathToResults(path + "/" + item + "/results.txt")))
    return min(results, key=lambda x: x[1])

print(bestInGen("/home/matthew/Documents/SpecificHeatFit/Inds/Run20/Gen225"))

```

A.6 Calculating RVEs

Script for running ParticlePack to generate a sample set of RVEs.

```

# -*- coding: utf-8 -*-
"""
Created August 2018.

@author: espencer
=====

RVE GENERATOR   AKA   ALUMINUM SPHERES / GRAINS SHAPE FILE EDITOR

Created by Erik Spencer, August 2018 for Applied Research Associates, Inc., ECD
Updated by Joey Maestas, 4/12/21 to align with WPG style guidelines.

```

This script will take 2 user specified shape files from ParticlePack and include one within the other, as well as edit the initial shape ID for the grains files so that the initial ID number will pick-up where the Aluminum shapes left off.

This script also allows the user to create a Gaussian distribution of sizes across a given sample size.

Units: The shape files created by ParticlePack are simply coordinates for ALE3D to use to create the shapes. Therefore, the units should be the same as the ALE3D input deck that is used.

```
=====
"""
import os
import sys
import subprocess
import shutil
from random import gauss
from fileinput import FileInput
import fileinput

# ===== USER DEFINED INPUTS =====

# Operation System (windows or linux)
operating_system = 'windows'

# Sample size and file/folder names
sample_size = 1 # number of files to create
# particlepack .ppin file for aluminum spheres, do not include extension
aluminum_file = 'Spheres2'
# particlepack .ppin file for grains, do not include extension
grains_file = 'Grains'

main_folder_name = 'Void_5' # User selected name of folder for sample batch
```

```

sub_folder_name = 'Sample' # User selected subfolder name for each sample

# ammonium perchlorate parameters
ap_radius = 1
ap_radius_deviation = 0.10 * ap_radius
ap_vol_fraction = 0.2
ap_vol_fraction_deviation = 0.10 * ap_vol_fraction

# RDX Class I parameters
rdx_classI_radius = 0.5
rdx_classI_radius_deviation = 0.01
rdx_classI_vol_fraction = 0.1
rdx_classI_vol_fraction_deviation = 0.10 * rdx_classI_vol_fraction

# FEM RDX parameters
rdx_fem_radius = 0.04
rdx_fem_radius_deviation = 0.001
rdx_fem_vol_fraction = 0.3
rdx_fem_vol_fraction_deviation = 0.10 * rdx_fem_vol_fraction

# aluminum parameters
aluminum_radius = 0.2
aluminum_radius_deviation = 0.10 * aluminum_radius
aluminum_vol_fraction = 0.3
aluminum_vol_fraction_deviation = 0.10 * aluminum_vol_fraction

# enclosure size
box_size = 4.0

# Defect parameters
# enter binder defect volume as a percentage (e.g. 20.0 for 20% target void)
binder_defect_volume = 5.0

# ===== PROGRAM RUN =====

```

```

# Loop through sample folders
for i in range(1, sample_size + 1):

    # make aluminum radius and vol fraction random
    aluminum_rad = round(gauss(aluminum_radius, aluminum_radius_deviation), 8)
    aluminum_vol_frac = round(gauss(aluminum_vol_fraction,
                                    aluminum_vol_fraction_deviation), 8)

    # make ap radius and vol fraction random
    ap_rad = round(gauss(ap_radius, ap_radius_deviation), 8)
    ap_vol_frac = round(gauss(ap_vol_fraction, ap_vol_fraction_deviation), 8)

    # make rdx_classI radius and vol fraction random
    rdx_classI_rad = round(gauss(rdx_classI_radius,
                                 rdx_classI_radius_deviation), 8)
    rdx_classI_vol_frac = round(gauss(rdx_classI_vol_fraction,
                                      rdx_classI_vol_fraction_deviation), 8)

    # make rdx_fem radius and vol fraction random
    rdx_fem_rad = round(gauss(rdx_fem_radius,
                              rdx_fem_radius_deviation), 8)
    rdx_fem_vol_frac = round(gauss(rdx_fem_vol_fraction,
                                   rdx_fem_vol_fraction_deviation), 8)

    # set and create path for each subfolder
    path = f"{main_folder_name}/{sub_folder_name}-{i}"
    os.makedirs(path, exist_ok=True)

    # add file extension to file names in order to move correct items
    aluminum = f"{aluminum_file}.ppin"
    grains = f"{grains_file}.ppin"

    # copy aluminum and grains file to the new path
    shutil.copy(aluminum, path)
    shutil.copy(grains, path)

```

```

# set new aluminum and grains files for editing
new_aluminum_file = f"{path}/{aluminum}"
new_grains_file = f"{path}/{grains}"

# ===== ALUMINUM FILE =====
# change container size
for line in FileInput(new_aluminum_file, inplace=True):
    if " 0.0 0.0 0.0  4.0 4.0 4.0 # cube min, max" in line:
        sys.stdout.write(" 0.0 0.0 0.0  "
                           f"{box_size} {box_size} {box_size} "
                           "# cube min, max\n")
    else:
        sys.stdout.write(line)

# remove 0 0 radius line from aluminum file
for line in FileInput(new_aluminum_file, inplace=True):
    if '(enter zero when done)' in line:
        continue
    else:
        sys.stdout.write(line)

# add randomized aluminum radius and 0 0 end line back to file
for line in FileInput(new_aluminum_file, inplace=True):
    if '# radius, volume fraction' in line:
        sys.stdout.write(f" {aluminum_rad} {aluminum_vol_frac}  "
                           "# radius, volume fraction\n"
                           " 0.0 0.0  "
                           "# (enter zero when done)\n")
    else:
        sys.stdout.write(line)

# ===== GRAINS FILE =====
# change container size
for line in FileInput(new_grains_file, inplace=True):

```

```

if '0.0 0.0 0.0 4.0 4.0 4.0 # cube min, max' in line:
    sys.stdout.write(' 0.0 0.0 0.0  '
                     f'{box_size} {box_size} {box_size} '
                     '# cube min, max\n')
else:
    sys.stdout.write(line)

# change 0 0 radius line to 'New radii' as a place holder to edit
for line in FileInput(new_grains_file, inplace=True):
    if '(enter zero when done)' in line:
        sys.stdout.write('New radii\n')
    else:
        sys.stdout.write(line)

# remove all other lines that have radius and volume fraction
for line in FileInput(new_grains_file, inplace=True):
    if '# radius, volume fraction' in line:
        continue
    else:
        sys.stdout.write(line)

# add the randomized parameters for all 3 grains, plus the 0 0 end line
for line in FileInput(new_grains_file, inplace=True):
    if 'New radii' in line:
        sys.stdout.write(f" {ap_rad} {ap_vol_frac}  "
                         "# radius, volume fraction - AP\n"
                         f" {rdx_classI_rad} {rdx_classI_vol_frac}  "
                         "# radius, volume fraction - RDX Class I\n"
                         f" {rdx_fem_rad} {rdx_fem_vol_frac}  "
                         "# radius, volume fraction - RDX FEM\n"
                         " 0.0 0.0 # (enter zero when done)\n")
    else:
        sys.stdout.write(line)

# this will change the target binder void volume to user specified amount

```

```

with fileinput.input(files=new_grains_file, inplace=True) as f:
    for line in f:
        if fileinput.lineno() == 28:
            sys.stdout.write(f" {binder_defect_volume / 100.}      "
                             "# defect volume fraction of enclosure "
                             "volume\n")
        else:
            sys.stdout.write(line)

# ===== RUN PARTICLEPACK =====

# ----- WINDOWS VERSION -----
if operating_system == 'windows':
    # set cmd prompt command to run particlepack
    run_aluminum = f'cmd /c \"cd {main_folder_name}\\{sub_folder_name}\\'
                    f'{i}\\ && particlepack < {aluminum}\\'
runGrains = f'cmd /c \"cd {main_folder_name}\\{sub_folder_name}\\'
            f'{i}\\ && particlepack < {grains}\\'

    # run particlepack in a new window for aluminum .rin file
    p = subprocess.call(run_aluminum,
                        creationflags=subprocess.CREATE_NEW_CONSOLE)

    # run particlepack in a new window for grains .rin file
    p = subprocess.call(runGrains,
                        creationflags=subprocess.CREATE_NEW_CONSOLE)

# ----- LINUX VERSION -----
if operating_system == 'linux':
    run_aluminum = f"cd {main_folder_name}/{sub_folder_name}{i}" \
                    f" && module load ale3d && ParticlePack < {aluminum}"
    run_grains = f"cd {main_folder_name}/{sub_folder_name}{i}" \
                  f" && module load ale3d && ParticlePack < {grains}"

    p = subprocess.run(run_aluminum, shell=True)

```

```

p = subprocess.run(run_grains, shell=True)

# ParticlePack is run at this point to create all of the output files for each
# sample. This will allow the rest of the script to perform the operations on
# the files and determine where to start the shape ID number for the grains.
# =====

# set variables for shape and defect file names
aluminum_shape_file = f"{path}/{aluminum_file}S.sph"
aluminum_defect_file = f"{path}/{aluminum_file}.defect"
grains_shape_file = f"{path}/{grains_file}S.grn"
grains_bkg_defect = f"{path}/{grains_file}Bkg.defect"
grains_defect = f"{path}/{grains_file}.defect"

# open aluminum file to read lines
lines = open(aluminum_shape_file).readlines()

# if a defect file for aluminum exists
if os.path.exists(aluminum_defect_file):

    defect_lines = open(aluminum_defect_file).readlines()

    # check the length of the defect file to make sure there are actually
    # defect shape commands. If it is longer than 5 lines, get the last
    # shape ID number to increment grains from there
    if len(defect_lines) > 5:

        # get first defect ID
        first = int(defect_lines[3].split()[2])

        # get last defect ID
        start = defect_lines[-2].index('+') + 1
        end = defect_lines[-2].index(' ')
        last = int(defect_lines[-2][start:end])

```

```

# if the length of the defect file is less than 5, then there are no
# defect shape commands so get the last shape ID number from the
# aluminum shape file
else:

    # get initial id value
    first = int(lines[0].split()[2])

    # get integer value for last shape id
    start = lines[-1].index('+') + 1
    end = lines[-1].index('}')
    last = int(lines[-1][start:end])

# if no defect file for aluminum exists get shape ID numbers from the
# aluminum shape file
else:

    # get initial id value
    first = int(lines[0].split()[2])

    # get integer value for last shape id
    start = lines[-1].index('+') + 1
    end = lines[-1].index('}')
    last = int(lines[-1][start:end])

# add initial ID to value of last shape and set new_id +1 higher
new_id = first + last + 1

# change the InitID value of the grains to the new_id value and add to
# aluminum shape file
for line in FileInput(grains_shape_file, inplace=True):
    if 'defstr InitID' in line:
        sys.stdout.write(f"defstr InitID {new_id} #Initial shape ID\n"
                        f"include {aluminum_file}S.sph InitID={first}\n")
    else:

```

```

        sys.stdout.write(line)

    # change the InitID value of the grain background defects to new_id
    for line in FileInput(grains_bkg_defect, inplace=True):
        if 'defstr InitID' in line:
            sys.stdout.write(f"defstr InitID {new_id} #Initial shape ID\n")
        else:
            sys.stdout.write(line)

    # change the InitID value of grain defects to new_id value
    for line in FileInput(grains_defect, inplace=True):
        if 'defstr InitID' in line:
            sys.stdout.write(f"defstr InitID {new_id} #Initial shape ID\n")
        else:
            sys.stdout.write(line)

    i += 1

```

A.7 Generating RVEs

Script for automatically running GEN3D to create RVE geometries.

```

"""

Run gen3d to generate RVE shapes.

Created by J. Maestas on 9 November 2020.

This script runs gen3d to initialize an ALE3D mesh and shape descriptions.
The resulting VisIt file is post-processed using another script.

"""

import os
import shutil
from pathlib import Path

# User defined inputs

```

```

homedir = Path.cwd() # get current working directory
input_name = 'gen3d_input' # ALE3D file name (base)
num_proc = 48 # number of processors to use
directory_name = 'GEN3D' # directory where databases are located

# Find all void folders within input directory
indir = Path(directory_name)
void_dirs = [x for x in indir.iterdir() if x.is_dir()]

# Create an array to store all sample directories
sample_dirs = []

# Loop through void folders
for v in void_dirs:
    # Gather sample directories
    sample_dirs.extend([x for x in v.iterdir() if x.is_dir()])

# Loop through all sample directories and run gen3d
for s in sample_dirs:

    # Copy input deck into directory
    shutil.copy2(input_name, s)

    # Go into GEN3D/sample directory
    os.chdir(s)

    # Check to see if GEN3D was already run
    if os.path.isfile(f"{input_name}_0{num_proc}_00000"):
        break

    # Run gen3d
    cmd = f"module load ale3d; mpirun -n {num_proc} gen3d -in {input_name}.ain"
    os.system(cmd)

    # Go to home directory

```

```
os.chdir(homedir)
```

A.8 Converting RVE to STL

Script for converting RVE geometry into STL format.

```
"""
ALE3D VisIt STL Export.

Created by J. Maestas on 6 November 2020.

This script uses VisIt to export an ALE3D mesh/shape database into STL files.
One STL file is generated per material.

"""
import os
from pathlib import Path

# User defined inputs
homedir = Path.cwd() # get current working directory
db_name = 'gen3d_input_048_00000' # ALE3D database name
directory_name = 'GEN3D' # directory where databases are located
output_directory = 'STL' # directory where STL files will be places

# Define function to run visit
def run_visit(in_db_name, out_db_name):

    # open a file to write visit commands
    f = open("visit_run", "w")

    file_inputs = ''
# Define materials and their database numbers
mats = {"void":("2 mat2_2", "3 mat3_3"),
        "aluminum":("5 mat5_5",),
        "ap":("6 mat6_6",),
```

```

        "rdx": ("7 mat7_7", "8 mat8_8")}]

# Open Database
OpenDatabase('' + in_db_name + '')

# Create materials plot
AddPlot("FilledBoundary", "material")
DrawPlots()

# Get export attributes
e = ExportDBAttributes()
e.db_type = "STL"
e.dirname = '' + out_db_name + ''

# Loop through the different materials
for m in list(mats):

    # Turn off all materials
    TurnMaterialsOff()

    # Turn on select material numbers
    TurnMaterialsOn(mats[m])

    # Export database
    e.filename = m
    ExportDatabase(e)

# Clean up
DeleteAllPlots()
ClearAllWindows()
exit()

'''

f.write(file_inputs)
f.close()

```

```

    # Run visit

    cmd = "visit -nowin -cli -s visit_run"
    os.system(cmd)

# Create output directory (if does not already exist)
outdir = Path(output_directory)
if not outdir.is_dir():
    os.mkdir(output_directory)

# Find all void folders within input directory
indir = Path(directory_name)
void_dirs = [x for x in indir.iterdir() if x.is_dir()]

# Create an array to store all sample directories
sample_dirs = []

# Loop through void folders
for v in void_dirs:
    # Gather sample directories
    sample_dirs.extend([x for x in v.iterdir() if x.is_dir()])

# Loop through all sample directories and run VisIt
for s in sample_dirs:

    # Create a directory for output
    stl_dir = homedir / outdir / s.parts[1] / s.parts[2]
    stl_dir.mkdir(parents=True)

    # Go into GEN3D/sample directory
    os.chdir(s)

    # Run visit
    run_visit(db_name, str(stl_dir))

```

```
# Go to home directory
os.chdir(homedir)
```

A.9 Mesoscale Adiabatic Compression

Script for post-processing mesoscale compression simulations and generating an unreacted equation of state.

```
# -*- coding: utf-8 -*-
"""
```

Created on April 5, 2020.

@author: jmaestas

ADIABATIC COMPRESSION ANALYSIS

This script processes the CTH adiabat compression simulations to generate a reactant EOS to use with the HERMES model. It first goes through each sample and generates a preliminary model. Those models are then averaged across samples to give a more accurate result. The HERMES model uses a cubic polynomial for the adiabat and one for the Gruneisen parameter as well.

Written by Joey Maestas for AFOSR.

"""

```
import os
import pickle
import scipy
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
```

```
# User inputs
```

```
size = 0.4 # cm, initial size of sample RVE (does not include surroundings)
```

```

matnums = [1, 2, 3, 4, 5]  # material numbers in sample
voids = [0]  # list of void sets
samples = [1, 2]  # list of sample numbers
gruneisen_method = 1  # 1 = simple fit, 2 = complex fit

# ===== FUNCTIONS =====

def get_mass(input_file, matnums):
    """Read CTH octh file and get the total mass of the sample.

    Parameters
    -----
    input_file : STRING
        CTH octh file

    matnums : ARRAY of INTEGERS
        Array of material numbers in sample

    Returns
    -----
    mass : FLOAT
        Total mass of sample

    """
    # Read file
    file = open(input_file, "r")

    i = -1
    total_mass = 0

    for line in file:

        if "CONSERVATION ACCOUNTING" in line and "TIME= 0.00000E+00" in line:
            i = 0

```

```

        continue

    # skip the next line
    if i == 0:
        i += 1
        continue

    # Gather masses
    if i >= 0 and i <= max(matnums):
        mat = int(line.split()[0])
        mass = float(line.split()[1])

        if mat in matnums:
            total_mass = total_mass + mass

        i += 1
    file.close()

    return total_mass


def cth_read(input_file, mass):
    """Read CTH tracer history file, make plots, and save state data.

    Parameters
    -----
    input_file : STRING
        CTH hsct (tracer history) file.

    mass: FLOAT
        Total mass of sample (g)

    Returns
    -----
    None.

```

```

"""

# Read in data

data = pd.read_csv(input_file, skiprows=[0, 2], skipinitialspace=True)
numnodes = int(data.columns[-1].split('.')[0])

# Convert time units

TIME = data.TIME * 1e6 # s to us

# Set up arrays for collecting average values
P_DATA = [[], []]
T_DATA = [[], []]
RHO_DATA = [[], []]

# Loop through time
for i in range(len(TIME)):

    p_set = []
    t_set = []
    rho_set = []

    # Loop through each tracer
    for j in range(1, numnodes + 1):

        # Get variables and convert units if necessary
        p = 'P.' + str(j)
        t = 'T.' + str(j)
        rho = 'DENS.' + str(j)
        surr = 'VOLM+6.' + str(j) # volume fraction of surroundings

        P = data[p].iloc[i] / 1.e10 # dyne/cm2 to GPa
        T = data[t].iloc[i] * 11604.5 # eV to K
        RHO = data[rho].iloc[i]
        SURR = data[surr].iloc[i]

```

```

    # If a tracer contains a lot of surrounding material, disregard it,
    # else average the state variables
    if SURR < 0.1:
        p_set.append(P)
        t_set.append(T)
        rho_set.append(RHO)

    # Find means and standard deviations
    P_DATA[0].append(np.mean(p_set))
    P_DATA[1].append(np.std(p_set))
    T_DATA[0].append(np.mean(t_set))
    T_DATA[1].append(np.std(t_set))
    RHO_DATA[0].append(np.mean(rho_set))
    RHO_DATA[1].append(np.std(rho_set))

# Create EOS table
eos = {}
eos['P_DATA'] = P_DATA
eos['T_DATA'] = T_DATA
eos['RHO_DATA'] = RHO_DATA

# Output to pickle file
with open('eos.pickle', 'wb') as f:
    # Pickle the eos dictionary using the highest protocol available.
    pickle.dump(eos, f, pickle.HIGHEST_PROTOCOL)

# Make plots

# ----- PRESSURE vs TIME -----
fig, ax = plt.subplots(1, 1)
ax.plot(TIME, np.array(P_DATA[0]))
ax.fill_between(TIME, np.array(P_DATA[0]) - np.array(P_DATA[1]),
               np.array(P_DATA[0]) + np.array(P_DATA[1]), alpha=0.3)

# Figure properties
fig.suptitle('PRESSURE', fontsize=20, fontweight='bold')

```

```

ax.set_xlabel(r'$Time \, (\mu s)$', fontsize=18)
ax.set_ylabel(r'$Pressure \, (GPa)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("PRESSURE_TIME.png", format='png', dpi=200)

# ----- TEMPERATURE vs TIME -----
fig, ax = plt.subplots(1, 1)
ax.plot(TIME, np.array(T_DATA[0]))
ax.fill_between(TIME, np.array(T_DATA[0]) - np.array(T_DATA[1]),
               np.array(T_DATA[0]) + np.array(T_DATA[1]), alpha=0.3)

# Figure properties
fig.suptitle('TEMPERATURE', fontsize=20, fontweight='bold')
ax.set_xlabel(r'$Time \, (\mu s)$', fontsize=18)
ax.set_ylabel(r'$Temperature \, (K)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("TEMPERATURE_TIME.png", format='png', dpi=200)

# ----- DENSITY vs TIME -----
fig, ax = plt.subplots(1, 1)
ax.plot(TIME, np.array(RHO_DATA[0]))
ax.fill_between(TIME, np.array(RHO_DATA[0]) - np.array(RHO_DATA[1]),
               np.array(RHO_DATA[0]) + np.array(RHO_DATA[1]), alpha=0.3)

# Figure properties
fig.suptitle('DENSITY', fontsize=20, fontweight='bold')
ax.set_xlabel(r'$Time \, (\mu s)$', fontsize=18)
ax.set_ylabel(r'$Density \, (g/cm^3)$', fontsize=18)
ax.grid(True)

```

```

plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("DENSITY_TIME.png", format='png', dpi=200)

# ===== READ IN DATA =====

# Loop through each void set
for vd in voids:
    os.chdir(f"Void_{vd}")

    # Loop through each sample
    for s in samples:
        os.chdir(f"Sample{s}")

        # Find mass of sample
        mass = get_mass('octh', matnums)

        # Read in data
        if not os.path.isfile("eos.pickle"):
            cth_read('hscth', mass)

        os.chdir("..")
    os.chdir("..")

# ===== PROCESS DATA AND FIND FITS =====

# Average data across samples and then fit cubic polynomials to the averaged
# data.

# Loop through each void set
for vd in voids:
    os.chdir(f"Void_{vd}")

    # Create arrays to gather average data.

```

```

# Column 0 = means, Column 1 = standard deviations
P_DATA = [], []
T_DATA = [], []
RHO_DATA = [], []

# Create arrays to average across
PMEANS = []
PVARs = []
TMEANS = []
TVARS = []
RHOMEANS = []
RHOVARs = []

# Loop through samples
for s in samples:

    # Grab the EOS values
    with open(f"Sample{s}/eos.pickle", 'rb') as f:
        eos = pickle.load(f)

    # Gather eos data
    PMEANS.append(np.array(eos['P_DATA'][0]))
    PVARs.append(np.array(eos['P_DATA'][1])**2)
    TMEANS.append(np.array(eos['T_DATA'][0]))
    TVARS.append(np.array(eos['T_DATA'][1])**2)
    RHOMEANS.append(np.array(eos['RHO_DATA'][0]))
    RHOVARs.append(np.array(eos['RHO_DATA'][1])**2)

# Find mean and standard deviations
P_DATA[0] = np.mean(np.array(PMEANS), axis=0)
P_DATA[1] = np.sqrt(np.mean(np.array(PVARs), axis=0))
T_DATA[0] = np.mean(np.array(TMEANS), axis=0)
T_DATA[1] = np.sqrt(np.mean(np.array(TVARs), axis=0))
RHO_DATA[0] = np.mean(np.array(RHOMEANS), axis=0)
RHO_DATA[1] = np.sqrt(np.mean(np.array(RHOVARs), axis=0))

```

```

# Make plots

# ----- PRESSURE vs DENSITY -----
fig, ax = plt.subplots(1, 1)
ax.plot(RHO_DATA[0], P_DATA[0])
ax.fill_between(RHO_DATA[0], P_DATA[0] - P_DATA[1],
               P_DATA[0] + P_DATA[1], alpha=0.3)
ax.plot(RHO_DATA[0] - RHO_DATA[1], P_DATA[0])
ax.fill_between(RHO_DATA[0] - RHO_DATA[1], P_DATA[0] - P_DATA[1],
               P_DATA[0] + P_DATA[1], alpha=0.3)
ax.plot(RHO_DATA[0] + RHO_DATA[1], P_DATA[0])
ax.fill_between(RHO_DATA[0] + RHO_DATA[1], P_DATA[0] - P_DATA[1],
               P_DATA[0] + P_DATA[1], alpha=0.3)

# Figure properties
fig.suptitle('Pressure Average', fontsize=20, fontweight='bold')
ax.set_xlabel(r'$Density \, (g/cm^3)$', fontsize=18)
ax.set_ylabel(r'$Pressure \, (GPa)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("P-V.png", format='png', dpi=200)

# ----- TEMPERATURE vs DENSITY -----
fig, ax = plt.subplots(1, 1)
ax.plot(RHO_DATA[0], T_DATA[0])
ax.fill_between(RHO_DATA[0], T_DATA[0] - T_DATA[1],
               T_DATA[0] + T_DATA[1], alpha=0.3)
ax.plot(RHO_DATA[0] - RHO_DATA[1], T_DATA[0])
ax.fill_between(RHO_DATA[0] - RHO_DATA[1], T_DATA[0] - T_DATA[1],
               T_DATA[0] + T_DATA[1], alpha=0.3)
ax.plot(RHO_DATA[0] + RHO_DATA[1], T_DATA[0])
ax.fill_between(RHO_DATA[0] + RHO_DATA[1], T_DATA[0] - T_DATA[1],

```

```

T_DATA[0] + T_DATA[1], alpha=0.3)

# Figure properties
fig.suptitle('Temperature Average', fontsize=20, fontweight='bold')
ax.set_xlabel(r'$Density \, (g/cm^3)$', fontsize=18)
ax.set_ylabel(r'$Temperature \, (K)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)
figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("T_V.png", format='png', dpi=200)

# -----
# Fit cubic polynomial to average
# -----

P = P_DATA[0]
T = T_DATA[0]
MU = RHO_DATA[0] / RHO_DATA[0][0] - 1.

# Pressure fit
A = np.vstack([MU, MU**2, MU**3]).T
A1, A2, A3 = np.linalg.lstsq(A, P, rcond=None)[0]

# Plot fit
fig, ax = plt.subplots(1, 1)
ax.plot(MU, P)
ax.plot(MU, A1 * MU + A2 * MU**2 + A3 * MU**3)

# Figure properties
fig.suptitle('Adiabat Fitting', fontsize=20, fontweight='bold')
ax.set_xlabel(r'$\mu$', fontsize=18)
ax.set_ylabel(r'$P \, (GPa)$', fontsize=18)
ax.grid(True)
plt.tick_params(axis='both', which='major', labelsize=15)

```

```

figure = plt.gcf() # get current figure
figure.set_size_inches(8, 6)
plt.savefig("adiabat_fit.png", format='png', dpi=200)

# Gruneisen parameter fit
if gruneisen_method == 1:
    logT = np.log(T)
    z = np.polyfit(MU, logT, 1)
    fit_func = np.poly1d(z)
    C1 = fit_func[1]
    B0 = C1
    B1 = 2 * C1
    B2 = C1

    # Plot fit
    fig, ax = plt.subplots(1, 1)
    ax.plot(MU, logT, MU, fit_func(MU))

    # Figure properties
    fig.suptitle('Gruneisen Fitting', fontsize=20, fontweight='bold')
    ax.set_xlabel(r'$\mu$', fontsize=18)
    ax.set_ylabel(r'$\ln T$', fontsize=18)
    ax.grid(True)
    plt.tick_params(axis='both', which='major', labelsize=15)
    figure = plt.gcf() # get current figure
    figure.set_size_inches(8, 6)
    plt.savefig("Gruneisen_fit.png", format='png', dpi=200)

if gruneisen_method == 2:
    # Filter the temperature signal
    # T_filt = scipy.signal.savgol_filter(T, 101, 2)

    # z = np.polyfit(MU, T, 6)
    # filt_func = np.poly1d(z)
    # T_filt = filt_func(MU)

```

```

cs = scipy.interpolate.CubicSpline(MU[0:-1:80], T[0:-1:80])
T_filt = cs(MU)

# Take natural log of temperature
logT = np.log(T_filt)

# Find derivative
dlnTdmu = np.gradient(logT, MU)

# Find fit
B = np.vstack([1. / (MU + 1)**2, MU / (MU + 1)**2,
               MU**2 / (MU + 1)**2]).T
B0, B1, B2 = np.linalg.lstsq(B, dlnTdmu, rcond=None)[0]

# Write out fitting parameters to file
out_file = open("fit_parameters.txt", "w")
out_file.write("----- ADIABAT FITTING PARAMETERS -----\\n")
out_file.write(f"A1 = {A1}\\n")
out_file.write(f"A2 = {A2}\\n")
out_file.write(f"A3 = {A3}\\n")
out_file.write("\\n")
out_file.write("----- GRUNEISEN FITTING PARAMETERS -----\\n")
out_file.write(f"B0 = {B0}\\n")
out_file.write(f"B1 = {B1}\\n")
out_file.write(f"B2 = {B2}\\n")
out_file.close()

os.chdir("../")

```