USEABLE COARSE-GRAINED MODELS FOR SEMICONDUCTING POLYMERS AND THERMOSETS

by

Michael Montgomery Henry

A dissertation

submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy in Materials Science and Engineering

Boise State University

December 2020

BOISE STATE UNIVERSITY GRADUATE COLLEGE

DEFENSE COMMITTEE AND FINAL READING APPROVALS

of the dissertation submitted by

Michael Montgomery Henry

Dissertation Title: Useable Coarse-Grained Models for Semiconducting Polymers and Thermosets

Date of Final Oral Examination: 30 October 2020

The following individuals read and discussed the dissertation submitted by student Michael Montgomery Henry, and they evaluated the student's presentation and response to questions during the final oral examination. They found that the student passed the final oral examination.

Eric Jankowski, Ph.D.	Chair, Supervisory Committee
Elton Graugnard, Ph.D.	Member, Supervisory Committee
Lan Li, Ph.D.	Member, Supervisory Committee
Kevin Ausman, Ph.D.	Member, Supervisory Committee

The final reading approval of the dissertation was granted by Eric Jankowski, Ph.D., Chair of the Supervisory Committee. The dissertation was approved by the Graduate College.

Dedicated to Zoey

ACKNOWLEDGMENTS

I would like first like to acknowledge the traditional, ancestral, unceded territory of the Shoshone-Bannock tribe on which I live and work. Despite hundreds of years of colonial theft and violence, this is their native land, and it will always be their native land. I hope those reading this will take time to research what native lands they are currently occupying.

Newton said it best in a letter to Robert Hooke in 1675, “if I have seen further, it is by standing on the shoulders of giants”. I have had the privilege to have been surrounded by many giants. Everyone in my life has been a part of this PhD. I would like to thank Dr. Aaron Santos for introducing me to Metropolis Monte Carlo sampling and also introducing me to Dr. Eric Jankowski.

CME Lab, past and present, I love you all. We are a lab of helpers and it shows. I knew that when I needed help, someone always had my back. Dr. Stephen and Dr. Evan, thank you both for blazing the trail and being excellent examples of PhD graduates from the CME lab. Dr. Matty, I couldn’t have done this without you. You would always challenge me to do things the correct way, even when it was the hard way. All of the CME lab members have Eric Jankowski to thank for creating such a powerfully positive environment.

Eric, I cannot thank you enough. You helped me take the first step in graduate education by helping me apply to the material science program at Boise State. It has been an incredible experience to be at the beginning of what will be your scientific legacy. You ended everyone of our meetings with “what’s in your way and how can I help?” and I never doubted that you wanted the absolute best for me.

I would also like to thank all of the MSMSE administrative staff that have supported me. My (sometimes very frequent) coffee trips to the office provided a much needed break from my work and I was always met with friendly conversation. I would especially like to thank Jamie Hayward for booking all of the plane tickets, conference registrations, and Airbnbs that made my PhD possible. Chad Watson was one of the first people I met at Boise State and helped me connect with other graduate students and feel very welcome to Boise, thank you. I would also like to thank Jessica Economy for helping me navigate life, graduate college paperwork, and department politics. I would like to all of our funding sources, especially the NASA Idaho Space Grant Consortium for awarding me a fellowship, which is funded by NASA (NNX15Ai04H).

To my parents, David and Kelly, thank you for enduring my constant stream asking “why?”. You never stifled my curiosity but instead nurtured it, and now I am a scientist. I will be forever grateful for always letting me choose the path that I wanted to take and enabling my dreams. Thank you for supporting me in all of my academic endeavors, and letting your first born leave to go to college out of state.

To my wife, Zoey, you have been a pillar of constant support. Thank you for everything you have sacrificed, for leaving your friends and family, to support me in my journey to obtain a PhD, I am forever grateful. I couldn’t ask for a better partner. You believed in me when I didn’t believe in myself, thank you.

ABSTRACT

This work aims to inform the formulation and processing of polymer mixtures through the use of models that have minimally sufficient complexity. Models with minimal complexity are easier to develop, understand, explain, and extend, all of which underpin model validation, verification, and reproducibility.

We develop simplified models for two different material systems, semiconducting polymers and thermosets. With the relatively low cost of predicting morphologies enabled by these models, we investigate structure-property-processing relationships in record system sizes and combinatorial parameter spaces. The insight from these models lays the foundation for improving the efficiency of organic solar cells and air travel.

The morphology of the active layer of an organic solar cell determines its efficiency, but is also the most difficult aspect to control during manufacturing. Morphology can in principle be controlled through the thermodynamic self-assembly of active layer components. We develop models of two semiconducting polymers. We find our predictions are validated by morphological and charge transport measurements from experiments and we provide guidance for optimizing conditions for self-assembly.

Thermoset polymers present a unique modeling challenge because their properties are sensitive to processing kinetics that are at odds with thermodynamic modeling frameworks. The primary source of this difficulty is bridging time (1×10^{-12} s) and length scales (1×10^{-10} m) of reaction dynamics with the time (1×10^{2} s) and length scales (1×10^{-6} m) of morphology evolution. We implement a coarse-grained model of toughened thermosets where each amine, epoxy, and toughener mer is treated as a single simulation element. This simplification allows us to reach the time and length scales necessary to model the epoxy amine reaction and observe curing-driven morphology evolution. We simulate curing of $(100\,\mathrm{nm})^3$, million-particle volumes, which allows observation of experimentally-relevant volume evolution.

To practice behaviors necessary for computational research to be usable and reproduced by others, we make available all the models, initial configurations, submission scripts, analysis scripts, and simulation data associated with this work with an open source, permissive license. We describe software development practices and design choices that make this possible and discuss opportunities for improvement in future computational materials research.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

44DDS – 4,4’-diaminodiphenyl sulphone

AAMD – All Atom Molecular Dynamics

AA – All-Atom

ABS – poly(acrylonitrile-butadiene-styrene)

ANN – Artificial Neural Network

BDT-TPD – poly(benzodithiophene-thienopyrrolo-dione)

BHJ – Bulk Heterojunction

CA – 4-methylhexahydrophthalic anhydride

CI – Continuous Integration

CGMD – Coarse Grained Molecular Dynamics

CG – Coarse-Grained

CHARM – Chemistry at Harvard Macromolecular Mechanics

CH – Cahn-Hilliard

COM – Center-of-Mass

DGEBA – diglycidyl-ether of bisphenol A

DGEBF – diglycidyl-ether of bisphenol F

DPD – Dissipative Particle Dynamics

EP – 3,4-Epoxycyclohexylmethyl-3,4-epoxycyclohexanecarboxylate

FENE – Finitely Extensible Non-Linear Elastic

FJC – Freely Joint Chains

FO – First Order

FRC – Freely Rotating Chains

GAFF – Generalized Amber Force-Field

GIXS – Grazing Incidence X-ray Scattering

GPU – Graphical Processing Unit

HFM-c – Constrained Hyperbola Fitting Method

HFM – Hyperbola Fitting Method

HOMO – Highest Occupied Molecular Orbital

HOOMD-Blue – Highly Optimized Object-oriented Many-particle Dynamics: Blue Edition

HPC – High Performance Computing

IBI – Iterative Boltzmann Inversion

ITIC – 3,9-bis(2-methylene-(3-(1,1-dicyanomethylene)-indanone))- 5,5,11,11- tetrakis (4-hexylphenyl)-dithieno[2,3-d:2',3'-d']-s-indaceno [1,2-b:5,6-b'] dithiophene

KG – Kremer-Grest

KMC – Kinetic Monte Carlo

KNN – K-Nearest Neighbors

LB – Lorentz-Berthelot

LJ – Lennard-Jones

LUMO – Lowest Unoccupied Molecular Orbital

MC – Monte Carlo

MD – Molecular Dynamics

ML – Machine Learning

MSD – Mean-Squared Displacement

MSIBI – Multistate Iterative Boltzmann Inversion

MTHPA – Methyl Tetrahydrophthalic Anhydride

NVT – Canonical ensemble: constant number of particles, volume and temperature

OLS – Ordinary Lease Squares

OPLS – Optimized Potentials for Liquid Simulations

OPV – Organic Photovoltaic

P3HT – Poly-(3-hexylthiophene)

PAH – Polycyclic Aromatic Hydrocarbon

PDMS – Polydimethylsiloxane

PEEK – poly ether ketone

PES – Poly(oxy-1,4-phenylsulfonyl-1,4-phenyl)

PE – Polyethylene

PLFM – Power Law Fitting Method

PP – Polypropylene

PRM-a – Automatic Piecewise Regression Method

PRM – Piecewise Regression Method

PS – Polystyrene

Pe – Perylene

Pt – Perylothiophene

QCC – Quantum Chemical Calculations

RDF – Radial Distribution Function

RF – Random Forest

SAFO – Self-accelerated First Order

SVM – Support Vector Machine

TraPPE – Transferable Potential for Phase Equilibria

UA – United-Atom

VRH – Variable Range Hopping

WLF – William-Landel-Ferrel

ZINDO/S – Zerner's Intermediate Neglect of Differential Overlap

TRUE – Transferable, reproducible, understandable and extensible

LIST OF SYMBOLS

A	Frequency factor (n_B/N_B)
D	Diffusivity
E_a	Activation energy of bond formation
E_{coh}	Cohesive energy density
F	Force
F_i^C	Conservative force
F_i^D	Dissipative force
F_i^R	Random force
K	Kelvin
N	Number of simulation elements
N_B	Total bonds possible in the system
P	The barostat step point pressure
T	Temperature
TPS	Timesteps per second
T^*	The dimensionless temperature of the system
T_{high}^a	The highest annealed temperature to which the system is cooled for T_g measurement

T_{low}^{a}	The lowest annealed temperature to which the system is cooled for T_g measurement
T_{high}^{q}	The highest quenched temperature to which the system is cooled for T_g measurement
T_{low}^{q}	The lowest quenched temperature to which the system is cooled for T_g measurement
T_C	Calibration temperature for the DPD model
T_g	Glass transition temperature
T_g^{exp}	The glass transition temperature found experimentally
T_g^{sim}	The glass transition temperature found from simulation
U	Energy, the subscript signifies what kind of energy
X	Cure fraction of the system
$\Delta E_{i,j}$	Energy difference
ΔT_{rxn}	Change in temperature of the system per reaction
Υ	E factor, a scaling factor for the LJ energy parameter ϵ
α	Cure fraction of the system
α_{cut}	Maximum cure fraction of the simulation after which bonding is stopped
α_{gel}	Cure fraction at gelation
α_{high}	The highest cure fraction considered in the study
α_{low}	The lowest cure fraction considered in the study
β	Scaling factor for activation energy of secondary bond formation
χ	Flory Huggins interaction parameter

δ	Hildebrand solubility parameter
ϵ	The Lennard-Jones interaction energy parameter
γ	Dissipative frag coefficient
$\hbar$	Planck's reduce constant
λ	The DiBenedetto equation interaction parameter
μm	micro meters
ϕ	Packing fraction
$\boldsymbol{v}_i$	Velocity of particles i
$\psi\prime$	The order parameter normalized by the relative standard deviations of bond lengths
	Order parameter describing the number of "large" clusters
ρ	Mass density, normally in units of g/cm^3
ρ_n	Number density
σ_{LJ}	Van der Waals radius
τ	Relaxation time of phase separation
τ_B	Bond period in time steps
τ_s	MD time unit
θ	The angle defining three simulation elements
θ_0	The equilibrium bond angle
ε	Depth of the Lennard-Jones potential well
ε_s	Scaling factor for ε, represents inverse solvent strength

φ	The dihedral angle describing four simulation elements
ς	Random force amplitude
$^{\circ}\mathbf{C}$	Degree Celcius
a_{ij}	DPD repulsion parameter
cm	centi meter
dt	Timestep
e_{coh}	Cohesive energy
$g(r)$	Radial distribution function
k_1	Primary reaction rate
k_2	Secondary reaction rate
k_B	Boltzmann's constant
$k_{\textbf{harmonic}}$	Harmonic bond constant
k_θ	The equilibrium bond angle
k_{b,θ,d_n}	Spring constant for bonds, angles or dihedrals
$k_{i,j}$	Rate coefficient
n_B	Total bonds made per bond step
q	Wave vector
q_{max}	The first peak in the scattering pattern which describes the characteristic feature size in the system
r	Separation
r_0	Equilibrium harmonic bond distance

s	second
$s(q)$	Structure factor
t	Time
t_a	The anneal time at each cooling temperature in time steps
t_q	The quench time at each cooling temperature in time steps
t_a	Decorrelation time
t_{gel}	Time taken to reach gelation
t_r	Relaxation time
v	velocity
x_r	Random number over [0, 1)
CA	Aromatic Carbon
CT	Aliphatic Carbon
E	Total energy of the system
L	Distance unit
N	Total number of particles in the system
O	Oxygen
S	Sulfur
U	Potential energy of the system
V	System volume
kcal	kilocalories
nm	nano meter

ps	pico second

CHAPTER 1

INTRODUCTION

1.1 Motivation

Global climate change is an existential threat to our species survival[1]. It is likely that we will experience a 1.5C increase in global mean surface temperature between 2030 and 2052[2]. Even if global emissions reach net zero by 2040, we will experience this temperature increase. While we must accept irreversible damage to our planet has already occurred, we must also act and reduce greenhouse gas emissions to mitigate even more damage.

Transportation and electrical power generation account for 28% and 27% of total U.S. greenhouse gas emissions, respectively[3]. Approximately 63% of the power generated in U.S. is from burning fossil fuels. Only 11% of power in the U.S. is generated from renewable sources[4], and of those sources, 43% of that power is from burning biomass. We need to expand the share of the non-carbon emitting energy sources in the U.S. power generation portfolio if we are to reduce greenhouse gas emissions. Solar power is a source of renewable and clean energy, which accounts for 1.8% of electricity produced in the United States. Current limitations to wide spread silicon based solar power include the weight of the panels[5], which limits installation options, and their expense, which is driven by manufacturing costs. These limitations are a materials problem, monocrystalline silicon is expensive to refine, purify, and is a heavy metalloid. We can overcome the limitations associated with traditional silicon based solar cells by using different materials.

Instead of silicon, organic based materials can be used to generate power. Organic photovoltaic (OPV) solar cells can overcome the limitations associated with silicon based solar. This is a result of how OPV devices are manufactured. OPVs use solution-based manufacturing processes, which are generally performed at lower temperatures than inorganic material purification, at atmospheric pressure, and often without specialist machinery[6,7]. Solution-based manufacturing enables easy, inexpensive batch-production of the photoactive materials, by utilizing large-scale commercial roll-to-roll manufacturing[6,7]. OPVs are lightweight, flexible, and can be incorporated into common building materials such as concrete and asphalt tiles[5]. These benefits will help to facilitate the adoption of OPV devices once the efficiency of these devices is improved.

It is estimated that once OPV devices have a power conversion efficiency (PCE) of 15%, it will be commercially viable to use OPVs for on-grid power generation and have energy payback time of a day[8–12]. There are OPV devices that have been created in a laboratory,[13,14] but these are "hero" experiments that when mass manufactured will have a PCE $\sim$5% of devices made in a lab[15]. The largest barrier to improving device efficiency is improving synthetic control of the active layer morphology[5]. By improving our understanding of how the active layer self-assembles, we can 1) improve the efficiency of OPVs and 2) ensure the self-assembly process is robust to reduce the PCE decrease that arises from mass manufacturing.

Understanding how the active layer self-assembles requires atomistic resolution, the exploration of different processing conditions, and the exploration of different OPV chemistries. Computer simulations provide atomistic insight and can explore the vast combination of different processing conditions and chemistries.

We need a diverse greenhouse gas reduction strategy to fight climate change. In addition to reducing greenhouse gas emissions from power generation, we need to reduce greenhouse gas emissions from other industrial sectors. Commercial air travel accounted for 6.9% of the United States total greenhouse gas production[3]. One way to improve the efficiency of air travel and reduce emissions is to replace metal parts on aircraft with composite material[16]. Technological advancements to improve efficiency when paired with with regulatory (taxes) efforts can reduce resource consumption[17]. Difficulty in reliably manufacturing composite materials for aircraft, which is the primary driver for their high cost, is one of the limiting factors in their widespread adoption. Improving our understanding of how to control the nanostructure of the thermoset matrix during curing will improve manufacturing reliability and decrease time and energy costs[18–20]. The material properties and morphology of the thermoset matrix change as the thermoset cures, further compounding the difficulty in understanding how to control the nanostructure. Computer simulations can help to elucidate the relationship between morphology, properties, and processing.

Beyond improving the efficiency of OPV devices and aircraft, we must also improve the efficiency of scientific research. This will have a force multiplying effect, the easier it is to train scientists and build computational research capacity, the quicker we can discover solutions to mitigate the effects of climate change. We can reduce the time it takes to train scientists to perform computational research by intentionally considering pedagogy related to on-boarding new researchers[21]. Creating simple models that are easy to extend to other systems reduces duplicated effort. Coarse-grained TRUE (transferable, reusable, usable and extensible) models are self-consistent, they are computationally and teaching efficient.

Understanding polymer self-assembly in material systems is important since the morphology that forms when polymers self-assemble determine the properties of the material system. Our current understanding of how polymers self-assemble is limited by our ability to observe self-assembly *in situ* and sufficiently explore the different processing conditions that effect self-assembly. This dissertation will use molecular dynamics to improve our understanding of polymer self-assembly. We will develop and use simplified models and generalized methodologies to study thermoset and OPV material systems at experimentally relevant length and time scales. We will characterize the morphologies that form through self-assembly to develop processing-structure relationships. By establishing the relationship between processing and morphology, we can engineer better materials to combat global climate change.

1.2 Outline

This dissertation is structured as follows: In Chapter 2 we review molecular dynamics, coarse graining, thermoset physics, and current scientific software best practices. In Chapter 3, we present our model of BDT-TPD where we looked at two different coarse-grain models comparing their structural predictions as well as their computational efficiency. In Chapter 4, we overcome some of the limitations of our first generation model (Ref. [22]) and demonstrate our models validity with experimental data and demonstrate our model's sensitivity to cure path. In Chapter 5 I outline my contributions to other papers and software. Lastly, in Chapter 6 we conclude and provide direction for figure work.

REFERENCES

[1] Chris D. Thomas, Alison Cameron, Rhys E. Green, Michel Bakkenes, Linda J. Beaumont, Yvonne C. Collingham, Barend F. N. Erasmus, Marinez Ferreira de Siqueira, Alan Grainger, Lee Hannah, Lesley Hughes, Brian Huntley, Albert S. van Jaarsveld, Guy F. Midgley, Lera Miles, Miguel A. Ortega-Huerta, A. Townsend Peterson, Oliver L. Phillips, and Stephen E. Williams. Extinction risk from climate change. *Nature*, 427(6970):145–148, jan 2004.

[2] Masson-Delmotte, V., P. Zhai, H.-O. Pörtner, D. Roberts, J. Skea, P.R. Shukla, A. Pirani, W. Moufouma-Okia, C. Péan, R. Pidcock, S. Connors, J.B.R. Matthews, Y. Chen, X. Zhou, M.I. Gomis, E. Lonnoy, T. Maycock, M. Tignor and T. Waterfield (eds.). 2018: Summary for Policymakers. Technical report, World Meteorological Organization, Geneva, Switzerland, 2018.

[3] EPA. Inventory of U.S. greenhouse gas emissions and sinks: 1990-2009. *Federal Register*, 76(36):10026, 2011.

[4] U.S. Energy Information Administration. *September Monthly Energy Review*, volume 0035. 2013.

[5] Letian Dou, Jingbi You, Ziruo Hong, Zheng Xu, Gang Li, Robert a Street, and Yang Yang. 25th Anniversary Article: A Decade of Organic/Polymeric Photovoltaic Research. *Advanced Materials*, 25(46):6642–6671, dec 2013.

[6] Ana Claudia Arias, J Devin MacKenzie, Iain McCulloch, Jonathan Rivnay, and Alberto Salleo. Materials and Applications for Large Area Electronics: Solution-Based Approaches. *Chemical Reviews*, 110(1):3–24, jan 2010.

[7] Frederik C Krebs, Thomas Tromholt, and Mikkel Jørgensen. Upscaling of Polymer Solar Cell Fabrication Using Full Roll-to-Roll Processing. *Nanoscale*, 2(6):873, 2010.

[8] Katherine a. Mazzio and Christine K Luscombe. The future of organic photovoltaics. *Chemical Society Reviews*, 44(1):78–90, sep 2015.

[9] Christoph J. Brabec, Srinivas Gowrisanker, Jonathan J M Halls, Darin Laird, Shijun Jia, and Shawn P Williams. Polymer-Fullerene Bulk-Heterojunction Solar Cells. *Advanced Materials*, 22(34):3839–3856, sep 2010.

[10] Sean E Shaheen, David S Ginley, and Ghassan E Jabbour. Organic-Based Photovoltaics: Toward Low-Cost Power Generation. *MRS Bulletin*, 30(01):10–19, jan 2005.

[11] Nieves Espinosa, Markus Hösel, Dechan Angmo, and Frederik C. Krebs. Solar Cells with One-Day Energy Payback for the Factories of the Future. *Energy & Environmental Science*, 5(1):5117, 2012.

[12] Shamsiah Ali Oettinger. Heliatek announces world record for organic cell, 2013.

[13] Jun Yuan, Yunqiang Zhang, Liuyang Zhou, Guichuan Zhang, Hin-Lap Yip, Tsz-Ki Lau, Xinhui Lu, Can Zhu, Hongjian Peng, Paul A. Johnson, Mario Leclerc, Yong Cao, Jacek Ulanski, Yongfang Li, and Yingping Zou. Single-Junction Organic Solar Cell with over 15% Efficiency Using Fused-Ring Acceptor with Electron-Deficient Core. *Joule*, pages 1–12, jan 2019.

[14] Lingxian Meng, Yamin Zhang, Xiangjian Wan, Chenxi Li, Xin Zhang, Yanbo Wang, Xin Ke, Zuo Xiao, Liming Ding, Ruoxi Xia, Hin-Lap Yip, Yong Cao, and Yongsheng Chen. Organic and solution-processed tandem solar cells with 17.3% efficiency. *Science*, 2612:eaat2612, 2018.

[15] Jon E. Carlé, Martin Helgesen, Ole Hagemann, Markus Hösel, Ilona M. Heckler, Eva Bundgaard, Suren A. Gevorgyan, Roar R. Søndergaard, Mikkel Jørgensen, Rafael García-Valverde, Samir Chaouki-Almagro, José A. Villarejo, and Frederik C. Krebs. Overcoming the Scaling Lag for Polymer Solar Cells. *Joule*, 1(2):274–289, 2017.

[16] Royal Aeronautical Society: Greener by Design Science and Technology Sub-Group. Air Travel – Greener by Design Mitigating the environmental impact of aviation: Opportunities and priorities. *The Aeronautical Journal*, 109(1099):361–416, sep 2005.

[17] Mathis Wackernagel and William E. Rees. Perceptual and structural barriers to investing in natural capital: Economics from an ecological footprint perspective. *Ecological Economics*, 20(1):3–24, jan 1997.

[18] J. Zhang, Y. C. Xu, and P. Huang. Effect of cure cycle on curing process and hardness for epoxy resin. *Express Polymer Letters*, 3(9):534–541, 2009.

[19] J W Sinclair. Effects of Cure Temperature on Epoxy Resin Properties. *The Journal of Adhesion*, 38(3-4):219–234, jul 1992.

[20] Fabrice Lapique and Keith Redford. Curing effects on viscosity and mechanical properties of a commercial epoxy resin adhesive. *International Journal of Adhesion and Adhesives*, 22(4):337–346, jan 2002.

[21] Eric Jankowski, Neale Ellyson, Jenny W Fothergill, Michael M Henry, Mitchell H Leibowitz, Evan D Miller, Mone't Alberts, Samantha Chesser, Jaime D Guevara, Chris D. Jones, Mia Klopfenstein, Kendra K Noneman, Rachel Singleton, Ramon A Uriarte-Mendoza, Stephen Thomas, Carla E Estridge, and Matthew L Jones. Perspective on Coarse-Graining , Cognitive Load , and Materials Simulation. *Computational Materials Science*, 169(109129):109129, jan 2020.

[22] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, may 2018.

CHAPTER 2

BACKGROUND

First we will discuss techniques and concepts that are used throughout this work (molecular dynamics in Section 2.1 and coarse graining in Section 2.2). Then we discuss relevant thermoset (Section 2.3) physics. We then end with background information on current scientific software engineering best practices (Section 2.4).

2.1 Molecular Dynamics

Molecular Dynamics (MD) simulations are in many ways like experiments performed in an experimental laboratory[1]. First, some material is prepared for study, then some property of the material is measured. When we perform a MD simulation, we first configure our initial configuration, then use Newton's equation of motion to update the particles locations until we reach equilibration, then we perform our measurement.

The first MD study was performed in 1957 by Alder and Wainbright (Ref. [2]) to calculate the phase transition for hard spheres. The largest system studied by Alder and Wainbright had 108 spheres and these simulations were performed on an IBM-704 which could perform 12,000 floating point additions per second[3]. The first MD study performed on a real material was performed by A. Rahman (Ref. [4]) in 1964. He simulated 864 argon atoms on a CDC 3600 computer that cost 1.2 million dollars and had 1.536 megabytes of memory and could perform seven hundred thousand instructions per second at a CPU frequency of 714 kHz. In this work the largest systems we examine have 4 million particles and the GPUs that we used in our work are capable of 3.5×10^{12} floating point operations a second, which is 100 million times faster than the IBM-704 used in the first MD simulation.

Molecular dynamics allows us to study molecular motion. This motion, while discretized, represents *a* physical trajectory that a molecule would experience in a physical experiment. This is important for our self-assembly studies because we are interested in both the thermodynamically stable structure and the pathway to that structure. First we will start with an example simulating argon using a Lennard-Jones pair potential (Equation 2.1).

$$V_{\mathrm{LJ}}(r) = 4\varepsilon \left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6} \right] \tag{2.1}$$

Where σ represents the size of the particle and ε is the depth of the potential well (see Figure 2.1. Since we are simulating only a single particle species, we can choose σ and ε to be one, and then scale by the parameters for argon when performing analysis later. Before moving onto a description of the steps in a molecular dynamic simulation, we have a simulation detail to consider, how do we treat our simulation volume? To reduce finite size effects, we treat our simulation volume as periodic, which means that particles that would venture past the edge of the simulation volume are wrapped back into the volume on the other side. The basic molecular dynamic steps are

1. Calculate potential energy between every simulation element

2. Calculate forces between every simulation element using the relation $F = -\nabla U$

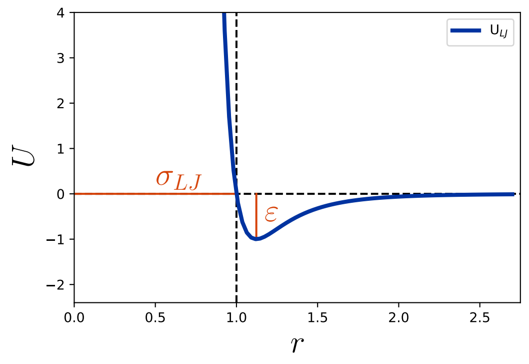


Figure 2.1 Plot of Lennard-Jones (Equation 2.1) potential energy (U) as a function of distance, r with $\sigma = 1$ and $\varepsilon = 1$.

3. Use Newton's second law ($a = \frac{F}{m}$) to calculate the acceleration for each element

4. Use acceleration to calculate velocity for each element

5. Displace particles r with $r = v\delta t$ and advance simulation time $t = t + \delta t$

This process is repeated as long as needed, typically until the simulation reaches thermodynamic equilibrium and enough statistically independent samples are collected. This basic outline of molecular dynamics is sufficient to understand what molecular dynamics *is* and a more thorough description of the current state-of-the art in molecular dynamics techniques is outside the scope of this brief background section. See Appendix F "Saving CPU Time" in Ref. [1] (Frenkel and Smit) for discussion on how modern molecular dynamics software accelerates computation. Our argon example was convenient since we had only a single potential to evaluate. When simulating more complex molecules, additional terms are needed to represent the interatomic forces. These are divided into pairs, triplets, and quadruplets of bonded atoms or bonds, angles, and dihedrals (see Figure 2.2). In molecular mechanics, it is

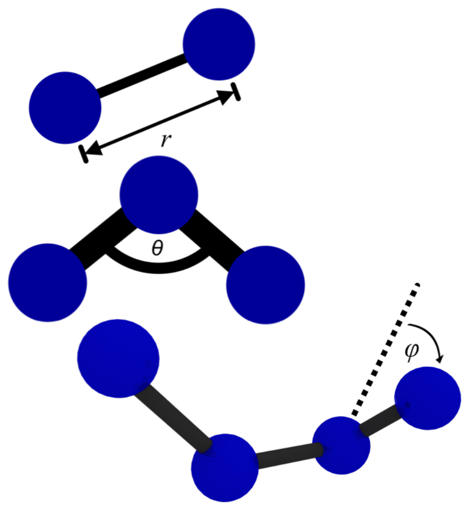


Figure 2.2 Illustration of bond distance r, bond angle θ, and dihedral angle ϕ.

common to use a harmonic oscillator to represent bonds with the form:

$$U_{\text{bond}}(r) = \frac{1}{2}k(r - r_0)^2, \tag{2.2}$$

where r is the distance between a pair of bonded atoms, k is the spring constant, and r_0 is the equilibrium bond length[5]. Angles are also often harmonic and in the form:

$$U_{\text{angle}}(\theta) = \frac{1}{2}k(\theta - \theta_0)^2, \tag{2.3}$$

where θ is the angle between the triplet of atoms, k the spring potential constant, and θ_0 the equilibrium angle[5]. Dihedral functional forms tend to vary more in literature than bond and angle terms. Two different dihedral forms are used in this work. In Chapter 3, dihedrals are in the "OPLS style" and have the form:

$$V(r) = \frac{1}{2}k_1\left(1+\cos\left(\phi\right)\right)+\frac{1}{2}k_2\left(1-\cos\left(2\phi\right)\right)+\frac{1}{2}k_3\left(1+\cos\left(3\phi\right)\right)+\frac{1}{2}k_4\left(1-\cos\left(4\phi\right)\right), \tag{2.4}$$

where ϕ is the angle between both sides of the dihedral and k_1, k_2, k_3, k_4 are the force coefficients. In Ref. [6], dihedrals are in the form of a multi-harmonic series[7]:

$$U_{\text{dihedral}}(\varphi) = \sum_{n=0}^{4} k_n \cos^n \varphi, \tag{2.5}$$

where ϕ is the angle between both sides of the dihedral and k_n are the force coefficients. In principle, any function can be used to describe the inter- and intra- atomic forces, but smooth, continuous, and differentiable functional forms help to ensure numerical stability.

2.2 Coarse Graining

In order to simulate longer time and length scales we have a few options. We could buy faster hardware, improve our algorithms, or combine some simulation elements together. By combining simulation elements together into a single simulation element, we reduce the number of elements that we need to keep track of, which results in faster simulations. This method is called “coarse-graining”. One method of coarse-graining is to create a particle at the center of mass of the simulation elements being combined and removing the elements that are now represented by a single particle. For a more thorough description of coarse-grain modeling, see Chapter 17 of [1], and review articles by Ingolfsson et al.[8] and Saunders and Voth[9]. With coarse-graning, we are sacrificing atomistic positional fidelity for computational efficiency.

In this work, we use two different combination strategies. In Chapter 3 and Ref. [6], we combine carbon atoms and all bonded hydrogens into a single "united-atom" (UA) site[10,11]. UA models are effective when the position and explicit electrostatic treatment of hydrogen does significantly affect the properties or process being modeled. Even more coarse-grained models are possible (see Figure 2.3). With each

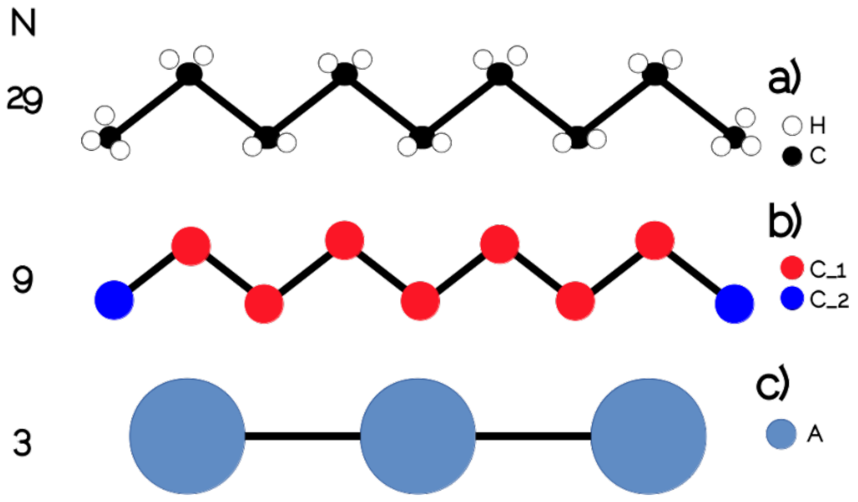


Figure 2.3 Illustration of successively coarser coarse-graining schemes. Scheme a) is an "all-atom" or "fine grain" representation of an alkane chain, where each atomic species is represented by a simulation element. Scheme b) is an "united-atom" model, where hydrogen is implicitly modeled by modifying the forcefield parameters of their constituent atoms. Scheme c) is a coarse-grain model, where each simulation element corresponds to multiple atoms. With each successive scheme, the number of simulation elements required to represent the alkane chain are reduced.

successively more coarse model in Figure 2.3, we trade-off explicit treatment of the degrees of freedom present in our molecule for an approximation of the underlying fine-grain elements position. Backmapping techniques[12] can be used to "recover" the probable location of the fine-grain elements if need to calculate some property that is unable to be accurately calculated with a coarse-grain model. In Ref. [13] and Chapter 4, we use a scheme where entire molecules are coarse-grain into a single simulation element. Coarse-grained models are necessary to study experimentally relevant systems of thermosets because the length scale associated with thermoset microstructure is in the nanometer to micrometer regime, which is inaccessible with all atom molecular dynamics (AAMD). One of the largest AAMD simulations with crosslinking had 230,000 atoms and a box edge length of 13.6 nm[14]. This simulation did not have a non-reacting toughener component and coarse grain models will be required to study cure induced phase separation of toughening agents in thermosets.

2.3 Thermoset

Polymers are macromolecules formed by repeat units, called monomers, that are covalently bonded. When polymers are heated above their glass transition temperature, they flow like a liquid. Below their glass transition temperature, they can either form a semi-crystalline solid or an amorphous glass (which has some short-range order but no long range order). Polymers are a category of material that has many sub-categories, in this section we will focus on thermosetting polymers (for a more general overview of polymers, see Chapter 14 and 15 of Ref [15]).

Thermosetting polymers form 3-dimensional networks when their monomers chemically react when heated[16]. This chemical reaction is irreversible and forms covalent bonds, which is the difference between thermosets and thermoplastics (which with temperature or solvents can be reshaped). Thermosets used for aerospace applications start with low-molecular weight monomers (frequently only dozens of atoms), which when reacted form dense, stiff, strong, and highly connected networks. These thermosets are lightweight and are commonly used as the matrix for fiber composites.

As the thermosets react and form bonds, their molecular weight increases which affects their material properties. This makes them difficult to process as glass transition temperature and viscosity vary as a function of degree of cure, α. In addition to the fact that material properties change as the thermoset cures, the reaction is exothermic[17], which results in autocatalytic[18] behavior. This makes it difficult to precisely control the temperature, which is necessary for both controlling the rate of the reaction, and also ensuring the prescribed cure cycle from the manufacture is followed. Controlling the temperature of the reaction is more difficult with more reacting material, preventing large scale parts from being reliably formed.

One material property of interest for thermosetting polymers is the gel point. In 1941 Flory[19] demonstrated, with a few assumptions, that the point of gelation is solely a function of the number of functional groups in the reacting species. This theory was further refined by Stockmayer in 1944[20] and is know as the Flory-Stockmayer theory of gelation. One central assumption is that steric affects are negligible, which causes the theory to underpredict the degree of cure at gelation[21]. We can measure the gel point in our simulations by observing when the molecular mass of the largest and second-largest network diverge, as seen in Figure 2.4.

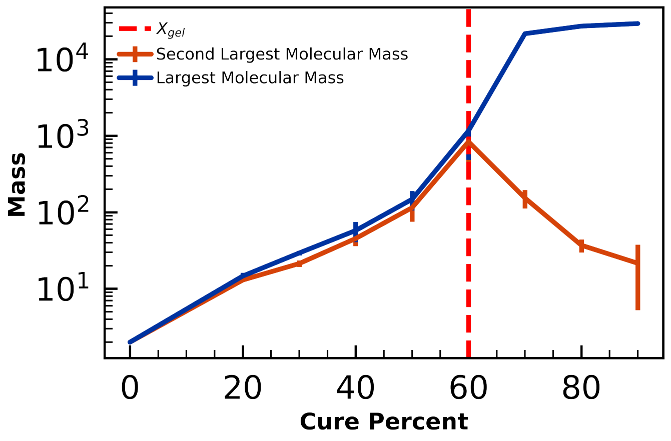


Figure 2.4 Example gel-point calculation. When the largest (blue) and second-largest (orange) molecular mass diverge, a thermoset is considered to have "gelled", here calculated at $\alpha = 60\%$.

The gel point is an important material property as the viscosity of the thermoset increases rapidly past the point of gelation, which can cause significant issues when processing a thermoset. Depending on the glass transition temperature at a given degree of cure, gelation can have the effect of locking in a phase separated toughener nanostructure, which depending on the properties of interested, may or may not be desirable. Above the glass transition temperature, the thermoset morphology may still evolve, but once the thermoset is below the glass transition temperature, vitrification occurs and thermoset becomes a glassy state. Being able to predict properties like point of gelation and the glass transition temperature as a function of degree of cure is key to improving the reliability of composite manufacture. For more background information on the molecular modeling of thermosets, see the excellent review article by Li et al.[16].

2.4 Current Software Engineering Best Practices

Reproducible results are one of the major tenets of the scientific method. However, there is a reproducibly crisis affecting many different scientific fields[22]. It is embarrassing that computational sciences have reproducibility issues when it should be "easy" to replicate simulation results since we (in principle) are able to control the environment where we conduct our experiments (*in silico*) precisely. While we do have full control over our software environment, we often fail to fully document our software environment.

Using open source code is essential to produce reproducible results. For more than a decade, the results of a simulation of super cooled water were disputed and the dispute was solved within two weeks of the code being made public[23]. Once the code was public, people were able to find an issue that was an implementation detail that was left ambiguous when the algorithm was described in the paper. Hence, it is not enough to publish the pseudo code of an algorithm, the source code of the implementation used to generate the results must also be provided. While releasing the source code does not magically fix bugs, having many eyes looking at the code can identify issues with the code. Providing input files and analysis scripts does not only enable reviewers to spot potential issues with the work, but also enables future work to easily build off of previous work.

It is frustrating to find a paper that does an impressive analysis or utilizes some novel initialization strategy, but because code and input files are not shared, in order to use the new method, one would have to re-implement everything from a terse algorithm description, which slows down scientific progress and creates duplicate work. Providing the raw data generated from the simulation also helps to improve reproducibility as then others can run the same analysis on the data. Here we have a unique opportunity. It would not be possible for an experimental group to share some new novel material sample with the entire scientific community. There is a finite amount of material, it may be too dangerous to ship or expensive to share with more than a few research groups. With simulations, we can share our outputs and raw data with the entire world.

It is difficult to faithfully reproduce a software environment. While a paper may mention the software used in the research, without knowing the version of the software, it may be impossible to reproduce as the authors may be relying on the behavior of some bug in an old version of code. Even with the software version, things like complier version, compile time options, dependency versions, and even operating system can have an effect. A python script used to calculate NMR shifts yields different results depending on the operating system due to assumptions made on how files were sorted when listed[24]. Software "Containers" can help solve these issues. Containers are similar to a virtual machines but have less overhead as they do not vitalize hardware. With a container, the software environment is defined in a declarative way and allows people to reproduce the same container on a different machine. Reproducible software environments are key for reproducible simulations and analysis. Containers can also reduce cognitive load when working with a heterogeneous computing environment as the same software stack can be replicated to multiple clusters.

Using public version control repositories for software development has benefits beyond just tracking changes to code. Version control enables an unambiguous way to "point" to a version of code used in research. Version control also tracks who makes what change which makes it easier to ensure that all code authors receive credit for their work. With services like Zenodo, one can obtain a DOI number for a code repository. There are two benefits to this practice. One, the code repository is easy to cite, making it easier to get "credit" for scientific software development. Two, the DOI links to a specific version of the code, meaning that future researchers know exactly what version of the code was used in the research. This is important for research reproducibility as some software bug or implementation detail can affect results.

Continuous integration (CI) is another practice for writing scientific software and could prevent some of the bugs previously discussed[24]. CI is a development practice where code is frequently committed to a shared repository and tested automatically in an isolated environment. By running tests in an isolated environment (instead of locally on the developers workstation) bugs related to different software environments are easily detected. Most developers use a single operating system to develop code, but with CI, tests can be conducted on a matrix of different operating systems and software versions. CI also enables software metrics to be tracked, such as test coverage, helping to identify parts of the code base that could benefit from additional testing. See these papers[25–27] for more discussion regarding current best practices, and Ref. [28].

REFERENCES

[1] Daan Frenkel and Berend Smit. *Understanding Molecular Simulations: From Algorithms to Applications.* Elsevier, 2 edition, 2008.

[2] B.J. Alder and T.E. Wainbright. Phase Transition for a Hard Sphere System. *Journal of Chemical Physics*, 27(1957):1208, 1957.

[3] IBM. 704 Data Processing System.

[4] A. Rahman. Correlations in the Motion of Atoms in Liquid Argon. *Physical Review A*, 136:A405–A411, 1964.

[5] Scott J Weiner, Peter a Kollman, David a Case, U Chandra Singh, Caterina Ghio, Giuliano Alagona, Salvatore Profeta, and Paul Weinerl. A New Force Field for Molecular Mechanical Simulation of Nucleic Acids and Proteins. *Journal of the American Chemical Society*, 106(17):765–784, 1984.

[6] Evan D Miller, Matthew Lewis Jones, Michael M Henry, Paul Chery, Kyle Miller, and Eric Jankowski. Optimization and Validation of Efficient Models for Predicting Polythiophene Self-Assembly. *Polymers*, 10(12):1305, nov 2018.

[7] Ram S Bhatta, Yeneneh Y Yimer, David S Perry, and Mesfin Tsige. Improved Force Field for Molecular Modeling of Poly(3-hexylthiophene). *The Journal of Physical Chemistry B*, 117(34):10035–10045, aug 2013.

[8] Helgi I. Ingólfsson, Cesar a. Lopez, Jaakko J Uusitalo, Djurre H de Jong, Srinivasa M Gopal, Xavier Periole, and Siewert J Marrink. The power of coarse graining in biomolecular simulations. *Wiley Interdisciplinary Reviews: Computational Molecular Science*, 4(3):225–248, may 2014.

[9] Marissa G. Saunders and Gregory A. Voth. Coarse-Graining Methods for Computational Biology. *Annual Review of Biophysics*, 42(1):73–93, 2013.

[10] William L. Jorgensen and Julian. Tirado-Rives. The OPLS [Optimized Potentials for Liquid Simulations] Potential Functions for Proteins, Energy Minimizations for Crystals of Cyclic Peptides and Crambin. *Journal of the American Chemical Society*, 110(6):1657–1666, mar 1988.

[11] Marcus G Martin and J Ilja Siepmann. Transferable Potentials for Phase Equilibria. 1. United-Atom Description of n-Alkanes. *Journal of Physical Chemistry B*, 102(14):2569–2577, 1998.

[12] Matthew Lewis Jones and Eric Jankowski. Computationally connecting organic photovoltaic performance to atomistic arrangements and bulk morphology. *Molecular Simulation*, 43(10-11):1–18, mar 2017.

[13] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, may 2018.

[14] Yasuyuki Shudo, Atsushi Izumi, Katsumi Hagita, Toshio Nakao, and Mitsuhiro Shibayama. Large-scale molecular dynamics simulation of crosslinked phenolic resins using pseudo-reaction model. *Polymer (United Kingdom)*, 103:261–276, 2016.

[15] William D. Callister and David G. Rethwisch. *Materials Science and Engineering: An Introduction.* John Wiley and Sons, 9 edition, 2013.

[16] Chunyu Li and Alejandro Strachan. Molecular scale simulations on thermoset polymers: A review. *Journal of Polymer Science Part B: Polymer Physics*, 53(2):103–122, jan 2015.

[17] J. Zhang, Y. C. Xu, and P. Huang. Effect of cure cycle on curing process and hardness for epoxy resin. *Express Polymer Letters*, 3(9):534–541, 2009.

[18] J. K. Gillham and G. Wisanrakkit. The glass transition temperature (Tg) as an index of chemical conversion for a High-Tg amine/epoxy system: Chemical and diffusion-controlled reaction kinetics.pdf. *Journal of Applied Polymer Science*, Volume 41(11-12):2885–2929, 1990.

[19] Paul J. Flory. Molecular Size Distribution in Three Dimensional Polymers. I. Gelation 1. *Journal of the American Chemical Society*, 63(11):3083–3090, nov 1941.

[20] Walter H. Stockmayer. Theory of Molecular Size Distribution and Gel Formation in Branched Polymers II. General Cross Linking. *The Journal of Chemical Physics*, 12(4):125–131, apr 1944.

[21] Dietrich Stauffer, Antonio Coniglio, and Mireille Adam. Gelation and critical phenomena. In *Polymer Networks*, pages 103–158. Springer Berlin Heidelberg, Berlin, Heidelberg.

[22] Monya Baker and Dan Penny. Is there a reproducibility crisis? *Nature*, 533(7604):452–454, 2016.

[23] Ashley G. Smart. The war over supercooled water. *Physics Today*, aug 2018.

[24] Jayanti Bhandari Neupane, Ram P. Neupane, Yuheng Luo, Wesley Y. Yoshida, Rui Sun, and Philip G. Williams. Characterization of Leptazolines A–D, Polar Oxazolines from the Cyanobacterium Leptolyngbya sp., Reveals a Glitch with the "Willoughby–Hoye" Scripts for Calculating NMR Chemical Shifts. *Organic Letters*, 21(20):8449–8453, oct 2019.

[25] Tom Crick, Benjamin A Hall, and Samin Ishtiaq. Reproducibility in Research: Systems, Infrastructure, Culture. *arXiv*, pages 1–12, mar 2015.

[26] Greg Wilson, D a Aruliah, C Titus Brown, Neil P Chue Hong, Matt Davis, Richard T Guy, Steven H D Haddock, Kathryn D Huff, Ian M Mitchell, Mark D Plumbley, Ben Waugh, Ethan P White, and Paul Wilson. Best practices for scientific computing. *PLoS biology*, 12(1):e1001745, jan 2014.

[27] Daniel S. Katz, Kyle E. Niemeyer, Sandra Gesing, Lorraine Hwang, Wolfgang Bangerth, Simon Hettrick, Ray Idaszak, Jean Salac, Neil Chue Hong, Santiago Núñez Corrales, Alice Allen, R. Stuart Geiger, Jonah Miller, Emily Chen, Anshu Dubey, and Patricia Lago. Report on the Fourth Workshop on Sustainable Software for Science: Practice and Experiences (WSSSPE4). *arXiv*, 1(1):0–3, may 2017.

[28] Eric Jankowski, Neale Ellyson, Jenny W Fothergill, Michael M Henry, Mitchell H Leibowitz, Evan D Miller, Mone't Alberts, Samantha Chesser, Jaime D Guevara, Chris D. Jones, Mia Klopfenstein, Kendra K Noneman, Rachel Singleton, Ramon A Uriarte-Mendoza, Stephen Thomas, Carla E Estridge, and Matthew L

Jones. Perspective on Coarse-Graining , Cognitive Load , and Materials Simulation. *Computational Materials Science*, 169(109129):109129, jan 2020.

CHAPTER 3

SIMPLIFIED MODELS FOR ACCELERATED STRUCTURAL PREDICTION OF CONJUGATED SEMICONDUCTING POLYMERS[1]

3.1 Introduction

Organic semiconducting polymers are promising components of next-generation electronics, as they can be used to create lightweight, flexible, and inexpensive devices such as organic thin film transistors (OTFTs), light emitting diodes (OLEDs), and photovoltaics (OPVs)[1]. These benefits arise from inexpensive, scalable, solution-based manufacturing processes, generally performed at lower temperatures than inorganic material purification, at atmospheric pressure, and often without specialist machinery[2,3]. Additionally, synthetic chemists can functionalize components of these molecules, chemically tuning the energetics to obtain enhanced electronic performance[4]. Organic semiconductors are of particular interest in the photovoltaics community, where opportunities exist for scalable manufacturing of inexpensive solar technologies[5]. Understanding how these molecules can be organized into structures

[1]This chapter has been published in *J. Phys. Chem. C* and is referenced as "Henry, M. M., Jones, M. L., Oosterhout, S. D., Braunecker, W. A., Kemper, T. W., Larsen, R. E., ... Jankowski, E. (2017). Simplified Models for Accelerated Structural Prediction of Conjugated Semiconducting Polymers. *The Journal of Physical Chemistry C*, 121(47), 26528–26538. https://doi.org/10.1021/acs.jpcc.7b09701"

that optimize conversion of light into electricity is a significant current challenge in the field[6]. The nanostructure of OPV active layers is critically important to the charge-carrier mobility – a property that strongly affects the resulting device performance – and depends on thermodynamic and kinetic factors that govern the self-assembly of its constituent molecules[7,8]. In particular, the spacing between polymer backbones, the sizes of these ordered domains, and their interconnectivity – all morphological characteristics – have a significant impact on OPV performance[9,10]. In order to rationally design organic electronics, we require improved understanding of how to select components optimized to assemble into the desired nanostructures.

Current understanding of how OPV active layer morphologies depend on their components and processing has been developed through both wet lab experimentation and computer simulation[11–15]. In the laboratory, active layer films are made by mixing organic semiconductors with a compatible solvent that is later evaporated through spin coating, drop casting, printing, or other deposition techniques onto a substrate[16,17]. The nanostructure of these films can be probed using grazing-incidence X-ray scattering (GIXS), which reveals ordering of the film components[18]. In computer simulations, molecular dynamics (MD) or Monte Carlo (MC) methods are used to sample equilibrium ensembles of configurations of the same molecules, and GIXS analysis can be extracted by transforming the atomic positions that result from these models[15]. The focus of this work is to enhance the complementarity of these techniques by improving the predictive capabilities of computer models.

The challenge in the lab is that determining the relationships between active layer morphology and device performance is hindered by the difficulty in reliably controlling the morphology of organic thin films. One barrier in predicting the structures that self-assemble in organic semiconducting polymers is the impact small changes in materials choice or processing have on molecular packing[19]. For instance, the molecular weight[20] and regioregularities[21,22] of poly(3-hexylthiophene) (P3HT) influences the conformation of backbone chains, structural feature sizes, and charge mobility of the deposited films. Furthermore, modifications to the casting solvent[23], annealing conditions[21,24], or the addition of organic dyes[25] can result in an active layer with drastically different structures and subsequent device performance. Another barrier to understanding active layer morphology is the difficulty of characterizing structure in relatively disordered layers with low scattering contrast[18]. Scanning microscopy techniques can provide useful insight into the surface features, but often miss the important structure of the bulk. Other techniques can provide additional information, but at the cost of destructively sampling the films[26].

The challenge with molecular simulations lies in faithfully representing experiments with models that are computationally tractable[15]. Increasing the number of simulation elements either with more detail or larger systems increases the number of calculations to perform and therefore the computation time required to obtain the final result of the simulation. The time needed to perform a simulation –its computational cost– depends on: (1) the cost of advancing from one configuration to the next, which scales worse than linearly with the number of simulation elements (atoms), and (2) the fact that systems with more simulation elements require more configurations to be sampled before relaxing to a steady-state.. In the case of organic semiconducting polymers, we can understand the scale of the problem by considering that coherence lengths of the order 3 to 30 nm are commonly observed in GIXS experiments. It stands to reason that, if a 30 nm coherence length is to be observed in a simulation, length scales of at least 30 nm must be represented. This is accomplished in MD simulations with a periodic unit cell, and filling it with carbons, hydrogens, oxygens, sulfurs, and nitrogens in representative ratios at realistic densities of the order 1 g/cm^3, which corresponds to between 0.7 million and 1.5 million atoms. Million-atom simulations are now routine on supercomputers, especially those with graphics processing units (GPUs), but are typically limited to accessing time scales of hundreds of nanoseconds at most[27]. Performing a million-atom simulation of organic oligomers on supercomputers that accesses the hundreds or thousands of nanoseconds required to equilibrate can take several months of computation time for a single state point. In the polymer limit of chain length (over hundreds of monomers), neither experiments nor simulations can be run long enough to equilibrate, with observed morphologies representing entangled, kinetically arrested configurations. Rather than month-long computation times accessing equilibrium, the ability to perform hundreds of simulations per month that predict thermodynamically-driven yet possibly-arrested assembly, is required to determine phase diagrams or to find optimal conditions for assembling target structures.

To lower the computational cost of a model (assuming recent GPU supercomputers are being used), approximations must be invoked to lower the number of simulation elements represented. This is the basis behind coarse-graining strategies, where typically spherical simulation elements are used to represent collections of neighboring atoms and their associated bonds. Such models have been used to overcome the atomistic simulation time/length barrier, providing insight into photoactive polymers, lipid bilayers, and macromolecular structures[13,14,28,29]. Another modeling approximation is to ignore the fast modes of fluctuations about relatively rigid bonded constraints. Recent work studying the self-assembly of aromatic molecules perylene and perylothiophene showed that using rigid bodies to model conjugated systems improved sampling by a factor of two or better, without affecting the observed phase behavior or self-assembled morphology[30]. The improvement was attributed to the combination of more timesteps accessible per CPU second, and shortened structural decorrelation times when a rigid body approximation was employed. Because conjugated systems are prevalent in organic electronics, coarse-grained models with rigid body approximations have potential to enable screening studies of organic electronic ingredients for those that robustly self-assemble into desirable nanostructures.

The success of computational screening studies depends on the accuracy and transferability of coarse-grained models. In this work we measure whether and to what degree rigid body approximations, combined with a united atom model, may be used to enhance the sampling efficiency of statepoints for a system that has characterized experimentally. We use MD simulations to investigate the self-assembly of poly(benzodithiophene-thienopyrrolodione) (BDT-TPD) pentamers, which are expected to be representative of donor-acceptor alternating copolymers. This class of molecules has demonstrated promise as a component of high power conversion efficiency photovoltaic devices, due to desirable optical and electrical properties[31,32]. We examine the evolution of the simulated morphologies after slow or fast cooling, and for cases when the conjugated moeities are described by rigid bodies or more expensive flexible bond, angle, and dihedral terms. Two different cooling schedules are used: an instantaneous "quench" from infinite temperature, and a more gradual "annealed" cooling schedule, where the temperature is decremented over a longer period of time. For all models, we evaluate the degree to which the simulations reproduce experimentally characterized film structures with simulated GIXS data, and quantify the computational efficiencies of each model. We aim to find the most efficient set of approximations to include in the modeling of an amorphous polymer that still faithfully predicts the morphologies observed in experiments. We determine that simulating BDT-TPD using rigid bodies results in a significant reduction to computational cost and reproduces experimentally-observed structure. We find that “annealing” results in structures that better match experiments than “quenching”, as expected, but with minor measurable differences in precise scattering peak locations. We also observe small measurable differences between the predictions of rigid and flexible models, though both accurately predict primary experimental features, and the disorder-to-order (glass) transition temperature.

In the Methods section we define the rigid and flexible models used in this investigation as well as the analysis techniques used to quantify structure and determine when a simulation has sampled sufficiently many configurations. In the Results section we detail the key simulation results and explain comparisons against experimental studies. We conclude with a discussion of the applicability of the present work and suggestions for future directions. The major modeling assumptions and investigation aims can be summarized in this question: To what degree do oligomers of BDT-TPD in implicit solvent and with implicit charges, modeled with OPLS-UA parameters unoptimized for BDT-TPD, faithfully represent self-assembly observed in experiments? We establish a minimum requirement for performing sufficiently accurate screening of thermodynamic state points, in order that promising structures can be quickly identified and submitted to further, more detailed, analysis using more expensive and rigorous methods. The enhanced sampling of time scales and length scales presented here enables the best to-date prediction of complex OPV oligomer self-assembly.

3.2 Methods

The chemical structure of BDT-TPD (3.1a) is described by a united atom model (3.1b), where hydrogen atoms are abstracted away into a "united" site that represents the carbon atom and all of its bonded hydrogens[33,34]. This technique reduces the number of simulation elements, improving computational efficiency while still maintaining good agreement with both atomistic models and experiment for a variety of systems. This technique has been shown to provide an improvement in computational efficiency, while still maintaining good agreement with both atomistic models and experiment for a variety of systems[30,35–39]. We employ the OPLS-UA (Optimized Potential for Liquid Simulations-United Atom) forcefield to model the non-bonded and bonded interactions[33]. The OPLS-UA forcefield includes constraints (bonds, angles, and dihedrals between pairs, triplets and quadruplets of simulation elements respectively) to model intramolecular structure, and describes non-bonded pair-wise interactions with Lennard-Jones potentials[33]. Reference units taken from the OPLS-UA force field are: length $\sigma = 3.55$ Å, energy $\varepsilon = 1.74 \times 10^{-21}$ J, and mass $\mathcal{M} = 32.06$ amu. The values of σ and ε correspond to the van der Waals radius and Lennard-Jones well depth of OPLS sulfur atoms, and $\mathcal{M}$ is the atomic mass of sulfur. The forcefield coefficients used in this investigation are described completely in the SI Section 1.

We consider two versions of the OPLS-UA model of BDT-TPD. The first version, termed "flexible", is a standard implementation of the OPLS-UA force field as described above. The second version, termed "rigid", represents each of the benzodithiophene and thienopyrrolodione moieties as *rigid bodies* (3.1c).[40]. Within each rigid body the constituent atoms are locked into place relative to each other,

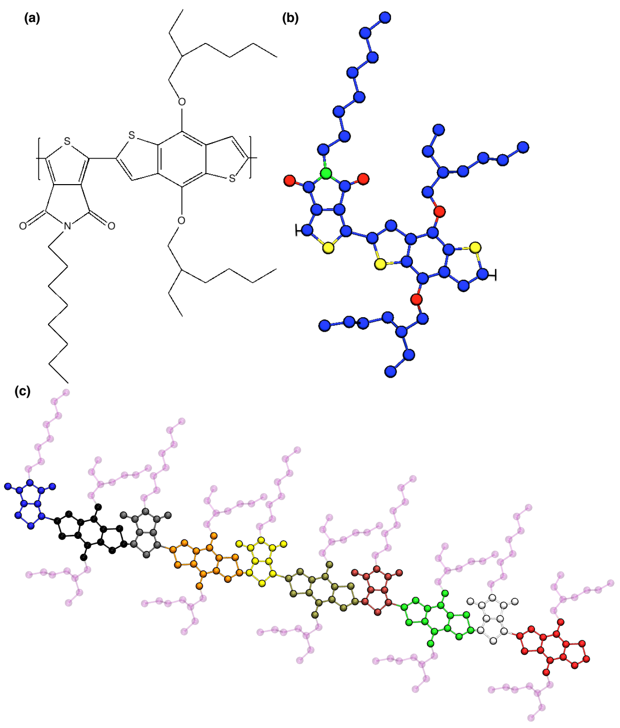


Figure 3.1 (a) Molecular structure of a BDT-TPD monomer. (b) United-atom topology of BDT-TPD, with implicitly modeled hydrogens. Blue spheres represent carbon atoms, red oxygen, yellow sulfur, and the green nitrogen. (c) An example BDT-TPD pentamer, colored by individual rigid bodies.

and a quaternion is used to encode the orientation of the individual benzodithiophene and thienopyrrolodione units. In both the rigid and flexible models, the oligomer sidechains are treated as flexible. Utilizing a rigid representation for these conjugated systems reduces the number of bond, angle, and dihedral degrees of freedom by 135, 200, and 290 respectively per pentamer, resulting in 53% fewer intramolecular constraints than in the flexible model. This reduction in the integrated degrees of freedom results in an increased quantity of simulated timesteps per CPU second.

The current work focuses on whether this improved computational efficiency results in compromised sampling times or the structural prediction capabilities of the model.

We study oligomers with five repeat units of isotactic, regioregular BDT-TPD (3.1), each with molecular weight of 3.542 kDa in implicit solvent. These molecular weights permit the simulation of sufficient material to access experimentally relevant length scales, while avoiding the longer relaxation times associated with longer polymers[13,14,41]. The implicit solvent quality is determined by a multiplicative scaling parameter e_s that modulates the Lennard-Jones well depths as implemented by Shin *et al*[42]. Experimentally, $e_s < 1.0$ corresponds to a solvent within which a solute can be dissolved easily, and $e_s > 1.0$ describes a solvent that is more difficult to dissolve in. At the number densities studied here, the implicit solvent represents 18.2% of the simulation volume, and is meant to capture the mobility-enhancing effect of the solvent before it is evaporated from the active layer. This method allows us to capture the effects of a solvent, without the added computational cost of directly simulating the solvent molecules. In this investigation, we use $e_s = 0.5$ throughout to investigate the structure of BDT-TPD in a relatively good solvent. Here we also assume long-range electrostatics play a negligible role in self-assembly, due to combined effects of charge screening by the implicit solvent and charge delocalization known to occur within conjugated systems. A performance benefit of the negligible charge assumption is that computationally expensive long-range electrostatic interactions need not be computed.

The reduced units of energy, distance, and mass determine derived units of time and temperature. The calculated units of time are therefore $\tau = \sqrt{\frac{\mathcal{M}\sigma^2}{\varepsilon}} = 1.97\times10^{-12}$ s. A unit of dimensionless temperature T corresponds to $\frac{\varepsilon}{k_B} = 126$ K, and dimensionless temperatures between 0.5 T (63 K) and 9.0 T (1134 K) are used as thermostat setpoints in this work. We use the symbol T without units to refer to dimensionless temperature and specify units of Kelvin otherwise. Molecular dynamics simulations are performed using HOOMD-Blue[40,43] on the Maverick and Kestrel high performance computing clusters outfitted with K20-architecture NVIDIA graphics processing units (GPUs). Simulations are performed in the canonical (constant number N, volume V, and temperature T) ensemble, regulated with the Nosé-Hoover thermostat[44] using the MTK equations[45,46]. Particle positions are updated via the velocity-Verlet integration of Newton's equations of motion, after every dimensionless timestep, $dt = 0.001 = 1.97$ fs[47]. We perform simulations across a range of temperatures, T, and unless otherwise specified the fiducial simulation parameters are listed in 3.1.

Table 3.1 Fiducial simulation parameters: Periodic box length L, number of molecules Nmol, mixing temperature Tmix, solvent quality es, and timestep size dt.

L	30.017σ	10.7 nm
T_{mix}	9.5	1198 K
dt	0.001	1.97×10^{-15} s
N_{mol}	197	197
e_s	0.5	good solvent

Initial configurations are generated through random chain placement, mixing, and shrinking to the target density described below. X-Ray reflectivity measurements of poly(BDT-TPD) report 1.17 g/cm^3 [48], and here we initialize volumes of BDT-TPD oligomers with a small amount (18.2% by volume) of implicit solvent. First, the 197 pentamers are randomly initialized in volume large enough to easily place them without overlap. Second, NVT simulations are run at $T_{\text{mix}} = 9.5$ (1198 K) for 1.0×10^5 timesteps (0.197 ns), allowing the pentamers to mix at high temperature and low density. Finally, an additional 1×10^5 timesteps (0.197 ns) of this simulation are performed while the periodic box axes are linearly scaled down to 30.017σ. This initialization protocol efficiently generates unique, randomized configurations of oligomers at $T = 9.5$ (1198 K) that are then annealed or quenched to lower temperatures. When cooled below the glass transition temperature, the oligomers phase separate from the implicit solvent so we expect the resulting structures to be comparable to neat BDT-TPD films after solvent evaporation.

The "annealed" simulations model gradual cooling of BDT-TPD films. Annealing is modeled here with a sequence of MD simulations performed at successively lower temperatures. The dimensionless temperatures in these simulations are decremented instantaneously by $\delta T = 0.5$ (63 K), every 1.2×10^7 timesteps ($\sim$ 24 ns) resulting in a cooling rate of 2.62 Kelvin per nanosecond. Though this annealing rate is extremely fast compared to those achievable in experiments, for the relatively small volumes simulated here it makes the difference between allowing volumes to relax towards thermodynamic equilibrium versus ensuring kinetic arrest. Nineteen simulations are performed for each annealing run, beginning at $T = 9.5$ (1198 K) and ending at $T = 0.5$ (63 K). The "quenched" simulations model cooling schedules that kinetically arrest the structure of BDT-TPD films before they are able to sample the thermodynamically stable configurations that drive self-assembly. Polymer films that have been drop-cast or spin-coated in experiments are assumed to be quenched, as subsequent annealing results in significant ordering[20,49–52] Here, we implement quenched cooling schedules by instantaneously changing the temperature of an initial $T = 9.5$ (1198 K) configuration to the desired set point. We quench to the same nineteen temperatures as sampled during the annealing schedule. This permits the differences in structure between cooling schedules to be observed at each of the 19 state points.

Quenched simulations can be performed in parallel if multiple processors or GPUs are available, whereas annealed simulations must be performed in series. Therefore, if all other factors are equal, it is computationally advantageous to be able to perform quenched simulations. All other factors are usually not equal when comparing equilibrated to non-equilibrium structures (initial conditions matter, integrators can matter, etc), so we expect the utility of quenching models to be limited to the specific case of modeling polymer films that are kinetically arrested.

Simulations are determined to be in thermodynamic equilibrium by comparing the fluctuations in potential energy of a simulation, run for at least twenty times the expected relaxation time from preliminary investigations. First, the evolution of the Lennard-Jones pair potential energy, E_{LJ}, is considered for each simulation, and split into 10 bins. For each bin, the standard deviation in E_{LJ} is calculated. Starting from the final bin and working backwards through simulation time, bins are added to the "equilibrated region" if the standard deviation of the bin's potential energy is no more than twice that of the previous bin in the region.

Once the equilibrated region is determined, its autocorrelation time is calculated to obtain the number of timesteps between statistically independent trajectory frames. An example is shown in figure 3.2, where data after 1×10^{-7} s are considered to be within the equilibrated window. The autocorrelation time measured by the first zero of the self-correlations of the equilibrated windows (SI Section 2) averages 1.08×10^{6} timesteps (2.12 ns) for the four combinations of model flexibility and cooling schedule. The slowest autocorrelation time measured was 2.11×10^{7} timesteps (41.6 ns). Simulation configurations are saved every 1×10^{6} timesteps in accordance with the average autocorrelation times.

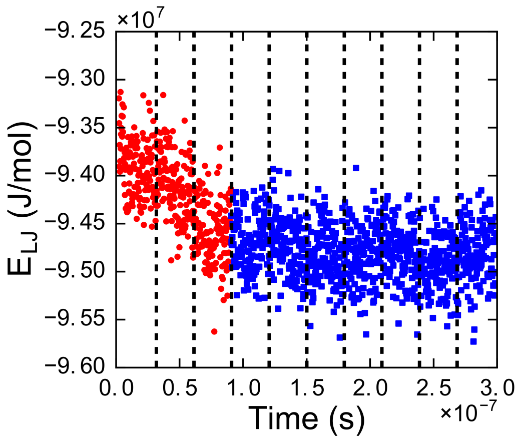


Figure 3.2 A representative non-bonded potential energy trajectory E_{LJ}. These data correspond to the flexible annealed simulation at $T = 2.0$ (252 K). The blue squares describe the equilibrated region of the system, where the standard deviation of energies within each bin (represented by the vertical, black dashes) is no more than twice that of the region to the right of it. Red circles describe the region of the simulation trajectory where the potential energy has not yet relaxed to equilibrium. The Python plotting library Matplotlib is used to generate the plots within this work[53].

3.2.1 Determining structure

The morphological structure of each cooled system is examined through a combination of cluster analysis and simulated X-ray diffraction. Here, neighboring oligomers are considered to be part of the same cluster if the centers-of-mass of at least two adjacent monomers on each chain are located within 1.6σ of each other. Ensuring that TPD moieties on each chain are within the defined center-of-mass cut-off suggests that there will be sufficient molecular orbital overlap between the regions of the molecule that a charge carrier is likely to be delocalized along, resulting in favorable charge transport. Defining clusters in this manner has the effect of identifying the aggregates within the morphology that would be expected to have good inter-chain electronic charge transport - an important characteristic for efficient photovoltaic devices[54,55]. 3.3

shows an example of two clusters. The chains within each cluster satisfy the clustering criterion, however the criterion is not satisfied between the two clusters because only a single monomer of each change lies within the 1.6σ cut-off. Distinguishing clusters of BDT-TPD by color gives a visual representation of structure, in which significant ordering is apparent as BDT-TPD is cooled, as seen in the Visual Molecular Dynamics visualizations (3.4)[56].

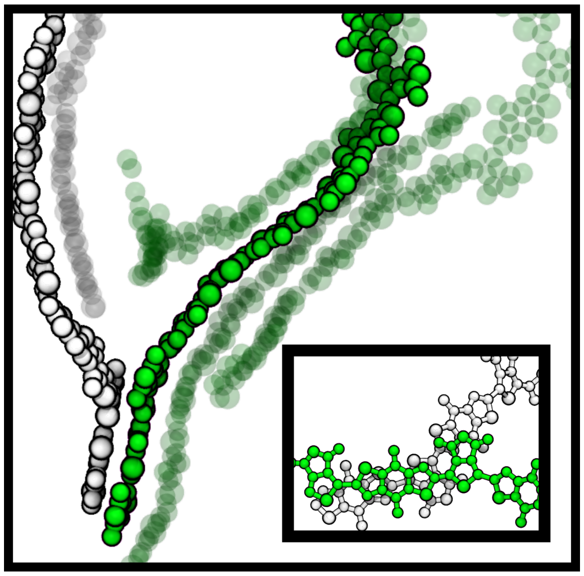


Figure 3.3 An example of two independent clusters (green and white) that do not satisfy our clustering criterion, due to an insufficient number of adjacent backbone moeities within the center-of-mass cut-off for the most external molecules. Only a single green monomer is within 1.6σ of a white monomer (see insert).

Quantification of the coherence length scales in each simulation snapshot is performed by a simulated grazing incidence X-Ray scattering technique. The full details of the simulation methodology, including the mathematical implementations of sample orientation and structure factor calculation, can be found in Ref. [15]. In order to automate the extraction of any coherence length scales and to quantify the degree to which they appear in a simulation snapshot, we perform ensemble averages of

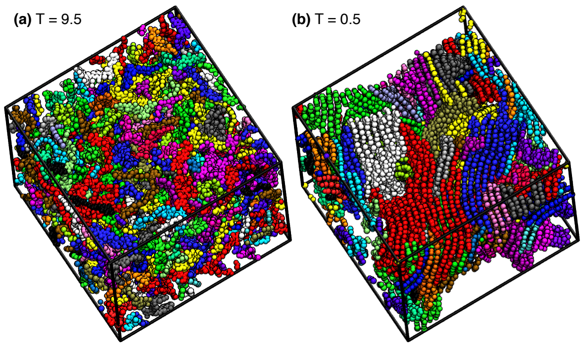


Figure 3.4 An example image of the clusters generated using our specified criteria within the flexible BDT-TPD morphology, taken at (a) the $T = 1198$ K, and (b) after gradual annealing down to $T = 63$ K. For clarity, only the chain backbone moieties are depicted, and like colorings indicate simulation elements that belong to the same cluster. Cluster domains are a few nanometres in size and the simulation volume is cubic.

scattering features over spherically-distributed orientations of the simulation snapshot. These coherence length scales represent averages over 100 orientations of the simulation volume, uniformly distributed about a sphere using the generalized spiral approximation[57]. Each individual scattering pattern captures any anisotropic structural features associated with its scattering orientation and the spherical averaging facilitates the extraction of the most significant structural periodicities. The chain packing is described by the arrangement of the polymer backbone moieties within the sample. As our simulations contain fewer, shorter chains and smaller simulation volumes than are available in experiment, we remove the aliphatic side-chains in order to amplify the signal associated with this backbone structure. To facilitate automated feature detection from the 2D GIXS patterns, we compute structure factors from the radial (q_r) average of diffraction intensities. An example of the structure factor plot at $T = 315$ K for all four models can be seen in 3.5. The location and amplitude of the various peaks in 3.5 are dependent on the combination of the pair interaction potentials, the choice of e_s, the final temperature of the simulation, and whether it was quenched or annealed.

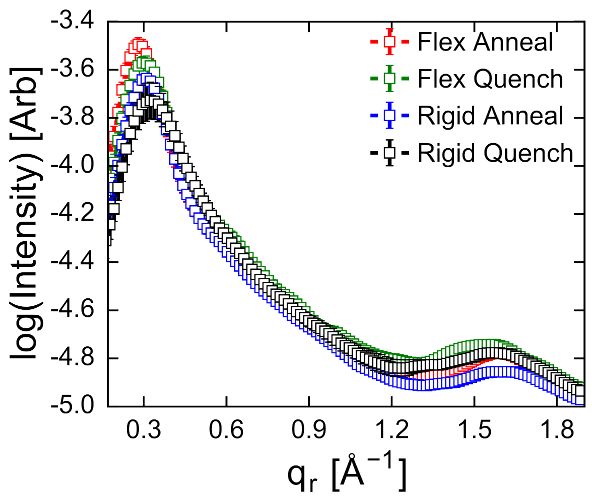


Figure 3.5 The logarithm of the intensity in the scattering pattern as a function of radially-averaged structure factors (q_r) for each model at $T = 315$ K.

We compare the calculated diffraction pattern with experimental GIXS data to validate our models. This practice is particularly useful when there is a match between experimental and simulated scattering patterns, because it gives insight into possible atomistic arrangements that occur in the experimental systems. Matching scattering patterns do not guarantee that the simulated structures are present in experimental films[58], yet represent the most detailed insight into possible structures without developing more sophisticated experimental characterization methods.

3.3 Results

MD simulations of the "rigid" and "flexible" models of BDT-TPD oligomers are equilibrated using the "annealed" and "quenched" cooling schedules at nineteen temperatures for each of the four combinations of model and cooling schedule (flexible-annealed, flexible-quenched, rigid-annealed, and rigid-quenched cases). Ensemble average properties are calculated from statistically independent configurations once the simulation has relaxed as described in the Methods section. On average it took 5×10^6 timesteps (9.85 ns) to reach a steady-state after which the average potential energy autocorrelation times for each state point were 1.08×10^6 timesteps (2.12 ns). We evaluate the computational performance of each model and cooling schedule combination. Additionally, we compute three ensemble properties to characterize each trajectory, (1) the non-bonded potential energy E_{LJ}, (2) the proportion of chains belonging to a cluster ζ, and (3) simulated scattering patterns.

3.3.1 Performance

The computational performance of the four cases are compared by evaluating timesteps per second (TPS), relaxation time, and autocorrelation time at each temperature. 3.6 shows the rigid model has roughly 14% higher TPS compared to the flexible model at the same temperature, and that there is little difference in TPS between quenching and annealing schedules, as expected. Prior work employing rigid models for perylene and perylothiophene showed the rigid model could have significantly different relaxation times and autocorrelation times[30]. Here we find the flexible and rigid models have identical autocorrelation and relaxation times, which means that TPS is an accurate metric for comparing computational efficiency. A more thorough and complete discussion of our simulations' relaxation and autocorrelation times is included in SI Section 2.

3.3.2 Potential Energy

Using E_{LJ} as a proxy for structure we find the annealed and quenched simulations generate identical potential energies when $T \geq 504$ K (3.7a). For cooler temperatures

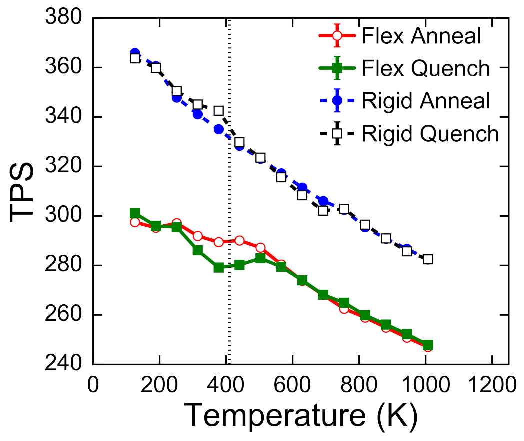


Figure 3.6 The TPS of the simulations explored in this investigation as a function of temperature. The black vertical line indicates the disorder-order transition tempera-ture $T_{\text{DO}} = 410$ K.

($T < 504$ K) we find the annealed runs achieve lower potential energies than their quenched counterparts, as expected. In each case, E_{LJ} is averaged over statistically independent samples. In all four cases, we observe a change in slope of the potential energy below 504 K, which is consistent with a structural phase transition. We observe the largest structural changes to occur between 441 K and 378 K, so we use the average as the disorder-order transition temperature $T_{\text{DO}} = 410$ K. This corresponds well with the glass transition temperature (411 K) measured in experiments[59], although we note the presence of significant (8%) uncertainties both here and in experiments. The non-bonded potential energies are more positive in the rigid case because pairwise interactions between components of a rigid body (which would typically be negative) are omitted. The differences in non-bonded potential energy between the rigid and flexible cases are not constant, which suggests that these models may give rise to different molecular arrangements.(3.7b). However, since differences in E_{LJ} are merely a proxy for structure, a more direct measurement of structure is warranted.

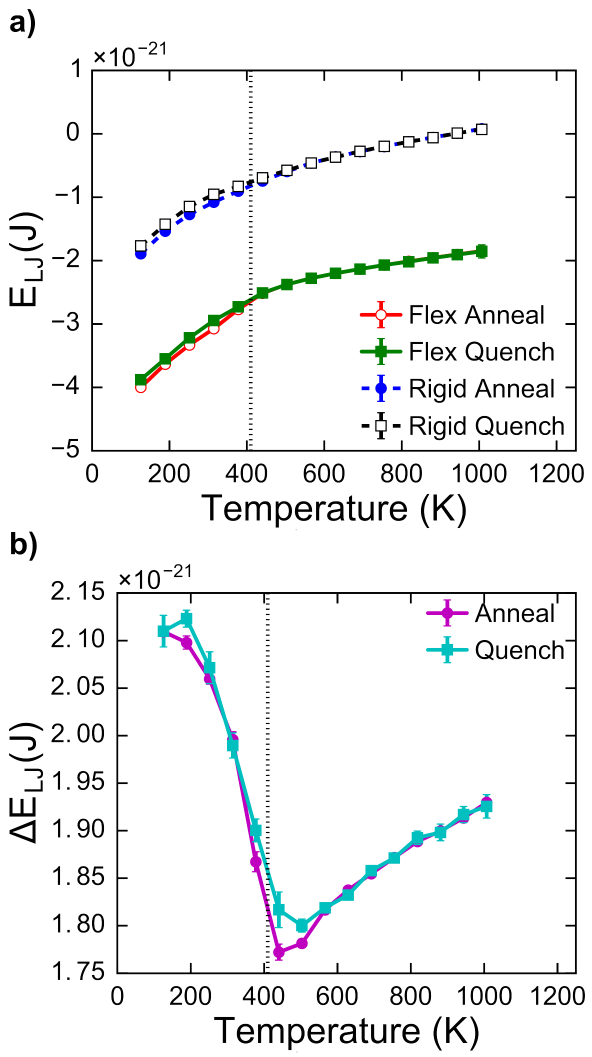


Figure 3.7: (a) Non-bonded potential energy E_{LJ} per atom as a function of temperature, T. (b) Energy difference per atom between the rigid and flexible models. Error bars indicate standard error. Black vertical lines indicate the locations of the disorder-order transition temperature $T_{\mathrm{DO}} = 410$ K.

3.3.3 Clustering

To provide more detailed structural information about morphologies around the transition temperature T_{DO}, we analyze and visualize clusters of oligomer backbones as described in the Methods section. 3.4 shows flexible-annealed morphologies above and below T_{DO}, with backbones colored according to the cluster that they belong to. The ordering of backbones within clusters can be seen in 3.4b. Details of a single cluster are shown in 3.8, where two orientations of the cluster describe the stacks of backbones that form "ribbons". This backbone aggregation corresponds to π-stacking observed experimentally, and we observe an average separation of around 4 Å. Such stacking is beneficial for charge transport, as closely stacked chains lead to increased orbital overlap and faster inter-molecular carrier hops, which can be critical in obtaining the high device efficiencies in organic thin-films.[54,55]. We find that the average spacing between ribbons, similar to the lamellar length scale observed experimentally, is around 21 Å for all four model/cooling combinations.

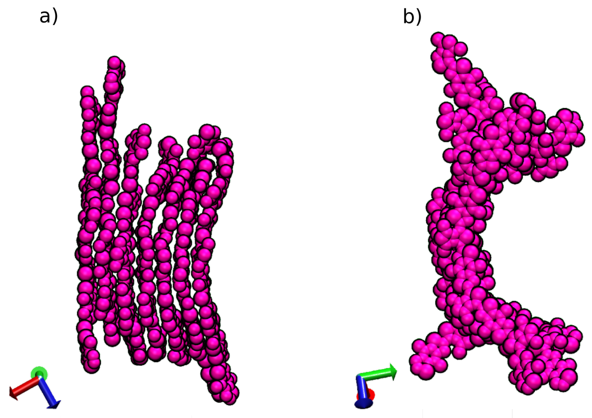


Figure 3.8 A detailed view of a single cluster, viewed from two orientations, taken from a $T = 315$ K flexible-annealed simulation snapshot. The red, green, and blue arrows represent the x, y, and z axis respectively. a) When the cluster is viewed along the y-axis, the aggregation of backbones through pi-stacking can be observed. b) When the cluster is viewed along the x-axis, the stack of backbones are seen to be mostly in-register. We refer to these clusters as "ribbons".

Analyzing the proportion of clustered backbones, ζ, as a function of temperature for the four cases (3.9) gives additional insight into their structural differences. As expected, both the quenching and annealing models give the same ζ when $T > T_{\mathrm{DO}}$.

However, the rigid models are more likely to have clustered chains when $T > T_{\mathrm{DO}}$ than the flexible models. Near T_{DO}, all four cases demonstrated sharp increases in ζ. For low temperatures $T < T_{\mathrm{DO}}$, the quenched cases demonstrated a relative decrease in ζ, which is consistent with the prior results indicating that they lack the thermal energy to rearrange into thermodynamically stable configurations. The annealed cases both show increased clustering as temperature is lowered, though the flexible-annealed case ordered more than the rigid-annealed case. These results reinforce the observations that a structural transition occurs around 410 K and that modeling conjugated systems with rigid bodies has a measurable impact on the π-π structural features that emerge.

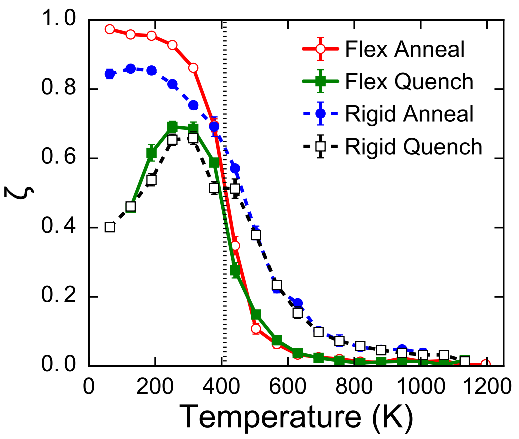


Figure 3.9 The proportion of chains that belong to a cluster containing two or more molecules(ζ), averaged over all statistically independent frames for each state point, cooling schedule and molecular model investigated. The black vertical line indicates the disorder-order transition temperature $T_{\mathrm{DO}} = 410$ K.

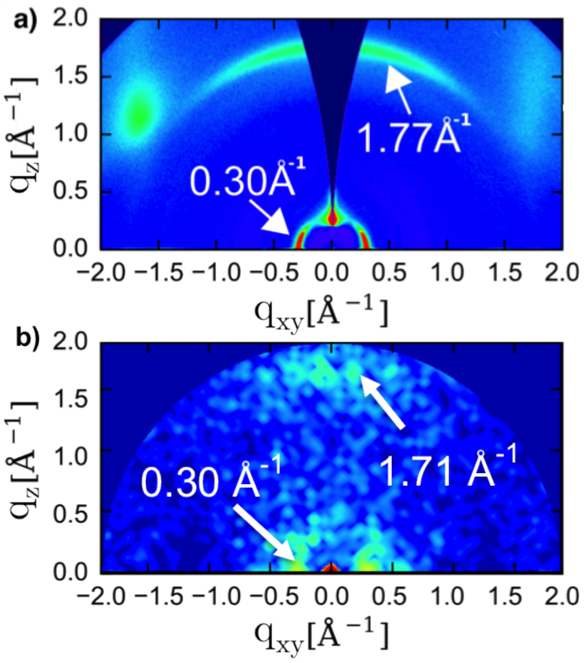


Figure 3.10 Comparison of a single a) experimental, and b) simulated GIXS scattering pattern of a BDT-TPD morphology. The simulated system contains oligomers described by rigid bodies, that were annealed to a temperature of 315 K, and the snapshot was taken from the sample orientation that most clearly showed perpendicular features.

3.3.4 Scattering

For each of the four combinations of flexibility and cooling schedule, we find simulated diffraction patterns that closely match experimental scattering patterns. Below the disorder-order transition temperature $T_{\text{DO}} = 410$ K, GIXS patterns for all four combinations have the same twofold rotational symmetry with orthogonal scattering peaks around 0.30 Å^{-1} and 1.77 Å^{-1}. Figure 3.10a presents the experimental X-ray scattering data obtained for BDT-TPD (synthesis described in SI Section 3 and Ref [60]), in which prominent peaks are observed at $q_r = 0.30$ Å^{-1} ($r_{\text{lamellar}} = 20.9$ Å) and 1.77 Å^{-1} ($r_{\pi-\pi} = 3.5$ Å). Figure 3.10b presents a representative simulated scattering pattern taken from a $T = 2.5$ (315 K) rigid annealed simulation. The simulated diffraction peaks are measured at 0.30 Å^{-1} and 1.71 Å^{-1}, echoing the length scales observed during our clustering analysis, and in excellent agreement with experiment.

We compare the average peak locations at $T = 2.5$ (315 K), where all four combinations of model and cooling schedule demonstrate significant clustering. Averaging over all independent frames and all scattering orientations at $T = 315$ K showed that the rigid annealed system most closely matched the experimental scattering patterns, with only a 3.34% error in lamellar spacing, and a 10.57% error in π-π-spacing. However, the π-stacking lengths across all four models lie within 15.71% (0.55 Å), and lamellar spacing within 6.70% (1.4 Å) of the experimental values, suggesting only minor structural differences between the models. In all four model and cooling schedule combinations, the π-stacking length scales ($3.87 < r_{\pi-\pi} < 4.05$ Å) are predicted to be larger than those observed in experiments ($r_{\pi-\pi} = 3.5$ Å), as seen in table 3.2. This corresponds to a 12.41% over-estimation of the physical π-stacking distance on average, which would be expected due the equilibrium distance between OPLS-UA sulfurs: The minimum of the Lennard-Jones potential is at $2^{1/6}\sigma$, or 1.12σ, which for sulfur is 3.98 Å. The OPLS-UA forcefield was not optimized for conjugated systems, and this observation suggests that new atom types with smaller diameters to represent conjugated carbons and sulfurs may be a small addition to OPLS-UA that will offer improved structural predictions for conjugated molecules. The periodicity of the long-ranged lamellar length scales (around $r_{\text{lamellar}} = 20.9$ Å) is in better experimental agreement, as all four of our models predict length scales $19.5 < r_{\text{lamellar}} < 22.2$ Å, within 6.70% of experimental GIXS data. The OPLS-UA forcefield is well parameterized for alkyl sidechains, which are expected to mediate the long length scales in our system, accounting for the good agreement with experiment.

Table 3.2 Comparison of the lamellar (r_{lamellar}) and π-stacking ($r_{\pi-\pi}$) structural features and their deviation from the experimental values ($r_{\text{expt},\,\pi-\pi} = 3.5$ Å, $r_{\text{expt, lamellar}} = 20.9$ Å) at $T = 2.5$. The subscript 'sim' corresponds to simulated peak locations.

	Long-range (r_{lamellar})			Short-range ($r_{\pi-\pi}$)		
Model	r_{sim} (Å)	$r_{\text{sim}} - r_{\text{expt}}$ (Å)	$\frac{\lvert r_{\text{sim}}-r_{\text{expt}}\rvert}{r_{\text{expt}}}\%$	r_{sim} (Å)	$r_{\text{sim}} - r_{\text{expt}}$ (Å)	$\frac{\lvert r_{\text{sim}}-r_{\text{expt}}\rvert}{r_{\text{expt}}}\%$
Flex Anneal	22.25 ± 0.06	+1.35	6.46	3.937 ± 0.008	+0.437	12.49
Flex Quench	20.9 ± 0.4	+0.0	0.00	4.05 ± 0.03	+0.55	15.71
Rigid Anneal	20.2 ± 0.2	−0.7	3.34	3.87 ± 0.04	+0.37	10.57
Rigid Quench	19.5 ± 0.2	−1.4	6.70	3.88 ± 0.04	+0.38	10.86

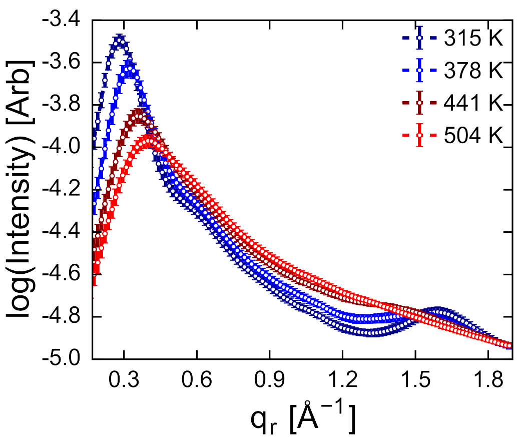


Figure 3.11 The logarithm of the scattering intensity as a function of radially-averaged structure factors (q_r) over each statistically independent frame for a representative flexible-annealed system at simulation temperatures above and below $T_{\text{DO}} = 410K$. Only one model is shown for clarity as all four combinations of cooling schedules and models demonstrated the same trend.

Emergence of increased ordering via simulated scattering analysis supports the observations from our non-bonded potential energy measurements and clustering data that $T_{\text{DO}} = 410K$. In 3.11, we consider the scattering peak intensities averaged over statistically independent frames at four temperatures for representative rigid-quenched simulations. As the temperature is lowered, the intensity of the low $q_r \sim 0.3$ Å^{-1} peak increases and shifts downwards, corresponding to longer length scales. This indicates that there is more and longer-range structural ordering present in the morphology at lower temperatures, in analogy to lamellar and liquid crystal formation observed in neat poly(3-hexylthiophene)-b-poly-(90,90-dioctylfluorene) (P3HT-b-PF) thin films[61]. At high $q_r \sim 1.6$ Å^{-1}, there is a local peak intensity maximum corresponding to increased π-stacking order in the system when cooled below T_{DO}, but this feature is not observed when $T > T_{\mathrm{DO}}$. The presence of the π-stacking peak at $T < 410$ K reinforces that a structural change is occuring in the system when cooling from above T_{DO} to below it.

3.4 Conclusions

The OPLS-UA model used in this investigation captures the phase behavior of BDT-TPD, with both rigid and flexible models showing a glass transition temperature around 410 ± 32 K, in agreement with the glass transition temperature (411 K) measured in experiments[59]. Utilizing rigid bodies to model conjugated systems in BDT-TPD results in 14% faster simulations that faithfully reproduce the structural characteristics observed in experiments. Cooling BDT-TPD oligomers below 441 K gives rise to increasingly ordered stacks of polymer backbones ("ribbons"), with π-stacking within the ribbons and the "lamellar" spacing between the ribbons for both rigid and flexible models, whether they are annealed or quenched. All four combinations of model and cooling schedule overpredict the π-stacking length (by 0.37 Å to 0.55 Å), which is not surprising considering the OPLS-UA forcefield is not optimized for these conjugated backbones. All of the model and cooling combinations predict the lamellar spacing within 1.35 Å of the experimental value, with the flexible-quench matching best. Overall, the rigid-annealed simulations best match experiments with the closest prediction of π-stacking and only 3.34% error in lamellar spacing, and the rigid-quenched simulations provide the most structural insight for the least computation. In short, we find the phase behavior and morphology of BDT-TPD to be accurately predicted by GPU-accelerated simulations of short oligomers in implicit solvent using the OPLS-UA force-field without explicit long-range electrostatics.

The accurate structural predictions observed here support the modeling assumptions that the partial charges of BDT-TPD, the solvent degrees of freedom, and the flexibility of each conjugated monomer unit play negligible roles in determining self-assembled structure. We interpret these results to indicate that these modeling assumptions are justified for accelerating the prediction of organic photovoltaic morphologies. This is an important result in the context of high-throughput simulations needed to screen thousands of candidate chemicals for those most likely to result in high-efficiency organic photovoltaics because it shows that "off-the-shelf" force fields that have not been optimized for a particular chemistry have high predictive utility. The computational efficiency of quenching compared to annealing is significant, as here a single 12-hour quench gives as experimentally-relevant results as over 144 hours of annealing. We recommended using computationally efficient techniques (rigid bodies, instantaneous quenching) for estimating phase transitions and identifying candidate phases, followed by more detailed explorations where appropriate. As one example relevant to organic photovoltaics, we show in other work how back-mapping atomistic detail for calculating properties such as charge mobility is essential[15]. Of course, there are certainly moeties for which charge, flexibility, and solvent assumptions made here will break down, so chemical intuition or first-principles calculations should be used before blindly applying them.

Using a simplified united atom model to predict BTD-TPD oligomer structure opens related questions that extend from this work. Firstly, to what degree can BDT-TPD and related organic semiconductors be further coarse-grained before the increases in sampling efficiency are outweighed by inaccuracies in structural predictions? Secondly, are there fundamental limits to using coarse-graining and back-mapping as a form of thermodynamic integration to more rigorously calculate free energy differences between materials? Thirdly, how generally applicable is the rigid-body assumption for conjugated systems? Answering these questions and further validating our modeling assumptions by predicting the morphologies of as yet unsynthesized organic semiconductors is the focus of future work.

REFERENCES

[1] Alan J. Heeger. Semiconducting polymers: the Third Generation. *Chemical Society Reviews*, 39(7):2354, 2010.

[2] Ana Claudia Arias, J Devin MacKenzie, Iain McCulloch, Jonathan Rivnay, and Alberto Salleo. Materials and Applications for Large Area Electronics: Solution-Based Approaches. *Chemical Reviews*, 110(1):3–24, jan 2010.

[3] Frederik C Krebs, Thomas Tromholt, and Mikkel Jørgensen. Upscaling of Polymer Solar Cell Fabrication Using Full Roll-to-Roll Processing. *Nanoscale*, 2(6):873, 2010.

[4] Yongfang Li. Molecular Design of Photovoltaic Materials for Polymer Solar Cells: Toward Suitable Electronic Energy Levels and Broad Absorption. *Accounts of Chemical Research*, 45(5):723–733, may 2012.

[5] Nieves Espinosa, Markus Hösel, Dechan Angmo, and Frederik C. Krebs. Solar Cells with One-Day Energy Payback for the Factories of the Future. *Energy & Environmental Science*, 5(1):5117, 2012.

[6] John E. Anthony. Organic electronics: Addressing challenges. *Nature Materials*, 13(8):773–775, 2014.

[7] H. Sirringhaus. 25th Anniversary Article: Organic Field-Effect Transistors: The Path Beyond Amorphous Silicon. *Advanced Materials*, 26(9):1319–1335, 2014.

[8] M. L. Jones, D. M. Huang, B. Chakrabarti, and Chris Groves. Relating Molecular Morphology to Charge Mobility in Semicrystalline Conjugated Polymers. *Journal of Physical Chemistry C*, 120(8):4240–4250, 2016.

[9] Jonathan A. Bartelt, Zach M. Beiley, Eric T. Hoke, William R. Mateker, Jessica D. Douglas, Brian A. Collins, John R. Tumbleston, Kenneth R. Graham, Aram Amassian, Harald Ade, Jean M. J. Fréchet, Michael F. Toney, and Michael D. McGehee. The Importance of Fullerene Percolation in the Mixed Regions of Polymer-Fullerene Bulk Heterojunction Solar Cells. *Advanced Energy Materials*, 3(3):364–374, mar 2013.

[10] Rodrigo Noriega, Jonathan Rivnay, Koen Vandewal, Felix P. V. Koch, Natalie Stingelin, Paul Smith, Michael F. Toney, and Alberto Salleo. A general relationship between disorder, aggregation and charge transport in conjugated polymers. *Nature Materials*, 12(11):1038–1044, nov 2013.

[11] Christoph J. Brabec, Srinivas Gowrisanker, Jonathan J M Halls, Darin Laird, Shijun Jia, and Shawn P Williams. Polymer-Fullerene Bulk-Heterojunction Solar Cells. *Advanced Materials*, 22(34):3839–3856, sep 2010.

[12] Letian Dou, Jingbi You, Ziruo Hong, Zheng Xu, Gang Li, Robert a Street, and Yang Yang. 25th Anniversary Article: A Decade of Organic/Polymeric Photovoltaic Research. *Advanced Materials*, 25(46):6642–6671, dec 2013.

[13] Eric Jankowski, Hilary S. Marsh, and Arthi Jayaraman. Computationally linking molecular features of conjugated polymers and fullerene derivatives to bulk heterojunction morphology. *Macromolecules*, 46(14):5775–5785, jul 2013.

[14] Hilary S Marsh, Eric Jankowski, and Arthi Jayaraman. Controlling the Morphology of Model Conjugated Thiophene Oligomers through Alkyl Side Chain Length, Placement, and Interactions. *Macromolecules*, 47(8):2736–2747, apr 2014.

[15] Matthew Lewis Jones and Eric Jankowski. Computationally connecting organic photovoltaic performance to atomistic arrangements and bulk morphology. *Molecular Simulation*, 43(10-11):1–18, mar 2017.

[16] Frederik C Krebs. Fabrication and processing of polymer solar cells: A review of printing and coating techniques. *Solar Energy Materials and Solar Cells*, 93(4):394–412, apr 2009.

[17] Uyxing Vongsaysy, Dario M. Bassani, Laurent Servant, Bertrand Pavageau, Guillaume Wantz, and Hany Aziz. Formulation strategies for optimizing the morphology of polymeric bulk heterojunction organic solar cells: a brief review. *Journal of Photonics for Energy*, 4(1):040998, jun 2014.

[18] Dean M DeLongchamp, R Joseph Kline, and Andrew Herzing. Nanoscale Structure Measurements for Polymer-Fullerene Photovoltaics. *Energy & Environmental Science*, 5(3):5980–5993, 2012.

[19] Jeffrey Peet, Michelle L. Senatore, Alan J. Heeger, and Guillermo C. Bazan. The Role of Processing in the Fabrication and Optimization of Plastic Solar Cells. *Advanced Materials*, 21(14-15):1521–1527, 2009.

[20] A Zen, J Pflaum, S Hirschmann, W Zhuang, F Jaiser, U. Asawapirom, J. P. Rabe, U Scherf, and D Neher. Effect of Molecular Weight and Annealing of Poly(3-hexylthiophene)s on the Performance of Organic Field-Effect Transistors. *Advanced Functional Materials*, 14(8):757–764, aug 2004.

[21] M Surin, Ph. Leclère, R Lazzaroni, J D Yuen, G Wang, D Moses, A J Heeger, S Cho, and K Lee. Relationship between the microscopic morphology and the charge transport properties in poly(3-hexylthiophene) field-effect transistors. *Journal of Applied Physics*, 100(3):033712, 2006.

[22] Youngkyoo Kim, Steffan Cook, Sachetan M. Tuladhar, Stelios A. Choulis, Jenny Nelson, James R. Durrant, Donal D.C. Bradley, Mark Giles, Iain McCulloch, Chang Sik Ha, and Moonhor Ree. A strong regioregularity effect in self-organizing conjugated polymer films and high-efficiency polythiophene:fullerene solar cells. *Nature Materials*, 5(3):197–203, mar 2006.

[23] Minh Trung Dang, Guillaume Wantz, Habiba Bejbouji, Mathieu Urien, Olivier J. Dautel, Laurence Vignau, and Lionel Hirsch. Polymeric Solar Cells Based on P3HT:PCBM: Role of the Casting Solvent. *Solar Energy Materials and Solar Cells*, 95(12):3408–3418, 2011.

[24] Do Hwan Kim, Yeong Don Park, Yunseok Jang, Sungsoo Kim, and Kilwon Cho. Solvent Vapor-Induced Nanowire Formation in Poly(3-hexylthiophene) Thin Films. *Macromolecular Rapid Communications*, 26(10):834–839, may 2005.

[25] Stefan Haid, Magdalena Marszalek, Amaresh Mishra, Mateusz Wielopolski, Joël Teuscher, Jacques E. Moser, Robin Humphry-Baker, Shaik M. Zakeeruddin, Michael Grätzel, and Peter Bäuerle. Significant Improvement of Dye-Sensitized Solar Cell Performance by Small Structural Modification in π-Conjugated Donor-Acceptor Dyes. *Advanced Functional Materials*, 22(6):1291–1302, 2012.

[26] Wei Chen, Maxim P. Nikiforov, and Seth B. Darling. Morphology characterization in organic and hybrid solar cells. *Energy & Environmental Science*, 5(8):8045–8074, 2012.

[27] Jan-Michael Y. Carrillo, Rajeev Kumar, Monojoy Goswami, Bobby G. Sumpter, and W. Michael Brown. New insights into the dynamics and morphology of P3HT:PCBM active layers in bulk heterojunctions. *Physical Chemistry Chemical Physics*, 15(41):17873, nov 2013.

[28] Siewert J Marrink and D Peter Tieleman. Perspective on the Martini model. *Chemical Society reviews*, 42(16):6801–6822, aug 2013.

[29] J Huang, Y Ie, M Karakawa, and Y Aso. Low band-gap donor–acceptor copolymers based on dioxocylopenta[c]thiophene derivatives as acceptor units: synthesis, properties, and photovoltaic performances. *J. Mater. Chem. A*, (207890), 2013.

[30] Evan D Miller, Matthew Lewis Jones, and Eric Jankowski. Enhanced Computational Sampling of Perylene and Perylothiophene Packing with Rigid-Body Models. *ACS Omega*, 2(1):353–362, jan 2017.

[31] Shu Liu, Xichang Bao, Wei Li, Kailong Wu, Guohua Xie, Renqiang Yang, and Chuluo Yang. Benzo[1,2- b :4,5- b ']dithiophene and Thieno[3,4- c]pyrrole-4,6-dione Based Donor-π-Acceptor Conjugated Polymers for High Performance Solar Cells by Rational Structure Modulation. *Macromolecules*, 48(9):2948–2957, may 2015.

[32] Serge Beaupré, Sepideh Shaker-Sepasgozar, Ahmed Najari, and Mario Leclerc. Random D–A 1 –D–A 2 terpolymers based on benzodithiophene, thiadiazole[3,4-e]isoindole-5,7-dione and thieno[3,4-c]pyrrole-4,6-dione for efficient polymer solar cells. *J. Mater. Chem. A*, 5(14):6638–6647, 2017.

[33] William L. Jorgensen and Julian. Tirado-Rives. The OPLS [Optimized Potentials for Liquid Simulations] Potential Functions for Proteins, Energy Minimizations for Crystals of Cyclic Peptides and Crambin. *Journal of the American Chemical Society*, 110(6):1657–1666, mar 1988.

[34] Marcus G Martin and J Ilja Siepmann. Transferable Potentials for Phase Equilibria. 1. United-Atom Description of n-Alkanes. *Journal of Physical Chemistry B*, 102(14):2569–2577, 1998.

[35] Do Y. Yoon, Grant D. Smith, and Tsunetoshi Matsuda. A Comparison in Simulations of a United Atom and an Explicit of polymethylene Atom Model. *The Journal of Chemical Physics*, 98(12):10037–10043, 1993.

[36] Wolfgang Paul, Do Y. Yoon, and Grant D. Smith. An Optimized United Atom Model for Simulations of polymethylene Melts. *Journal of Chemical Physics*, 103(4):1702–1709, 1995.

[37] John D. McCoy and John G. Curro. Mapping of Explicit Atom onto United Atom Potentials. *Macromolecules*, 31(26):9362–9368, 1998.

[38] Mesfin Tsige, John G. Curro, Gary S. Grest, and John D. McCoy. Molecular Dynamics Simulations and Integral Equation Theory of Alkane Chains: Comparison of Explicit and United Atom Models. *Macromolecules*, 36(6):2158–2164, 2003.

[39] Chunxia Chen, Praveen Depa, Victoria García Sakai, Janna K. Maranas, Jeffrey W. Lynn, Inmaculada Peral, and John R. D. Copley. A Comparison of United Atom, Explicit Atom, and Coarse-grained Simulation Models for poly(ethylene oxide). *Journal of Chemical Physics*, 124(23):234901:1–234901:11, 2006.

[40] Trung Dac Nguyen, Carolyn L Phillips, Joshua a. Anderson, and Sharon C Glotzer. Rigid Body Constraints Realized in Massively-parallel Molecular Dynamics on Graphics Processing Units. *Computer Physics Communications*, 182(11):2307–2313, nov 2011.

[41] Kyra N Schwarz, Tak W Kee, and David M Huang. Coarse-grained simulations of the solution-phase self-assembly of poly(3-hexylthiophene) nanostructures. *Nanoscale*, 5(5):2017–2027, mar 2013.

[42] Hyeyoung Shin, Tod a. Pascal, William a. Goddard, and Hyungjun Kim. Scaled effective solvent method for predicting the equilibrium ensemble of structures with analysis of thermodynamic properties of amorphous polyethylene glycol-water mixtures. *Journal of Physical Chemistry B*, 117(3):916–927, 2013.

[43] Joshua A. Anderson, Chris D. Lorenz, and A. Travesset. General Purpose Molecular Dynamics Simulations Fully Implemented on Graphics Processing Units. *Journal of Computational Physics*, 227(10):5342–5359, may 2008.

[44] William G. Hoover. Canonical Dynamics: Equilibrium Phase-space Distributions. *Physical Review A*, 31(3):1695–1697, 1985.

[45] A. Denner, S. Dittmaier, M. Roth, and L. H. Wieders. Complete electroweak O(alpha) corrections to charged-current e+e- –> 4fermion processes. *The Journal of Chemical Physics*, 101(5):4177–4189, feb 2005.

[46] J. Cao and G. J. Martyna. Adiabatic path integral molecular dynamics methods. II. Algorithms. *The Journal of Chemical Physics*, 104(5):2028–2035, feb 1996.

[47] W. C. Swope, H. C. Andersen, P. H. Berens, and K. R. Wilson. A Computer Simulation Method for the Calculation of Equilibrium Constants for the Formation of Physical Clusters of Molecules: Application to Small Water Clusters. *Journal of Chemical Physics*, 76(1):637–649, 1982.

[48] William R. Mateker, Thomas Heumueller, Rongrong Cheacharoen, I. T. Sachs-Quintana, Michael D. McGehee, Julien Warnan, Pierre M. Beaujuge, Xiaofeng Liu, and Guillermo C. Bazan. Molecular Packing and Arrangement Govern the Photo-Oxidative Stability of Organic Photovoltaic Materials. *Chemistry of Materials*, 27(18):6345–6353, sep 2015.

[49] Miguel a. Modestino, Elaine R Chan, Alexander Hexemer, Jeffrey J Urban, and Rachel a. Segalman. Controlling Nanorod Self-Assembly in Polymer Thin Films. *Macromolecules*, 44(18):7364–7371, sep 2011.

[50] Eric Verploegen, Rajib Mondal, Christopher J Bettinger, Seihout Sok, Michael F Toney, and Zhenan Bao. Effects of Thermal Annealing Upon the Morphology of Polymer-Fullerene Blends. *Advanced Functional Materials*, 20(20):3519–3529, oct 2010.

[51] Benjamin H. Wunsch, Mariacristina Rumi, Naga Rajesh Tummala, Chad Risko, Dun Yen, Xerxes Steirer, Jeremy Gantz, Marcel M Said, Neal R. Armstrong, Jean-Luc Luc Brédas, Dun-Yen Kang, K. Xerxes Steirer, Jeremy Gantz, Marcel M Said, Neal R. Armstrong, Jean-Luc Luc Brédas, David Bucknall,

and Seth R. Marder. Structure–processing–property correlations in solution-processed, small-molecule, organic solar cells. *Journal of Materials Chemistry C*, 1(34):5250, 2013.

[52] Jeffrey M. Lucas, Joelle a. Labastide, Lang Wei, Jonathan S. Tinkham, Michael D. Barnes, and Paul M. Lahti. Carpenter's Rule Folding in Rigid–Flexible Block Copolymers with Conjugation-Interrupting, Flexible Tethers Between Oligophenylenevinylenes. *The Journal of Physical Chemistry A*, page 150630131149000, 2015.

[53] J D Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[54] Lei Zhai, Saiful I. Khondaker, Jayan Thomas, Chen Shen, and Matthew McInnis. Ordered conjugated polymer nano- and microstructures: Structure control for improved performance of organic electronics. *Nano Today*, 9(6):705–721, dec 2014.

[55] Masahiro Sato, Akiko Kumada, Kunihiko Hidaka, Toshiyuki Hirano, and Fumitoshi Sato. Quantum chemical calculation of hole transport properties in crystalline polyethylene. *IEEE Transactions on Dielectrics and Electrical Insulation*, 23(5):3045–3052, oct 2016.

[56] William Humphrey, Andrew Dalke, and Klaus Schulten. VMD: Visual molecular dynamics. *Journal of Molecular Graphics*, 14(1):33–38, feb 1996.

[57] E. A. Rakhmanov, E. B. Saff, and Y. M. Zhou. Minimal Discrete Energy on the Sphere. *Mathematical Research Letters*, 1(6):647–662, 1994.

[58] Herbert Hauptman. The phase problem of x-ray crystallography, 1983.

[59] Yingping Zou, Ahmed Najari, Philippe Berrouard, Serge Beaupre, Badrou Réda Aïch, Ye Tao, and Mario Leclerc. A Thieno[3, 4-c]pyrrole-4, 6-dione-Based Copolymer for Efficient Solar Cells. *Journal of the American Chemical Society*, 132(15):5330–5331, 2010.

[60] Wade A Braunecker, Zbyslaw R Owczarczyk, Andres Garcia, Nikos Kopidakis, Ross E Larsen, Scott R Hammond, David S Ginley, and Dana C Olson.

Benzodithiophene and Imide-Based Copolymers for Photovoltaic Applications. *Chemistry of Materials*, 24(7):1346–1356, apr 2012.

[61] Yen-Hao Lin, Kevin G. Yager, Bridget Stewart, and Rafael Verduzco. Lamellar and liquid crystal ordering in solvent-annealed all-conjugated block copolymers. *Soft Matter*, 10(21):3817–3825, 2014.

CHAPTER 4

GENERAL-PURPOSE COARSE-GRAINED TOUGHENED THERMOSET MODEL FOR 44DDS/DGEBA/PES[1]

4.1 Introduction

Lightweight composites are increasingly used as alternatives to metal components of aircraft, especially over the last decades. Initially reserved for the most demanding aerospace applications, such as fighter aircraft, composite components are now prevalent in commercial aircraft, including 50% of the weight of the Boeing 787[1]. This proliferation is enabled by improvements in composite formulations and processing, yet there exist significant opportunities to improve the reliable manufacturing of composite aerospace parts. Specifically, control of the thermoset matrix nanostructure (*morphology*) during the curing is currently underdeveloped and improvements could drastically increase the reliability and reduce the time and energy costs of part fabrication[2–4]. The challenge lies in understanding how morphology depends on the conditions experienced by the part during curing, and which morphologies have sufficient material properties for specific applications. Improved ability to predict properties from morphologies and morphologies from processing will enable:

[1]This chapter has been published to *Polymers* and is referneced as "Henry, M. M., Thomas, S., Alberts, M., Estridge, C. E., Farmer, B., McNair, O., & Jankowski, E. (2020). General-Purpose Coarse-Grained Toughened Thermoset Model for 44DDS/DGEBA/PES. Polymers, 12(11), 2547. https://doi.org/10.3390/polym12112547."

1. Predicting how deviations from process specifications impact performance.

2. Composite formulations optimized for manufacturing requirements.

3. Temperature schedules (termed *cure profiles*) optimized for speed and reproducibility.

Embedding fibers in a matrix of polymers serves to support the fibers and transfers loads between them, providing the attractive bulk mechanical properties of fiber-based composites. The main chemical components of a thermoset are an epoxy species, an amine species, and sometimes a toughening agent. Here we focus on the epoxy bisphenol A diglycidyl ether (DGEBA), amine 4,4'-diaminodiphenyl Sulfone (44DDS) mixed with toughener Poly(oxy-1,4-phenylsulfonyl-1,4-phenyl) (PES), a toughened thermoset found in aerospace applications (Figure 4.1). Thermoset manufacturers recommended cure profiles for matrix formulations based on cure requirements of the crosslinked polymer. Recommended cure profiles are empirically determined and are not necessarily the most efficient paths to sufficiently cured parts.

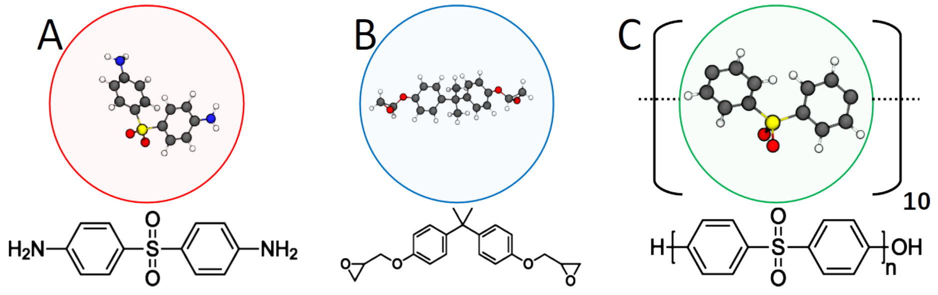


Figure 4.1 Coarse-grained representations of 44DDS (A), DGEBA (B), and PES (C) repeat units. The amines (A) can bond to up to four epoxies (B), which can each bond to up to two amines. All toughener molecules are linear 10-mers of (C).

During curing, the crosslinking of DGEBA and 44DDS lowers the miscibility of PES, and this reaction-induced phase separation (RIPS) results in toughener-rich domain formation[5–9]. Early work by Sultan and McGarry (Reference [10]) used rubber additives to improve fracture toughness in exchange for lower thermal stability at high temperatures[6]. Since then, control over the toughener domains has been shown to increase fracture toughness without sacrificing other desirable mechanical properties [7,8,11–17]. The toughener domains improve mechanical properties though a variety of mechanisms, including crack tip blunting, voiding at the interface between thermoset and toughener, and shear yielding[18,19]. Smaller domain sizes are argued to improve mechanical properties, as it results in higher surface area between the thermoplastic and thermoset domains[7]. Block copolymers have also been deployed to control toughener morphology and composite mechanical properties[20–22]. Regardless of mechanism, understanding and controlling the morphology of tougheners whose phase-separation is induced by the crosslinking is central to controlling the mechanical properties of the matrix.

Temperature deviations away from a desired cure profile increase the probability that the morphology and material properties of a part are compromised, and these parts must undergo material review to confirm whether this is the case. Material review involves the creation of a sample volume cured with the same temperature deviation as the original part, which then undergoes mechanical testing. Throwing away the deviant part and curing a new one usually costs less time and effort than replicating the deviation and validating the sample volume, which is wasteful in the cases of sufficiently strong deviants. Avoiding this waste would be possible if the sensitivity of mechanical properties to cure profile deviations were more fully understood.

Computer simulations are needed for making sense of cure profile sensitivity because the parameter space combinatorics prohibit experimental enumeration, compounded by the impracticality of obtaining atomic-level detail of each cured morphology. Formulating a thermoset includes choosing the chemistry and proportions of epoxy, crosslinker, toughener, and additives compounds, resulting in combinatorial explosion of candidate formulations. Further, each formulation can result in a wide range of morphologies that depend upon cure profile, the number of which adds another factor to the intractability of enumeration. Models for thermoset curing implemented in computer simulations provide a proxy for part fabrication that are faster and less expensive to perform, and can provide insight into how atomic-level structure evolves and impacts properties. Further, modern GPU (graphics processing unit) hardware enables sensitivity analysis and optimizing cure profiles for desired morphologies because screenings of independent formulations and cure profiles can be performed in parallel.

Computationally predicting morphology requires models that faithfully capture the thermodynamics and kinetics of the crosslinking reaction between amine and epoxy molecules, and resulting phase separation of any tougheners present. Doing so is challenging because reactions dynamics occur at fast ($1 \times 10^{-12}\,\mathrm{s}$) and small ($1 \times 10^{-10}\,\mathrm{m}$) scales, while morphology evolution occurs at slow ($1 \times 10^{2}\,\mathrm{s}$) and large ($1 \times 10^{-6}\,\mathrm{m}$) scales. Accurately simulating the cross-linking of the epoxy and amine species is crucial when modeling these systems as the bonding network influences the properties of the thermoset [23,24], in particular the relationship between the glass transition temperature T_g and cure fraction α described by the DiBenedetto equation [23,25–31]. Atomistic molecular dynamics (MD) simulations with temperature-independent bonding models have been successfully deployed to generate crosslinked nanostructures and glass transition temperatures T_g, but are limited to simulation volumes around ($13\,\mathrm{nm}^3$) [32–36]. The work of Li, Strachan and coworkers [32,33] demonstrates atomistic simulations of DGEBA reacted with 44DDS, 33DDS, and other crosslinkers to predict mechanical properties including T_g, density, modulus, and expansion coefficients. In the case of T_g for 44DDS/DGEBA, the atomistic simulations performed overpredict $T_{g,sim} = 525$ K compared to DSC experiments $T_{g,exp} = 450$ K at 92% cure, though no empirical fitting is performed and cooling-rate-dependent corrections help explain the discrepancy[32,33]. Khare and Phelan investigate similar, untoughened DGEBA (2-mers) and 44DDS and predict 489 K$\leq T_{g,sim}(\alpha = 100\%) \leq$ 556 K, depending on cooling rate[36].

Coarse-grained (CG) approaches demonstrate the ability to access substantially larger simulation volumes and time scales than atomistic approaches, and mapping atomistic degrees of freedom into crosslinked networks enables calculation of material properties[37–40]. In both References [38] and [40], one-site dissipative particle dynamics (DPD) models are used to represent reacting monomers of 44DDS/DGEBA and DGEBA/DETA (Diethylenetriamine) , respectively. In both cases, experimentally reasonable T_g are calculated after backmapping, and the case is made for large system sizes for observing toughener microstructure[38] and sufficient structural relaxation[40]. Langeloth et al. develop a coarse-grained model of intermediate resolution to study toughened DGEBA/DETA and show significant discrepancies in $T_g(\alpha)_{CG} < T_g(\alpha)_{AA}$. Earlier this year Pervaje et al. develop another intermediate-resolution coarse-grained model of reacting thermosets parameterized by SAFT-γ Mie calculations, which includes temperature-dependent reactions and a novel bonding algorithm[41]. Applied to polyester-polyol resins, T_g predictions from the coarse model are in agreement with experiments[41]. While the exact details and experimental validations depend on the themoset formulation and the force fields used, multiscale approaches that use coarse models to access long times, large volumes, and high cure fractions $0.9 < \alpha < 0.95$

and atomistic simulations for mechanical property calculations have begun spanning the ≈ 12 orders of magnitude between reaction dynamics and phase separation.

However, to predict how thermoset microstructure depends on cure profiles, temperature-dependent reaction models are necessary. In our prior work developing *epoxpy*[42] , we implemented such a reaction model with DPD coarse-grained simulations. Here, we extend *epoxpy* and focus on simulation workflows for parameterizing, validating, and exploring materials behaviors of reacting thermosets with 44DDS/DGEBA toughened with PES as a case study. While prior studies[32,33,36,38–41,43,44] have included or implemented (1) Reaction rates calibrated against experimentally observed reaction models, (2) Microphase separation of toughener, or (3) $T_g(\alpha)$ validated against experiments, this work is distinguished by the inclusion of all three simultaneously, and crucially (4) We demonstrate for the first time structural sensitivity to cure profile.

4.2 Model

Spherical simulation elements ("beads") are used to represent monomers of amine 44DDS (A), epoxy DGEBA (B), and each repeat unit of PES (C) 10-mers (Figure 4.1). Non-bonded interactions are modeled with the 12-6 Lennard-Jones (LJ) potential

$$\begin{aligned} V_{\mathrm{LJ}}(r) =& 4\varepsilon\left[\left(\frac{\sigma}{r}\right)^{12} - \left(\frac{\sigma}{r}\right)^{6}\right] && r < r_{\mathrm{cut}} \\ =& 0 && r \geq r_{\mathrm{cut}}, \end{aligned}$$

where the parameters σ represent "size" of simulation elements and ε sets the magnitude of the potential energy minimum between two simulation elements. Throughout this work σ is used as the dimensionless length scale and $\sigma_A = \sigma_B = \sigma_C = \sigma = 1$

nm. We note that the relatively hard-core repulsion of the LJ potential prevents chain crossing that is commonplace in DPD simulations, with impacts on network structure and T_g calculations. Energy scales ε calculated from cohesive energy calculations described in Section 4.4.1 and are summarized in Table 4.1. Interactions between dissimilar simulation elements ("cross" interactions) are obtained using Lorentz-Berthelot (LB) mixing rules applied in prior DGEBA studies[45–47], where

$$\epsilon_{AB} = \sqrt{\epsilon_A \epsilon_B} \tag{4.1}$$

and

$$\sigma_{AB} = \frac{\sigma_A + \sigma_B}{2}. \tag{4.2}$$

Harmonic potentials are used to model bond stretching between pairs of bonded simulations elements. Harmonic angle potentials are used to model bending among triplets of bonded PES (type C) simulation elements, but no angle potentials are used for epoxy-amine triplets. No dihedral or improper constraints are implemented here.

Table 4.1 Interaction strengths (ε_{ij}) determined by cohesive energy calculations.

	(A) 44DDS	**(B) DGEBA**	**(C) PES**
(A) 44DDS	0.9216	0.9600	0.9026
(B) DGEBA		1.0000	0.9402
(C) PES			0.8840

Bond formation between amine and epoxy simulation elements is modeled through the stochastic creation of harmonic bonds between A and B beads that are sufficiently close by an activated process with probability of bond formation

$$p = e^{-\frac{E_a \Upsilon}{k_B T}}, \tag{4.3}$$

where E_a is activation energy and bond-order factor $\Upsilon = 1.0$ if the bond being proposed is the first bond to form for either bead and $\Upsilon = 1.2$ otherwise.

By design, the energy scale for modeling pairwise interactions is distinct from the energy scale for modeling bond formation, which are both distinct from the energy scale for modeling vitrification. This modeling choice facilitates the empirical bridging of timescales that is the focus of the present work through exploitation of temperature-time superposition[24]. We report dimensionless simulation temperatures $T = \frac{k_B T^K}{\epsilon}$ throughout this work, where k_B is Boltzmann's constant, T^K is temperature in Kelvin, and ϵ is an energy unit for either pairwise interactions, bonding reactions, or vitrification. These energy scales span about three orders of magnitude, with $\epsilon_{pair} = \varepsilon = 2.1 \times 10^{-22}$ J, $\epsilon_{rxn} = 1.78 \times 10^{-19}$ J, and $\epsilon_{vit} = 6.63 \times 10^{-21}$ J. The pairwise energy scale is derived from cohesive energy described in Section 4.4.1, the reaction energy scale is set from experimental measurements of activation energy[48], and the vitrification energy scale is set by equating the dimensionless $T_g^{sim}(\alpha = 1)$ to an experimental measurement of $T_g^{exp}(\alpha = 1) = 480$ K[5].

4.3 Methods

Simulations of curing epoxy thermosets (with and without toughener) are implemented with the open source dynamic bonding plugin "dybond"[49] written for the HOOMD-blue[50] molecular dynamics engine. Data storage, retrieval, and job submission is done with the signac[51,52] framework. System initialization is performed with mBuild[53]. Plots are created using matplotlib[54] and all scripts used for job submission and data analysis are available at this repository[55]. We use the bonding algorithm as outlined in our previous work[42]. Briefly, every τ_B molecular dynamics steps we attempt to form n_B possible bonds where center-to-center distance between an epoxy and amine simulation element is $r \leq 1.0\sigma$ and with probability as in Equation (4.3). Here, $n_B = 0.005n_T$, where n_T is the total number of bonds that can be formed, equal to four times the number of A beads for the stoichiometric mixtures of A and B. Simulation element positions and velocities are integrated forward in time according to Langevin equations of motion with drag coefficient $\gamma = 4.5$ and step size $\delta t = 0.01$. Random initial configurations are used for each independent simulation run. We calculate the toughener (PES-PES, C-C) structure factor $S(q)$ for simulation snapshots using the "diffract" utility described in Reference [56], enabling identification of any periodic domain features that could indicate phase separation. Unless otherwise noted, simulation parameters summarized in Table 4.2 are used throughout.

Table 4.2 Fiducial simulation parameters. Note that in the present CG model, monomer% and volume% are equivalent but are not identical to corresponding experimental fractions.

Parameter	**Value**
Bond equilibrium (A-B,C-C) (r_o)	$1.0\ \sigma$
Bond force constant (A-B,C-C) (k)	$100\ \frac{\epsilon_{pair}}{\sigma^2}$
Angle equilibrium (C-C-C) (θ_0)	$109.5°$
Angle force constant (C-C-C) (k_{angle})	$25\ \frac{\epsilon_{pair}}{\sigma^2}$
Non-bonded interaction cutoff r_{cut}	$2.5\ \sigma$
Number density ($\rho_n = N/V$)	1.0
Activation Energy (E_A)	$3.0\ \epsilon_{rxn}$
Bonding distance maximum	$1.0\ \sigma$
Secondary bond weight (Υ)	1.2
Enthalpy of Reaction (ΔT_{rxn})	0.0
Bond Period (τ_B)	1.0
Maximum attempted bonds (n_b)	$0.005\ n_T$
Langevin drag (γ)	4.5
%monomers 44DDS:DGEBA:PES	20:40:40
Cure temperature (T)	3.0
Step size (δt)	0.01

Glass transition temperatures are calculated directly from coarse-grained simulation volumes as described in Section 4.3.3 of Reference [57]. Briefly, snapshots of simulations that have reached a specified degree of cure α are used to initialize new simulations that are instantaneously quenched across a range of temperatures to identify T_g, below which the self-diffusion coefficient $\mathcal{D}$ vanishes (Figure 4.2).

Diffusion coefficients $\mathcal{D} = \frac{d\text{MSD}}{6dt}$ are measured directly from quenched trajectories, where MSD is the mean-squared displacement averaged over "B" (DGEBA) simulation elements. We employ piecewise regression to identify the discontinuity in $\mathcal{D}(T)$. Calculations of $T_g(\alpha)$ are validated against theory by measuring the R-squared fit of the DiBenedetto equation[58] modifed by Pascault and Williams[31]

$$T_g(\alpha) = \frac{\lambda\alpha(T_{g1} - T_{g0})}{1 - \alpha(1 - \lambda)} + T_{g0}, \tag{4.4}$$

where λ is chemistry specific and represents the non-linear relationship between T_g and degree of cure and varies from 0 to 1[31], T_{g0} is the glass transition temperature at zero percent cure, and T_{g1} is the glass transition temperature at one hundred percent cure ($\alpha = 1$). We set $\lambda = 0.5$ for its quality of fit here, and note it is larger than λ from prior work on 44DDS/DGEBA (0.34[59]—0.38[60]).

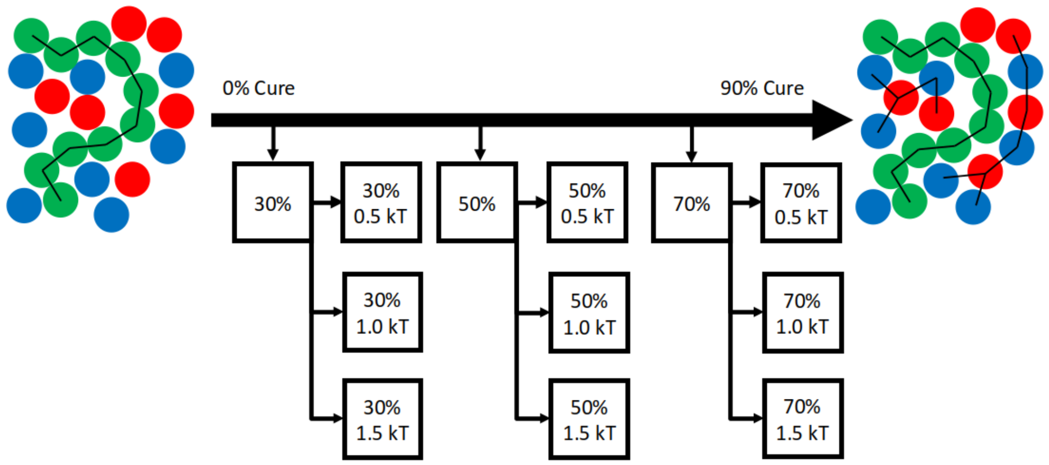


Figure 4.2 T_g prediction workflow: Snapshots at specified α are copied from a curing simulation to initialize instantaneous quenches across candidate low temperatures to identify where the self-diffusion coefficient $\mathcal{D}$ vanishes.

4.4 Results

The 7849 independent MD simulations performed in this work fall into three categories:

1. Setup

2. Validation

3. Exploration

In total, approximately 15,000 GPU-hours of simulation time are performed over about three months. Descriptions of analysis and simulation methods specific to each type of simulation are included in the appropriate subsections that follow.

4.4.1 Setup Simulations

We perform 33 all-atom simulations to determine coarse-grained forcefield parameters, 4480 coarse-grained simulations to calibrate reaction kinetics, and 1448 coarse-grained simulations check for finite size effects before peforming validation and exploration studies.

Forcefield Parameterization

We perform 33 all-atom MD simulations to calculate cohesive energies e_{coh} of amine 44DDS (A), epoxide DGEBA (B), and toughener PES (C) moieties to parameterize their non-bonded interactions of their coarse-grained simulation elements ε_i. In liquids, e_{coh} represents the energy required to separate molecules from the liquid state into isolated molecules in the vapor phase

$$e_{coh} = E_{bulk} - E_{isolated} \tag{4.5}$$

and is calculated from the difference in average molar potential energies E between bulk and isolated molecules[42,61]. Cohesive energies have been used to estimate macroscopic miscibility[62] and parameterize coarse LJ models[61] and we do the same in the present work. We use the OPLS-2005 force field and NPT simulations at $P = 1$ atm, and simulate 11 temperatures equally spaced over $T \in [273, 600]$ K. Each simulation volume is initialized with 500 molecules (monomers of DGEBA and 44DDS, 10-mers of PES) at a density of 1 g/cm^3. After equilibration, densities in agreement with experiments of 0.8–1.14 g/cm^3 (DGEBA), 1.3–1.1 g/cm^3 (44DDS), and 1.3–1.2 g/cm^3 (PES) are observed. Averaging over temperatures, we calculate e_{coh} for DGEBA, 44DDS and PES monomers as 30.36 kcal/mol, 27.98 kcal/mol and 26.84 kcal/mol respectively. We de-demensionalizes pairwise interactions in the coarse-grained models by normalizing by the DGEBA cohesive energy, resulting in the interaction potentials of Table 4.1.

Reaction Kinetics Calibration

Two parameters are tuned to calibrate reaction kinetics: The maximum number of bonds attempted per bonding step n_B and the number of time steps between bonding steps τ_B. Reaction calibration is important for two primary reasons: First, the higher the ratio of n_B/τ_B, the faster simulations can cure to higher α, which saves time. Therefore, the largest n_B/τ_B that replicates experimental reaction dynamics optimizes computational throughput. Second, validating first-order reaction dynamics lays the foundation for exploratory simulations with self-accelerated reactions. We perform 20 independent coarse-grained simulations of 44DDS/DGEBA/PES at each of 224 combinations of (n_B, τ_B, T) to identify the combinations that best fit a first-order reaction model from experimental data[48]. Each simulation has $N = 50000$ (10000 A, 20000 B, and 2000 10-mer chains of C) coarse simulation elements and is cured isothermally at $T \in \{0.2, 0.5, 1.0, 2, 3, 4, 5, 6\}$. Reaction parameters are sampled over the sets $n_B \in \{2.5 \times 10^{-5}, 5 \times 10^{-5}, 1 \times 10^{-4}, 1 \times 10^{-2}\} \times n_T$ (where n_T is the total number of bonds that can be formed, 40,000 here) and $\tau_B \in \{1, 2, 10, 20, 40, 80, 100\}$. We find $n_B = 2.5 \times 10^{-5} n_T = 1.0$ and $\tau_B = 1.0$ here, and use $n_B = 2.5 \times 10^{-5} n_T$ for other system sizes.

Finite Size Effects

Here we investigate the effect of small system sizes on the prediction of glass transition temperatures and morphology.

Glass Transition—Small Systems

We perform curing simulations and $T_g(\alpha)$ calculations of small $N = 500$ volumes and find deviations relative to $N = 50,000$ predictions of $T_g(\alpha)$. For each $N = 500$ and $N = 50,000$, DGEBA/44DD/PES blends are cured isothermally at $T = 3$. Simulation snapshots at intervals $\alpha \in \{0, 0.3, 0.5, 0.7\}$ are used to initialize new trajectories that are quenched to $T = \{0.05, 0.15 \ldots, 2.95, 3.0\}$. Three independent quenches are performed for each of the 60 quench temperatures. T_g calculated from the quenches and the DiBenedetto fits are presented in Figure 4.3.

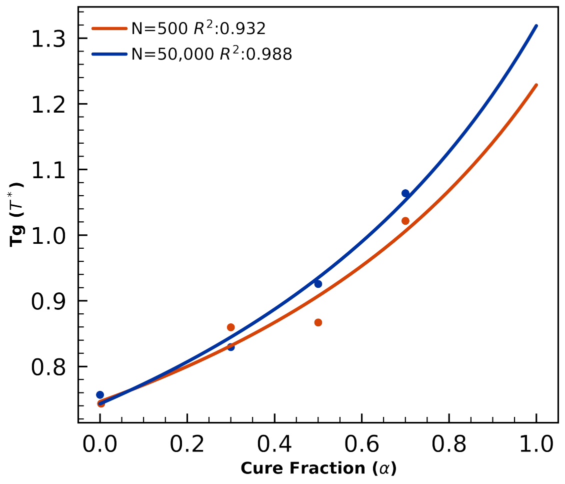


Figure 4.3 $T_g(\alpha)$ calculations and DiBenedetto fits for $N = 500$ (orange) and $N = 50,000$ volumes of coarse-grained 44DDS/DGEBA/PES show the smaller system sizes result in noiser T_g predictions.

While the smaller systems are noisier, the qualitative trend in $T_g(\alpha)$ is not without value, as these predictions can be used for estimates bounds of T_g that will lower the computaitonal cost of measuring the glass transition in larger systems.

Morphology—Small Systems

We next apply our model to study the domain sizes of PES toughener that evolve over the course of curing. We use the PES-PES structure factor to quantify the domain size of the PES toughener. We expect sufficiently large system sizes to demonstrate PES domain sizes independent of simulation volume, but to find volumes below which microphase separation cannot be resolved. Throughout this work we use *microphase separation* and *macrophase separation* to distinguish characteristic length scales of the tougheners: In the case of microphase separation, we measure charasteristic spacings of toughener (with a local peak in the structure factor $S(q)$ that are smaller than half the smallest periodic simulation axis $L_{min}/2$) whether or not they or ordered or disordered. In the case of macrophase separation, divergence of $S(q)$ for $q < 4\pi/L_{min}$ indicates toughener has aggregated into a domain large enough where microphase separation can no longer be resolved.

Three replicates of system sizes with $N \in \{5 \times 10^4, 8 \times 10^4, 1 \times 10^5, 2 \times 10^5, 4 \times 10^5, 6 \times 10^5, 8 \times$ are cured isothermally to 90% with fiducial parameters shown in Table 4.2 and simulations were run for $1 \times 10^7\ \Delta t$. The resulting structure factors $S(q)$ are summarized in Figure 4.4 and local maxima in $S(q)$ (red dots) indicate PES domains with a characteristic spacing of 26 ± 2 nm emerge in $N \geq 2 \times 10^5$ systems.

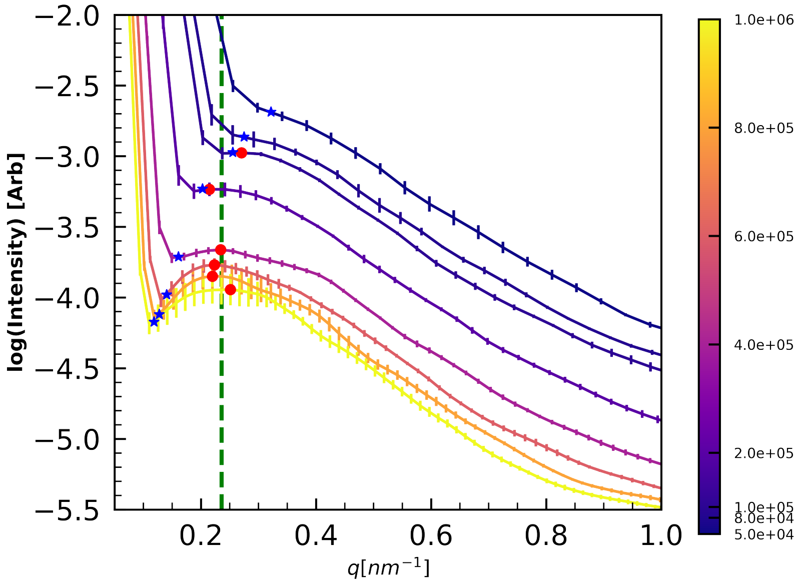


Figure 4.4 PES-PES structure factor in $\alpha = 0.9$ simulations shows emergence of a 0.236 ± 0.019 nm^{-1} (26 ± 2 nm) feature (dashed green line), too large to resolve in simulations where $N \leq 2 \times 10^5$. The color bar indicate system size (N). The blue star indicate half of the box length.

Importantly, cubic simulation volumes below $N = 2 \times 10^5$ are too small to resolve these 26 nm PES features, as the half-box-length (blue stars) for these volumes are smaller than 26 nm (recall conversion factor $l = \frac{2\pi}{q}$ between lengths l and wavenumbers q). Note that in the too-small volumes, no local maxima (red dots) are observed, and $S(q)$ appears to diverge at low q. Therefore, for studies of microphase separation in 44DDS/DGEBA/PES, system sizes of at least $N = 2 \times 10^5$ are necessary. More broadly, microphase separation on length scales larger than half the periodic box length manifest as macrophase separation because local maxima in $S(q)$ cannot be resolved for $q < \frac{\pi}{L}$ for box length L.

4.4.2 Validation Simulations

Validation simulations comprise 1785 coarse-grained MD simulations for calculating gel points, glass transition temperatures, and morphology of toughened 44DDS/DGEBA/PES and untoughened 44DDS/DGEBA blends.

Gel-Point Validation

Isothermal curing simulations of the fiducial $N = 50,000$ toughened 44DDS/DGEBA/PES volumes are performed to predict gelation. The gel-point is dependent on the underlying bonding network that forms as the amine and epoxy react, and is therefore a useful metric for validation in addition to T_g and $S(q)$. We calculate the gel-point by examining at what degree of cure α the molecular weight of the largest and second largest chain diverge. We use the NetworkX[63] python package to measure the size of molecules as curing proceeds.

We sample 26 independent isothermally cured ($T = 3$), toughened volumes spanning cure fractions from $\alpha = 0\%$ to $\alpha = 92.4\%$ and find the gel-point measured by molecular mass at $\alpha_{gel} = 60\%$ (Figure 4.5, in good agreement with theory and experiments. Flory-Stockmayer theory of gelation[64,65] predicts that gelation of 44DDS/DGEBA (a bifunctional monomer and a tetrafunctional monomer) at $\alpha_{gel} = 58\%$[66]. Flory-Stockmayer theory is known to underpredict the cure fraction at gelation, as steric hindrance prevents functional groups reacting with equal probability[67]. Experiments of 44DDS/DGEBA curing measure $\alpha_{gel} > 50\%$[68] and $\alpha_{gel} = 60\%$[69].

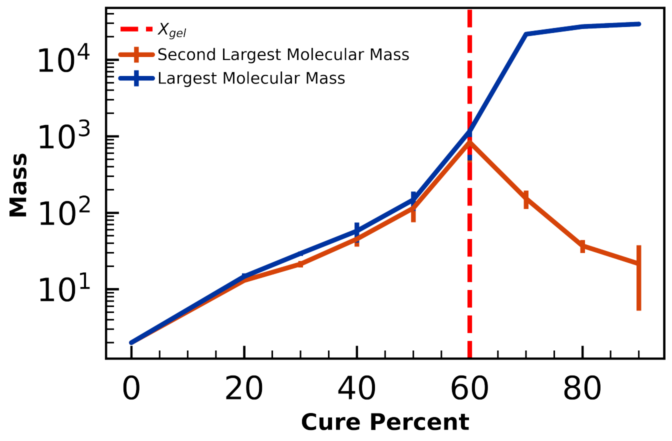


Figure 4.5 Divergence of the largest (blue) and second-largest (orange) molecular mass indicates gelation, here calculated at $\alpha = 60\%$, in agreement with theory (58%) and experiments (60%). Error bars denote standard deviations of 3 independent samples, except the 90% cure case, which have 2 samples.

Glass Transition Validation

A total of 1770 coarse-grained MD simulations are performed to validate predicted $T_g(\alpha)$ against experimental data and theoretical fit to the DiBenedetto equation. First, three independent isothermal curing simulations are performed for $N = 50,000$ systems at the fiducial simulation paramaters. Independent snapshots from $\alpha = 0$ to $\alpha = 0.9$ at intervals of $d\alpha = 0.1$ are taken from each curing simulation to initialize independent quenches (Figure 4.2). These 30 independent snapshots representing the full range of cure fractions are each quenched in independent simulations to each of the 40 dimensionless temperatures from 0.05 to 2.0 at intervals of $dT = 0.05$, plus each of the 15 temperatures from 2.1 to 3.5 in intervals of $dT = 0.1$, plus $T \in \{3.6, 4.0, 4.5, 5.0\}$. From these simulations we focus on $\alpha \in \{0, 0.3, 0.5, 0.7\}$ for determining fits to the DiBenedetto equation, and temperatures $0.1 < T_{quench} < 2.5$ for identifying glass transition temperatures.

We use piecewise regression to identify T_g from diffusivity measurements from each of the aforementioned simulations (Figure 4.6a), and fit with the DiBenedetto equation (Figure 4.6b).

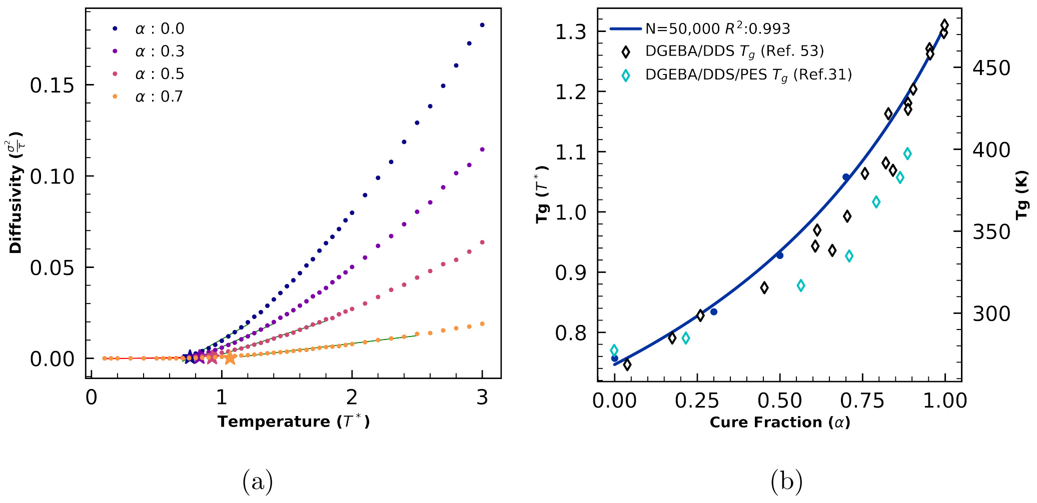
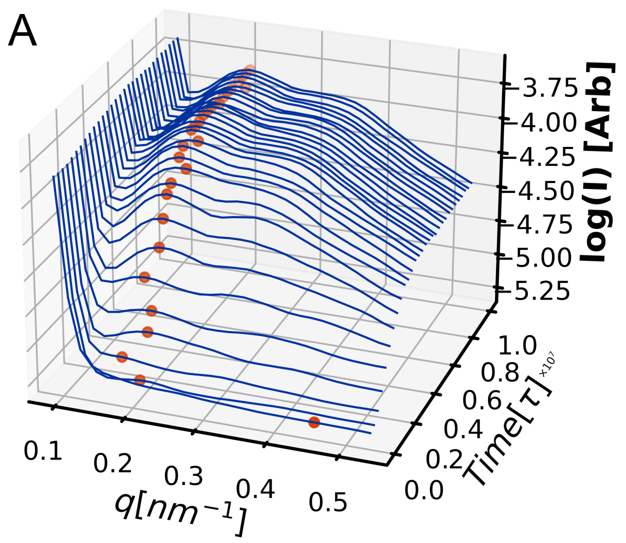


Figure 4.6 (a) Diffusivities measured from quenches of 44DDS/DGEBA/PES as a function of cure fraction and temperature. Green lines indicate linear fits of mid-T diffusivities used to calculate T_g, which are indicated by stars. (b) $T_g(\alpha)$ (blue symbols) and the DiBenedetto fit (blue curve) from (a). The simulated T_g at low and high cure fractions shows close agreement with T_g values measured from an experimental 44DDS/DGEBA system[70] (open black diamonds) and 44DDS/DGE-BA/PES[5] (open cyan diamonds).

We validate against experiments of 44DDS/DGEBA by setting the extrapolated dimensionless value of $T_g(\alpha = 1) = 1.32$ equal the experimental measurement 480 K and then checking intermediate $\alpha = 0.4$ predictions. Here, our predicted $T_g(\alpha = 0.4) = 320$ K is 6.7% higher than the experimental interpolation of 300 K for PES-toughened 44DDS/DGEBA[5], and 6.5% higher than the experimental interpolation of 310 K for the untoughened system[70] (Figure 4.6b). Several other untoughened epoxy systems which have a similar epoxy/amine chemistry also shows a similar trend in the DiBenedetto equation where the $T_g(\alpha = 0.4) \approx 300\ K$[66,70,71]. It is also known from experiments that the uncured 44DDS/DGEBA/PES system is completely miscible and flows at room temperature. Both conditions ($T_g(\alpha = 0) < 293$ K, and $T_g(\alpha = 0.4) \approx 300$ K) are satisfied by the current model.

Morphology Validation

To validate predictions of microphase separated morphology we first perform 3 independent curing simulations at $T = 3$ of the fiducial simulations (Table 4.2) at each of 5 system sizes ($N = \{4 \times 10^5, 6 \times 10^5, 8 \times 10^5, 1 \times 10^6\}$). These sizes are chosen because $N = 4 \times 10^5$ corresponds to cubic simulation volumes with side length $L = 74$ nm, far larger than needed to measure 26 nm periodic features with Fourier-based $S(q)$ analysis (see Section 4.4.1). As in the simulations for understanding minimum simulation sizes, we measure the structure factor $S(q)$–specifically the wave number of any local maxima—to quantify microphase separation and when systems reach steady states. A representative time evolution of $S(q)$ is shown in Figure 4.7A for an $N = 1 \times 10^6$ system, which reaches steady state after 7×10^6 steps.



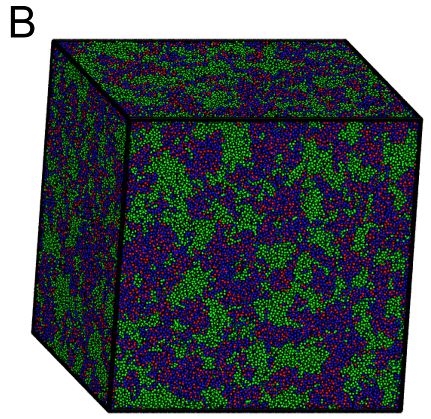


Figure 4.7 (**A**) Structure factor evolution of PES correlations for $N = 1 \times 10^6$ is used to quantify equilibration. Red symbols indicate the wavenumber q_{max} of a local maximum in $S(q)$. (**B**) representative $N = 1 \times 10^6$ morphology after achieving steady state.

Figure 4.7B shows a representative $N = 1 \times 10^6$ morphology after achieving steady state. The average PES-PES $S(q)$ measured for fiducial systems with $N \geq 4 \times 10^5$ has a local maximum at $q_{max} = 0.235 \pm 0.020$ nm^{-1}, corresponding to feature spacings of 26.6 ± 2.5 nm.

In experiments by Rosetti et al.[7], chemically similar DGEBF/44DDS toughened with PES is observed to undergo increasing reaction-induced phase separation that increases with increasing cure temperature. Nonfunctional PES, most similar to the system studied here, remains mixed at a cure temperature of 363 K, phease separates into 250 nm domains when cured at 403 K, and 400 nm domains when cured at 423 K. The length scales of nonfunctional PES phase separation we predict here are smaller than those reported in Reference[7], but we observe the same qualitative trend of larger domain sizes with higher cure temperatures in the cure-path-dependent simulations forthcoming in Section 4.4.3. Phenoxy-functionalized PES, which can participate in crosslinking, is observed by Rosetti et al. that smaller PES nodular domains phase separate (40 nm at 4033 K and 150 nm at 423 K). Smaller PES-rich domains are observed in experiments with a tri-functional epoxy, 44DDS, and functionalized PES, around 20 nm[6]. To fully resolve phase separation of 250 nm domains, (500 nm)^3 simulation volumes are needed, a factor of 5 larger than the largest volumes cured here. In summary, the simulations presented here demonstrate toughener phase separation on length scales smaller than similar-but-not-equivalent experiments, and $N = 1 \times 10^6$ systems corresponding to (100 nm)^3 volumes can routinely be cured to $\alpha = 0.9$ in one week.

4.4.3 Exploration Simulations

Exploration simulations are performed to measure the effect of including reaction enthalpy (80 simulations) and the dependence of cure profile on final morphologies (23 simulations).

Enthalpy Experiment

With temperature-dependent reaction rates in the present model, we perform non-isothermal reaction simulations of otherwise fiducial systems to investigate what models of reaction enthalpy are sufficient for modeling self-accelerated first-order reaction kinetics. In the present case we assume the change in energy associated with the crosslinking reaction is instantaneously distributed among all simulation degrees of freedom, corresponding to an increase in temperature where $\Delta H_{rxn} = C_v \Delta T_{rxn}$ for heat capacity C_v in the NVT ensembles studied here. We perform simulations with per-bond $\Delta T_{rxn} = 0.0, 1 \times 10^{-6}, 1 \times 10^{-5}, 1 \times 10^{-4}$ in addition to the same n_B and τ_B ranges described in Section 4.4.3.

Results summarized in Figure 4.8 validate first-order reaction kinetics are accurately modeled when $\Delta T \leq 1 \times 10^{-6}$, and that $\Delta T = 1 \times 10^{-4}$ is sufficiently large for self-accelerated first-order kinetics to always beat first-order kinetic fits to concentration profiles. Unlike the isothermal simulation cases where $\Delta T = 0$ and reaction kinetics become more accurate as A is decreased, in the self-accelerated first-order kinetic models there exist optimal $A \approx 1$.

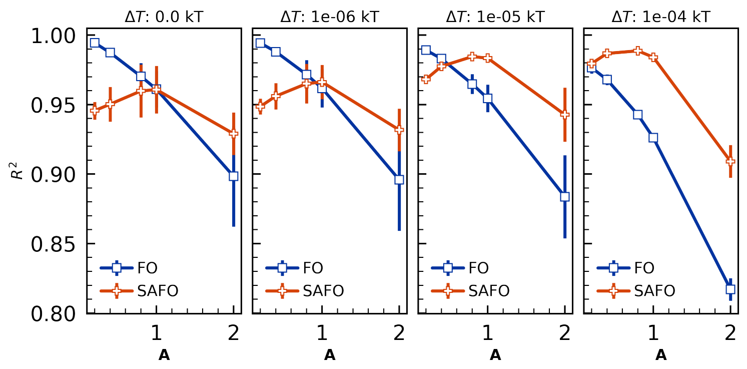


Figure 4.8 Quality of fit for first-order (FO) and self-accelerated first-order (SAFO) reaction models as a function of ΔT_{rxn} and $A = \frac{n_B}{\tau_B}$ validate FO kinetics are most accurate for $\Delta T = 0$, and that SAFO kinetics best fit the concentration profiles when $\Delta T = 1e - 4$. Error bars show standard error in R^2 value averaged across cure temperatures $T = 0.5, 1.0, 2.0, 4.0, 6.0$ kT

In sum, the present model permits straightforward modeling of self-accelerated reactions through the inclusion of a per-bond change in temperature that is validated against kinetic models.

Sensitivity to Cure Profile

The final studies in this work investigatethe dependence on structure of nonisothermal cure profiles meant to be representative of industrial temperature schedules. We first perform 17 simulations of otherwise fiducial $N = 5 \times 10^4$ volumes that step up from $T = 2.0$ to $T = 3.5$ instantaneously at time t_1 ranging between 1.5×10^4 steps and 4×10^6 steps. We next perform 3 replicate simulations of $N = 4 \times 10^5$ volumes that each experience two changes in temperature: From $T_1 = 1.0$ up to $T_2 = 2.0$ at $t_1 = 1 \times 10^5$ steps, followed by a quench down to $T_3 = 1.2$ at either $t_2 = 2 \times 10^6$ steps or $t_2 = 9.5 \times 10^6$ steps. Except for the instantaneous temperature changes described above, the simulations performed in this section are all isothermal. We calculate the time of gelation and $S(q)$ to quantify structure.

Results from the temperature steps from $T = 2$ to $T = 3.5$ are summarized in Figure 4.9, and demonstrate that gelation before 1e6 steps have elapsed is independent of initial time when $t_1 < 2 \times 10^5$. Inset in Figure 4.9b are the cure profiles on semilog axes with open squares indicating gelation times, which are summarized in the main plot.

The delay in gelation with longer times at low T is expected because the more time spent at higher temperature, the faster curing occurs, and the faster gelation will occur. Bicontinuous microphase separated morphologies are observed for all simulations here, but no measurable differences in periodic length scales are observed. These results demonstrate that modifying the cure profile enables control over how quickly systems gel.

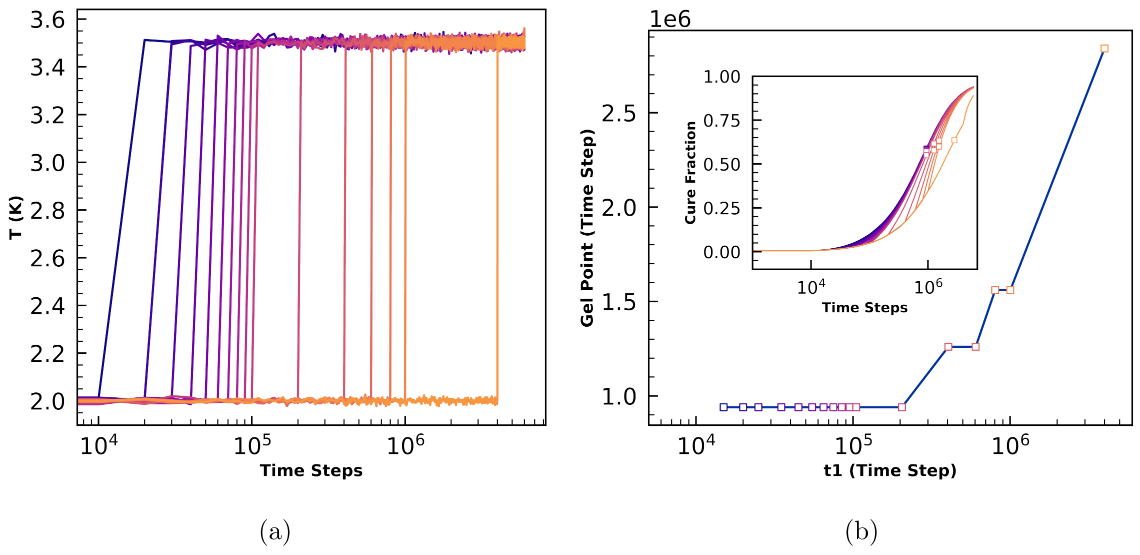


Figure 4.9 (**a**) Temperature profiles where the initial ramp up time (t_1) is varied.(**b**) Time to gelation is not affected by $t_1 < 2 \times 10^5\,\Delta t$. t_1 time denotes the time at which the cure temperature is ramped up and held constant. Inset in (**b** are the cure profiles on semilog axes with open squares indicating gelation times.

The final 6 simulations of $N = 4 \times 10^5$ volumes are cured isothermally at $T_1 = 1$ for 1×10^5 steps before being instantaneously heated to $T_2 = 2$. Three simulations are quenched to $T_3 = 1.2$ before gelation at $t_2 = 2 \times 10^6$ steps, and held there until a total of 3×10^7 steps have elapsed. The other three simulations are quenched to $T_3 = 1.2$ after gelation at $t_2 = 9.5 \times 10^6$ steps, and held there until a total of 1×10^7 steps have elapsed. Note that $T_g(\alpha = 0.87) = 1.2$, so systems with $\alpha < 0.87$ will be above the glass transition temperature at all points during these cure profiles. Temperature schedules, gel points, and cure profiles for these pre- and post-gelation quenches are summarized in Figure 4.10.

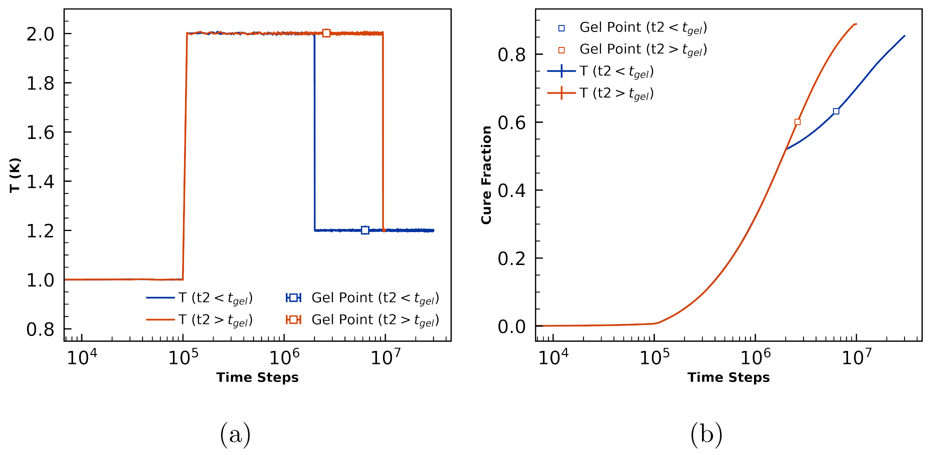


Figure 4.10 Temperatures profiles (**a**) and curing profiles (**b**) for $t_2 < t_{gel}$ ($t_2 = 2 \times 10^6$ Δt) and $t_2 > t_{gel}$ ($t_2 = 9.5 \times 10^6$ Δt). The hollow squares show gel point. T2 is chosen to be higher than and T3 is chosen to be slightly lower than the T_g of the fully cured system ($T_g(\alpha = 1.0) = 480$ K).

The temperature set points correspond to $T_1 = 365$ K, $T_2 = 730$ K, and $T_3 = 438$ K. T2 is chosen such that it is much higher than $T_g(\alpha = 1.0) = 480$ K, facilitating diffusion especially before gelation. We analyze morphologies with final cure fraction $\alpha = 0.855$ for both pre-gelation (blue data) and post-gelation (orange) quenches, neither of which is ever below its glass transition temperature.

Average $S(q)$ for the pre- and post-gelation cures are shown in Figure 4.11.

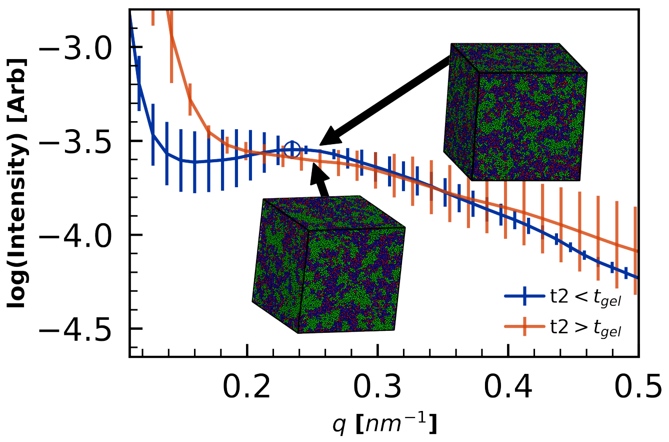


Figure 4.11 PES-PES structure factor shows difference in morphology as a result of varying t$_2$ of the "Step" curing profile. Both simulation volumes are cured to $\alpha = 0.855$. Error bars represent standard error from the three replicate simulations. The length scales of microphase separation are much smaller in the pre-gelation quench (blue), whereas $S(q)$ diverges around $q_{L/2} = 0.17$ nm^{-1}, indicating a higher degree of phase separation that is apparent in the more distinct clumping of the inset visualizations.

Two features of the $S(q)$ stand out—first, the length-scales of phase separation are smaller for the pre-gelation quench. Second, there is higher variance in the measured $S(q)$ in the pre-gelation quenches.

The observations of increased phase separation in the post-gelation quench are consistent with experiments demonstrating increased phase separation with higher cure temperatures[5,70]. These observations are also consistent with two different mechanistic explanations: (1) Higher temperatures increases curing rates, which increase reaction-induced phase separation, and (2) Quenching pre-gelation keeps the morphology from being kinetically arrested, and so the tougheners can more easily mix and distribute in the unvitrified volume if thermodynamically favorable. These results demonstrate that thermoset volumes with identical cure fractions can have significant cure-path-dependent microstructures.

4.5 Conclusions and Outlook

We demonstrate a coarse-grained model of toughened epoxy thermosets that

1. Offers straightforward forcefield parameterization.

2. Can capture first-order and self-accelerated first order reaction dynamics.

3. Is validated against experimental gel points, glass transition temperatures, and morphology for 44DDS/DGEBA/PES blends.

4. Does not require backmapping for T_g calculation.

5. Can cure million-particle volumes (corresponding to 31-million atoms and (100 nm)^3 periodic boxes) to $\alpha = 0.9$ in under one week.

6. Demonstrates for the first time sensitivity of morphology to cure profile.

To summarize, the present work represents progress towards efficient prediction of the morphology and properties of realistic toughened thermosets and provides template workflows for calibrating models to specific formulations and cure profiles. These functionalities offer opportunity to develop a deeper understanding of aerospace-grade thermosets and more reliable manufacturing processes. As an example, datasets generated here lay the foundation to answer questions about how the degree of phase separation contribute to changes of T_g and gelation, which should find applicability beyond the single formulation studied here.

The main shortcomings of this work are the degree of validation against experimental T_g and morphology. While the low and high cure fractions matched experimental glass transition temperatures for 44DDS/DGEBA, the curvature of our DiBenedetto fit was smaller than observed in experiments. We expect subsequent work in improved

T_g detection from diffusivity data, calculation of T_g from back-mapped morphologies to provide better predictions of T_g across the full spectrum of cure fractions. While we recognize experiments characterizing toughener phase separation on the 10 nm–50 nm length scales are challenging, additional work in this area would provide key datasets to validate against. Alternatively, applying the workflows presented here to thermoset formulations with small-scale phase separation characterized would be a information-rich extension of this work. Finally, this work sets the stage for investigations that simultaneously calibrate the energy scales of monomer interactions, reaction kinetics, vitrification to experimental curing profiles that measure the degree to which hour-long curing profiles can accurately be predicted by a few billion steps of a coarse-grained model.

REFERENCES

[1] George Marsh. Reclaiming value from post-use carbon composite. *Reinforced Plastics*, 52(7):36–39, 2008.

[2] J. Zhang, Y. C. Xu, and P. Huang. Effect of cure cycle on curing process and hardness for epoxy resin. *Express Polymer Letters*, 3(9):534–541, 2009.

[3] J W Sinclair. Effects of Cure Temperature on Epoxy Resin Properties. *The Journal of Adhesion*, 38(3-4):219–234, jul 1992.

[4] Fabrice Lapique and Keith Redford. Curing effects on viscosity and mechanical properties of a commercial epoxy resin adhesive. *International Journal of Adhesion and Adhesives*, 22(4):337–346, jan 2002.

[5] W Jenninger, J. E.K. Schawe, and I Alig. Calorimetric studies of isothermal curing of phase separating epoxy networks. *Polymer*, 41(4):1577–1588, 2000.

[6] Bong Sup Kim, Tsuneo Chiba, and Takashi Inoue. Morphology development via reaction-induced phase separation in epoxy/poly(ether sulfone) blends: morphology control using poly(ether sulfone) with functional end-groups. *Polymer*, 36(1):43–47, jan 1995.

[7] Yann Rosetti, Pierre Alcouffe, Jean-Pierre Pascault, Jean-François Gérard, and Frédéric Lortie. Polyether Sulfone-Based Epoxy Toughening: From Micro- to Nano-Phase Separation via PES End-Chain Modification and Process Engineering. *Materials*, 11(10):1960, oct 2018.

[8] Tae Ho Yoon, Duane B. Priddy, Gregory D. Lyle, and James E. McGrath. Mechanical and morphological investigations of reactive polysulfone toughened epoxy networks. *Macromolecular Symposia*, 98(1):673–686, jul 1995.

[9] Roberto J. J. Williams, Boris A. Rozenberg, and Jean-Pierre Pascault. Reaction-induced phase separation in modified thermosetting polymers. pages 95–156. 1997.

[10] Jacques N. Sultan and Frederick J. McGarry. Effect of rubber particle size on deformation mechanisms in glassy epoxy. *Polymer Engineering and Science*, 13(1):29–34, jan 1973.

[11] Alexander J. MacKinnon, Stephen D. Jenkins, Patrick T. McGrail, and Richard A. Pethrick. A dielectric, mechanical, rheological and electron microscopy study of cure and properties of a thermoplastic-modified epoxy resin. *Macromolecules*, 25(13):3492–3499, jun 1992.

[12] Alexander J. MacKinnon, Stephen D. Jenkins, Patrick T. McGrail, and Richard A. Pethrick. Dielectric, mechanical and rheological studies of phase separation and cure of a thermoplastic modified epoxy resin: incorporation of reactively terminated polysulfones. *Polymer*, 34(15):3252–3263, jan 1993.

[13] A. J. Kinloch, M. L. Yuen, and S. D. Jenkins. Thermoplastic-toughened epoxy polymers. *Journal of Materials Science*, 29(14):3781–3790, jul 1994.

[14] Ping Huang, Sixun Zheng, Jinyu Huang, Qipeng Guo, and Wei Zhu. Miscibility and mechanical properties of epoxy resin/polysulfone blends. *Polymer*, 38(22):5565–5571, oct 1997.

[15] Shi Ru Jong and Tzyy Lung Yu. Physical aging of epoxy resin blended with a medium molecular weight poly(ether sulfone), 1999.

[16] R J Varley, J H Hodgkin, D G Hawthorne, G P Simon, and D McCulloch. Toughening of a trifunctional epoxy system Part {III}. Kinetic and morphological study of the thermoplastic modified cure process. *Polymer*, 41(9):3425–3436, apr 2000.

[17] J H Hodgkin, G P Simon, and R J Varley. Thermoplastic toughening of epoxy resins: a critical review. *Polymers for Advanced Technologies*, 9(1):3–10, jan 1998.

[18] Lorena Ruiz-Pérez, Gareth J. Royston, J. Patrick A. Fairclough, and Anthony J. Ryan. Toughening by nanostructure. *Polymer*, 49(21):4475–4488, oct 2008.

[19] Jia (Daniel) Liu, Zachary J. Thompson, Hung-Jue Sue, Frank S. Bates, Marc A. Hillmyer, Marv Dettloff, George Jacob, Nikhil Verghese, and Ha Pham. Toughening of Epoxies with Block Copolymer Micelles of Wormlike Morphology. *Macromolecules*, 43(17):7238–7243, sep 2010.

[20] Miren Blanco, Marta López, Galder Kortaberria, and Iñaki Mondragon. Nanostructured thermosets from self-assembled amphiphilic block copolymer/epoxy resin mixtures: effect of copolymer content on nanostructures. *Polymer International*, 59(4):523–528, apr 2010.

[21] Marc A. Hillmyer, Paul M. Lipic, Damian A. Hajduk, Kristoffer Almdal, and Frank S. Bates. Self-Assembly and Polymerization of Epoxy Resin-Amphiphilic Block Copolymer Nanocomposites. *Journal of the American Chemical Society*, 119(11):2749–2750, mar 1997.

[22] Sajeev Martin George, Debora Puglia, Jose M. Kenny, Jyotishkumar Parameswaranpillai, and Sabu Thomas. Reaction-induced phase separation and thermomechanical properties in epoxidized Styrene- block -butadiene- block -styrene triblock copolymer modified epoxy/DDM system. *Industrial and Engineering Chemistry Research*, 53(17):6941–6950, 2014.

[23] Chunyu Li and Alejandro Strachan. Molecular scale simulations on thermoset polymers: A review. *Journal of Polymer Science Part B: Polymer Physics*, 53(2):103–122, jan 2015.

[24] André Lee and Gregory B McKenna. Effect of crosslink density on physical ageing of epoxy networks. *Polymer*, 29(10):1812–1817, 1988.

[25] G. Rajagopalan, J. W. Gillespie, and S. H. McKnight. Diffusion of reacting epoxy and amine monomers in polysulfone: A diffusivity model. *Polymer*, 41(21):7723–7733, 2000.

[26] J. P. Pascault and R. J. J. Williams. Glass transition temperature versus conversion relationships for thermosetting polymers. *Journal of Polymer Science Part B: Polymer Physics*, 28(1):85–95, jan 1990.

[27] A. T. DiBenedetto. Prediction of the glass transition temperature of polymers: A model based on the principle of corresponding states. *Journal of Polymer Science Part B: Polymer Physics*, 25(9):1949–1969, 1987.

[28] John B Enns and John K Gillham. Time–temperature–transformation (TTT) cure diagram: Modeling the cure behavior of thermosets. *Journal of Applied Polymer Science*, 28(8):2567–2591, 1983.

[29] Sue Ann Bidstrup, Norman F. Sheppard, and Stephen D. Senturia. Dielectric analysis of the cure of thermosetting epoxy/amine systems. *Polymer Engineering and Science*, 29(5):325–328, mar 1989.

[30] Olivier Georjon, Jocelyne Galy, and Jean-Pierre Pascault. Isothermal curing of an uncatalyzed dicyanate ester monomer: Kinetics and modeling. *Journal of Applied Polymer Science*, 49(8):1441–1452, aug 1993.

[31] Catherine Jordan, Jocelyne Galy, and Jean-Pierre Pascault. Measurement of the extent of reaction of an epoxy–cycloaliphatic amine system and influence of the extent of reaction on its dynamic and static mechanical properties. *Journal of Applied Polymer Science*, 46(5):859–871, oct 1992.

[32] Chunyu Li, Grigori A. Medvedev, Eun Woong Lee, Jaewoo Kim, James M. Caruthers, and Alejandro Strachan. Molecular dynamics simulations and experimental studies of the thermomechanical response of an epoxy thermoset polymer. *Polymer (United Kingdom)*, 53(19):4222–4230, 2012.

[33] Chunyu Li, Eric Coons, and Alejandro Strachan. Material property prediction of thermoset polymers by molecular dynamics simulations. *Acta Mechanica*, 225:1187–1196, 2014.

[34] Lauren J. Abbott, Justin E. Hughes, and Coray M. Colina. Virtual synthesis of thermally cross-linked copolymers from a novel implementation of polymatic. *Journal of Physical Chemistry B*, 118(7):1916–1924, 2014.

[35] Yasuyuki Shudo, Atsushi Izumi, Katsumi Hagita, Toshio Nakao, and Mitsuhiro Shibayama. Large-scale molecular dynamics simulation of crosslinked phenolic resins using pseudo-reaction model. *Polymer (United Kingdom)*, 103:261–276, 2016.

[36] Ketan S. Khare and Frederick R. Phelan. Quantitative Comparison of Atomistic Simulations with Experiment for a Cross-Linked Epoxy: A Specific Volume-Cooling Rate Analysis. *Macromolecules*, 51(2):564–575, 2018.

[37] A Prasad, Tarun Grover, and Sumit Basu. Coarse – grained molecular dynamics simulation of cross – linking of DGEBA epoxy resin and estimation of the

adhesive strength. *International Journal of Engineering, Science and Technology*, 2(4):17–30, sep 2010.

[38] Hong Liu, Min Li, Zhong-Yuan Lu, Zuo-Guang Zhang, Chia-Chung Sun, and Tian Cui. Multiscale Simulation Study on the Curing Reaction and the Network Structure in a Typical Epoxy System. *Macromolecules*, 44(21):8650–8660, nov 2011.

[39] Michael Langeloth, Taisuke Sugii, Michael C. Böhm, and Florian Müller-Plathe. The glass transition in cured epoxy thermosets: A comparative molecular dynamics study in coarse-grained and atomistic resolution. *The Journal of Chemical Physics*, 143(24):243158, 2015.

[40] Gokhan Kacar, Elias A J F Peters, and Gijsbertus De With. Multi-scale simulations for predicting material properties of a cross-linked polymer. *Computational Materials Science*, 102:68–77, 2015.

[41] Amulya K. Pervaje, Joseph C. Tilly, Andrew T. Detwiler, Richard J. Spontak, Saad A. Khan, and Erik E. Santiso. Molecular Simulations of Thermoset Polymers Implementing Theoretical Kinetics with Top-Down Coarse-Grained Models. *Macromolecules*, 53(7):2310–2322, 2020.

[42] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, may 2018.

[43] Jacob R Gissinger, Benjamin D Jensen, and Kristopher E Wise. Modeling chemical reactions in classical molecular dynamics simulations. *Polymer*, 128:211–217, 2017.

[44] Pavel V Komarov, Chiu Yu-tsung, Chen Shih-ming, Pavel G Khalatur, and Peter Reineker. Highly Cross-Linked Epoxy Resins: An Atomistic Molecular Dynamics Simulation Combined with a Mapping/Reverse Mapping Procedure. *Macromolecules*, 40(22):8104–8113, 2007.

[45] Mar\'\ia A Pérez-Maciá, David Curcó, Roger Bringué, Montserrat Iborra, and Carlos Alemán. Atomistic simulations of the structure of highly crosslinked

sulfonated poly (styrene-co-divinylbenzene) ion exchange resins. *Soft matter*, 11(11):2251–2267, 2015.

[46] Zhan Liu, Junhui Li, Can Zhou, and Wenhui Zhu. A molecular dynamics study on thermal and rheological properties of BNNS-epoxy nanocomposites. *International Journal of Heat and Mass Transfer*, 126:353–362, 2018.

[47] Arun Srikanth, Emre Kinaci, John Vergara, Giuseppe Palmese, and Cameron F Abrams. The effect of alkyl chain length on mechanical properties of fatty-acid-functionalized amidoamine-epoxy systems. *Computational Materials Science*, 150:70–76, 2018.

[48] Michael Aldridge, Alan Wineman, Anthony Waas, and John Kieffer. In situ analysis of the relationship between cure kinetics and the mechanical modulus of an epoxy resin. *Macromolecules*, 47(23):8368–8376, 2014.

[49] Stephen Thomas and Mike Henry. dybond_plugin, oct 2017.

[50] Joshua A. Anderson, Jens Glaser, and Sharon C. Glotzer. HOOMD-blue: A Python package for high-performance molecular dynamics and hard particle Monte Carlo simulations. *Computational Materials Science*, 173(October 2019):109363, 2020.

[51] Carl Simon Adorf, Vyas Ramasubramani, Bradley D Dice, Michael M Henry, Paul M Dodd, and Sharon C Glotzer. glotzerlab/signac, feb 2019.

[52] Carl S. Adorf, Paul M. Dodd, Vyas Ramasubramani, and Sharon C. Glotzer. Simple data and workflow management with the signac framework. *Computational Materials Science*, 146(C):220–229, 2018.

[53] Christoph Klein, János Sallai, Trevor J Jones, Christopher R Iacovella, Clare McCabe, and Peter T Cummings. A Hierarchical, Component Based Approach to Screening Properties of Soft Matter. *Foundations of Molecular Modeling and Simulation*, 2016.

[54] J D Hunter. Matplotlib: A 2D graphics environment. *Computing In Science & Engineering*, 9(3):90–95, 2007.

[55] Stephen Thomas and Mike Henry. notebooks, sep 2020.

[56] Matthew Lewis Jones and Eric Jankowski. Computationally connecting organic photovoltaic performance to atomistic arrangements and bulk morphology. *Molecular Simulation*, 43(10-11):1–18, mar 2017.

[57] Stephen Thomas. *New Methods for Understanding and Controlling the Self-Assembly of Reacting Systems Using Coarse-Grained Molecular Dynamics*. PhD thesis, Boise State University, 2018.

[58] Lawrence E Nielsen. Cross-Linking–Effect on Physical Properties of Polymers. *Journal of Macromolecular Science, Part C*, 3(1):69–103, jan 1969.

[59] Jean Pascal Eloundou, Ohandja Ayina, Hippolyte Ntede Nga, Jean François Gerard, Jean Pierre Pascault, Gisèle Boiteux, and Gérard Seytre. Simultaneous kinetic and microdielectric studies of some epoxy-amine systems. *Journal of Polymer Science, Part B: Polymer Physics*, 36(16):2911–2921, 1998.

[60] J. P. Pascault and R. J.J. Williams. Relationships between glass transition temperature and conversion - Analyses of limiting cases. *Polymer Bulletin*, 24(1):115–121, 1990.

[61] Wen Sheng Xu, Jack F. Douglas, and Karl F. Freed. Influence of Cohesive Energy on the Thermodynamic Properties of a Model Glass-Forming Polymer Melt. *Macromolecules*, 49(21):8341–8354, 2016.

[62] Chunyu Li and Alejandro Strachan. Cohesive energy density and solubility parameter evolution during the curing of thermoset. *Polymer (United Kingdom)*, 135:162–170, 2018.

[63] Aric a. Hagberg, Daniel a. Schult, and Pieter J. Swart. Exploring network structure, dynamics, and function using NetworkX. *Proceedings of the 7th Python in Science Conference (SciPy2008)*, 836:11—-15, 2008.

[64] Paul J. Flory. Molecular Size Distribution in Three Dimensional Polymers. I. Gelation 1. *Journal of the American Chemical Society*, 63(11):3083–3090, nov 1941.

[65] Walter H. Stockmayer. Theory of Molecular Size Distribution and Gel Formation in Branched Polymers II. General Cross Linking. *The Journal of Chemical Physics*, 12(4):125–131, apr 1944.

[66] Monoj Pramanik, Eric W. Fowler, and James W. Rawlins. Another look at epoxy thermosets correlating structure with mechanical properties. *Polymer Engineering and Science*, 54(9):1990–2004, 2014.

[67] Dietrich Stauffer, Antonio Coniglio, and Mireille Adam. Gelation and critical phenomena. In *Polymer Networks*, pages 103–158. Springer Berlin Heidelberg, Berlin, Heidelberg.

[68] S. R. White, P. T. Mather, and M. J. Smith. Characterization of the cure-state of DGEBA-DDS epoxy using ultrasonic, dynamic mechanical, and thermal probes. *Polymer Engineering & Science*, 42(1):51–67, jan 2002.

[69] A. Bonnet, J. P. Pascault, H. Sautereau, M. Taha, and Y. Camberlin. Epoxy-diamine thermoset/thermoplastic blends. 1. Rates of reactions before and after phase separation. *Macromolecules*, 32(25):8517–8523, 1999.

[70] B. G. Min, Z. H. Stachurski, and J. H. Hodgkin. Cure kinetics of elementary reactions of a DGEBA/DDS epoxy resin: 1. Glass transition temperature versus conversion. *Polymer*, 34(23):4908–4912, 1993.

[71] Jürgen E K Schawe. The Interplay between Molecular Dynamics and Reaction Kinetics during Curing Reactions. 100002:1–6, 2017.

CHAPTER 5

CONTRIBUTIONS TO PAPERS

In this appendix I describe my contributions to papers where I am listed as a co-author using CRediT Contributor Roles Taxonomy[1].

In Chapter 3 "Simplified models for accelerated structural prediction of conjugated semiconducting polymers" I contributed to the conceptualization, methodology, software, validation, formal analysis, investigation, data curation, manuscript original draft, manuscript review and editing, and visualization. I conducted all of the simulations and analysis. This included updating our laboratories software from python 2 to python 3, as well updating our software to work with the major API changes from hoomd version 1.x to hoomd version 2.x.

In Chapter 4 "General-purpose coarse-grained toughened thermoset model for 44DDS/DGEBA/PES" I contributed to the conceptualization, methodology, software, validation, formal analysis, investigation, data curation, manuscript original draft, manuscript review and editing, and visualization. I reprocessed the raw simulation data and analyzed simulations performed by another graduate student, as well as performed additional simulations.

In Ref. [2], "Optimization and Validation of Efficient Models for Predicting Polythiophene Self-Assembly" I contributed to the data curation, investigation, software, visualization, and manuscript review and editing. A key aspect of this paper was the development of an efficient computational model for P3HT. In order to benchmark the performance of this model, I created a software image to ensure the same software stack was used on three different HPC clusters. In Figure 2c, I did the benchmarking on different clusters and then collated the results.

In Ref. [3], "Machine learning predictions of electronic couplings for charge transport calculations of P3HT", I contributed to software, validation, and manuscript review and editing. I helped to develop the code and API that extracted features from simulation data. I also helped to test and refine the model on a different chemical system.

In Ref. [4], "Routine million-particle simulations of epoxy curing with dissipative particle dynamics" I contributed to methodology, software, validation, visualization, and manuscript review and editing. I developed the coarse-grained model and developed the algorithm and wrote the pure python implementation of the dybond plugin. I added functionality to mbuild software package used to support million particle packing which was critical to this work.

In Ref. [5], "Application of artificial neural networks to identify equilibration in computer simulations" I contributed to software and manuscript review and editing. I contributed to the autocorrelation code that is in the paper, helped to prepare manuscript for submission.

In Ref. [6], "Perspective on coarse-graining, cognitive load, and materials simulation" I contributed to methodology, software, investigation, data curation, visualization, manuscript original draft, and manuscript review and editing. I contributed mostly to section three "Best (*sic*) practices and cognitive load". This section is a comprehensive literature review of the current best practices utilized in scientific software engineering. These practices are given additional context though the lens of reducing cognitive load. Additionally, figure 4 is taken from my diffraction analysis I did for my BDT-TPD paper (Chapter 3). Figures 7 and 8 are taken from my work on demonstrating the affect of different curing profiles have on thermoset morphology.

REFERENCES

[1] Amy Brand, Liz Allen, Micah Altman, Marjorie Hlava, and Jo Scott. Beyond authorship: attribution, contribution, collaboration, and credit. *Learned Publishing*, 28(2):151–155, apr 2015.

[2] Evan D Miller, Matthew Lewis Jones, Michael M Henry, Paul Chery, Kyle Miller, and Eric Jankowski. Optimization and Validation of Efficient Models for Predicting Polythiophene Self-Assembly. *Polymers*, 10(12):1305, nov 2018.

[3] Evan D Miller, Matthew Lewis Jones, Mike M. Henry, Bryan Stanfill, and Eric Jankowski. Machine learning predictions of electronic couplings for charge transport calculations of P3HT. *AIChE Journal*, 65(12):1–10, dec 2019.

[4] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, may 2018.

[5] Mitchell H Leibowitz, Evan D Miller, Michael M Henry, and Eric Jankowski. Application of artificial neural networks to identify equilibration in computer simulations. *Journal of Physics: Conference Series*, 921(1):012013, nov 2017.

[6] Eric Jankowski, Neale Ellyson, Jenny W Fothergill, Michael M Henry, Mitchell H Leibowitz, Evan D Miller, Mone't Alberts, Samantha Chesser, Jaime D Guevara, Chris D. Jones, Mia Klopfenstein, Kendra K Noneman, Rachel Singleton, Ramon A Uriarte-Mendoza, Stephen Thomas, Carla E Estridge, and Matthew L Jones. Perspective on Coarse-Graining , Cognitive Load , and Materials Simulation. *Computational Materials Science*, 169(109129):109129, jan 2020.

CHAPTER 6

CONCLUSIONS AND SUGGESTIONS FOR FUTURE WORK

6.1 Conclusions

We have studied the self-assembly of organic polymers and toughened thermosets using molecular dynamics and coarse-grained models. We found that using coarse-grained models we were able to perform some of the largest and most structurally accurate simulations for these systems. We developed an united atom coarse-grained model of a BDT-TPD, a promising donor/acceptor polymer, using a forcefield from literature and rigid bodies on conjugated elements. We also developed a general purpose methodology for modeling toughened thermosets which includes using a single simulation element for each amine, epoxy, and toughener-mer, and an open-source dynamic bonding plugin for HOOMD-Blue to model the crosslinking reaction between the amine and epoxy.

With both P3HT (Ref. [1]) and BDT-TPD (Chapter 3), we were able to predict self-assembled structures while simplifying our model by:

1. Modeling hydrogen implicitly
2. Modeling electrostatics implicitly
3. Modeling solvent implicitly

4. Utilizing rigid bodies on conjugated elements

5. Instantaneous quenches to different state points to establish phase boundaries

We demonstrate that even when treating some of those items explicitly, the increased accuracy is not enough to offset the advantages of our simplifications, computational speedups, ease of implementation, and lower cognitive load. However, when we do identify a limitation of our model, we add complexity. We do this in two cases, 1) when exploring phase space in finer detail, simulated solvent evaporation is required to accelerate equilibration and avoid kinetic arrest and 2) we add hydrogens back into our model when performing charge transport calculations as full atomic resolution is required when calculating properties that depend on electron orbitals. Our simulations of semiconducting polymers are some of the largest and most structurally accurate performed. We conclude that coarse-grained models of organic photovoltaic polymers are able to faithfully capture the morphology of a self-assembled bulk heterojunction and coarse-grained models with rigid bodies are required to overcome the long relaxation times associated with polymers.

We developed (Ref. [2]) and refined (Chapter 4) a generalizable coarse-grained model for toughened thermosets. Each amine, epoxy, and toughener mer is represented by a single spherical simulation element (see Figure 6.1).

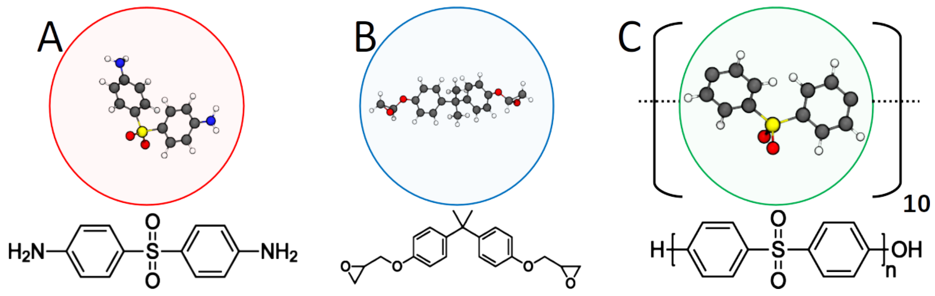


Figure 6.1 Coarse-grained representations of 44DDS (A), DGEBA (B), and PES (C) repeat units. The amines (A) can bond to up to four epoxies (B), which can each bond to up to two amines. All toughener molecules are linear 10-mers of C.

This simplification reduces the amount of work required to parametrize our coarse-grained model. We have only 3 non-bonded parameters, and 3 interatomic parameters (A-B bond, C-C bond, and C-C-C angle) that require parametrization. Our model is further simplified by using a radius of 1 nm and a mass of 1 mass unit for each simulation element. Despite what may appear as an over simplification, our model is validated against experimental gel-point, glass transition temperature, phase behavior, and morphology. The computational speedup associated with these simplifications enable simulations of (100 nm)^3 million-particle volumes that can reach a 90% degree of cure in $\approx$ 1 week. We also demonstrate, for the first time in literature, sensitivity to cure profile. We conclude that we are able to capture important thermodynamic properties and processes (gel-point, glass transition temperature, and reaction induced phase separation) and morphology using a single site coarse-grain model and a dynamic bonding algorithm. We now have the tools to answer the question: "How do changes in cure profile affect structure and properties in toughened thermosets".

We make our models as usable as possible. All of our models, initial configurations, submission scripts, analysis scripts, and simulation data associated with this work with an open source, permissive license. We also follow current scientific software best practices when developing code:

1. Continuous integration

2. Unit testing

3. Code review

4. Automatic code formatting

5. Distributed version control

6. Open source licensing

While employing these practices, we have found that continuous integration, unit testing, code review, and automatic code formatting help to verify and ensure that the code is correct. We conclude that using distributed version control, code review, and open source licensing facilitates open collaboration between research groups. Open collaboration helps to accelerate scientific discovery.

Given the challenges and threat to our species survival from climate change, we need to maximize scientific discovery to develop the next-generation of materials that will enable greater efficiency in organic solar cells and reduce the emissions from air travel. We have outlined the simplified models and the scientific software engineering practices used to create, publish, and distribute these models for utilization by the greater scientific community. We can use these models now to further our understanding of polymer self-assembly and create more efficient solar devices and air travel, reducing greenhouse gas emissions and combating global climate change.

6.2 Suggestions for Future Work

We now have the tools, protocols, and methodology to study the self-assembly of polymers and reacting toughened thermosets. We validated our approach on novel and well studied material systems. All of the input scripts, simulation code, analysis code, and notebooks used to produce figures used in this work have been made available under a permissive open-source license.

The next step for screening candidate OPV morphologies is to continue development of an automated pipeline that will enable high-throughput screening. The methodology and analysis used in the Chapter 3 and Ref. [1] is generally applicable, but the software infrastructure is not amenable to high throughput screening. For example, the code and methodology used to identify clusters in Chapter 3 was written with only the BDT-TPD chemistry in mind and the order parameter ψ used in Ref. [1] was developed to only work with P3HT. A set of non-chemistry specific order parameters (in addition to standard $g(r)$ and $S(q)$ measurements) will be required to facilitate high-throughput screening and identify interesting OPV morphologies. Our new Python software package Planckton [3] has been designed with high throughput screening as the primary objective. Planckton employs the current software engineering best practices outlined in Ref. [4] and automates atom typing, which was a significant bottleneck to exploring new compounds using the "opv_cg" software package that was used in Chapter 3 and Ref. [1]. Using Planckton, it would be interesting to explore how coarse grain of a model could be used that still faithfully replicates experimental morphology.

The weakest aspect of our toughened thermoset work in Ref. [2] and 4 is the glass transition analysis. Our current method for T_g calculation is labor intensive, as we define by hand the vitrified regime and rubbery regime in the plot of the diffusivity as a function of temperature for each cure fraction that we are examining. We are capable of generating more data than we are capable of analyzing. Automating the glass transition calculation is an immediate next step in this research. Another area of exploration would be to compare experimental species concentration as a function of degree of cure. Using the species concentration will enable direct comparison to experimental kinetic measurements and allow direct comparison to experimental timescales and simulation timescales. This approach will require some work with experimental collaborators to generate datasets that are comparable to chemistries that we have simulated. The analysis code for species determination has been written and is a part of the *epoxpy* analysis suite.

REFERENCES

[1] Evan D Miller, Matthew Lewis Jones, Michael M Henry, Paul Chery, Kyle Miller, and Eric Jankowski. Optimization and Validation of Efficient Models for Predicting Polythiophene Self-Assembly. *Polymers*, 10(12):1305, nov 2018.

[2] Stephen Thomas, Monet Alberts, Michael M Henry, Carla E Estridge, and Eric Jankowski. Routine million-particle simulations of epoxy curing with dissipative particle dynamics. *Journal of Theoretical and Computational Chemistry*, 17(03):1840005, may 2018.

[3] Evan D Miller, Michael M Henry, Matthew Lewis Jones, and Eric Jankowski. Planckton, mar 2019.

[4] Eric Jankowski, Neale Ellyson, Jenny W Fothergill, Michael M Henry, Mitchell H Leibowitz, Evan D Miller, Mone't Alberts, Samantha Chesser, Jaime D Guevara, Chris D. Jones, Mia Klopfenstein, Kendra K Noneman, Rachel Singleton, Ramon A Uriarte-Mendoza, Stephen Thomas, Carla E Estridge, and Matthew L Jones. Perspective on Coarse-Graining , Cognitive Load , and Materials Simulation. *Computational Materials Science*, 169(109129):109129, jan 2020.

APPENDIX A

SIMPLIFIED MODELS FOR ACCELERATED STRUCTURAL PREDICTION OF CONJUGATED SEMICONDUCTING POLYMERS — SUPPORTING INFORMATION[1]

[1]This appendix has been published as supporting information for a paper published in *J. Phys. Chem. C* and the paper is referenced as "Henry, M. M., Jones, M. L., Oosterhout, S. D., Braunecker, W. A., Kemper, T. W., Larsen, R. E., … Jankowski, E. (2017). Simplified Models for Accelerated Structural Prediction of Conjugated Semiconducting Polymers. *The Journal of Physical Chemistry C*, 121(47), 26528–26538. https://doi.org/10.1021/acs.jpcc.7b09701"

A.1 Simulation Code

The xml representation of the BDT-TPD oligomers, forcefield parameters, simulation scripts, and source code for generating scattering patterns are available in bitbucket code repositories: `https://bitbucket.org/cmelab/opv\_cg` and `https://bitbucket.org/cmelab/cme\_utils`. The model file with the force field parameters used in this work is located at `mlibs/models/mike_ua/model.xml`, with the model name `ua_e` in commit 8d19157 in the `opv_cg` repository. The rigid model and flexible model topology files are located in the same folder as the force field parameters and are labeled `rbdt-5-scaled.xml` and `bdt.xml` respectively. The diffraction methods are located in `cme_utils/analyze/diffractometer.py` of the `cme_utils` repository.

A.2 Determining Equilibrium

To determine which configurations belong to the equilibrium distribution for each simulation, the following method was used:

First, the time evolution of the non-bonded Lennard-Jones potential (E_{LJ}) was considered for each simulation, and split into 10 bins. For each bin, the standard deviation in E_{LJ} was calculated. Starting from the final bin and working backwards through simulation time, bins were added to the "equilibrated region" if the standard deviation of the bin's potential energy was no more than twice that of the previous bin in the region. Relaxation time is the time it takes for our simulation to reach equilibrium.

Once the equilibrated region was determined, the autocorrelation time was calculated to obtain the number of time steps between statistically independent frames.

The autocorrelation $R_{E_{LJ}}$ was calculated as:

$$R_{E_{LJ}} = \frac{\mathrm{IFT}\left[\mathrm{FT}\left[E_{LJ}\right]\mathrm{FT}^{*}\left[E_{LJ}\right]\right]}{N_{E_{LJ}}\sigma^{2}_{E_{LJ}}}, \tag{A.1}$$

where FT and IFT are the Fourier transform and its inverse respectively, $\mathrm{FT}^{*}[E_{LJ}]$ denotes the complex conjugate of the Fourier transform of E_{LJ}, $N_{E_{LJ}}$ is the number of elements in E_{LJ}, and $\sigma^{2}_{E_{LJ}}$ is the variance of E_{LJ}. We used the time at which the self-correlation time crossed zero as our autocorrelation criterion. The slowest autocorrelation time was calculated to be 4.16×10^{-8} seconds for all of the simulations studied in this investigation and the fastest autocorrelation time was calculated to be around 1.97×10^{-10} seconds. An example is shown in Figure 2 in our paper, where the blue data points are considered to be taken from within the equilibrated window.

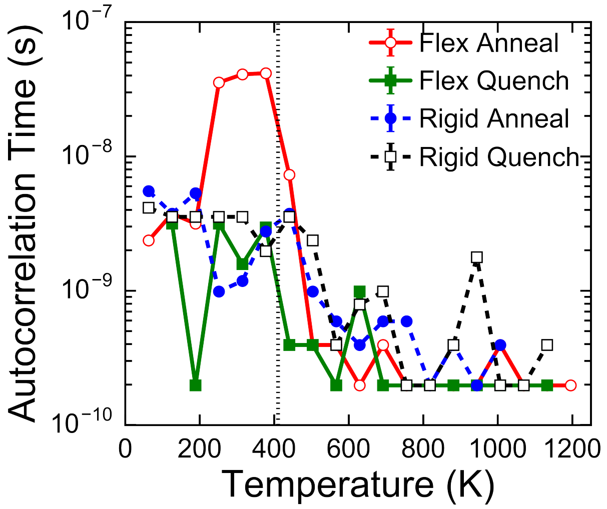


Figure A.1 Autocorrelation time for each state point. At higher temperatures the autocorrelation time is generally lower. The minimum autocorrelation time reflects the frequency at which we write the log file. The black vertical line indicates the disorder-order transition temperature $T_{\mathrm{DO}} = 410$ K.

A.1 shows some dependence of the autocorrelation time on temperature. At higher temperatures the autocorrelation time is lower which means that our system becomes decorrelated faster, suggesting more independent samples for the same number of timesteps. Note that the minimum possible autocorrelation time is 1×10^5 time steps (0.197 ns) since the E_{LJ} is only recorded in the log file every 1×10^5 time steps. There is no apparent trend in autocorrelation time between flexible or rigid systems, nor between annealing and quenched cooling mechanisms. We record the system's trajectory information every 1×10^6 time steps (1.97 ns). For quantities that require trajectory information (diffraction and clustering) we can use every trajectory frame (after the relaxation time) when the system's autocorrelation time is less than or equal to 1×10^6 time steps. For systems with an autocorrelation time greater than our trajectory recording time, several consecutive trajectory frames are skipped in order to ensure that multiple independent samples are considered.

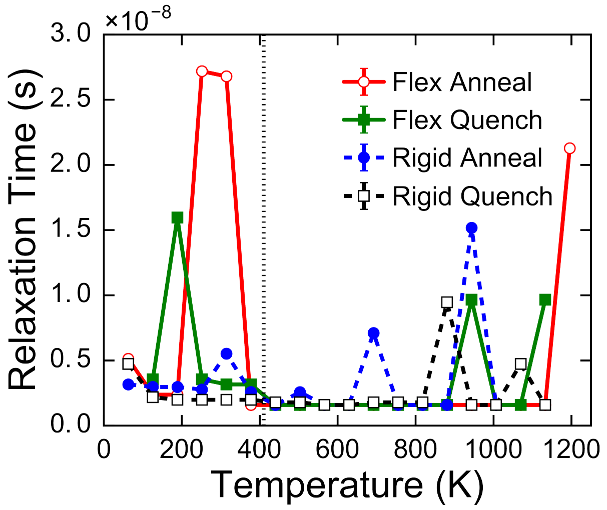


Figure A.2 Relaxation time for each state point. The black vertical line indicates the disorder-order transition temperature $T_{\mathrm{DO}} = 410$ K.

In A.2 we do not see any clear trend between relaxation time and state point. We found that most systems relaxed after 5 ns. This is about an order of magnitude greater than our autocorrelation time.

A.3 BDT-TPD Synthesis

The polymers used here are identical to those synthesized in Ref. [1]: "Polymer molecular weight was determined with the following process: The BDT-TPD polymer was dissolved in HPLC grade chloroform (~1 mg/mL), stirred and heated at 50°C for several hours under nitrogen, stirred overnight at r.t., and then filtered through a 0.45 μm PVDF filter. Size exclusion chromatography was then performed on a PL-Gel 300 × 7.5 mm (5 μm) mixed D column using an Agilent 1200 series autosampler, inline degasser, and refractometer. The column and detector temperatures were 35 °C. HPLC grade chloroform was used as eluent (1 mL/min). Linear polystyrene standards were used for calibration. The number average molecular weight of the polymer used in this work was determined to be 37 kg/mol, with a polydispersity index of 2.5."

A.4 Hardware

Access to Maverick, located at the Texas Advanced Computing Center (TACC) was provided through the NSF-supported XSEDE gateway[2]. Maverick has NVIDIA Tesla K40 "Atlas" GPUs with 12 GB of RAM and two Intel Xeon E5-2680 v2 "Ivy Bridge" CPUs per node. Kestrel has 2 NVIDIA Tesla K20 "Kepler" GPUs with 5GB of RAM and 2 Intel Xeon E5-2600 processors per node.

REFERENCES

[1] Wade A Braunecker, Zbyslaw R Owczarczyk, Andres Garcia, Nikos Kopidakis, Ross E Larsen, Scott R Hammond, David S Ginley, and Dana C Olson. Benzodithiophene and Imide-Based Copolymers for Photovoltaic Applications. *Chemistry of Materials*, 24(7):1346–1356, apr 2012.

[2] John Towns, Timothy Cockerill, Maytal Dahan, Ian Foster, Kelly Gaither, Andrew Grimshaw, Victor Hazlewood, Scott Lathrop, Dave Lifka, Gregory D. Peterson, Ralph Roskies, J. Ray Scott, and Nancy Wilkens-Diehr. XSEDE: Accelerating Scientific Discovery. *Computing in Science & Engineering*, 16(5):62–74, sep 2014.

APPENDIX B

ANALYSIS CODE

In Section B.1, the notebooks and python scripts used to generate the figures in Chapter 3 are included. In Section B.2, the notebooks and python scripts used to generate the figures in Chapter 4 are included.

B.1 Code for Chapter 3

scatter.py Page 1/3

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]
    y_err = log_data[:, 2]
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"$\zeta$")
ax.set_ylim([0,1])
ax.set_xlim([0,1250])
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5)
plt.axvline(x=410, color='k', linestyle=':')
```

scatter.py Page 3/3

```
plt.savefig("clust.pdf")
#plt.show()
```

doasq.py Page 1/3

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
from matplotlib.ticker import FormatStrFormatter
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
import glob
import re
from sys import argv

from matplotlib.ticker import FormatStrFormatter
import matplotlib.ticker as ticker
legend = ["Flex-Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]

system = "Flex-Anneal"
cold = '/Users/mikehenry/Projects/data/jan/paper-data/links/' + system + '/*T3.0/diffract/*/asq.txt'
hot =  '/Users/mikehenry/Projects/data/jan/paper-data/links/' + system + '/*T3.5/diffract/*/asq.txt'
colder = '/Users/mikehenry/Projects/data/jan/paper-data/links/' + system + '/*T2.5/diffract/*/asq.txt'
hoter =  '/Users/mikehenry/Projects/data/jan/paper-data/links/' + system + '/*T4.0/diffract/*/asq.txt'

dirs_list = [glob.glob(colder), glob.glob(cold), glob.glob(hot), glob.glob(hoter
)]

colors = ['darkblue', 'blue', 'darkred', 'red']
#legend = ["378 K","441 K","315 K","504 K"]
legend = ["315 K","378 K","441 K","504 K"]
bin_size = 0.013400650261999991
fig1, ax1 = plt.subplots()
scale = (1/1.6090103219409952)*1.7699113541350948
for i, dirs in enumerate(dirs_list):
    temps= []
    err = []
    for d in dirs:
        if True:
            data = np.genfromtxt(d)
            if len(data.shape) ≡ 1:
```

doasq.py Page 2/3

```
                pass
            else:
                try:
                    x = float((re.search("T(\d\.\d)", d).groups()[0]))
                except (AttributeError):
                    x = float((re.search("T(\d\d\.\d)", d).groups()[0]))
                if True:#x == 2.0 or x == 5.0:
                    frame_num = (d.split("/")[11:12][0])
                    if frame_num ≡ "difout" ∨ frame_num ≡ "newdiff" ∨ frame_num ≡ 'newdifout':
                        pass
                    else:
                        frame = int(frame_num)
                    window = np.genfromtxt("/".join(d.split("/")[:10])+"/LJ_window.txt", delimiter=" ")
                    skip = (int(np.ceil(window[0]/10)))
                    if frame ≥ skip:
                        temps.append(x)
                        err.append(data)
                    else:
                        pass
    #print(err)
    ax1.errorbar(data[:,0] * scale,
                 np.mean(err, axis=0)[:,1],
                 yerr=np.std(err, axis=0)[:,1],
                 label=legend[i],
                 marker = '.',
                 markeredgewidth=1,
                 markersize=8,
                 markerfacecolor="white",
                 color=colors[i],
                 markeredgecolor=colors[i],
                 linestyle="--")
    #plt.rcParams.update({'mathtext.default':  'regular' })
    #plt.xlabel(r"$q_r$ [$\AA^{-1}$]")
    #plt.ylabel(r"$\log(Intensity)$ [Arb]")
```

doasq.py Page 3/3

```
    #ax1.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, bo
rderaxespad = 0)

    #ax1.yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
    #ax1.set_xlim([.17,1.9])
    #ax1.set_ylim([-5,-3.4])
    #ax1.set_xlim([1.5,1.8])
    #ax1.set_ylim([-5,-4.6])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"$q_r$ [$\AA^{-1}$]")
plt.ylabel(r"$\log(Intensity)$ [Arb]")
#ax.set_ylim([0,-5])
#ax.yaxis.major.formatter._useMathText = True
ax1.xaxis.set_major_locator(ticker.MultipleLocator(.3))
ax1.yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
ax1.set_xlim([.17,1.9])
ax1.set_ylim([-5,-3.4])
ax1.xaxis.set_major_locator(ticker.MultipleLocator(.3))
#plt.yscale("log")
legend = ax1.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5,
borderaxespad = 0)
#plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0), useMathText=True)
plt.savefig("doasq.png", transparent=True)
#plt.show()
```

alle.py Page 1/3

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

print(matplotlib.matplotlib_fname())

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
```

```python
colors = ['r', 'g', 'b', 'k']

# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]/47280*1.74e-21#*4184*(0.25)
    y_err = log_data[:, 2]/47280*1.74e-21#*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])


plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"E$_{LJ} (J)$")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, borderaxespad = 0)
plt.axvline(x=410, color='k', linestyle=':')
plt.savefig("energy.png")
#plt.show()
```

ediff.py Page 1/2

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Anneal","Quench","Rigid Anneal","Rigid Quench"]
colors = ['m', 'c', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]/47280*1.74e-21#*4184*(0.25)
    y_err = log_data[:, 2]/47280*1.74e-21#*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i % 2 == 0:
        ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])
    else:
        ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])


plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"$\Delta$E$_{LJ} (J)$")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, b
orderaxespad = 0)
plt.axvline(x=410, color='k', linestyle=':')
plt.savefig("energydiff.png")
#plt.show()
```

fancy_asq.py Page 1/3

```
import numpy as np
import matplotlib
matplotlib.use('AGG')
from matplotlib.ticker import FormatStrFormatter
import matplotlib.ticker as ticker
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]


```

fancy_asq.py Page 2/3

```
legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']


#legend = ["Rigid Quench"]
#colors = ['k']

# Rigid dashed and open
# annealed circles quenched squares
local_y_max = []
local_y_min = []

'''
for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]
    local_y_max.append(np.max(y))
    local_y_min.append(np.min(y))

y_max = np.max(local_y_max)
y_maxi = np.argmax(local_y_max)

y_min = np.min(local_y_min)
y_mini = np.argmin(local_y_min)


for i, log in enumerate(list_of_logs):
    if i == y_mini:
        log_data = get_data(log)
        y = log_data[:, 1]
        yi = np.argmin(y)
#       alpha_1 = log_data[:, 2][yi]

for i, log in enumerate(list_of_logs):
    if i == y_maxi:
        log_data = get_data(log)
    y = log_data[:, 1]
    yim = np.argmin(y)
#    alpha_1_max = log_data[:, 2][yim]
'''

scale = (1/1.6090103219409952)*1.7699113541350948 # gets us to 3.55 sigma


for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1][13:]
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0][13:]*scale#*(1/0.87)#
    ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                        markeredgewidth=1, markersize=8, markerfacecolor="white",
                        color=colors[i], markeredgecolor=colors[i], linestyle="--")


plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"$q_r$ [$\AA^{-1}$]")
plt.ylabel(r"$\log(Intensity)$ [Arb]")
#ax.set_ylim([0,-5])
#ax.yaxis.major.formatter._useMathText = True
ax.yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
ax.set_xlim([.17,1.9])
ax.set_ylim([-5,-3.4])
ax.xaxis.set_major_locator(ticker.MultipleLocator(.3))
#plt.yscale("log")
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, b
orderaxespad = 0)
#plt.ticklabel_format(style='sci', axis='y', scilimits=(0,0), useMathText=True)
plt.savefig("mt.png")
#plt.show()
```

newfacnyasq.py Page 1/2

```python
import os
import glob
import numpy as np
import re
import matplotlib
matplotlib.use('AGG')

import matplotlib.pyplot as plt
from matplotlib.ticker import FormatStrFormatter
import matplotlib.ticker as ticker
dirs_list = [glob.glob('/Users/mikehenry/Projects/data/jan/paper-data/links/Flex-Anneal/*/diffract/*/asq.txt'),
             glob.glob('/Users/mikehenry/Projects/data/jan/paper-data/links/Flex-Quench/*/diffract/*/asq.txt'),
             glob.glob('/Users/mikehenry/Projects/data/jan/paper-data/links/Rigid-Anneal/*/diffract/*/asq.txt'),
             glob.glob('/Users/mikehenry/Projects/data/jan/paper-data/links/Rigid-Quench/*/diffract/*/asq.txt')]
colors = ['r', 'g', 'b', 'k']
legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
bin_size = 0.0134006502619999991
fig1, ax1 = plt.subplots()
scale = (1/1.6090103219409952)*1.7699113541350948
for i, dirs in enumerate(dirs_list):
    temps= []
    err = []
    for d in dirs:
        if True:
            data = np.genfromtxt(d)
            if len(data.shape) ≡ 1:
                pass
            else:
                try:
                    x = float((re.search("T(\d\.\d)", d).groups()[0]))
                except (AttributeError):
                    x = float((re.search("T(\d\d\.\d)", d).groups()[0]))
                if x ≡ 2.0 :
                    frame_num = (d.split("/")[11:12][0])
                    if frame_num ≡ "difout" ∨ frame_num ≡ "newdiff" ∨ frame_num ≡ 'newdifout':
                        pass
                    else:
                        frame = int(frame_num)
                    window = np.genfromtxt("/".join(d.split("/")[:10])+"/LJ_window.txt", delimiter=" ")
                    skip = (int(np.ceil(window[0]/10)))
                    if frame ≥ skip:
                        temps.append(x)
                        err.append(data)
                    else:
                        pass
    ax1.errorbar(data[:,0]*scale,np.mean(err, axis=0)[:,1],yerr=np.std(err, axis=0)[:,1],label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i], linestyle="--")
    plt.rcParams.update({'mathtext.default':  'regular' })
    plt.xlabel(r"$q_r$ [$\AA^{-1}$]")
    plt.ylabel(r"$\log(Intensity)$ [Arb]")
    ax1.set_xlim([2.5,3.5])
    ax1.set_ylim([0.26,0.36])
    ax1.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, borderaxespad = 0)

    ax1.yaxis.set_major_formatter(FormatStrFormatter('%.1f'))
    ax1.set_xlim([.17,1.9])
    ax1.set_ylim([-5,-3.4])
    ax1.xaxis.set_major_locator(ticker.MultipleLocator(.3))
plt.savefig("new.png")
```

scatter.py Page 1/4

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

scatter.py Page 2/4

```
# Rigid dashed and open
# annealed circles quenched squares

#For the addition (shift) you add in quadrature (sqrt(alpha_{1}^{2} + alpha_{2}^{2}))
#Then for the multiplication (scaling) you add the fractional uncertainties in quadrature
#sqrt( (alpha_{1}/val_{1})^{2} + (alpha_{2}/val_{2})^{2} )



local_y_max = []
local_y_min = []

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]
    local_y_max.append(np.max(y))
    local_y_min.append(np.min(y))

y_max = np.max(local_y_max)
y_maxi = np.argmax(local_y_max)

y_min = np.min(local_y_min)
y_mini = np.argmin(local_y_min)


for i, log in enumerate(list_of_logs):
    if i ≡ y_mini:
        log_data = get_data(log)
        y = log_data[:, 1]
        yi = np.argmin(y)
        alpha_1 = log_data[:, 2][yi]

for i, log in enumerate(list_of_logs):
    if i ≡ y_maxi:
```

scatter.py Page 3/4

```
        log_data = get_data(log)
        y = log_data[:, 1]
        yim = np.argmin(y)
        alpha_1_max = log_data[:, 2][yim]

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = (log_data[:, 1]+abs(y_min))/(abs(y_min)+y_max)
    y_err = log_data[:, 2]
    y_err = np.sqrt(alpha_1**2 + y_err**2) + np.sqrt(alpha_1**2 + alpha_1_max**2
) #+ np.sqrt((alpha_1/y_min)**2+(y_err/y)**2)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"|$\mathcal{I}$(T)|")
ax.set_ylim([0,1])
```

scatter.py Page 4/4

```
ax.set_xlim([0,1100])
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, borderaxespad = 0,
handlelength=1.5)
#ax.yaxis.labelpad = 10
plt.savefig("longrange.png")
#plt.show()
```

short_peak.py Page 1/4

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares
#For the addition (shift) you add in quadrature (sqrt(alpha_{1}^{2} + alpha_{2}^{2}))
#Then for the multiplication (scaling) you add the fractional uncertainties in quadrature
#sqrt( (alpha_{1}/val_{1})^{2} + (alpha_{2}/val_{2})^{2} )



local_y_max = []
local_y_min = []

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 3]
    local_y_max.append(np.max(y))
    local_y_min.append(np.min(y))

y_max = np.max(local_y_max)
y_maxi = np.argmax(local_y_max)

y_min = np.min(local_y_min)
y_mini = np.argmin(local_y_min)


for i, log in enumerate(list_of_logs):
    if i ≡ y_mini:
        log_data = get_data(log)
        y = log_data[:, 3]
        yi = np.argmin(y)
        alpha_1 = log_data[:, 4][yi]

for i, log in enumerate(list_of_logs):
    if i ≡ y_maxi:
        log_data = get_data(log)
        y = log_data[:, 3]
        yim = np.argmin(y)
        alpha_1_max = log_data[:, 4][yim]

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = (log_data[:, 3]+abs(y_min))/(abs(y_min)+y_max)
    y_err = log_data[:, 4]
    y_err = np.sqrt(alpha_1**2 + y_err**2) + np.sqrt(alpha_1**2 + alpha_1_max**2) #+ np.sqrt((alpha_1/y_min)**2+(y_err/y)**2)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"|$\mathcal{I}$(T)|")
ax.set_ylim([0,1])
ax.set_xlim([0,1100])
```

short_peak.py Page 4/4

```
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, borderaxespad = 0)
plt.savefig("short_peak.png")
#plt.show()
```

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, -1]*1.97e-15#*2.16e-15
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"Autocorrelation Time (s)")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
plt.yscale("log")
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, b
```

act.py Page 3/3

```
orderaxespad = 0)
plt.axvline(x=410, color='k', linestyle=':')
plt.savefig("autocorrtime.png", transparent=True)
#plt.show()
```

relax.py Page 1/3

```
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]*1e5*1.97e-15#2.16e-15#*1e6
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                    linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                    markeredgewidth=1, markersize=8, markerfacecolor="white",
                    color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                    markeredgewidth=1, markersize=8,
                    color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular'  })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"Relaxation Time (s)")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
#plt.yscale("log")
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='upper right', shadow=False, prop={'size':20}, handlelength=1.
```

relax.py Page 3/3

```
5, borderaxespad = 1)
plt.axvline(x=410, color='k', linestyle=':')
plt.savefig("relaxtime.png", transparent = True)
#plt.show()
```

scalefactor.py Page 1/3

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                        markeredgewidth=1, markersize=8,
                        linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                        markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                        linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 == 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                        markeredgewidth=1, markersize=8, markerfacecolor="white",
                        color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                        markeredgewidth=1, markersize=8,
                        color=colors[i], markeredgecolor=colors[i])

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"Scale Factor")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
#plt.yscale("log")
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, b
```

scalefactor.py Page 3/3

```
    orderaxespad = 0)
plt.savefig("scalefactor.png")
#plt.show()
```

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv

def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
colors = ['r', 'g', 'b', 'k']

```

```
# Rigid dashed and open
# annealed circles quenched squares

for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 1]
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0]*125.867 # Temp conversion
    if i > 1:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                        markeredgewidth=1, markersize=8,
                        linestyle='--', color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                        markeredgewidth=1, markerfacecolor="white" ,markersize=8,
                        linestyle='--', color=colors[i], markeredgecolor=colors[i])

    else:
        if i % 2 ≡ 0:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 'o',
                        markeredgewidth=1, markersize=8, markerfacecolor="white",
                        color=colors[i], markeredgecolor=colors[i])
        else:
            ax.errorbar(x,y,yerr=y_err,label=legend[i], marker = 's',
                        markeredgewidth=1, markersize=8,
                        color=colors[i], markeredgecolor=colors[i])
    print(np.mean(y))

plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Temperature (K)")
plt.ylabel(r"TPS")
#ax.set_ylim([0,1])
ax.set_xlim([0,1250])
ax.yaxis.major.formatter._useMathText = True
legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5, b
```

tps.py Page 3/3

```
orderaxespad = 0)
plt.axvline(x=410, color='k', linestyle=':')
plt.savefig("TPS.png", transparent=True)
#plt.show()
```

window.py Page 1/3

```python
import numpy as np
import matplotlib
matplotlib.use('AGG')
import matplotlib.pyplot as plt
import re
from sys import argv
from matplotlib.ticker import FormatStrFormatter


def get_data(log_file):
    try:
        data = np.genfromtxt(log_file)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None


def get_data_with_headers(log_file):
    try:
        data = np.genfromtxt(log_file, comments="@", names=True)
        return data
    except (ValueError, IOError):
        print("Problem with this log file")
        print(log_file)
        print("\n")
        return None



fig, ax = plt.subplots()
list_of_logs = [vals for vals in argv[1:]]

legend = ["Flex Anneal","Flex Quench","Rigid Anneal","Rigid Quench"]
```

window.py Page 2/3

```
colors = ['r', 'g', 'b', 'k']

# Rigid dashed and open
# annealed circles quenched squares
# % * Change the blue datapoints to be square (makes it clearer for black-and-wh
ite representations of figures and it is generally good practice)
# % * Change the vertical dotted lines to be black to make them more distinct
# 10 bins#
SLICE = 410
for i, log in enumerate(list_of_logs):
    log_data = get_data(log)
    y = log_data[:, 6]*4184*(0.25)
    y = y[10:]
    y_err = 0#log_data[:, 2]*4184*(0.25)
    x = log_data[:, 0]*2.16e-15 #:Time conversion
    x = x[10:]
    ax.scatter(x[:SLICE], y[:SLICE],label=legend[i], marker = 'o',
                    color=colors[i])

    ax.scatter(x[SLICE:], y[SLICE:],label=legend[i], marker = 's',
                    color='b')



#print(len(x))

vslice = x[::int(len(x)/10)]
#print(vslice)
vslice = vslice[1:-1]
for line in vslice:
    plt.axvline(line, ls ='--', color = 'k')
plt.rcParams.update({'mathtext.default':  'regular' })
plt.xlabel(r"Time (s)")
plt.ylabel(r"E$_{LJ}$ (J/mol)")
#ax.set_ylim([0,1])
plt.xticks(np.array([0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0])*1e-7)
```

window.py   Page 3/3

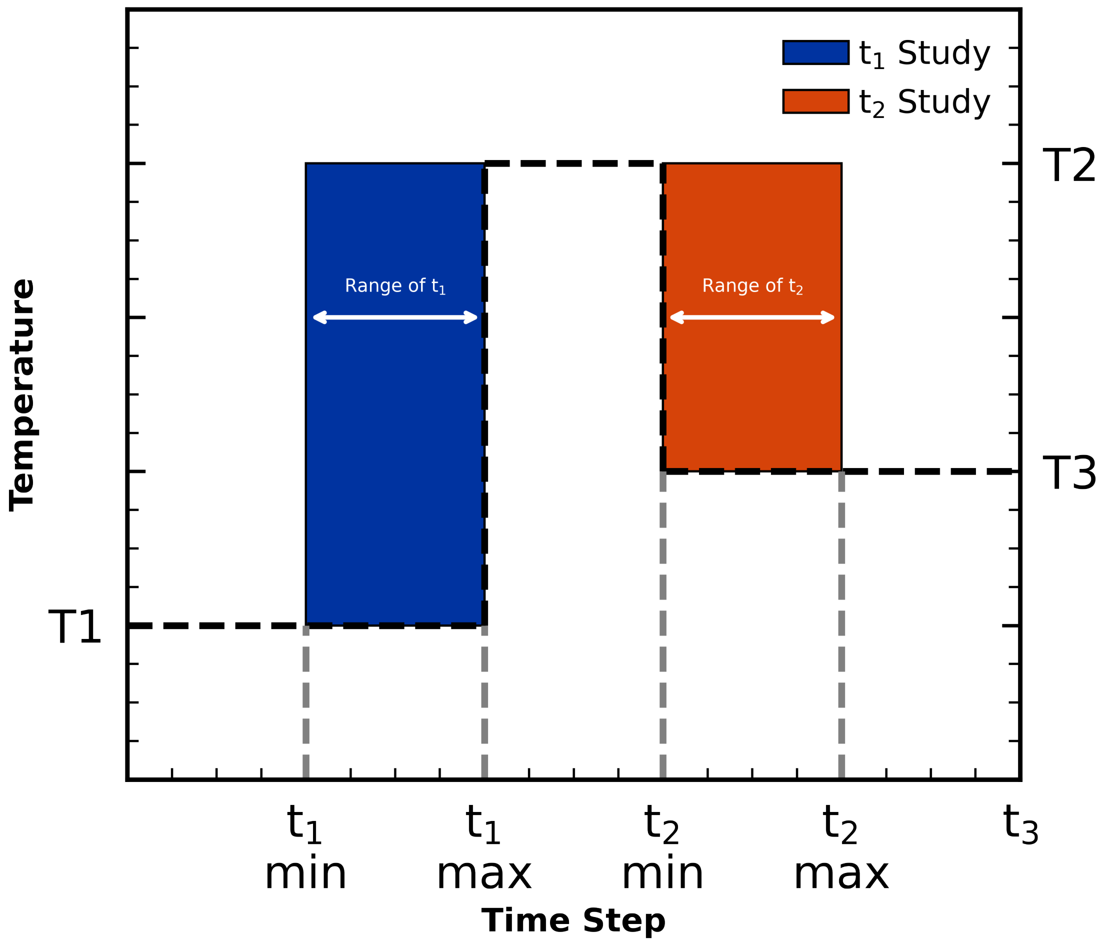
```
ax.set_xlim([0.0,3.0e-7])
ax.yaxis.major.formatter._useMathText = True
ax.xaxis.major.formatter._useMathText = True
plt.ticklabel_format(style='sci', axis='x', scilimits=(0,0), useMathText=True)
#legend = ax.legend(loc='best', shadow=False, prop={'size':20}, handlelength=1.5
, borderaxespad = 0)
plt.savefig("window.png")
#plt.show()
```

B.2 Code for Chapter 4

```
In [1]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"

        import matplotlib.pyplot as plt

        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"
        %matplotlib inline

In [2]: fig, ax = plt.subplots(dpi=600, figsize=(7, 6))

        main_style = {"linestyle": "--", "color": "k"}

        # vline x, ymin, ymax
        # hline y, xmin, xmax

        eye_guide_style  = {"linestyle": "--", "color": "grey"}

        # t1 study
        #t1_style = {"linestyle": "--", "color": BSU_BLUE}

        #b_t1 = ax.vlines(0.2, 0.2, 0.8, **t1_style)
        #c_t1 = ax.hlines(0.8, 0.2, 1, **t1_style)
        t1_shade = ax.axvspan(
            0.2,
            0.4,
            ymin=0.2,
            ymax=0.8,
            facecolor=BSU_BLUE,
            hatch="",
            edgecolor="k",
            label=r"t$_1$ Study",
        )


        # t2 study
        #t2_style = {"linestyle": "--", "color": BSU_ORANGE}

        #c_t2 = ax.hlines(0.8, 0.6, 0.8, **t2_style)
        #d_t2 = ax.vlines(0.8, 0.4, 0.8, **t2_style)




        # e_t2 = ax.hlines(.4, .75, 1, **t2_style)

        t2_shade = ax.axvspan(
            0.6,
            0.8,
            ymin=0.4,
            ymax=0.8,
            facecolor=BSU_ORANGE,
            hatch="",
            edgecolor="k",
            label=r"t$_2$ Study",
        )

        # main lines
        a = ax.hlines(0.2, 0, 0.4, **main_style)
        b = ax.vlines(0.4, 0.2, 0.8, **main_style)
        c = ax.hlines(0.8, 0.4, 0.6, **main_style)
        d = ax.vlines(0.6, 0.4, 0.8, **main_style)
        e = ax.hlines(0.4, 0.6, 1, **main_style)

        # t1 eye guide
        ax.vlines(0.2, 0.0, 0.2, **eye_guide_style)
        ax.vlines(0.4, 0.0, 0.2, **eye_guide_style)
```

```
# t2 eye guide
ax.vlines(0.8, 0.0, 0.4, **eye_guide_style)
ax.vlines(0.6, 0.0, 0.4, **eye_guide_style)

ax.set_xlim(0, 1)
ax.set_ylim(0, 1)

ax.set_ylabel("Temperature")
ax.set_xlabel("Time Step")

# ax.set_yticklabels([])
# ax.set_xticklabels(["test"])
x_labels = [item.get_text() for item in ax.get_xticklabels()]
x_labels[1] = "t$_1$ \nmin"
x_labels[2] = "t$_1$ \nmax"
x_labels[3] = "t$_2$ \nmin"
x_labels[4] = "t$_2$ \nmax"
x_labels[5] = "t$_3$"
ax.set_xticklabels(x_labels)


y_labels = [item.get_text() for item in ax.get_yticklabels()]
y_labels[1] = "T1"

ax.set_yticklabels(y_labels)

ax1 = ax.twinx()
y_labels = [item.get_text() for item in ax1.get_yticklabels()]
y_labels[2] = "T3"
y_labels[4] = "T2"
ax1.set_yticklabels(y_labels)

#plt.text(0.2, 0.15, "T1", horizontalalignment="center")
#plt.text(0.5, 0.85, "T2", horizontalalignment="center")
#plt.text(0.8, 0.35, "T3", horizontalalignment="center")

arrow_x = 0.2
arrow_y = 0.6
arrow_dx = 0.2
ax.annotate(
    text="",
    xy=(arrow_x, arrow_y),
    xytext=(arrow_x + arrow_dx, arrow_y),
    arrowprops=dict(arrowstyle="<->",lw=2,color="white"),
)
ax.annotate(
    text=r"Range of t$_1$",
    xy=((arrow_x + arrow_dx / 2), arrow_y + 0.04),
    size=8,
    color="white",
    ha="center",
    va="center",
)


arrow_x = 0.25 + 0.35
arrow_y = 0.6
arrow_dx = 0.2
ax.annotate(
    text="",
    xy=(arrow_x, arrow_y),
    xytext=(arrow_x + arrow_dx, arrow_y),
    arrowprops=dict(arrowstyle="<->",lw=2, color="white"),
)
ax.annotate(
    text=r"Range of t$_2$",
    xy=((arrow_x + arrow_dx / 2), arrow_y + 0.04),
    size=8,
    color="white",
    ha="center",
    va="center",
)


for axis in ['top','bottom','left','right']:
    ax.spines[axis].set_linewidth(2)

ax.tick_params(width=1.5)

ax.legend()


plt.savefig("new_step_cartoon.png", transparent=True)
```

```
<ipython-input-2-330cee01a3e4>:78: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_xticklabels(x_labels)
<ipython-input-2-330cee01a3e4>:84: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax.set_yticklabels(y_labels)
<ipython-input-2-330cee01a3e4>:90: UserWarning: FixedFormatter should only be used
together with FixedLocator
  ax1.set_yticklabels(y_labels)
```



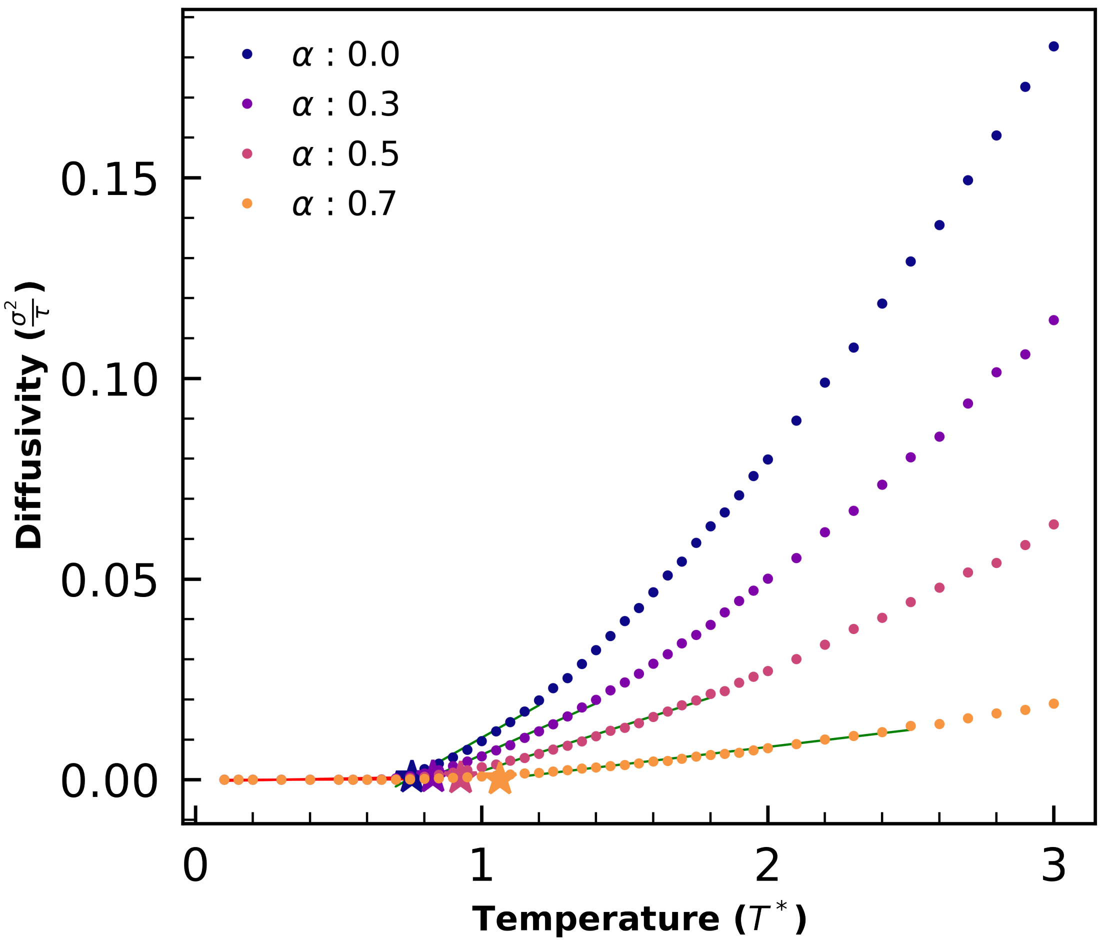
```
In [ ]:
```

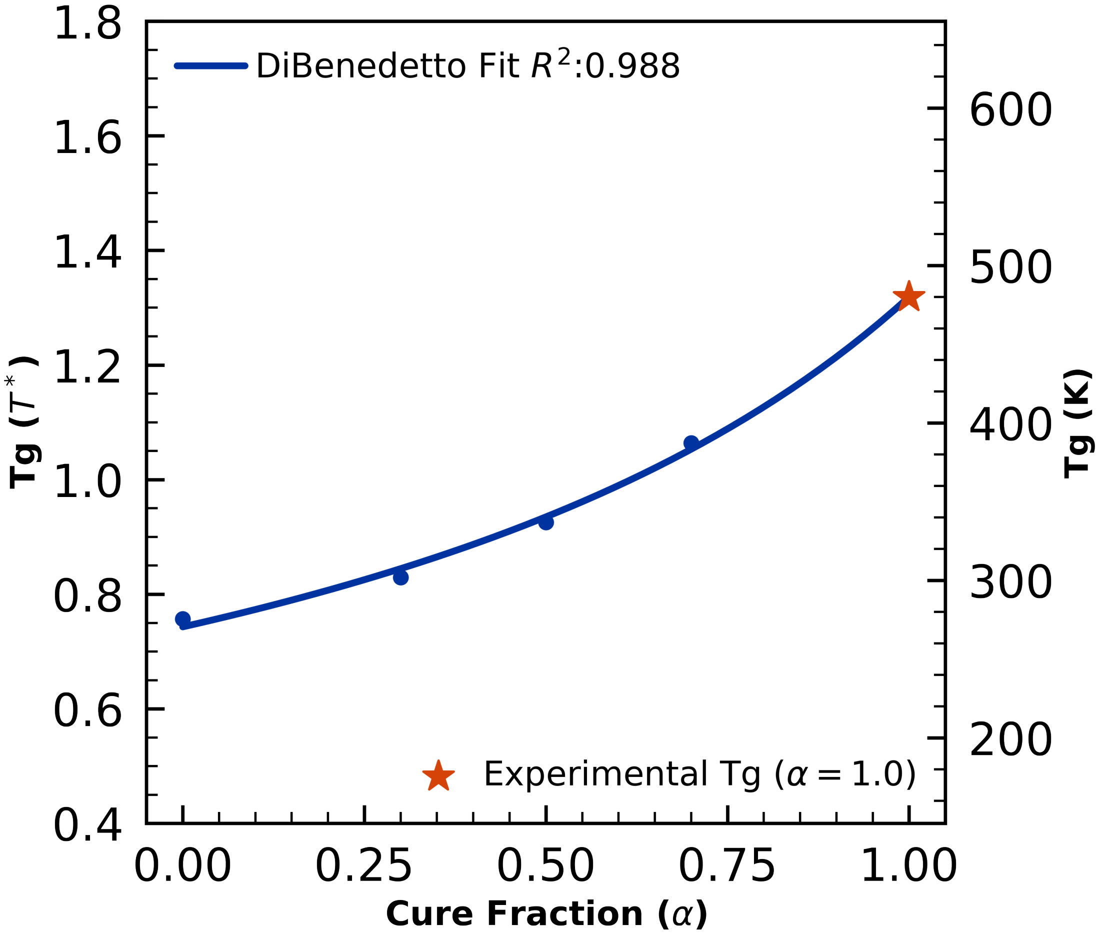
```
In [5]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"

        data_path = '/home/sthomas/projects/LB_mixing'

        from common import getDiffusivities, line_intersect, fit_Tg_to_DiBenedetto, DiBenedetto,
        Fit_Diffusivity1

        import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib

        from piecewise.regressor import piecewise
        #https://www.datadoghq.com/blog/engineering/piecewise-regression/
        from piecewise.plotter import plot_data_with_regression
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools

        %matplotlib inline
        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        project = signac.get_project(data_path)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(['_id'])
        statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
        df = pd.DataFrame(statepoints).T.join(df_index)
        df = df.sort_values('T')
        def get_custom_ranges(cooling_method):
            if cooling_method=='quench':
                custom_ranges_l1={00.0:[0.1,0.8],
                                  30.0:[0.1,0.8],
                                  50.0:[0.1,0.8],
                                  70.0:[0.1,0.8]}
                custom_ranges_l2={00.0:[0.7,1.2],
                                  30.0:[0.85,1.4],
                                  50.0:[1.0,1.8],
                                  70.0:[1.15,2.5]}
            elif cooling_method=='anneal':
                custom_ranges_l1={00.0:[0.1,0.8],
                                  30.0:[0.1,0.8],
                                  50.0:[0.1,0.8],
                                  70.0:[0.1,0.8]}
                custom_ranges_l2={00.0:[0.7,1.2],
                                  30.0:[0.85,1.4],
                                  50.0:[1.0,1.8],
                                  70.0:[1.15,2.5]}
            else:
                raise ValueError(cooling_method+'is unknown')
            return custom_ranges_l1, custom_ranges_l2

        PROP_NAME
        ='bparticles'#'volume'#'pair_lj_energy','bond_harmonic_energy'#'potential_energy'
        filter_saps=[0.0,30.,50.,70.]#,100.]#,100.]#[0.0,50.0,100.0]#,30,50,70]#,90]
        colors = plt.cm.plasma(np.linspace(0,0.75,len(filter_saps)))
        Tgs=[]
        Tgs_tangent=[]
        cure_percents = []
        Cure_Ts=[]
        markers=['+','.']
        markersize=[10,10]
```

```
cooling_method='quench'
fig, ax1 = plt.subplots(dpi=600, figsize=(7,6))

df_filtered=df[(df.quench_T<=3.0)&
               (df.quench_T>=0.1)&
               (df.CC_bond_angle!=109.5)&
(df.cooling_method==cooling_method)]#(df.quench_T<=3.0)&(df.quench_T>=0.05)&
for i,sap in enumerate(filter_saps):
    cooling_colors = plt.cm.plasma(np.linspace(0,0.75,2))
    for j,(cooling_method,df_grp) in enumerate(df_filtered.groupby('cooling_method')):
        df_curing = df_grp[(df_grp.bond==False)&
                       (df_grp.calibrationT==305)&
                       (df_grp.cooling_method==cooling_method)&
                       (df_grp.stop_after_percent==sap)]
        cure_percent = df_curing.cure_percent.mean()
        cure_percents.append(cure_percent)
        Ts,Ds=getDiffusivities(project,df_curing,name=PROP_NAME)
        Cure_Ts.append(Ts)

        mul_fact=1000000
        Ds_scaled=Ds*mul_fact
        custom_ranges_l1, custom_ranges_l2 = get_custom_ranges(cooling_method)
        print(custom_ranges_l1[sap])
        Tg,Tg_prop,line_vals = Fit_Diffusivity1(Ts,
                                    Ds_scaled,
                                    method='use_viscous_region',
                                    min_D=0,
                                    ver=4,
                                    viscous_line_index=0,
                                    l1_T_bounds=custom_ranges_l1[sap],
                                    l2_T_bounds=custom_ranges_l2[sap])
        xs = Ts#np.linspace(0.1,4)
        plt.plot(Tg,
                 Tg_prop/mul_fact,
                 marker='*',
                 color=colors[i],
                 markersize=15)#,

        plt.plot(Ts,
                 Ds,
                 marker='.',
                 color=colors[i],#cooling_colors[j],
                 linewidth=0.0,
                 label='$\\alpha$ : {:.1f}'.format(sap/100))


        l_colors=['r','g']
        for li,line_val in enumerate(line_vals):
            xs=line_val[0]
            ys=line_val[1]/mul_fact
            plt.plot(xs,
                     ys,
                     color=l_colors[li],
                     zorder=0,
                     linewidth=1)
        Tgs.append(Tg)
    #break
plt.legend(fontsize=15)
plt.ticklabel_format(axis='y', style='sci', scilimits=(-2,2))
Tgs = np.asarray(Tgs)
cure_percents = np.asarray(cure_percents)
data=[cure_percents,Tgs]

plt.xlabel('Temperature ($T^*$)',fontsize=15)
plt.ylabel('Diffusivity ($\\frac{\\sigma^2}{\\tau}$)',fontsize=15)


plt.savefig("some_alpha.png", transparent=True)
```

```
        cure_percents = np.asarray(cure_percents)
        fig, ax1 = plt.subplots(dpi=600, figsize=(7,6))
        ax2=ax1.twinx()
        Tgs = np.asarray(Tgs)
        Tgs_tangent = np.asarray(Tgs_tangent)
        print(Tgs)
        Tg_data = np.asarray([cure_percents/100.,Tgs])
        cure_percents_ss = cure_percents#[:-1]
        Tgs_ss = Tgs#[:-1]
        print(cure_percents_ss)
        print(Tgs_ss)
        R2,fit_Tgs,T1,inter_parm,T0 = fit_Tg_to_DiBenedetto(cure_percents_ss/100.,
                                                           Tgs_ss,
                                                           T1=None,
                                                           T0=None)
        print('T1',T1,'lambda',inter_parm)
        alphas = np.linspace(0,1)
        fit_ydata = DiBenedetto(alphas,T1,T0=T0,inter_param=inter_parm)
        ax1.plot(alphas,fit_ydata,label='DiBenedetto Fit $R^2$:{}'.format(round(R2,3)),
        color=BSU_BLUE)
        ax1.scatter(cure_percents/100.,
                    Tgs,
                    color=BSU_BLUE)

        Tg_sim = T1
        Tg_exp = 480
        roomT_exp = 300
        Tex_toTsim = Tg_exp/Tg_sim
        roomT_sim =  Tg_sim*roomT_exp/Tg_exp
        Tg0_exp = Tg_exp*T0/Tg_sim
        print('300 K in T*:',roomT_sim)
        ax2.scatter(1.00,Tg_exp,marker='*',color=BSU_ORANGE,s=200,label='Experimental Tg
        ($\\alpha=1.0$)')
        ax2.set_ylabel('Tg (K)')

        sim_low_lim = 0.4
        ex_low_lim = sim_low_lim*Tex_toTsim
        sim_up_lim = 1.8
        ex_up_lim = sim_up_lim*Tex_toTsim
        ax2.set_ylim(ex_low_lim,ex_up_lim)
        ax1.set_ylim(sim_low_lim,sim_up_lim)
        show_roomT=False
        if show_roomT:
            ax1.axhline(y=roomT_sim,linewidth=1.1,linestyle='--',label='simulated 300 K')
        ax1.set_xlabel('Cure Fraction ($\\alpha$)',fontsize=15)
        ax1.set_ylabel('Tg ($T^*$)',fontsize=15)
        ax1.legend(fontsize=15,loc='upper left')
        ax2.legend(fontsize=15,loc='lower right')
        plt.ticklabel_format(axis='y',style='plain')
        plt.savefig("DB_fit.png", transparent=True)
```

```
[0.1, 0.8]
in common, indices: (array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 1
[0.1, 0.8]
in common, indices: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 0
[0.1, 0.8]
in common, indices: (array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 1
[0.1, 0.8]
in common, indices: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 0
[0.75670876 0.82950327 0.9256867  1.06374355]
[2.49999994e-03 3.00000000e+01 5.00000000e+01 7.00000000e+01]
[0.75670876 0.82950327 0.9256867  1.06374355]
T1 1.318812860773745 lambda 0.5
300 K in T*: 0.8242580379835907
```



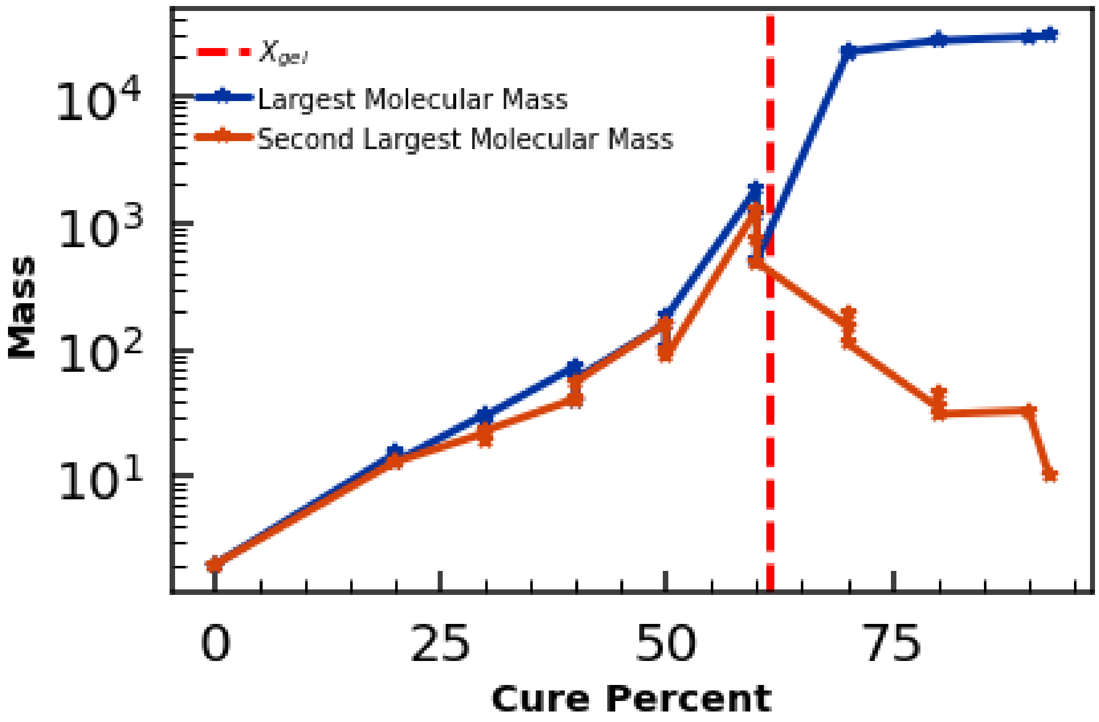

DiBenedetto Fit R^2:0.988
Experimental Tg ($\alpha = 1.0$)
Tg (T^*)
Tg (K)
Cure Fraction (α)
1.8
1.6
1.4
1.2
1.0
0.8
0.6
0.4
600
500
400
300
200
0.00
0.25
0.50
0.75
1.00

```
In [ ]:
```

```
In [ ]: data_path = "/home/sthomas/projects/LB_mixing"
        import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"
```

```
In [2]: import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib
        %matplotlib inline
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools


        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        project = signac.get_project(data_path)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(['_id'])
        statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
        df = pd.DataFrame(statepoints).T.join(df_index)
        df = df.sort_values('T')
```

```
In [3]: df_filtered = df[(df.bond==True)]
        df_sorted = df_filtered.sort_values('cure_percent')
        df_100 = df_filtered[df_filtered.stop_after_percent>80]

        plt.axvline(x=df_100['curing_at_gel_point'][0],
                    color='r',
                    linestyle='--',
                   label='$X_{gel}$')
        plt.plot(df_sorted['cure_percent'],
                 df_sorted['largest_network'],
                 marker='*',
                 label='Largest Molecular Mass',
                color=BSU_BLUE)
        plt.plot(df_sorted['cure_percent'],
                 df_sorted['second_largest_network'],
                 marker='*',
                 label='Second Largest Molecular Mass',
                color=BSU_ORANGE)
        plt.yscale('log')
        plt.xlabel('Cure Percent')
        plt.ylabel('Mass')
        plt.legend(fontsize=10)
        plt.savefig("gel_point.png", transparent=True)
```

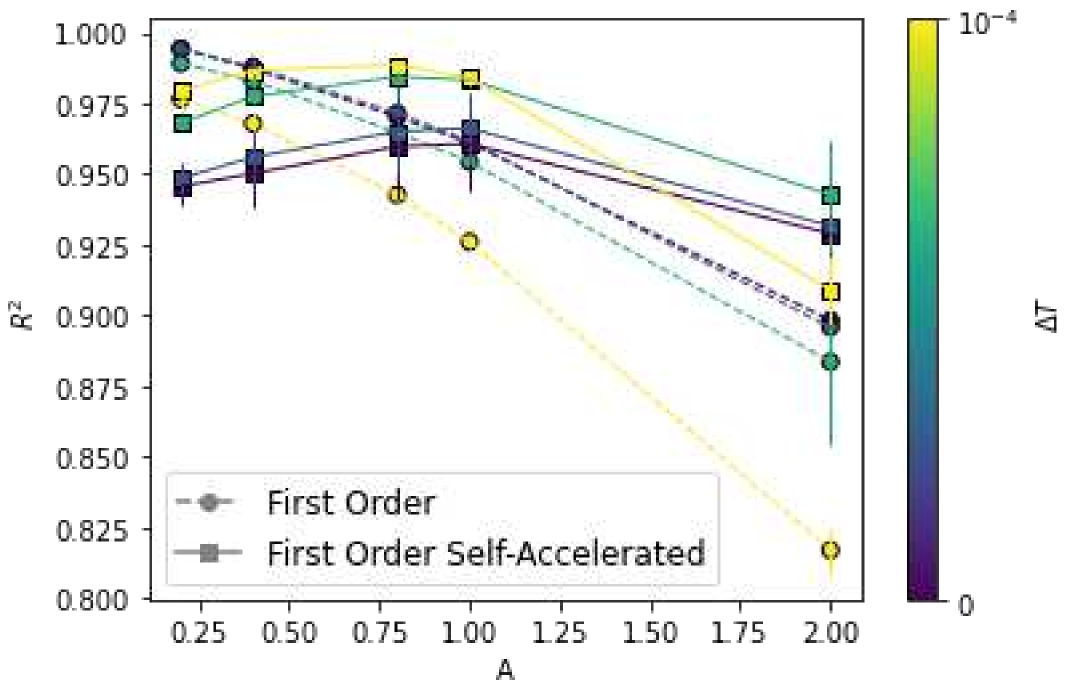

X_{gel}
Largest Molecular Mass
Second Largest Molecular Mass
Mass
Cure Percent
10^1
10^2
10^3
10^4
0
25
50
75

```
In [ ]:
```

```
In [1]: import matplotlib.pyplot as plt
        import matplotlib as cm
        import matplotlib.lines as mlines


        import numpy as np
        from collections import OrderedDict


        A = [0.2, 0.4, 0.8, 1.0, 2.0]
        deltaT= [0, 1e-06, 1e-05, 0.0001]
        FO_0_mean = [0.994388, 0.987342, 0.970278, 0.961052, 0.898360]
        FO_0_sem = [0.0011599593740658951, 0.0024308681875413827, 0.009510626980093839,
        0.01311594231997513, 0.03607616595453638]

        FO_1e06_mean =[0.9942860750595337, 0.9880332636504043, 0.9715727286807505,
        0.9617348596652091, 0.895853274381769]
        FO_1e06_sem = [0.0011950288033497475, 0.0027625968946956327, 0.010346970623441662,
        0.013708840275554593, 0.03664748969973626]

        FO_1e05_mean = [0.9891900062751051, 0.9830138183585232, 0.9647697478186913,
        0.954392840081353, 0.8836898304889049]
        FO_1e05_sem  = [0.001522794529324003, 0.001954618710525918, 0.007076488011568168,
        0.009890860112780323, 0.029829038695698272]

        FO_0001_mean = [0.9765340308833451, 0.9679304230504282, 0.9427140731911086,
        0.9261284736630658, 0.8170891346516121]
        FO_0001_sem =[0.0041862994458542425, 0.003888607082289215, 0.0030449517296994824,
        0.0017131920455586425, 0.008133915592155674]


        SAFO_0_mean = [0.945619103017723, 0.9502542889066092, 0.9598076788566482,
        0.9607301037624483, 0.9289951020475868]
        SAFO_0_sem = [0.006392732210594329, 0.0124956825095459, 0.019040485607138728,
        0.01716467117011829, 0.015309757691479006]


        SAFO_1e06_mean = [0.9485914887891203, 0.9560205476716985, 0.9650941367218115,
        0.9662292362702924, 0.931752284665132]
        SAFO_1e06_sem =  [0.005625036624910599, 0.00948104936120714, 0.014276453108781461,
        0.012399245208401697, 0.015227730575043147]

        SAFO_1e05_mean = [0.9681606710891607, 0.9774884845930968, 0.984524089818837,
        0.9834009700698025, 0.942708220784915]
        SAFO_1e05_sem  = [0.0025019918928122736, 0.0023519309457531038, 0.0014005409206292304,
        0.0013345367141796671, 0.01939687875342732]

        SAFO_0001_mean = [0.9794764769613135, 0.9867889115270978, 0.9886633811268632,
        0.9840540474306009, 0.9091319649266506]
        SAFO_0001_sem = [0.0022878305443372587, 0.0013229661902640793, 0.0012035959203720908,
        0.0020346156964257483, 0.011740217875926318]

        class Data():
            A = A


        FO_0 = Data()
        FO_0.mean = FO_0_mean
        FO_0.sem = FO_0_sem
        FO_0.kT = 0
        FO_0.name = "First Order"

        FO_1e06 = Data()
        FO_1e06.mean = FO_1e06_mean
        FO_1e06.sem = FO_1e06_sem
        FO_1e06.kT = 1e-06
        FO_1e06.name = "First Order"
```

```
FO_1e05 = Data()
FO_1e05.mean = FO_1e05_mean
FO_1e05.sem = FO_1e05_sem
FO_1e05.kT = 1e-05
FO_1e05.name = "First Order"

FO_0001 = Data()
FO_0001.mean = FO_0001_mean
FO_0001.sem = FO_0001_sem
FO_0001.kT = 1e-4
FO_0001.name = "First Order"

SAFO_0 = Data()
SAFO_0.mean = SAFO_0_mean
SAFO_0.sem = SAFO_0_sem
SAFO_0.kT = 0
SAFO_0.name = "First Order Self-Accelerated"

SAFO_1e06 = Data()
SAFO_1e06.mean = SAFO_1e06_mean
SAFO_1e06.sem = SAFO_1e06_sem
SAFO_1e06.kT = 1e-06
SAFO_1e06.name = "First Order Self-Accelerated"

SAFO_1e05 = Data()
SAFO_1e05.mean = SAFO_1e05_mean
SAFO_1e05.sem = SAFO_1e05_sem
SAFO_1e05.kT = 1e-05
SAFO_1e05.name = "First Order Self-Accelerated"

SAFO_0001 = Data()
SAFO_0001.mean = SAFO_0001_mean
SAFO_0001.sem = SAFO_0001_sem
SAFO_0001.kT = 1e-4
SAFO_0001.name = "First Order Self-Accelerated"

data_points = [FO_0,FO_1e06,FO_1e05,FO_0001]
data_points += [SAFO_0,SAFO_1e06,SAFO_1e05,SAFO_0001]




deltaT= [0, 1e-06, 1e-05, 0.0001]

COLOR_MAP = "viridis"

FO_MARKER = "o"
FO_LS = "--"

SAFO_MARKER = "s"
SAFO_LS = "-"

norm = cm.colors.SymLogNorm(linthresh=1e-6, vmax=max(deltaT), vmin=min(deltaT),
clip=False)

cmap = cm.cm.get_cmap(COLOR_MAP)
plt.set_cmap(COLOR_MAP)



for dp in data_points:
    if dp.name == "First Order":
        marker = FO_MARKER
        linestyle = FO_LS
    else:
        marker = SAFO_MARKER
        linestyle = SAFO_LS
    plt.errorbar(dp.A, dp.mean, yerr= dp.sem, marker=None, c=cmap(norm(dp.kT)),
linewidth=1, linestyle=linestyle)
    plt.scatter(dp.A, dp.mean, c=[dp.kT]*len(dp.A), norm=norm, marker=marker,
label=dp.name, edgecolors="black", linewidths=1)


cbar = plt.colorbar()


first_order = mlines.Line2D([], [], color='grey', linestyle=FO_LS, marker=FO_MARKER,
label="First Order")
first_order_SA = mlines.Line2D([], [], color='grey', linestyle=SAFO_LS,
marker=SAFO_MARKER, label="First Order Self-Accelerated")

handles, labels = plt.gca().get_legend_handles_labels()
by_label = OrderedDict(zip(labels, handles))
plt.legend(handles=[first_order, first_order_SA], prop={'size': 12})


plt.xlabel("A")
plt.ylabel("$R^2$")
cbar.set_label("$\Delta T$")
cbar.set_ticks([0,0.0001])
plt.savefig("draft_A_deltaT.png", transparent=True, dpi=300)
```

```
<ipython-input-1-e447fb990a95>:106: MatplotlibDeprecationWarning: default base will
change from np.e to 10 in 3.4.  To suppress this warning specify the base keyword
argument.
  norm = cm.colors.SymLogNorm(linthresh=1e-6, vmax=max(deltaT), vmin=min(deltaT),
clip=False)
```



```
In [ ]:
```

```
In [1]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"
        from common import *
        data_path = "/home/mikehenry/epoxy-stuff/tuningrxn"

In [2]: import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib
        %matplotlib inline
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools


        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        def init_project():
            df_index = pd.DataFrame(project.index())
            df_index = df_index.set_index(['_id'])
            statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
            #print(statepoints)
            df = pd.DataFrame(statepoints).T.join(df_index)
            df = df.sort_values('T')
            return df
        project = signac.get_project(data_path)
        df = init_project()

In [3]: #ADDING A VALUE TO DATA FRAME
        df['A']=df.num_a*df.n_mul*4*df.percent_bonds_per_step/100/df.bond_period

        #IGNORING NEGATIVE R^2 VALUES SINCE THE MODEL DOES NOT FIT THE DATA AT ALL
        df.loc[df['FO_model_R2']<0, 'FO_model_R2'] = None
        df.loc[df['SAFO_model_R2']<0, 'SAFO_model_R2'] = None
        df.loc[df['SO_model_R2']<0, 'SO_model_R2'] = None
        df.loc[df['SASO_model_R2']<0, 'SASO_model_R2'] = None

In [4]: df_filtered=df[(df.n_particles==50000)&
                         (df.trial==0)&
                         #(df.kT==0.5)&
                         (df.sec_bond_weight==2.0)&
                         (df.t_Final==6000000.0)]
        N = len(df_filtered.index)
        for i,jobid in enumerate(df_filtered.index):
            print('\r{}/{}'.format(i,N),end='', flush=True)
            job=project.open_job(id=jobid)
            #print(job)
            if job.isfile('out.log'):
                success,r_squared,C,H, ts,X, first_index,last_index =
        fit_curing_profile_with_model(job,'FO')
                if success:
                    df.at[jobid,'FO_model_R2']=r_squared
                    df.at[jobid,'FO_model_C']=None
                    df.at[jobid,'FO_model_H']=H
                else:
                    df.at[jobid,'FO_model_R2']=None
                    df.at[jobid,'FO_model_C']=None
                    df.at[jobid,'FO_model_H']=None
                success,r_squared,C,H, ts,X, first_index,last_index =
        fit_curing_profile_with_model(job,'SO')
                if success:
                    df.at[jobid,'SO_model_R2']=r_squared
                    df.at[jobid,'SO_model_C']=None
                    df.at[jobid,'SO_model_H']=H
                else:
                    df.at[jobid,'SO_model_R2']=None
                    df.at[jobid,'SO_model_C']=None
                    df.at[jobid,'SO_model_H']=None
                success,r_squared,C,H, ts,X, first_index,last_index =
        fit_curing_profile_with_model(job,'SAFO')
                if success:
                    df.at[jobid,'SAFO_model_R2']=r_squared
                    df.at[jobid,'SAFO_model_C']=C
                    df.at[jobid,'SAFO_model_H']=H
                else:
                    df.at[jobid,'SAFO_model_R2']=None
                    df.at[jobid,'SAFO_model_C']=None
                    df.at[jobid,'SAFO_model_H']=None
                success,r_squared,C,H, ts,X, first_index,last_index =
        fit_curing_profile_with_model(job,'SASO')
                if success:
                    df.at[jobid,'SASO_model_R2']=r_squared
                    df.at[jobid,'SASO_model_C']=C
                    df.at[jobid,'SASO_model_H']=H
                else:
                    df.at[jobid,'SASO_model_R2']=None
                    df.at[jobid,'SASO_model_C']=None
                    df.at[jobid,'SASO_model_H']=None
```

```
559/560
```

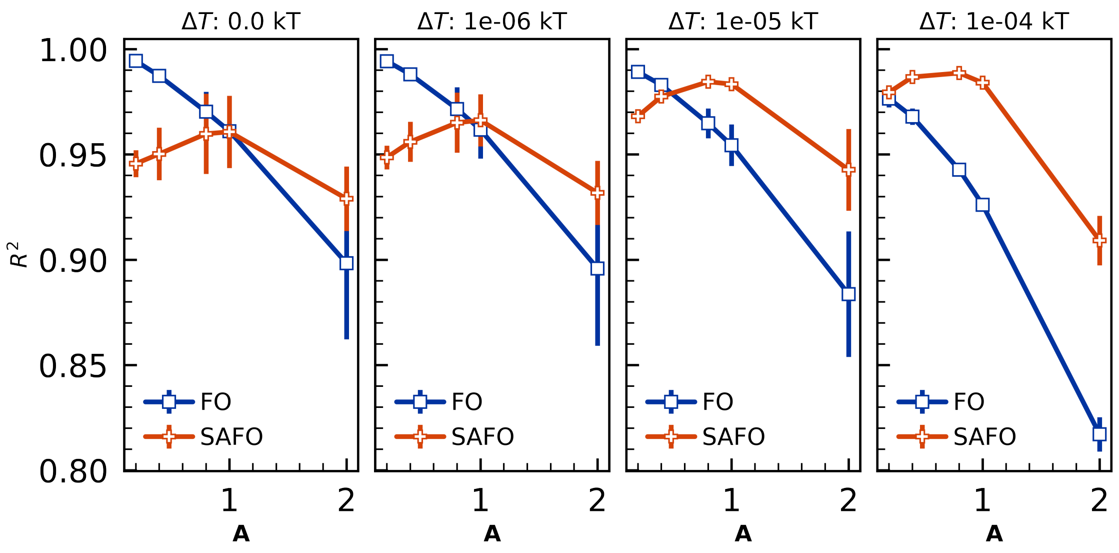
```
In [6]: from scipy import stats
        fig, axs = plt.subplots(1,4,sharey=True, dpi=600, figsize=(10,5))
        #axs = axs.ravel()
        df_filtered = df[(df.n_particles==50000)&
                         (df.trial==0)&
                         #(df.kT==0.5)&
                         ((df.kT==0.5)|(df.kT==1.0)|(df.kT==2.0)|(df.kT==4.0)|(df.kT==6.0))&
                         #(df.deltaT==1e-05)&
                         #(df.deltaT==0.0)&
                         (df.sec_bond_weight==2.0)&
                         (df.t_Final==6000000.0)&
                         (df.A<=2)&
                         (df.A>0.1)]
        df_sorted = df_filtered.sort_values('A')
        models = ['SAFO','SASO']#['FO','SAFO','SO','SASO']
        for i,(deltaT,df_deltaT) in enumerate(df_sorted.groupby('deltaT')):
            group_agg_mean = df_deltaT.groupby(['A']).aggregate(['mean','sem'])

            colors = itertools.cycle(["royalblue", "g", "orange", "r"])
            markers = itertools.cycle(["s", "P", "D", "H"])

            color = next(colors)
            marker = next(markers)
            if color is 'royalblue':
                facecolor='royalblue'
            else:
                facecolor='white'
            if True:
                #print("dt", deltaT)
                #print("mean", group_agg_mean.FO_model_R2['mean'].values.tolist())
                #print("sem", group_agg_mean.FO_model_R2['sem'].values.tolist())
                axs[i].errorbar(group_agg_mean.index,
                             group_agg_mean.FO_model_R2['mean'],
                             group_agg_mean.FO_model_R2['sem'],
                             color=BSU_BLUE,
                             marker=marker,
                             markeredgewidth=1,
                             markerfacecolor="white",
                             markersize=8,
                             #linestyle='--',
                             markeredgecolor=BSU_BLUE,
                             label='FO')

            color = next(colors)
            marker = next(markers)
            if color is 'royalblue':
                facecolor='royalblue'
            else:
                facecolor='white'
            if True:
                print("dt", deltaT)
                print(group_agg_mean.SAFO_model_R2['mean'])
                print("mean", group_agg_mean.SAFO_model_R2['mean'].values.tolist())
                print("sem", group_agg_mean.SAFO_model_R2['sem'].values.tolist())
                axs[i].errorbar(group_agg_mean.index,
                             group_agg_mean.SAFO_model_R2['mean'],
                             group_agg_mean.SAFO_model_R2['sem'],
                             color=BSU_ORANGE,
                             marker=marker,
                             markeredgewidth=1,
                             markerfacecolor="white",
                             markersize=8,
                             #linestyle='--',
                             markeredgecolor=BSU_ORANGE,
                             label='SAFO')

            color = next(colors)
            marker = next(markers)
            if color is 'royalblue':
                facecolor='royalblue'
            else:
                facecolor='white'
            if False:
                axs[i].errorbar(group_agg_mean.index,
                             group_agg_mean.SO_model_R2['mean'],
                             group_agg_mean.SO_model_R2['sem'],
                             color=color,
                             marker=marker,
                             markeredgewidth=1,
                             markerfacecolor=facecolor,
                             markersize=8,
                             #linestyle='--',
                             markeredgecolor=color,
                             label='SO')

            color = next(colors)
            marker = next(markers)
            if color is 'royalblue':
                facecolor='royalblue'
            else:
                facecolor='white'
            if False:
                axs[i].errorbar(group_agg_mean.index,
                             group_agg_mean.SASO_model_R2['mean'],
                             group_agg_mean.SASO_model_R2['sem'],
                             color=color,
                             marker=marker,
                             markeredgewidth=1,
                             markerfacecolor=facecolor,
                             markersize=8,
                             #linestyle='--',
                             markeredgecolor=color,
                             label='SASO')

            if deltaT==0:
                axs[i].set_title('$\Delta T$: {} kT'.format(deltaT),fontsize=15)
            else:
```

```
                axs[i].set_title('$\Delta T$: {:.0e} kT'.format(deltaT),fontsize=15)
            axs[i].legend(fontsize=15, loc="lower left")
            axs[i].set_xlabel('A')

            #plt.yscale('log')
            #plt.ylim(0.3,1.1)
            axs[i].set_xlim(0.1,2.1)
        axs[0].set_ylabel('$R^2$')
        plt.tight_layout()
        plt.savefig("all_delta.png", transparent=True, dpi=300)
        plt.show()
```

```
<>:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:46: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:69: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:88: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:24: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:46: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:69: SyntaxWarning: "is" with a literal. Did you mean "=="?
<>:88: SyntaxWarning: "is" with a literal. Did you mean "=="?
<ipython-input-6-11672b461b57>:24: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if color is 'royalblue':
<ipython-input-6-11672b461b57>:46: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if color is 'royalblue':
<ipython-input-6-11672b461b57>:69: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if color is 'royalblue':
<ipython-input-6-11672b461b57>:88: SyntaxWarning: "is" with a literal. Did you mean
"=="?
  if color is 'royalblue':
```

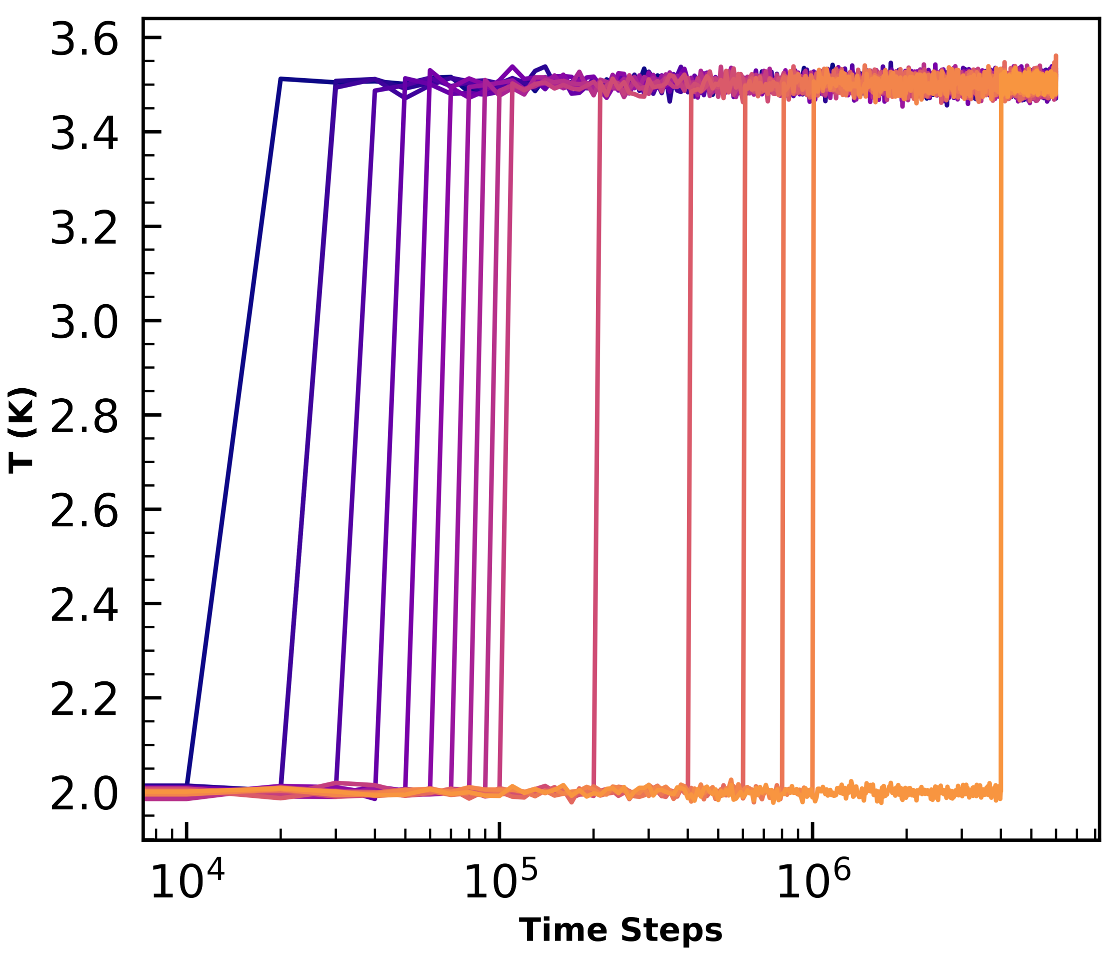
```
dt 0.0
A
0.2    0.945619
0.4    0.950254
0.8    0.959808
1.0    0.960730
2.0    0.928995
Name: mean, dtype: float64
mean [0.9456191030177226, 0.950254288906609, 0.9598076788566487, 0.9607301037624485,
0.9289951020475877]
sem [0.006392732210594506, 0.012495682509545891, 0.019040485607138315,
0.017164671170118294, 0.015309757691478826]
dt 1e-06
A
0.2    0.948591
0.4    0.956021
0.8    0.965094
1.0    0.966229
2.0    0.931752
Name: mean, dtype: float64
mean [0.9485914887891203, 0.956020547671699, 0.9650941367218113, 0.9662292362702919,
0.9317522846651322]
sem [0.005625036624910592, 0.009481049361207005, 0.014276453108781284,
0.012399245208402227, 0.015227730575042836]
dt 1e-05
A
```

```
0.2    0.968161
0.4    0.977488
0.8    0.984524
1.0    0.983401
2.0    0.942708
Name: mean, dtype: float64
mean [0.9681606710891606, 0.9774884845930968, 0.9845240898188369, 0.9834009700698028,
0.942708220784915]
sem [0.002501991892812254, 0.0023519309457531094, 0.0014005409206293497,
0.0013345367141796738, 0.01939687875342723]
dt 0.0001
A
0.2    0.979476
0.4    0.986789
0.8    0.988663
1.0    0.984054
2.0    0.909132
Name: mean, dtype: float64
mean [0.9794764769613133, 0.9867889115270978, 0.9886633811268632, 0.9840540474306009,
0.9091319649266504]
sem [0.002287830544337261, 0.0013229661902640715, 0.001203595920372104,
0.002034615696425728, 0.011740217875926431]
```



```
In [ ]:
```

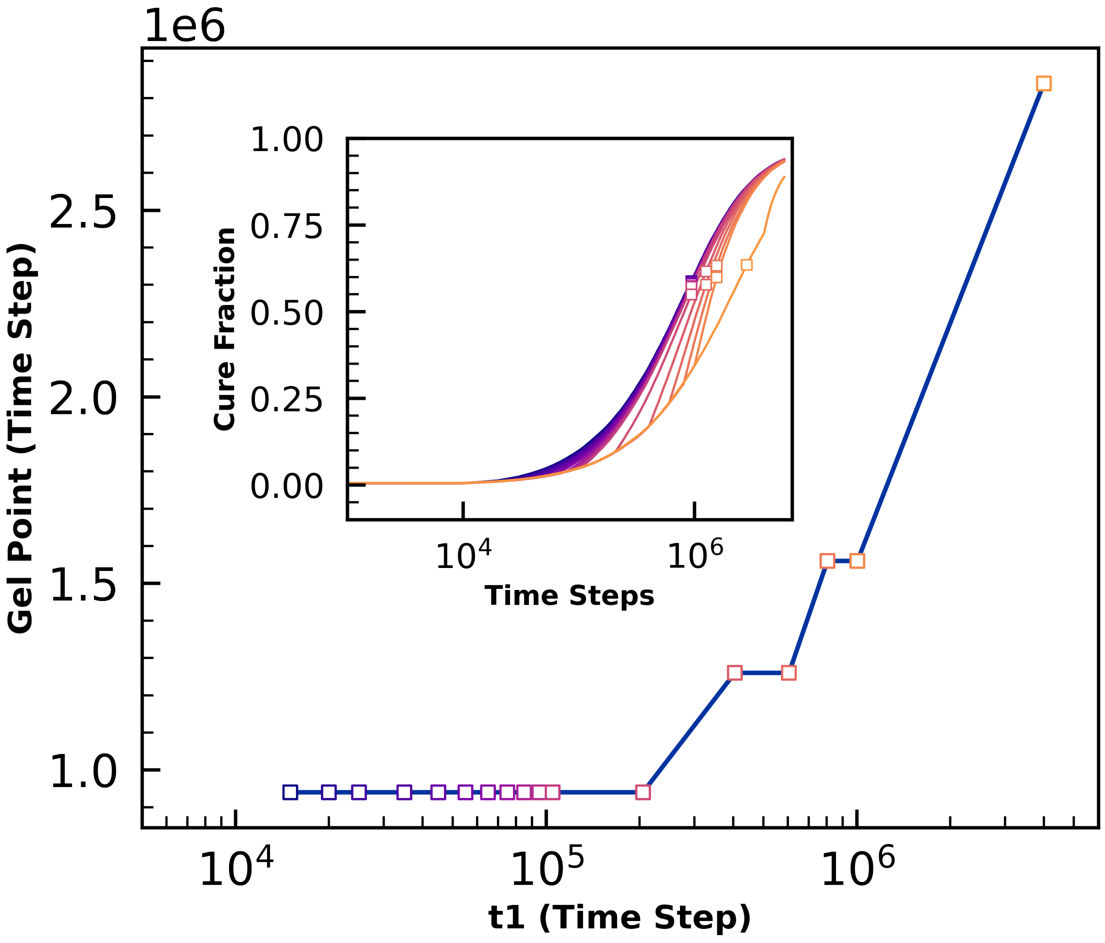
```
In [7]: data_path = '/home/sthomas/projects/sensitivity_analysis'
        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"

In [2]: import os
        os.environ['MATPLOTLIBRC'] = "../../matplotlibrc"
        import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib
        %matplotlib inline
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools
        from common import *

        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        def init_project():
            df_index = pd.DataFrame(project.index())
            df_index = df_index.set_index(['_id'])
            statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
            #print(statepoints)
            df = pd.DataFrame(statepoints).T.join(df_index)
            df = df.sort_values('T')
            return df
        project = signac.get_project(data_path)
        df = init_project()

In [3]: df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                         (df.profile=='ramp_up_and_down')]
        df['t1'] = df_filtered['temp_prof'].str[1].str[0]
        df['t2'] = df_filtered['temp_prof'].str[3].str[0]

In [4]: df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                         (df.profile=='step_SA')]
        df['t1'] = df_filtered['temp_prof'].str[1].str[0]
        df_sorted = df.sort_values('t1')

In [5]: fig = plt.figure(dpi=600, figsize=(7, 6))
        ax1 = fig.add_subplot(111)
        lines=[]
        time_conversion = 1.05e-11 #s
        distance_conversion = 1.06 #nm
        TemperatureConversion = 365.01 #K
        df_filtered = df[(df.activation_energy==3.0)&
                        (df.stop_after_percent==100.0)&
                        (df.profile=='step_SA')]
        colors = plt.cm.plasma(np.linspace(0,0.75,len(df_filtered.groupby('t1'))))
        for i,(key,dfgrp) in enumerate(df_filtered.groupby('t1')):
            for jobid in dfgrp.index:
                job=project.open_job(id=jobid)
                data = np.genfromtxt(job.fn('out.log'),names=True)
                ax1.plot(data['timestep'],
                         data['temperature'],#*TemperatureConversion,
                         linewidth=2.0,
                        color=colors[i])
        ax1.set_xlabel('Time Steps')
        ax1.set_ylabel('T (K)')
        plt.xscale('log')
        plt.savefig("temperature_profiles.png", transparent=True)
        plt.show()
```



```
In [10]: fig = plt.figure(dpi=600, figsize=(7, 6))
         ax1 = fig.add_subplot(111)
         gel_points=[]
         ramp_up_times=[]
         # These are in unitless percentages of the figure size. (0,0 is bottom left)

         df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                          (df.profile=='step_SA')]
         colors = plt.cm.plasma(np.linspace(0,0.75,len(df_filtered.groupby('t1'))))
         for i,(key,dfgrp) in enumerate(df_filtered.groupby('t1')):
             for jobid in dfgrp.index:
                 job=project.open_job(id=jobid)
                 x=job.document['gel_point']
                 gel_points.append(x)
                 ramp_up_times.append(key)
                 print(key,job.document['gel_point'])
                 ax1.scatter(key,
                             job.document['gel_point'],
                            marker='s',
                             facecolor='w',
                             linewidth=1.0,
                             color=colors[i],
                            zorder=1)
         ax1.plot(ramp_up_times,
                  gel_points,
                  color=BSU_BLUE,
                  linewidth=2.0,
                 label='Gel Point',
                 zorder=0)
         #print(ramp_up_times)
         ax1.ticklabel_format(axis='y', style='sci', scilimits=(-2,2))
         ax1.set_xlabel('t1 (Time Step)')
         ax1.set_ylabel('Gel Point (Time Step)')
         ax1.set_xscale('log')
         ax1.set_xlim(5e3,6e6)

         left, bottom, width, height = [0.32, 0.45, 0.39, 0.39]
         ax2 = fig.add_axes([left, bottom, width, height])
         colors = plt.cm.plasma(np.linspace(0,0.75,len(df_filtered.groupby('t1'))))
         for i,(key,dfgrp) in enumerate(df_filtered.groupby('t1')):
             for jobid in dfgrp.index:
                 job=project.open_job(id=jobid)
                 data = np.genfromtxt(job.fn('out.log'),names=True)
                 ax2.plot(data['timestep'],
                          data['bond_percentAB']/100,
                          label='t1: {:.1e}'.format(key),
                          color=colors[i],
                          linewidth=1.0,
                         zorder=0)
                 ax2.scatter(job.document['gel_point'],
                            job.document['curing_at_gel_point']/100,
                             marker='s',
                             facecolor='w',
                             linewidth=0.7,
                             s=20,
                             color=colors[i],
                            zorder=1)
         ax2.tick_params(axis = 'both', which = 'major', labelsize = 15)
         ax2.set_ylim(-0.1,1)
         ax2.set_xlim(1e3,7e6)
         ax2.set_xlabel('Time Steps',fontsize=12.0)
         ax2.set_ylabel('Cure Fraction',fontsize=12.0)
         ax2.set_xscale('log')
         plt.savefig("gel_points.png", transparent=True)
         plt.show()
```

```
15000.0 940000.0
20000.0 940000.0
25000.0 940000.0
35000.0 940000.0
45000.0 940000.0
55000.0 940000.0
65000.0 940000.0
75000.0 940000.0
85000.0 940000.0
95000.0 940000.0
105000.0 940000.0
205000.0 940000.0
405000.0 1260000.0
605000.0 1260000.0
805000.0 1560000.0
1005000.0 1560000.0
4005000.0 2840000.0


<ipython-input-10-c6f62127bb6b>:65: UserWarning: This figure includes Axes that are
not compatible with tight_layout, so results might be incorrect.
  plt.savefig("gel_points.png", transparent=True)
/home/mikehenry/miniconda3/envs/tg-plots/lib/python3.8/site-
```

```
packages/IPython/core/pylabtools.py:132: UserWarning: This figure includes Axes that
are not compatible with tight_layout, so results might be incorrect.
  fig.canvas.print_figure(bytes_io, **kw)
```



```
In [ ]:
```

```
In [11]: data_path = '/home/sthomas/projects/sensitivity_analysis'
         BSU_BLUE = "#0033A0"
         BSU_ORANGE = "#D64309"
```

```
In [2]: import os
        os.environ['MATPLOTLIBRC'] = "../../matplotlibrc"
        import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib
        %matplotlib inline
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools
        from common import *

        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        def init_project():
            df_index = pd.DataFrame(project.index())
            df_index = df_index.set_index(['_id'])
            statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
            #print(statepoints)
            df = pd.DataFrame(statepoints).T.join(df_index)
            df = df.sort_values('T')
            return df
        project = signac.get_project(data_path)
        df = init_project()
```

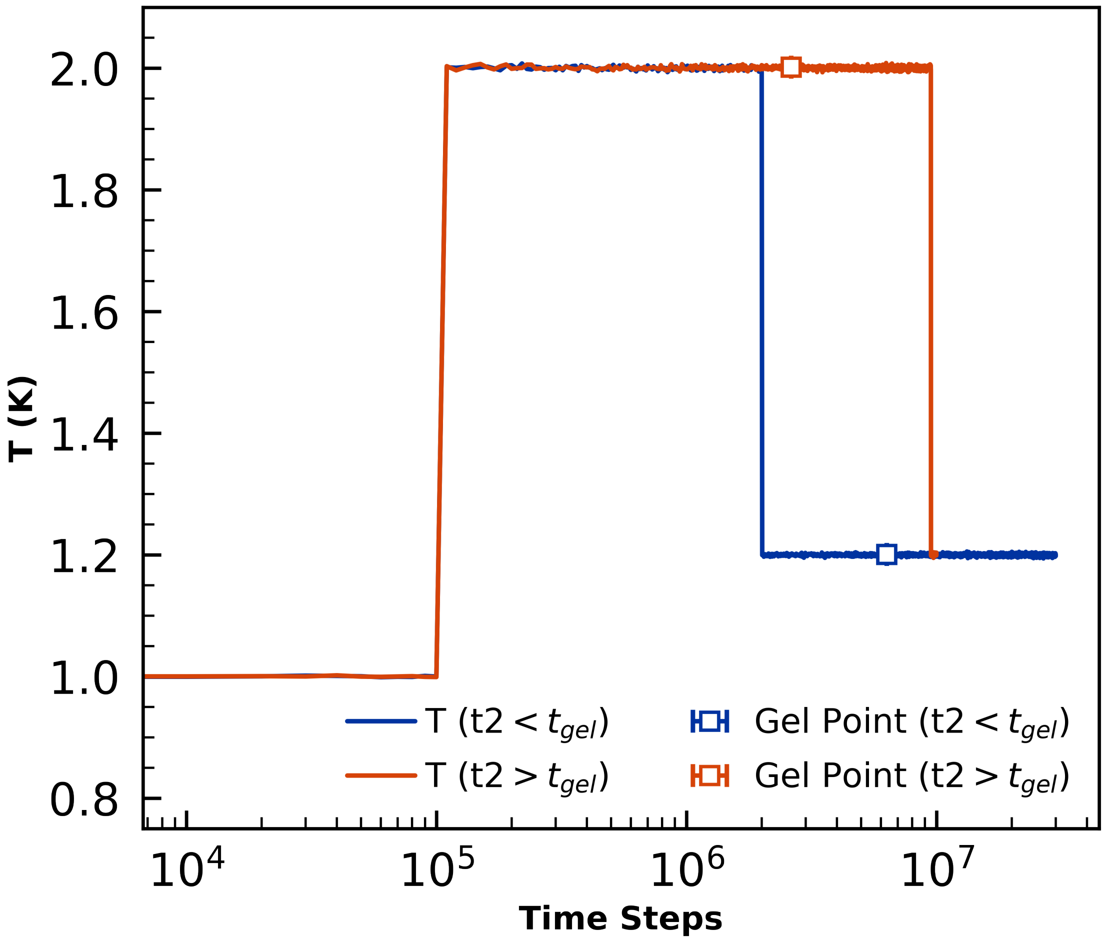
```
In [3]: df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                         (df.profile=='step_SA')]
        df['t1'] = df_filtered['temp_prof'].str[1].str[0]
```

```
In [4]: df_filtered = df[(df.stop_after_percent==100.0)&
                         (df.profile=='ramp_up_and_down')]
        df['t1'] = df_filtered['temp_prof'].str[1].str[0]
        df['t2'] = df_filtered['temp_prof'].str[3].str[0]
        df['T2'] = df_filtered['temp_prof'].str[4].str[1]
        df['t_SetT'] = df['t2']
```

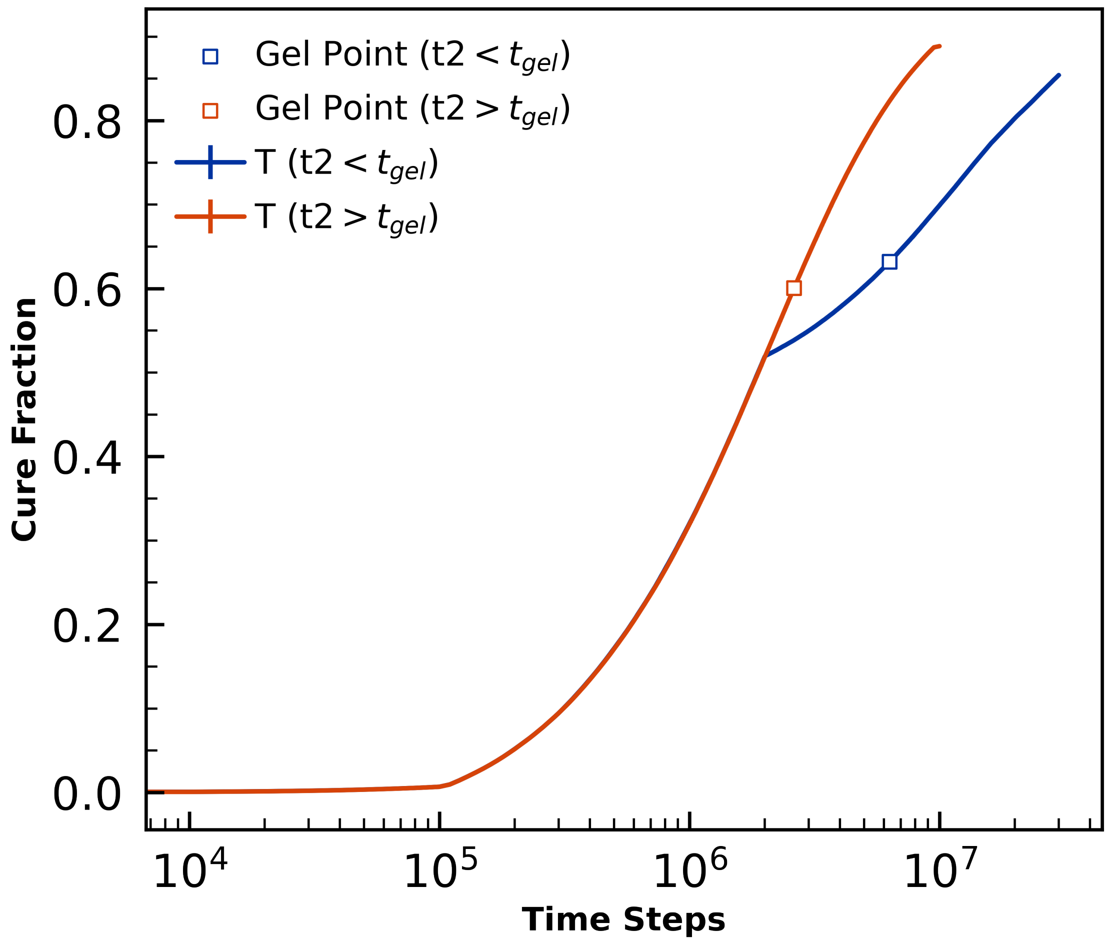
```
In [21]: import matplotlib.patheffects as pe
         fig = plt.figure(dpi=600, figsize=(7, 6))
         ax1 = fig.add_subplot(111)
         lines=[]
         time_conversion = 1.05e-11 #s
         distance_conversion = 1.06 #nm
         TemperatureConversion = 365.01 #K
         df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                          (df.t1==1.05e5)&
                          (df.E_factor==1.0)&
                          (df.T2==1.2)&
                          (((df.t2==2005001)&(df.t_Final==3e7))|
                           ((df.t2==9505001)&(df.t_Final==1.0e7)))&
                          (df.n_particles==4e5)&
                          (df.profile=='ramp_up_and_down')]
         df_sorted = df_filtered.sort_values('t2')
         colors = [BSU_BLUE,BSU_ORANGE]
         #plt.cm.plasma(np.linspace(0,0.75,len(df_sorted.groupby('t2'))))
         for i,(key,dfgrp) in enumerate(df_sorted.groupby('t2')):
             for jobid in dfgrp.index:
                 job=project.open_job(id=jobid)
```

```
        if job.isfile('out.log') and ('gel_point' in job.document) :
            #print(job,key)
            data = np.genfromtxt(job.fn('out.log'),names=True)
            timesteps = data['timestep']
            temperature = data['temperature']
            pops=[]
            pops = [ temp_i+1 for temp_i, (x, y) in
enumerate(zip(timesteps,timesteps[1:])) if x>=y]
            if len(pops)>0:
                #print(len(pops),pops)
                timesteps = np.asarray([x for i,x in enumerate(timesteps) if i not in
pops])
                temperature   = np.asarray([x for i,x in enumerate(temperature) if i not
in pops])
            temperature = temperature#*TemperatureConversion
            gel_point = job.document['gel_point']
            T_at_gel = temperature[np.isclose(timesteps,gel_point)]
            #print(T_at_gel)
            if job.sp.trial==1:
                if key<gel_point:
                    label='T (t2'+'$< t_{gel}$)'
                else:
                    label='T (t2'+'$> t_{gel}$)'
                print(i)
                ax1.plot(timesteps,
                         temperature,
                         linewidth=2.0,
                        color=colors[i],
                         label=label)
                        #label='T (t2 : {:.1e})'.format(key))

            if False:
                ax1.axvline(x=job.document['gel_point'],
                           color=colors[i],
                           linestyle='--',
                           linewidth=1.0,
                           label='Gel Point')
    if True:
        if key<gel_point:
            label='Gel Point (t2'+'$< t_{gel}$)'
        else:
            label='Gel Point (t2'+'$> t_{gel}$)'
        ax1.errorbar(x=dfgrp.gel_point.mean(),
                    y=T_at_gel,#400,
                    xerr=dfgrp.gel_point.std(),
                     label=label,#'$t_{gel}$'+' (t2 : {:.1e})'.format(key),
                     color=colors[i],#'r',
                     fmt='s',
                     markerfacecolor='w',
                     lw=3,
                     ms=8,
                     capthick=2,
                     elinewidth=2,
                     capsize=5,
                    zorder=2)
if False:
    ax1.axvline(x=df_filtered.gel_point.mean(),
                color='r',
                linestyle='--',
                linewidth=4.0,
                path_effects=[pe.Stroke(linewidth=5, foreground='k'), pe.Normal()])
    ax1.errorbar(x=df_filtered.gel_point.mean(),
                y=300,
                xerr=df_filtered.gel_point.std(),
                 label='Gel Point',
                 color='r',
                 fmt='o',
                 markerfacecolor='w',
                 markeredgecolor='r',
                 lw=3,
                 ms=8,
                 capthick=2,
                 elinewidth=3,
                 capsize=5,
                zorder=2)

ax1.set_xlabel('Time Steps')
ax1.set_ylabel('T (K)')
ax1.set_ylim(0.75,2.1)
#ax1.set_xlim(0.0,2.2)

#ax1.set_ylim(200,800)
#ax1.set_xlim(5e4,7e6)
plt.legend(fontsize=15,loc='lower right',ncol=2)
plt.xscale('log')
plt.savefig("temperature_profiles.png", transparent=True)
plt.show()
```

```
0
1
```



```
In [22]: fig = plt.figure(dpi=600, figsize=(7, 6))
         ax1 = fig.add_subplot(111)
         lines=[]
         df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                          (df.t1==1.05e5)&
                          (df.E_factor==1.0)&
                          (df.T2==1.2)&
                          (((df.t2==2005001)&(df.t_Final==3e7))|
                           ((df.t2==9505001)&(df.t_Final==1.0e7)))&
                          (df.n_particles==4e5)&
                          (df.profile=='ramp_up_and_down')]
         df_sorted = df_filtered.sort_values('t2')
         colors = colors = [BSU_BLUE,BSU_ORANGE] #
         plt.cm.plasma(np.linspace(0,0.75,len(df_sorted.groupby('t2'))))
         for i,(key,dfgrp) in enumerate(df_sorted.groupby('t2')):
             times=[]
             cure_fractions=[]
             gel_points=[]
             cure_at_gel=[]
             #print(dfgrp.t_Final)
             for jobid in dfgrp.index:
                 job=project.open_job(id=jobid)
                 if job.isfile('out.log') and ('gel_point' in job.document) :
                     data = np.genfromtxt(job.fn('out.log'),names=True)
                     timesteps = data['timestep']
                     cure_fraction = data['bond_percentAB']/100
                     pops = [ i for i, (x, y) in enumerate(zip(timesteps[:-1],timesteps[1:])) if
         x>=y]
                     if len(pops)>0:
                         popi=pops[0]+1
                         timesteps = np.asarray([x for i,x in enumerate(timesteps) if i!=popi])
                         cure_fraction = np.asarray([x for i,x in enumerate(cure_fraction) if
         i!=popi])

                     times.append(timesteps)
                     cure_fractions.append(cure_fraction)
                     gel_points.append(job.document['gel_point'])
                     cure_at_gel.append(job.document['curing_at_gel_point']/100)
             if key<gel_point:
                 label='T (t2'+'$< t_{gel}$)'
             else:
                 label='T (t2'+'$> t_{gel}$)'
             ax1.errorbar(np.mean(times,axis=0),
                          np.mean(cure_fractions,axis=0),
                          np.std(cure_fractions,axis=0),
                          label=label,#'t2: {:.1e}'.format(key),
                          color=colors[i],
                          linewidth=2.0,
                          zorder=0)
             if key<gel_point:
                 label='Gel Point (t2'+'$< t_{gel}$)'
             else:
                 label='Gel Point (t2'+'$> t_{gel}$)'
             ax1.scatter(np.mean(gel_points),
                        np.mean(cure_at_gel),
                         marker='s',
                         facecolor='w',
                         linewidth=1.0,
                         color=colors[i],
                        zorder=1,
                        label=label)#'Gel Point({:.1e})'.format(key))
         ax1.set_xlabel('Time Steps')
         ax1.set_ylabel('Cure Fraction')
         plt.legend(fontsize=15)
         plt.xscale('log')
         plt.savefig("cure_profiles.png", transparent=True)
         plt.show()
```

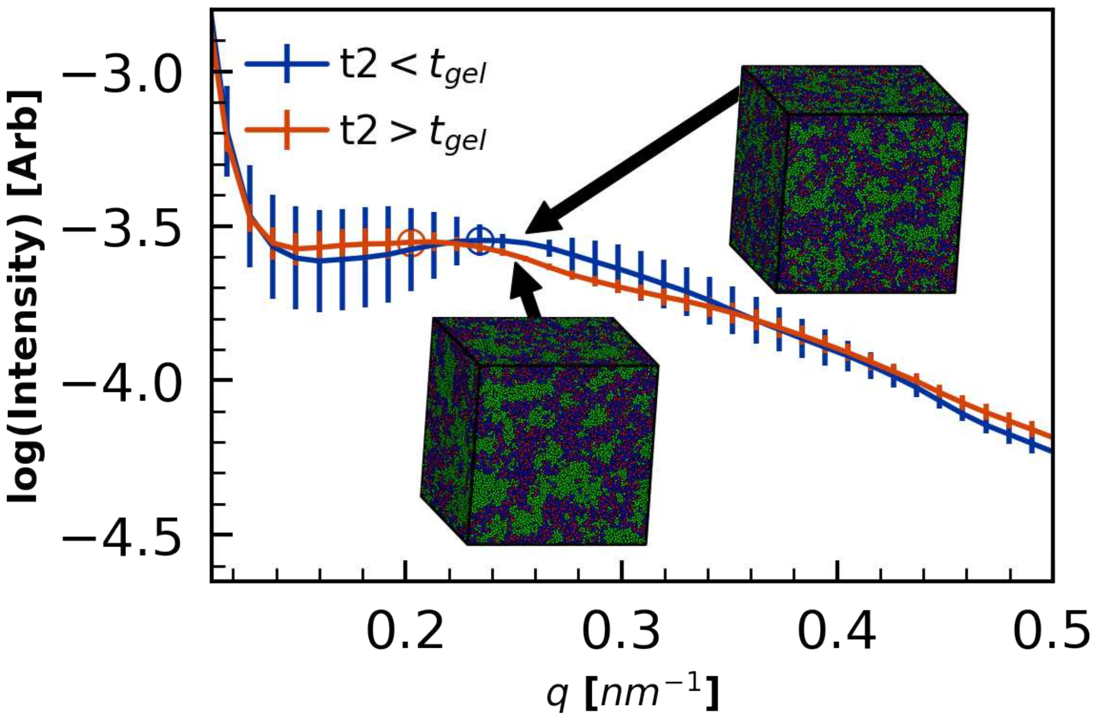


```
In [23]: from scipy import stats
         from scipy import interpolate
         fig = plt.figure(dpi=200)
         ax1 = fig.add_subplot(111)
         left, bottom, width, height = [0.34, 0.26, 0.3, 0.3]#[0.6, 0.57, 0.3, 0.3]#max t2
         ax2 = fig.add_axes([left, bottom, width, height])
         ax2.axis('off')
         left, bottom, width, height = [0.61, 0.59, 0.3, 0.3]#[0.23, 0.27, 0.3, 0.3]#min t2
         ax3 = fig.add_axes([left, bottom, width, height])
         ax3.axis('off')
         df_filtered = df[(df.activation_energy==3.0)&
                          (df.stop_after_percent==100.0)&
                          (df.t1==1.05e5)&
                          (df.E_factor==1.0)&
                          (df.T2==1.2)&
                          (((df.t2==2005001)&(df.t_Final==3e7))|
                           ((df.t2==9505001)&(df.t_Final==1.0e7)))&
                          (df.n_particles==4e5)&
                          (df.profile=='ramp_up_and_down')]
         #print(df_filtered.profile)
         df_sorted = df_filtered.sort_values('t2')
         print(df_sorted.t2.values)
         print('min t2',df_sorted.t2[3])#.min())
         print('max t2',df_sorted.t2.max())
         t2_min = 2005001#df_sorted.t2[3]#.min()
```

```
t2_max = 9505001#df_sorted.t2.max()#df_sorted.t2.max()
colors = colors = [BSU_BLUE,BSU_ORANGE]
#plt.cm.plasma(np.linspace(0,0.75,len(df_sorted.groupby('t2'))))

df_gp = df_sorted.groupby('t2')
for i,(key,df_grp) in enumerate(df_gp):
    qs_all_trials = []
    Is_all_trials = []
    box_len = df_grp.Lx.mean()
    cure_percent = df_grp.cure_percent.mean()
    #print('cure_percent',cure_percent)
    if key ==t2_max  or key==t2_min:
        #print('key',key)
        for jobid in df_grp.index:
            job=project.open_job(id=jobid)
            if job.isfile('out.log') and ('gel_point' in job.document) :
                #print(job,key)
                gel_point = job.document['gel_point']
A=job.sp.num_a*job.sp.n_mul*4*job.sp.percent_bonds_per_step/100/job.sp.bond_period
                if job.isfile('diffract_type_2/asq.txt'):
                    data = np.genfromtxt(job.fn('diffract_type_2/asq.txt'))
                    qs=data[:,0]
                    Is=data[:,1]
                    qs_all_trials.append(qs)
                    Is_all_trials.append(Is)
                    if job.isfile('final_snapshot.png'):
                        im = plt.imread(job.fn('final_snapshot.png'))
                        if key ==t2_max:
                            #ax2.set_title('t2 : {:.1e} $\\Delta$
t'.format(key),fontsize=15)
                            ax2.imshow(im)
                        if key ==t2_min:
                            #ax3.set_title('t2 : {:.1e} $\\Delta$
t'.format(key),fontsize=15)
                            ax3.imshow(im)
                    #print('Is',Is)
                else:
                    raise FileNotFoundError('Cannot find diffract_type_2/asq.txt
for'+job)
            else:
                print(job,' not completed!')
    if len(Is_all_trials)>0:
        mean_qs = np.mean(qs_all_trials,axis=0)
        mean_Is = np.mean(Is_all_trials,axis=0)
        if key<gel_point:
            label='t2'+'$< t_{gel}$'
        else:
            label='t2'+'$> t_{gel}$'
        ax1.errorbar(mean_qs,
                     mean_Is,
                     yerr=stats.sem(Is_all_trials,axis=0),
                     linewidth=2.0,
                     zorder=1,
                     color=colors[i],
                 label=label)#'t2 : {:.1e} $\\Delta$ t'.format(key,
                   #
cure_percent/100))
        if True:
            qms=[]
            Ims=[]
            for qs_t,Is_t in zip(qs_all_trials,Is_all_trials):
                first_peak_q,first_peak_i = get_highest_maxima(box_len,qs_t,Is_t)
                qms.append(first_peak_q)
                Ims.append(first_peak_i)
            #print(qms)
            fn = interpolate.interp1d(mean_qs,mean_Is,kind='cubic')
            first_peak_q=np.mean(qms)
            first_peak_i=fn(first_peak_q)
        else:
            first_peak_q,first_peak_i = get_highest_maxima(box_len,
                                                           mean_qs,
                                                           nmean_Is)
        print(first_peak_q,first_peak_i)
        if first_peak_q is None:
            #print(q_half_length)
            fn = interpolate.interp1d(qs,Is,kind='cubic')
            first_peak_q=q_half_length
            first_peak_i=fn(first_peak_q)
        if first_peak_q >0.8:
            fn = interpolate.interp1d(qs,Is,kind='cubic')
            first_peak_q=q_half_length
            first_peak_i=0#fn(first_peak_q)
        ax1.scatter(first_peak_q,
               first_peak_i,
               color=colors[i],
                zorder=0,
                    s=100,
                    facecolor='w',
                marker='o')

ax1.set_xlabel(r"$q$ [$nm^{-1}$]")
ax1.set_ylabel("log(Intensity) [Arb]")
ax1.set_xlim(0.11,0.5)
ax1.set_ylim(-4.65,-2.8)
ax1.legend(fontsize=15,loc='upper left')
ax1.annotate('', xy=(0.25, -3.6), xytext=(0.27, -4.0),arrowprops=dict(facecolor='black',
shrink=0.05))
ax1.annotate('', xy=(0.25, -3.55), xytext=(0.37, -3.),arrowprops=dict(facecolor='black',
shrink=0.05))
plt.savefig("sf_vertical.png", transparent=True)
```

```
[2005001. 2005001. 2005001. 9505001. 9505001. 9505001.]
min t2 9505001.0
max t2 9505001.0
a52a2aacf366f4bfdb526c3c90a6e553  not completed!
0.23446066894915651 -3.5471132590244476
0.20248875954699877 -3.5527970868063505
```

```
<ipython-input-23-30ae73c22877>:118: UserWarning: This figure includes Axes that are
not compatible with tight_layout, so results might be incorrect.
  plt.savefig("sf_vertical.png", transparent=True)
/home/mikehenry/miniconda3/envs/tg-plots/lib/python3.8/site-
packages/IPython/core/pylabtools.py:132: UserWarning: This figure includes Axes that
are not compatible with tight_layout, so results might be incorrect.
  fig.canvas.print_figure(bytes_io, **kw)
```



```
In [ ]:
```

```
In [1]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"
        from common import *

In [2]: data_path = "/home/sthomas/projects/LJ_System_Size"
        import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import pandas as pd
        import matplotlib
        %matplotlib inline
        from scipy.signal import argrelextrema as argex
        import matplotlib.cm as cm
        import itertools
        import os
        #import structure_factor as sf
        import math
        from scipy import interpolate
        import gsd
        import gsd.fl
        import gsd.hoomd

        from matplotlib import cm
        from mpl_toolkits.mplot3d import Axes3D



        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        project = signac.get_project(data_path)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(['_id'])
        statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
        df = pd.DataFrame(statepoints).T.join(df_index)
        df = df.sort_values('T')
        #df.head()

In [3]: typeId=2
        n_views=40
        grid_size=512
        gammas = [4.5]#,18,36,72]#[4.5,9,18,36,45,52,72]
        colors = itertools.cycle(cm.rainbow(np.linspace(0, 1, len(gammas))))
        for gamma in gammas:
            df_filtered = df[(df.n_particles==1000000)&
                              (df.t_Final==1e7)&
                                (df.trial==0)]
                              # (df.t_Final==6e6)]


            #print(df_filtered)
            #grpedByGamma = df_filtered.groupby('profile')#.apply(lambda x: x.sort_values('T'))
            times_for_all_trials=[]
            qs_for_all_trials=[]
            Is_for_all_trials=[]

            qms_all=[] #qmax
            Is_all=[]
            Qs_all=[]
            times_all=[]
            cure_all=[]
            color=next(colors)
            for signac_id in df_filtered['signac_id']:
                job = project.open_job(id=signac_id)
                if 'Lx' in job.document:
                    half_box_length = job.document['Lx']/2
                else:
                    print('Lx not found in',job)

                q_half_length = 2*math.pi/(half_box_length/1.06)
                diffract_dir_pattern
        ='diffract_type_{}_n_views_{}_grid_size_{}_frame'.format(typeId,
        n_views,
        grid_size)
                directories = os.listdir(job.workspace())
                directories = [d for d in os.listdir(job.workspace()) if
        d.startswith(diffract_dir_pattern)]
                directories.sort(key = lambda x: int(x.split('_')[-1]))
                #print(len(directories))
                #print(directories)
                num_frames = len(directories)

                qs_for_all_times=[]
                Is_for_all_times=[]
                times_for_all_times=[]
                qs_list = []
                times_list = []
                Is_list = []
                Qs_list=[]
                for i,diffract_dir in enumerate(directories):
                    print("Progress {:2.1%}".format(i / num_frames), end="\r")
                    if diffract_dir.startswith(diffract_dir_pattern):
                        frame = int(diffract_dir.split('_')[-1])
                        if frame%1==0 and frame >0e6/job.sp.dcd_write:#==119 or frame==123:#%100
        == 0:#num_frames/30:
                            if job.isfile('{}/asq.txt'.format(diffract_dir)):
                                #print(job.fn('{}/asq.txt'.format(diffract_dir)))
                                data=np.genfromtxt(job.fn('{}/asq.txt'.format(diffract_dir)))
                                time = round(frame*job.sp.dcd_write)
                                legend = '{} $\\Delta t(\Gamma:{})$'.format(time,job.sp.gamma)
                                qs = data[:,0]
                                Is = data[:,1]
                                #print(qs.shape)
                                qs_for_all_times.append(qs)
                                Is_for_all_times.append(Is)
                                times_for_all_times.append(time)

                                dq=qs[1]-qs[0]
                                Is_exp = np.exp(Is)
                                q_sq = qs**2
                                Q = np.sum(Is_exp*qs*dq)
                                Qs_list.append(Q)
                                first_peak_q,first_peak_i =
        get_highest_maxima(job.document['Lx'],qs,Is)
                                #print(first_peak_q,first_peak_i)
                                if first_peak_q is None:
                                    #print(q_half_length)
                                    fn = interpolate.interp1d(qs,Is,kind='cubic')
                                    first_peak_q=q_half_length
                                    first_peak_i=fn(first_peak_q)
                                if first_peak_q >0.8:# and time > 2.0e5:
                                    fn = interpolate.interp1d(qs,Is,kind='cubic')
                                    first_peak_q=q_half_length
                                    first_peak_i=0#fn(first_peak_q)
                                qs_list.append(first_peak_q)
                                times_list.append(time)
                                Is_list.append(first_peak_i)
                            else:
                                print(job,'did not contain diffraction data in ',diffract_dir)

                qs_for_all_trials.append(qs_for_all_times)#this is to plot the S(q)
```

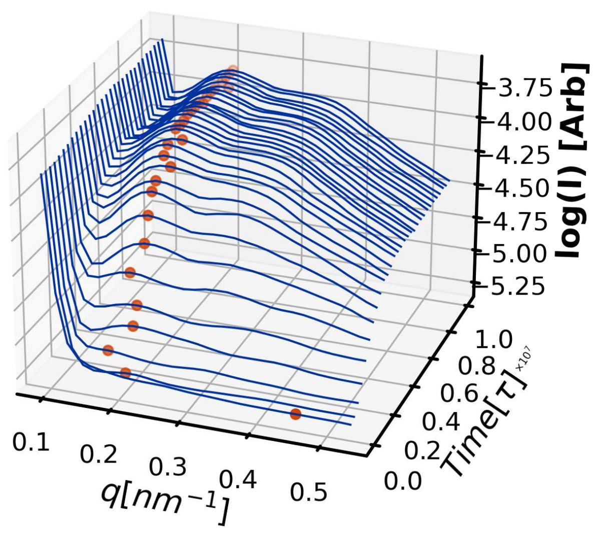
```
            Is_for_all_trials.append(Is_for_all_times)
            times_for_all_trials.append(times_for_all_times)

            qms_all.append(np.asarray(qs_list)) #this is to plot q_max
            Is_all.append(np.asarray(Is_list))
            Qs_all.append(np.asarray(Qs_list))
            log_data = np.genfromtxt(job.fn('out.log'))
            times = log_data[:,0]#/(job.sp.dt*job.sp.dcd_write)
            cure = log_data[:,9]
            times_all.append(times)
            cure_all.append(cure)

        #print(qs_all)
        qs_for_all_trials=np.asarray(qs_for_all_trials)
        Is_for_all_trials=np.asarray(Is_for_all_trials)
        times_for_all_trials=np.asarray(times_for_all_trials)
        q_mean = np.mean(qs_for_all_trials,axis=0)
        I_mean = np.mean(Is_for_all_trials,axis=0)
        time_mean= np.mean(times_for_all_trials,axis=0)

        qms_all = np.asarray(qms_all)
        Is_all = np.asarray(Is_all)
        Qs_all = np.asarray(Qs_all)
        times_all = np.asarray(times_all)
        cure_all = np.asarray(cure_all)
        Qs_av = np.mean(Qs_all,axis=0)
        Qs_std = np.std(Qs_all,axis=0)
        #print(Is_all)
        qs_av = np.mean(qms_all,axis=0)
        qs_std = np.std(qms_all,axis=0)
        Is_av = np.mean(Is_all,axis=0)
        times_av = np.mean(times_all,axis=0)
        cure_av = np.mean(cure_all,axis=0)
        cure_std = np.std(cure_all,axis=0)
        #print(len(times_list),len(qs_av))
    t_colors = colors = plt.cm.plasma(np.linspace(0,0.75,len(q_mean)))
```

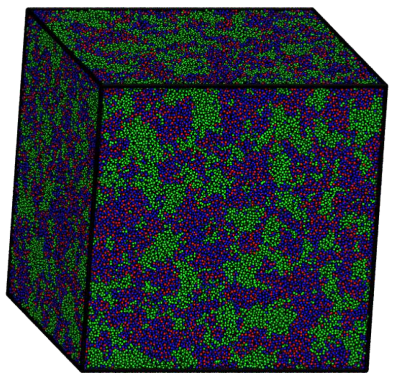
```
Progress 96.7%
```

```
In [4]: from matplotlib import rcParams
        rcParams["xtick.minor.visible"] = False
        rcParams["ytick.minor.visible"] = False

        fig = plt.figure(dpi=300)
        ax = fig.add_subplot(111, projection='3d')
        #ax.set_title('z-axis left side')
        #ax = fig.add_axes(MyAxes3D(ax, 'l'))
        left, bottom, width, height = [0.88, 0.2, 0.6, 0.6]#max t2
        ax2 = fig.add_axes([left, bottom, width, height])
        ax2.axis('off')
        qlim_max=35
        qlim_min=6
        Ilim=20
        q = q_mean[0][qlim_min:qlim_max]
        I=I_mean[:,qlim_min:qlim_max]
        #print(q)
        #print(I)
        #X,Y = np.meshgrid(q_mean[0],time_mean)
        X,Y = np.meshgrid(q,time_mean)

        Z=I
        #matplotlib.rcParams['xtick.labelsize'] = 15
        #matplotlib.rcParams['ytick.labelsize'] = 15
        #matplotlib.rcParams['xtick.major.pad'] = 2
        #matplotlib.rcParams['ytick.major.pad'] = 2
        #matplotlib.rcParams['ztick.major.pad'] = 1
        #surf = ax.plot_surface(X, Y, Z, cmap=cm.plasma,)#,rstride=1, cstride=1,linewidth=1,
        antialiased=True)
```

```
surf = ax.plot_wireframe(X, Y, Z, linewidth=1.0,zorder=0.2,rstride=1, cstride=0,
antialiased=True, color=BSU_BLUE)
view_1 = (0, 180)#back
view_2 = (25, -70)#angled
view_3 = (25, 0)#front from top
view_4 = (-5, 90)#right
view_5 = (10, -90)#angled
init_view = view_2
ax.view_init(*init_view)
ax.set_xlabel(r"$q [nm^{-1}]$", fontsize=15,rotation=150)
ax.set_ylabel(r"$Time[\tau]$",fontsize=15, rotation=150)
ax.set_zlabel("log(I) [Arb]", fontsize=15,rotation=90, labelpad=10)
#ax.ticklabel_format(style='sci', axis='y', scilimits=(0,0),
useMathText=True,rotation=0,labelsize=1)
ax.ticklabel_format(style='sci', axis='y', scilimits=(0,0), useMathText=True)
#ax.ticklabel_format(style='sci', axis='x', scilimits=(0,0), useMathText=True)
ax.yaxis.offsetText.set_fontsize(5)
ax.tick_params(axis = 'both',labelsize=12,pad=3)
#print(q_half_length,Is_av[:101])
ax.scatter3D(qs_av, time_mean,
Is_av,color=BSU_ORANGE,zorder=0.5,marker='o',antialiased=True,s=20)
if job.isfile('final_snapshot.png'):
    im = plt.imread(job.fn('final_snapshot.png'))
    #ax2.set_title('job:{}'.format(job),fontsize=8)
    ax2.imshow(im,zorder=1)
plt.savefig('morphology_evolution_1e6.png',transparent=True, bbox_inches='tight')
plt.show()
```

```
<ipython-input-4-e9c9f644d038>:51: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  plt.savefig('morphology_evolution_1e6.png',transparent=True, bbox_inches='tight')
/home/mikehenry/miniconda3/envs/tg-plots/lib/python3.8/site-
packages/IPython/core/pylabtools.py:132: UserWarning: This figure includes Axes that
are not compatible with tight_layout, so results might be incorrect.
  fig.canvas.print_figure(bytes_io, **kw)
```





In [ ]:

```
In [1]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        from common import *
        from common import MyAxes3D
        import signac
        import matplotlib.pyplot as plt
        import numpy as np
        import matplotlib.gridspec as gridspec
        import pandas as pd
        import matplotlib
        import numpy as np
        %matplotlib inline

In [2]: data_path = "/home/sthomas/projects/LJ_System_Size"

        names={'iso':'Isothermal','lin_ramp':'Linear Ramp','step':'Step'}
        colors={'iso':'C0','lin_ramp':'C1','step':'C2'}
        markers={'iso':'s','lin_ramp':'P','step':'>'}
        linestyles={'iso':'-','lin_ramp':'--','step':'-.'}

        project = signac.get_project(data_path)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(['_id'])
        statepoints = {doc['_id']: doc['statepoint'] for doc in project.index()}
        df = pd.DataFrame(statepoints).T.join(df_index)
        #df.head()

In [3]: from matplotlib import cm
        import matplotlib as mpl
        from mpl_toolkits.mplot3d import Axes3D
        from scipy import interpolate


        import os
        import math
        import gsd
        import gsd.fl
        import gsd.hoomd


        df_filtered = df[(df.t_Final==1e7)&
                         ((df.n_particles>=1e5)|
                          (df.n_particles==5e4)|
                          (df.n_particles==8e4))]

        df_sorted=df_filtered.sort_values('n_particles')
        #Ns=df_sorted.n_particles.unique()
        #print(Ns)
        Ns=[]
        mean_qs_for_Ns = []
        mean_Is_for_Ns = []
        mean_times_for_Ns = []
        std_qs_for_Ns = []
        std_Is_for_Ns = []
        q_hbls=[]
        USE_INDE_FRAMES=False
        for N,df_grp in df_sorted.groupby('n_particles'):
            times_for_all_trials=[]
            qs_for_all_trials=[]
            Is_for_all_trials=[]

            qms_all=[] #qmax
            Is_all=[]
            Qs_all=[]
            times_all=[]
            cure_all=[]
            mean_qs=[]
            mean_Is=[]
            std_qs=[]
            std_Is=[]
            mean_qms=[]
            mean_Is_of_qm=[]

            for signac_id in df_grp.index:
                job = project.open_job(id=signac_id)
                #print(job)
                if 'Lx' in job.document:#checking if the job completed
                    #print(job)
                    if USE_INDE_FRAMES:
                        mqfif,stdqfif,mifif,stdifif =
        get_mean_sf_from_independent_frames(job,equilibrated_percent=90)
                        #print(N,mqfif)
                        mean_qs.append(mqfif)
                        std_qs.append(stdqfif)
                        mean_Is.append(mifif)
                        std_Is.append(stdifif)
                    else:
                        diffract_dir ='diffract_type_2'
                        n_particles=job.sp.n_particles
                        if job.isfile('{}/asq.txt'.format(diffract_dir)):
                            data=np.genfromtxt(job.fn('{}/asq.txt'.format(diffract_dir)))
                            #print(np.shape(data[:,0]))
                            q=data[:,0]
                            I=data[:,1]
                            mean_qs.append(q)
                            #std_qs.append(stdqfif)
                            mean_Is.append(I)
                            #std_Is.append(stdifif)

                        else:
                            print(job)
                            print('Final frame is not diffracted')
                else:
                    print('Lx not found for',job)
            if len(mean_qs)>0:
                print('q calculated from',len(mean_qs),'trials for N=',N)
                mean_q_for_N = np.mean(mean_qs,axis=0)
                mean_qs_for_Ns.append(mean_q_for_N)
                std_q_for_N = stats.sem(mean_qs,axis=0)
                std_qs_for_Ns.append(std_q_for_N)
                mean_I_for_N = np.mean(mean_Is,axis=0)
                mean_Is_for_Ns.append(mean_I_for_N)
                std_I_for_N = stats.sem(mean_Is,axis=0)
                std_Is_for_Ns.append(std_I_for_N)
                Ns.append(N)
        #print((mean_qs_for_Ns))
        #print(std_Is_for_Ns)
        #print(Ns)
```

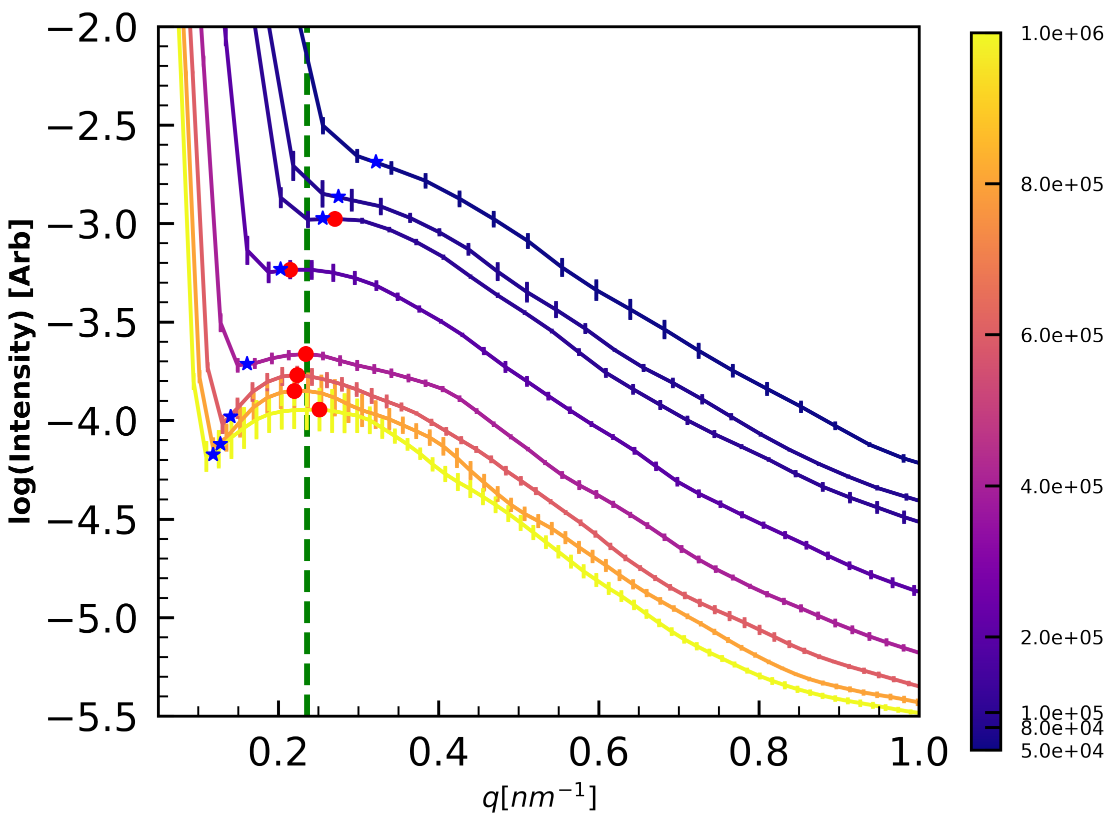
```
q calculated from 3 trials for N= 50000.0
q calculated from 3 trials for N= 80000.0
q calculated from 3 trials for N= 100000.0
q calculated from 3 trials for N= 200000.0
q calculated from 3 trials for N= 400000.0
q calculated from 3 trials for N= 600000.0
q calculated from 3 trials for N= 800000.0
q calculated from 3 trials for N= 1000000.0
Lx not found for b0bc66fa9ae800b457ccbc67f8debeaa
Lx not found for d02d14b7995d6a2866ba7a2c4c9597f6
Lx not found for b2aea2daba3d84311a6e664fbb4ad3f1
```

```
In [7]: fig = plt.figure(dpi=600,figsize=(7, 6))
        ax = fig.gca()
```

```
ax1 = fig.add_axes([ 1, 0.10,0.03, 0.85])

#ax1 = fig.add_axes([ 0.95, 0.10,0.03, 0.85])

ticks=Ns

cmap = mpl.cm.plasma
norm = mpl.colors.Normalize(vmin=np.min(Ns), vmax=np.max(Ns))
cb1 = mpl.colorbar.ColorbarBase(ax1, cmap=cmap,
                                norm=norm,
                                ticks=ticks,
                                format = "%.1e",
                                spacing='uniform',
                                orientation='vertical',)
cb1.ax.minorticks_off()

for l in ax1.yaxis.get_ticklabels():
    #l.set_weight("bold")
    l.set_fontsize(10)
print(Ns)
cmap_indices = []
maxN = np.max(Ns)
minN = np.min(Ns)
for N in Ns:
    cmap_i = (N-minN)/(maxN-minN)
    cmap_indices.append(cmap_i)
#cmap = [plt.cm.plasma(i) for i in np.linspace(0, 1, len(Ns))]
cmap = [plt.cm.plasma(i) for i in cmap_indices]
ax.set_prop_cycle(color=cmap)
offsets = np.linspace(0,4,num=len(Ns))
first_peak_qs=[]
for i,N in enumerate(Ns):
    #print('N',N)
    #i=l-i_temp-1
    q=mean_qs_for_Ns[i]
    I=mean_Is_for_Ns[i]
    I_std=std_Is_for_Ns[i]
    #print('I_std',I_std)
    #print(len(I))

    offset = 0#offsets[i]
    #time=time_mean[i]
    legend='{:.1e}'.format(N)
    fn = interpolate.interp1d(q,I,kind='cubic')
    ax.errorbar(q,
                I+offset,
                I_std,
                #marker='.',
                markersize=5,
                linewidth=2,
                capsize=1,
                label=legend,
                zorder=1)
    #df_filtered= df[(df.t_Final==6e6)&
    #                (df.n_particles==N)]
    df_filt = df_filtered[df_filtered.n_particles==N]
    sids = df_filt.signac_id
    job = project.open_job(id=sids[0])
    if 'Lx' not in job.document:
        print(job,'does not contain Lx')
        continue
    first_peak_q,first_peak_i = get_highest_maxima(job.document['Lx'],q,I)
    #print(first_peak_q,first_peak_i)
    if first_peak_q is not None:
        if first_peak_q <0.8:
            first_peak_qs.append(first_peak_q)
            ax.scatter(first_peak_q,
                       first_peak_i+offset,
                        color='r',s=50,zorder=2)
            half_box_length = job.document['Lx']/2
            q_hbl = 2*math.pi/(half_box_length*1.06)
            ax.scatter(q_hbl,
                        fn(q_hbl)+offset,
                        marker='*',
                        s=50,
                        color='b',
                        zorder=2)#,s=10)

    print(first_peak_qs)
    mean_q=np.mean(first_peak_qs)
    print('mean first peak',mean_q)
    ax.axvline(x=mean_q,linestyle='--',color='g',zorder=0)#,label='$\\langle
    q_{max}\\rangle$')

    ax.set_xlabel(r"$q [nm^{-1}]$")
    ax.set_ylabel("log(Intensity) [Arb]")
    #ax.legend(fontsize=10)
    ax.set_xlim(0.05,1.0)
    ax.set_ylim(-5.5,-2.0)
    plt.savefig("lj_finite_size_effect.png", transparent=True, bbox_inches='tight')
```

```
[50000.0, 80000.0, 100000.0, 200000.0, 400000.0, 600000.0, 800000.0, 1000000.0]
[0.2706786474320498, 0.2148377725895124, 0.2344606689491565, 0.2234402894435762,
0.21992640103854047, 0.25127578526617805]
mean first peak 0.2357699274531689
```

```
<ipython-input-7-ce00a7b2306d>:91: UserWarning: This figure includes Axes that are not
compatible with tight_layout, so results might be incorrect.
  plt.savefig("lj_finite_size_effect.png", transparent=True, bbox_inches='tight')
```



```
In [ ]:
```

```
In [1]: import os
        os.environ['MATPLOTLIBRC'] = "../matplotlibrc"
        import sys
        from tg_analysis import get_tg_data
        import signac
        import pandas as pd
        import matplotlib.pyplot as plt
        import numpy as np
```

```
In [2]: BSU_BLUE = "#0033A0"
        BSU_ORANGE = "#D64309"

        data_path_small = '/home/mikehenry/small_tg_test/epoxpy-flow'
        data_path_large = '/home/sthomas/projects/LB_mixing/'


        print("building small df")
        project = signac.get_project(data_path_small)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(["_id"])
        statepoints = {doc["_id"]: doc["statepoint"] for doc in project.index()}
        df_S = pd.DataFrame(statepoints).T.join(df_index)
        df_S = df_S.sort_values("T")
        print("done")
        print("building large df")
        project = signac.get_project(data_path_large)
        df_index = pd.DataFrame(project.index())
        df_index = df_index.set_index(["_id"])
        statepoints = {doc["_id"]: doc["statepoint"] for doc in project.index()}
        df_L = pd.DataFrame(statepoints).T.join(df_index)
        df_L = df_L.sort_values("T")
        print("done")
```

```
building small df
done
building large df
done
```

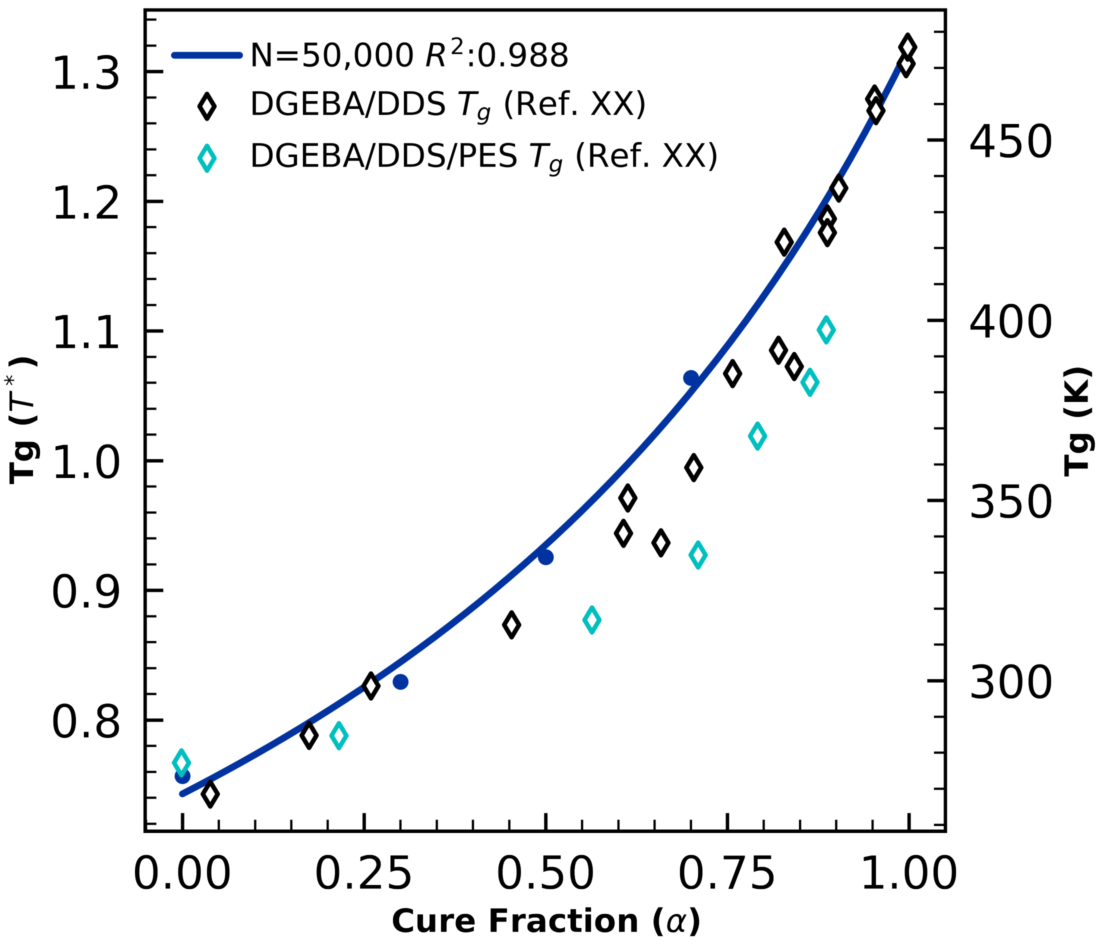
```
In [3]: alphas_S, fit_ydata_S, R2_S, cure_percents_S, Tgs_S = get_tg_data(data_path_small, df_S)
        alphas_L, fit_ydata_L, R2_L, cure_percents_L, Tgs_L = get_tg_data(data_path_large, df_L)
        print("done")
```

```
in common, indices: (array([ 0,  1,  2,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16, 17, 18,
       19, 20, 21, 22, 23, 24, 25, 26, 27, 28]),)
00 0
in common, indices: (array([ 0,  1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28]),)
00 0
in common, indices: (array([ 0,  1,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16, 17, 18,
       19, 20, 21, 22, 23, 24, 25, 26, 27, 28]),)
00 0
in common, indices: (array([ 1,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15,
16, 17, 18,
       19, 20, 21, 22, 23, 24, 25, 26, 27, 28]),)
00 1
in common, indices: (array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
```

```
00 1
in common, indices: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 0
in common, indices: (array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14,
15, 16, 17,
       18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34,
       35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 1
in common, indices: (array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13,
14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33,
       34, 35, 36, 37, 38, 39, 40, 41, 42, 43, 44, 45]),)
00 0
done
```

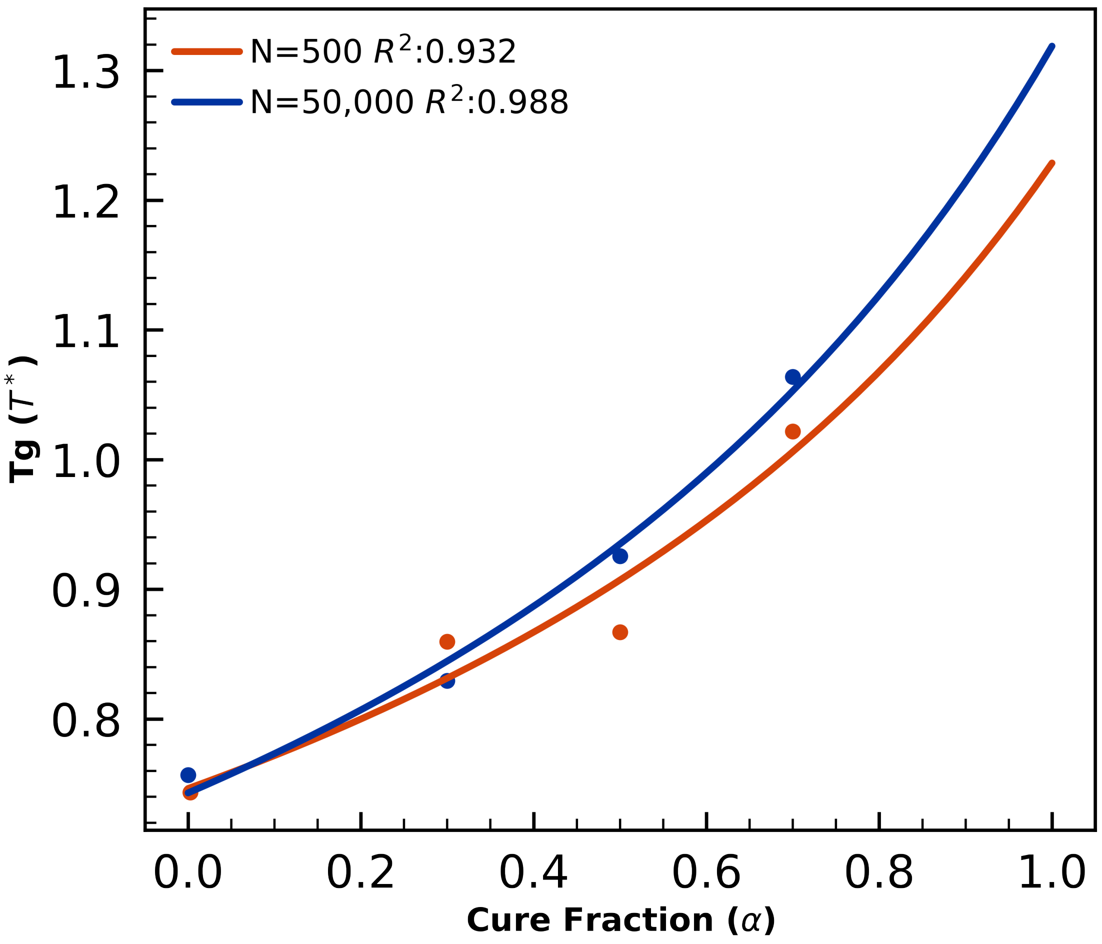
```
In [8]: fig, ax1 = plt.subplots(dpi=600, figsize=(7, 6))
        ax2=ax1.twinx()

        #ax1.scatter(cure_percents_S/100., Tgs_S, color='r')
        ax1.scatter(cure_percents_L/100., Tgs_L, color=BSU_BLUE)
        #ax1.plot(alphas_S,fit_ydata_S,label='N=500 $R^2$:{}'.format(round(R2_S,3)), color="r")
        ax1.plot(alphas_L,fit_ydata_L,label='N=50,000 $R^2$:{}'.format(round(R2_L,3)),
        color=BSU_BLUE)

        exp1_data = np.genfromtxt('Min1993.txt',delimiter=',')

        ax2.scatter(exp1_data[:,0],
                        exp1_data[:,1],
                        marker='d',
                        facecolor='w',
                        linewidth=2,
                        edgecolor='k',
                        s=60,
                        #label='$E_a$:{}'.format(activation_energy),
                        color='k',
                       zorder=1,
                       label='DGEBA/DDS $T_g$ (Ref. XX)')


        exp2_data = np.genfromtxt('Jenninger2000.txt',delimiter=',')
        ax2.scatter(exp2_data[:,0],
                        exp2_data[:,1],
                        marker='d',
                        facecolor='w',
                        linewidth=2,
                        edgecolor='c',
                        s=60,
                        #label='$E_a$:{}'.format(activation_energy),
                        color='c',
                       zorder=1,
                       label='DGEBA/DDS/PES $T_g$ (Ref. XX)')




        #ax2.scatter(1.00,Tg_exp,marker='*',color='r',s=200,label='Experimental Tg
        ($\\alpha=1.0$)')

        h1, l1 = ax1.get_legend_handles_labels()
        h2, l2 = ax2.get_legend_handles_labels()
        ax1.legend(h1+h2, l1+l2, loc=2)
```

```
ax1.set_xlabel('Cure Fraction ($\\alpha$)')
ax1.set_ylabel('Tg ($T^*$)')
ax2.set_ylabel('Tg (K)')
plt.savefig("Tg_N_exp.png", transparent=True)
#ax1.legend(fontsize=10,loc='best')
#ax2.legend(fontsize=10,loc='center left')
```



```
In [7]: #plt.style.use('matplotlibrc')
        fig, ax1 = plt.subplots(dpi=600, figsize=(7, 6))

        ax1.scatter(cure_percents_S/100., Tgs_S, color=BSU_ORANGE)
        ax1.scatter(cure_percents_L/100., Tgs_L, color=BSU_BLUE)
        ax1.plot(alphas_S,fit_ydata_S,label='N=500 $R^2$:{}'.format(round(R2_S,3)),
        color=BSU_ORANGE)
        ax1.plot(alphas_L,fit_ydata_L,label='N=50,000 $R^2$:{}'.format(round(R2_L,3)),
        color=BSU_BLUE)


        #ax2.scatter(1.00,Tg_exp,marker='*',color='r',s=200,label='Experimental Tg
        ($\\alpha=1.0$)')

        ax1.legend()
        ax1.set_xlabel('Cure Fraction ($\\alpha$)')
```

```
ax1.set_ylabel('Tg ($T^*$)')
plt.savefig("Tg_N_sim.png", transparent=True)
#ax1.legend(fontsize=10,loc='best')
#ax2.legend(fontsize=10,loc='center left')
```



```
In [ ]:
```

common.py Page 1/24

```python
import cme_utils
from cme_utils.analyze import autocorr
import numpy as np
import matplotlib.pyplot as plt


def get_split_quench_job_msd(job,prop_name):
    times = []
    prop_vals = []
    qTs=[]
    if job.isfile('msd.log'):
        log_path = job.fn('msd.log')
        data = np.genfromtxt(log_path, names=True)
        PROP_NAME =prop_name
        prop_values = data[PROP_NAME]#'pair_lj_energy']
        time_steps = data['timestep']
        len_prof = len(job.sp.quench_temp_prof)
        for i in range(0,len_prof,2):
            current_point = job.sp.quench_temp_prof[i]
            next_point = job.sp.quench_temp_prof[i+1]
            start_time = current_point[0]
            end_time = next_point[0]
            if current_point[1]≠next_point[1]:
                print('WARNING! Detected a non isothermal step')
            target_T = current_point[1]
            #print(time_steps)
            #print(start_time,end_time)
            indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
            start_index = indices[0][0]
            end_index = indices[0][-1]
            sliced_ts = time_steps[start_index:end_index+1]
            sliced_prop_vals = prop_values[start_index:end_index+1]
            #sliced_pe = pe[start_index:end_index+1]
            #mean,std = get_mean_and_std(job,sliced_ts,sliced_prop_vals,sliced_p
e)
            #means.append(mean)
            #stds.append(std)
            times.append(sliced_ts)
            prop_vals.append(sliced_prop_vals)
            qTs.append(target_T)
    return times,prop_vals,qTs

def _get_decorrelation_time(prop_values,
                             time_steps):
    t = time_steps - time_steps[0]
    dt = t[1] - t[0]
    acorr = autocorr.autocorr1D(prop_values)
    for acorr_i in range(len(acorr)):
        if acorr[acorr_i]<0:
            break
    lags = [i*dt for i in range(len(acorr))]

    decorrelation_time = int(lags[acorr_i])
    if decorrelation_time ≡ 0:
        decorrelation_time = 1
    decorrelation_stride = int(decorrelation_time/dt)
    nsamples = (int(t[-1])-t[0])/decorrelation_time
    temps = "There are %.5e steps, (" % t[-1]
    temps = temps + "%d" % int(t[-1])
    temps = temps + " frames)\n"
    temps = temps + "You can start sampling at t=%.5e" % t[0]
    temps = temps + " (frame %d)" % int(t[0] )
    temps = temps + " for %d samples\n" % nsamples
    temps = temps + "Because the autocorrelation time is %.5e" % lags[acorr_i]
    temps = temps + " (%d frames)\n" % int(lags[acorr_i])
    #print(temps)
    return decorrelation_time, decorrelation_stride

def get_mean_and_std_from_time_step(job, time_steps, prop_values,start_t):
    start_i = np.where(time_steps ≥ start_t)[0]
    if len(start_i) >0:
        start_i=start_i[0]
```

```python
    else:
        start_i = 0

    if start_i < len(time_steps):
        independent_vals_i = np.arange(start_i, len(prop_values)-1, 1)
        independent_vals = prop_values[independent_vals_i]
        #print(independent_vals)
        mean=np.mean(independent_vals)
        std=np.std(independent_vals)
    else:
        print('the {} values given have not reached equilibrium.'.format(prop))
        mean = None
        std = None
    return mean, std

def get_mean_and_std(job, time_steps, prop_values,pe,mean_from_second_half=False
):
    if mean_from_second_half:
        start_i = int(len(time_steps)*0.75)
        start_t = time_steps[start_i]
    else:
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, pe)

    if start_i < len(time_steps):
        decorrelation_time, decorrelation_stride = _get_decorrelation_time(prop_
values[start_i:], time_steps[start_i:])
        #print('decorrelation_time:',decorrelation_time)
        independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrelatio
n_stride)
        independent_vals = prop_values[independent_vals_i]
        #print(independent_vals)
        mean=np.mean(independent_vals)
        std=np.std(independent_vals)
    else:
        print('the {} values for {} given have not reached equilibrium.'.format(job,prop))
        mean = None
        std = None
    return mean, std



def get_mean_and_std_from_log(job, prop):
    if job.isfile('out.log'):
        log_path = job.fn('out.log')
        data = np.genfromtxt(log_path, names=True)
        prop_values = data[prop]
        time_steps = data['timestep']
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, prop_va
lues)
        if start_i < len(time_steps):
            decorrelation_time, decorrelation_stride = _get_decorrelation_time(p
rop_values[start_i:], time_steps[start_i:])
            independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrel
ation_stride)
            independent_vals = prop_values[independent_vals_i]
            #print(independent_vals)
            mean=np.mean(independent_vals)
            std=np.std(independent_vals)
        else:
            print('the {} values given have not reached equilibrium.'.format(prop))
            mean = None
            std = None
    else:
        print('could not find log file for {}'.format(job))
        mean=None
        std=None
    #print(mean)
    return mean, std


def plot_equilibriation(df_filtered,
                        project,
                        prop_name,
                        draw_decorrelated_samples=False,
                        draw_equilibrium_window=True,
                        mean_from_second_half=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])
    df_grouped = df_sorted.groupby('quench_T')


    quenchTs=[]
    mean_vols=[]
    vol_stds=[]
    colors = plt.cm.plasma(np.linspace(0,1,len(df_grouped)))
    i=0
    for name,group in df_grouped:
        time_steps_temp = []
        mean_vals_temp = []
        val_stds_temp = []
        for job_id in group.index:
    #for i,job_id in enumerate(df_sorted.index):
            job = project.open_job(id=job_id)
            #print(job)
            if job.isfile('out.log'):
                log_path = job.fn('out.log')
                data = np.genfromtxt(log_path, names=True)
                PROP_NAME =prop_name
                prop_values = data[PROP_NAME]#'pair_lj_energy']
                time_steps = data['timestep']
                if mean_from_second_half:
                    start_i = int(len(time_steps)*.75)
                    #print(job,'start_i',start_i,len(time_steps),time_steps)
                    start_t = time_steps[start_i]
                else:
                    start_i, start_t = autocorr.find_equilibrated_window(time_st
eps, data['potential_energy'])
                decorrelation_time, decorrelation_stride = _get_decorrelation_ti
me(data['potential_energy'][start_i:], time_steps[start_i:])
```

```
                #print('decorrelation_time:',decorrelation_time)
                independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrelation_stride)
                independent_vals = time_steps[independent_vals_i]
                #starttime_steps.index(start_t)

                if 'quench_T' in job.sp:
                    label = 'q_T:{},cure:{}'.format(job.sp.quench_T,job.sp.stop_after_percent)
                    #label = 'tau:{}, tauP:{}'.format(job.sp.tau,job.sp.tauP)
                else:
                    label = 'kT:{},cure:{}'.format(job.sp.kT,job.sp.stop_after_percent)
                time_steps_temp.append(time_steps)
                mean_vals_temp.append(prop_values)
            else:
                print('did not find out.log for',job)
        mean_time_steps = np.mean(time_steps_temp,axis=0)
        mean_prop_values = np.mean(mean_vals_temp,axis=0)
        plt.plot(mean_time_steps,mean_prop_values,label=label,color=colors[i],linewidth=1.0)
        i+=1
        if draw_decorrelated_samples:
            for xval in independent_vals:
                plt.axvline(x=xval,linestyle='--',linewidth=0.2)
        if draw_equilibrium_window:
            plt.plot(mean_time_steps[start_i],
                      mean_prop_values[start_i],
                      marker='*',
                      color='r',
                      markersize=10)
            #print(time_steps)
            #decorr_i = np.where(time_steps >= decor_time)[0][0]
            #print(decorr_


def get_values_for_quenchTs(df_filtered,project, prop,mean_from_second_half=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])
    df_grouped = df_sorted.groupby('quench_T')
    quenchTs=[]
    mean_vals=[]
    val_stds=[]
    for name,group in df_grouped:
        quench_Ts_temp = []
        mean_vals_temp = []
        val_stds_temp = []

        for job_id in group.index:
            #job_id = group.signac_id
            #print(name,job_id)
            job = project.open_job(id=job_id)
            #print(job)
            if job.isfile('out.log'):
                log_path = job.fn('out.log')
                data = np.genfromtxt(log_path, names=True)
                prop_value = data[prop]
                time_steps = data['timestep']
                pe = data['potential_energy']
                #print(job)
                mean,std = get_mean_and_std(job,time_steps,prop_value,pe,mean_from_second_half)
                if mean is ¬ None:
                    quench_Ts_temp.append(job.sp.quench_T)
                    mean_vals_temp.append(mean)
                    val_stds_temp.append(std)

        quenchTs.append(np.mean(quench_Ts_temp))
        mean_vals.append(np.mean(mean_vals_temp))
        val_stds.append(np.mean(val_stds_temp))
    return quenchTs,mean_vals,val_stds
```

```


def line_intersect(m1, b1, m2, b2):
    if m1 == m2:
        print ("These lines are parallel!!!")
        return None
    # y = mx + b
    # Set both lines equal to find the intersection point in the x direction
    # m1 * x + b1 = m2 * x + b2
    # m1 * x - m2 * x = b2 - b1
    # x * (m1 - m2) = b2 - b1
    # x = (b2 - b1) / (m1 - m2)
    x = (b2 - b1) / (m1 - m2)
    # Now solve for y -- use either line, because they are equal here
    # y = mx + b
    y = m1 * x + b1
    return x,y

from scipy.optimize import curve_fit
from scipy.interpolate import InterpolatedUnivariateSpline
from piecewise.regressor import piecewise #https://www.datadoghq.com/blog/engine
ering/piecewise-regression/
from piecewise.plotter import plot_data_with_regression

def DiBenedetto(alphas,T1,T0,inter_param):
    Tgs = []
    for alpha in alphas:
        Tg = inter_param*alpha*(T1-T0)/(1-(alpha*(1-inter_param))) +T0
        Tgs.append(Tg)
    return Tgs

def fit_Tg_to_DiBenedetto(alphas,Tgs,T1,T0=None):
    import warnings
    np.seterr(all='raise')
    plot_fit_fails=True
    inter_parm=0.5
    try:
        if T1≡None ∧ T0≡None:
            smallestTg=Tgs[0]
            largestTg=Tgs[-1]
            popt, pcov = curve_fit(lambda Xs,T1,T0: DiBenedetto(Xs,T1,T0,inter_parm),
                                   alphas,Tgs,
                                   #p0=[0,0],
                                   p0=[largestTg,smallestTg],
                                   #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                   bounds=([0,0],[largestTg*1.5,smallestTg*1.2]))#,maxfev=200000)
        elif T1≡None ∧ T0≠None:
            popt, pcov = curve_fit(lambda Xs,T1: DiBenedetto(Xs,T1,T0,inter_parm),
                                   alphas,Tgs,
                                   #p0=[0,0],
                                   p0=[1],
                                   #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                   bounds=([0],[np.infty]))#,maxfev=200000)
        else:
            popt, pcov = curve_fit(lambda Xs,T0: DiBenedetto(Xs,T1,T0,inter_parm),
                                       alphas,Tgs,
                                       #p0=[0,0],
                                       p0=[0],
                                       #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                       bounds=([-np.infty],[np.infty]))#,maxfev=200000)
        #print('found fit')
    except FloatingPointError:
        print('Curve fitting failed(FloatingPointError)')
    except RuntimeError:
            print('Curve fitting failed(RuntimeError)')
        except TypeError:
            print('Curve fitting failed(TypeError)')
        except ValueError:
            print('Curve fitting failed(ValueError)')

    ydata = np.asarray(Tgs)
    if T1≡None ∧ T0≡None:
        fit_ydata = DiBenedetto(alphas,*popt,inter_parm)
    elif T1≡None ∧ T0≠None:
        fit_ydata = DiBenedetto(alphas,*popt,T0,inter_parm)
    else:
        fit_ydata = DiBenedetto(alphas,T1,*popt,inter_parm)
    residuals = ydata - fit_ydata
    ss_res = np.sum(residuals**2)
    ss_tot = np.sum((ydata-np.mean(ydata))**2)
    #print('ss_res',ss_res,'ss_tot',ss_tot)
    if ss_tot ≡ 0:
        #print('found ss_tot: 0')
        r_squared = 0
    else:
        r_squared = 1 - (ss_res / ss_tot)
    if T1≡None ∧ T0≡None:
        return r_squared,fit_ydata,popt[0],inter_parm,popt[1]
    else:
        return r_squared,fit_ydata,popt[0],inter_parm#,popt[1]

def find_Tg(quenchTs, mean_vals,sap):
    print(sap)
    if True:#sap<=50.:
        use_first_deviation = False
        if use_first_deviation:
            model = piecewise(quenchTs, mean_vals)
            if len(model.segments) ≡ 2:
                lines = []
                l1 = model.segments[0]
                m1 = l1.coeffs[1]
                b1 = l1.coeffs[0]
                l2 = model.segments[1]
                m2 = l2.coeffs[1]
                b2 = l2.coeffs[0]
            f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
            dxdT = f.derivative(n=1)
            dx_dTs = dxdT(quenchTs)
            dev_index = np.where(np.abs(dx_dTs)>m1)[0][0]
            x=quenchTs[dev_index]
            y=mean_vals[dev_index]
        else:
            print(’using derivatives’)
            f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
            dxdT = f.derivative(n=1)
            d2xdT = f.derivative(n=2)
            dx_dTs = dxdT(quenchTs)
            d2x_dT2s = d2xdT(quenchTs)
            max_dx2 = np.max(d2x_dT2s)
            min_dx2 = np.min(d2x_dT2s)
            max_i = np.where(d2x_dT2s≡max_dx2)[0][0]
            min_i = np.where(d2x_dT2s≡min_dx2)[0][0]
            x = (quenchTs[min_i]+quenchTs[max_i])/2
            y = (mean_vals[min_i]+mean_vals[max_i])/2
    else:
        print(’using line iftting’)
        #plot_data_with_regression(quenchTs, mean_vals)
        model = piecewise(quenchTs, mean_vals)
        #print(model)
        if len(model.segments) ≡ 2:
            lines = []
            l1 = model.segments[0]
            m1 = l1.coeffs[1]
            b1 = l1.coeffs[0]
            l2 = model.segments[1]
            m2 = l2.coeffs[1]
```

```python
                b2 = l2.coeffs[0]
                x,y = line_intersect(m1,b1,m2,b2)

            else:
                print('WARNING: found more or less than 2 line segments in regression!')
        return x,y

def plot_this(job,time_steps,prop_values,pe,color,label=None,normalize_by_mean=False,mean_from_second_half=True):
    if mean_from_second_half:
        start_i = int(len(time_steps)*.75)
        start_t = time_steps[start_i]
    else:
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, pe)
    decorrelation_time, decorrelation_stride = _get_decorrelation_time(prop_values[start_i:], time_steps[start_i:])
    #print('decorrelation_time:',decorrelation_time)
    independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrelation_stride)
    independent_vals = time_steps[independent_vals_i]
    #starttime_steps.index(start_t)
    #for xval in independent_vals_i:
    #    plt.axvline(x=xval,linestyle='--',linewidth=0.2)
    indices = list(range(0,len(prop_values)))
    if len(indices) ≠ len(prop_values):
        print('Check the length of arrays')
    #print(indices)
    #print(prop_values)
    if normalize_by_mean:
        mean,std = get_mean_and_std(job,time_steps,prop_values,pe)
        prop_values = prop_values/mean
        plt.axhline(y=1.0,linewidth=1.0,linestyle='--')
    plt.plot(indices,prop_values,label=label,linewidth=1,color=color)
    plt.plot(start_i,prop_values[start_i],marker='*',color='r', markersize=10)

def get_split_quench_job_property_mean_std(job,prop_name):
    means = []
    stds = []
    times = []
    temps = []
    if job.isfile('out.log'):
        log_path = job.fn('out.log')
        data = np.genfromtxt(log_path, names=True)
        PROP_NAME =prop_name
        prop_values = data[PROP_NAME]#'pair_lj_energy']
        time_steps = data['timestep']
        pe = data['potential_energy']
        print(job)
        len_prof = len(job.sp.quench_temp_prof)
        for i in range(0,len_prof,2):
            current_point = job.sp.quench_temp_prof[i]
            next_point = job.sp.quench_temp_prof[i+1]
            start_time = current_point[0]
            end_time = next_point[0]
            if current_point[1]≠next_point[1]:
                print('WARNING! Detected a non isothermal step')
            target_T = current_point[1]
            #print(time_steps)
            #print(start_time,end_time)
            indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
            start_index = indices[0][0]
            end_index = indices[0][-1]
            sliced_ts = time_steps[start_index:end_index+1]
            sliced_prop_vals = prop_values[start_index:end_index+1]
            sliced_pe = pe[start_index:end_index+1]
            mean,std = get_mean_and_std(job,sliced_ts,sliced_prop_vals,sliced_pe
)
            means.append(mean)
            stds.append(std)
            times.append((start_time,end_time))
            temps.append(target_T)
    return means,stds,times,temps
```

```

def split_log(df_filtered,project,prop_name,filter_temp,rtol=0.1,show_all=True,normalize_by_mean=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])


    for job_id in df_sorted.index:
        job = project.open_job(id=job_id)
        #print(job)
        if job.isfile('out.log'):
            log_path = job.fn('out.log')
            data = np.genfromtxt(log_path, names=True)
            PROP_NAME =prop_name
            prop_values = data[PROP_NAME]#'pair_lj_energy']
            time_steps = data['timestep']
            pe = data['potential_energy']
            print(job)
            len_prof = len(job.sp.quench_temp_prof)
            colors = plt.cm.plasma(np.linspace(1,0,len_prof/2))
            for i in range(0,len_prof,2):
                current_point = job.sp.quench_temp_prof[i]
                next_point = job.sp.quench_temp_prof[i+1]
                start_time = current_point[0]
                end_time = next_point[0]
                if current_point[1]≠next_point[1]:
                    print('WARNING! Detected a non isothermal step')
                target_T = current_point[1]
                #print(start_time,end_time)
                #print(time_steps)
                if np.isclose(target_T,filter_temp,rtol=rtol) ∨ show_all:
                    #print(time_steps)
                    #print(start_time,end_time)
                    indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
                    #print(indices)
                    start_index = indices[0][0]
                    end_index = indices[0][-1]
                    #print('start_index',start_index,'end_index',end_index)
                    #print('start_index',start_index,'end_index',end_index)
                    sliced_ts = time_steps[start_index:end_index+1]
                    sliced_prop_vals = prop_values[start_index:end_index+1]
                    sliced_pe = pe[start_index:end_index+1]
                    #print(sliced_ts)
                    #print(sliced_prop_vals)
                    label = 'T:{}'.format(target_T)
                    #print(i/2)
                    plot_this(job,
                              sliced_ts,
                              sliced_prop_vals,
                              sliced_pe,
                              colors[int(i/2)],
                              label,
                              normalize_by_mean=normalize_by_mean)


def line_intersect(m1, b1, m2, b2):
    if m1 == m2:
        print ("These lines are parallel!!!")
        return None
    # y = mx + b
    # Set both lines equal to find the intersection point in the x direction
    # m1 * x + b1 = m2 * x + b2
    # m1 * x - m2 * x = b2 - b1
    # x * (m1 - m2) = b2 - b1
    # x = (b2 - b1) / (m1 - m2)
    x = (b2 - b1) / (m1 - m2)
    # Now solve for y -- use either line, because they are equal here
    # y = mx + b
    y = m1 * x + b1
    return x,y

def find_Tg(quenchTs, mean_vals):
    if False:#sap<=50.:
        use_first_deviation = True
        if use_first_deviation:
            model = piecewise(quenchTs, mean_vals)
            if len(model.segments) ≡ 2:
                lines = []
                l1 = model.segments[0]
                m1 = l1.coeffs[1]
                b1 = l1.coeffs[0]
                l2 = model.segments[1]
                m2 = l2.coeffs[1]
                b2 = l2.coeffs[0]
            f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
            dxdT = f.derivative(n=1)
            dx_dTs = dxdT(quenchTs)
            dev_index = np.where(np.abs(dx_dTs)>m1)[0][0]
            x=quenchTs[dev_index]
            y=mean_vals[dev_index]
        else:
            print('using derivatives')
            f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
            dxdT = f.derivative(n=1)
            d2xdT = f.derivative(n=2)
            dx_dTs = dxdT(quenchTs)
            d2x_dT2s = d2xdT(quenchTs)
            max_dx2 = np.max(d2x_dT2s)
            min_dx2 = np.min(d2x_dT2s)
            max_i = np.where(d2x_dT2s≡max_dx2)[0][0]
            min_i = np.where(d2x_dT2s≡min_dx2)[0][0]
            x = (quenchTs[min_i]+quenchTs[max_i])/2
            y = (mean_vals[min_i]+mean_vals[max_i])/2
    else:
        print('using line iftting')
        #plot_data_with_regression(quenchTs, mean_vals)
        model = piecewise(quenchTs, mean_vals)
        #print(model)
        if len(model.segments) ≡ 2:
            lines = []
            l1 = model.segments[0]
            m1 = l1.coeffs[1]
            b1 = l1.coeffs[0]
            l2 = model.segments[1]
            m2 = l2.coeffs[1]
            b2 = l2.coeffs[0]
            x,y = line_intersect(m1,b1,m2,b2)

        else:
            print('WARNING: found {} line segments in regression!Expecting 2'.format(len(model.
segments)))
    return x,y

def Fit_Diffusivity1(Ts,
                     Ds,
                     method='use_viscous_region',
                     min_D=1e-8,
                     ver=1,
                     viscous_line_index=1,
                     l1_T_bounds=[0,1],
                     l2_T_bounds=[0,1]):
    indices = np.where(Ds>min_D)#0.00000095)
    print("in common, indices:",indices)
    print("00", indices[0][0])
    start_index = indices[0][0]
    D_As=Ds[start_index:]
    quenchTs=Ts[start_index:]
    #print('quenchTs',quenchTs)
    model = piecewise(quenchTs, D_As)
    #print(ver)
    if ver≡4:
        #print('ver 4')
        line_vals=[]
        Ts_low_i = np.where(Ts≥l1_T_bounds[0])[0]
        if len(Ts_low_i)≡0:
            raise ValueError('lower bound for T fitting of line 1 too low. Use a higher T')
        l1_low_i = Ts_low_i[0]
        Ts_low_i = np.where(Ts≥l2_T_bounds[0])[0]
        if len(Ts_low_i)≡0:
            raise ValueError('lower bound for T fitting of line 2 too low. Use a higher T')
        l2_low_i = Ts_low_i[0]

        Ts_high_i = np.where(Ts≤l1_T_bounds[1])[0]
        if len(Ts_high_i)≡0:
            raise ValueError('upper bound for T fitting of line 1 too high. Use a lower T')
        l1_high_i = Ts_high_i[-1]
        Ts_high_i = np.where(Ts≤l2_T_bounds[1])[0]
        if len(Ts_high_i)≡0:
            raise ValueError('upper bound for T fitting of line 2 too high. Use a lower T')
        l2_high_i = Ts_high_i[-1]
        #print('Ts_high_i',Ts_high_i)
        l1Ts=Ts[l1_low_i:l1_high_i+1]
        l1Ds=Ds[l1_low_i:l1_high_i+1]
        #print(l1_low_i,l1_high_i,l1Ts)
        l2Ts=Ts[l2_low_i:l2_high_i+1]
        l2Ds=Ds[l2_low_i:l2_high_i+1]
        #print(l2_low_i,l2_high_i,l2Ts,'Ts',Ts)
        par = np.polyfit(l1Ts, l1Ds, 1, full=True)
        m1 = par[0][0]#0-slope, 1-intercept
        b1 = par[0][1]
        xs = np.linspace(l1Ts[0],l1Ts[-1])
        ys = m1*xs+b1
        line_vals.append((xs,ys))

        par = np.polyfit(l2Ts, l2Ds, 1, full=True)
        m2 = par[0][0]#0-slope, 1-intercept
        b2 = par[0][1]
        xs = np.linspace(l2Ts[0],l2Ts[-1])
        ys = m2*xs+b2
        line_vals.append((xs,ys))

        x,y = line_intersect(m1,b1,m2,b2)
        Tg=x
        Tg_prop = y

        return Tg,Tg_prop,line_vals
    elif ver≡3:
        line_vals=[]
        Ts_low_i = np.where(Ts≥l1_T_bounds[0])[0]
        if len(Ts_low_i)≡0:
            raise ValueError('lower bound for T fitting of line 1 too low. Use a higher T')
        l1_low_i = Ts_low_i[0]
        Ts_low_i = np.where(Ts≥l2_T_bounds[0])[0]
        if len(Ts_low_i)≡0:
            raise ValueError('lower bound for T fitting of line 2 too low. Use a higher T')
        l2_low_i = Ts_low_i[0]

        Ts_high_i = np.where(Ts≤l1_T_bounds[1])[0]
        if len(Ts_high_i)≡0:
            raise ValueError('upper bound for T fitting of line 1 too high. Use a lower T')
        l1_high_i = Ts_high_i[-1]
        Ts_high_i = np.where(Ts≤l2_T_bounds[1])[0]
        if len(Ts_high_i)≡0:
            raise ValueError('upper bound for T fitting of line 2 too high. Use a lower T')
        print('Ts_high_i',Ts_high_i)
        l2_high_i = Ts_high_i[-1]

        l1Ts=Ts[l1_low_i:l1_high_i]
        l1Ds=Ds[l1_low_i:l1_high_i]
        print(l1_low_i,l1_high_i,l1Ts)
        l2Ts=Ts[l2_low_i:l2_high_i]
        l2Ds=Ds[l2_low_i:l2_high_i]
        print(l2_low_i,l2_high_i,l2Ts)
        par = np.polyfit(l1Ts, l1Ds, 1, full=True)
        m1 = par[0][0]#0-slope, 1-intercept
        b1 = par[0][1]
        xs = np.linspace(l1Ts[0],l1Ts[-1])
        ys = m1*xs+b1
        line_vals.append((xs,ys))

        par = np.polyfit(l2Ts, l2Ds, 1, full=True)
        m2 = par[0][0]#0-slope, 1-intercept
        b2 = par[0][1]
        xs = np.linspace(l2Ts[0],l2Ts[-1])
        ys = m2*xs+b2
        line_vals.append((xs,ys))
        if viscous_line_index≡0:
            Tg = -b1/m1
            Tg_prop = 0.
        elif viscous_line_index≡1:
            Tg = -b2/m2
            Tg_prop = 0.
        else:
            x,y = line_intersect(m1,b1,m2,b2)
            Tg=x
            Tg_prop = y

        return Tg,Tg_prop,line_vals
    elif ver≡2:
        n_lines=len(model.segments)
        if n_lines ≡ 0:
            raise ValueError('Found zero lines in piecewise fitting')
        lines=[]
        line_vals=[]
        for i in range(n_lines):
            line = model.segments[i]
            lines.append(line)
            xs = np.linspace(line.start_t,line.end_t)
            ys = line.coeffs[1]*xs+line.coeffs[0]
            line_vals.append((xs,ys))

        if method≡'use_viscous_region':
            if n_lines>1:
                l2=lines[viscous_line_index]
            else:
                l2=lines[0]
            m2 = l2.coeffs[1]
            b2 = l2.coeffs[0]
            Tg = -b2/m2
            Tg_prop = 0.
        else:
            Tg,Tg_prop=find_Tg(mean_vals=Ds,quenchTs=Ts)
        return Tg,Tg_prop,line_vals
    elif ver≡1:
        if len(model.segments) ≡ 2:
            l1 = model.segments[0]
            m1 = l1.coeffs[1]
            b1 = l1.coeffs[0]
            l2 = model.segments[1]
            m2 = l2.coeffs[1]
            b2 = l2.coeffs[0]
            x,y = line_intersect(m1,b1,m2,b2)
            xs1 = np.linspace(l1.start_t,l1.end_t)#np.linspace(l1.start_t,(x+(l1.end_t-l1.start_t)*0.2))
            ys1 = l1.coeffs[1]*xs1+l1.coeffs[0]
            xs2 = np.linspace(l2.start_t,l2.end_t)#np.linspace((x-(l2.end_t-l2.start_t)*0.2),l2.end_t)
            ys2 = l2.coeffs[1]*xs2+l2.coeffs[0]

            if method≡'use_viscous_region':
                Tg = -b2/m2
                Tg_prop = 0.
            elif method ≡ 'intersection':
                Tg=x
                Tg_prop=y
        else:
            print('WARNING: found {} line segments in regression!'.format(len(model.segments)))
```

```python

        return Tg,Tg_prop,xs1,ys1,xs2,ys2


def Calc_Diffusivity(eq_time,
                     eq_msd,
                     fit_method='curve_fit'):
    #fit_method='curve_fit'#'power_law','poly_fit'
    if fit_method=='curve_fit':
        norm_eq_time = (eq_time-eq_time[0])
        #print(norm_eq_time,eq_msd)
        popt, pcov = curve_fit(lambda t,m,b: m*t+b ,
                                         eq_time,
                                         eq_msd,
                                         p0=[1.,0.0],
                                         bounds=([-1,0.0],[np.infty,np.infty]))
        drdt_A = popt[0]
        m=popt[0]
        b=popt[1]
    elif fit_method=='poly_fit':
        par = np.polyfit(time, msd, 1, full=True)
        drdt_A = par[0][0]#0-slope, 1-intercept
        m=par[0][0]
        b=par[0][1]
    elif fit_method=='power_law':
        popt, pcov = curve_fit(lambda t,w,x1: (w*t)**x1 ,
                               time,
                               msd,
                               p0=[0.2,1.0],
                               #p0=[1.0],
                               #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                               #bounds=([0],[4.0]))
                               maxfev=2000000,
                               bounds=([0.0,0.0],[1.0,4.0]))
        raise NotImplementedError('Diffusivity not determined')

    #calculate the diffusion coefficient
    dimensions=3
    D = drdt_A/(2*dimensions)
    return D,m,b

def getDiffusivities(project,df_curing,sortby='quench_T',name='bparticles',quench_time=1e7,use_first_trial=True):
    """
  returns diffusivity in units of D^2/tau where D and tau are distance and time units.
  Note that time is not in time steps.
  """
    Ts=[]
    Ds=[]
    for key,df_grp in df_curing.groupby('cooling_method'):
        if key≡'quench' ∧ quench_time is ¬ None:
            df_filt = df_grp[(df_grp.quench_time≡quench_time)]
        else:
            df_filt = df_grp
        df_sorted=df_filt.sort_values(sortby)
        for q_T,q_T_grp in df_sorted.groupby('quench_T'):
            for job_id in q_T_grp.index:
                job = project.open_job(id=job_id)
                if job.isfile('msd.log'):
                    log_path = job.fn('msd.log')
                    data = np.genfromtxt(log_path, names=True)
                    prop_values = data[name]#'pair_lj_energy']
                    equilibriated_ts_percentage = 0.5
                    if key≡'anneal':
                        times,msds,qTs = get_split_quench_job_msd(job,name)
                        for j,msd in enumerate(msds):
                            start_index = int(len(times[j])*equilibriated_ts_percentage)
                            time=times[j]*job.sp.md_dt
                            quench_T = qTs[j]
                            eq_msd = msd[start_index:]
                            eq_time = time[start_index:]
                            D_A,m,b = Calc_Diffusivity(eq_time,eq_msd,'curve_fit')
                            Ts.append(quench_T)
                            Ds.append(D_A)
                    else:
                        all_time_steps = data['timestep']
                        start_index = int(len(all_time_steps)*equilibriated_ts_percentage)
                        time=all_time_steps*job.sp.md_dt
                        quench_T = job.sp.quench_T
                        eq_msd = prop_values[start_index:]
                        eq_time = time[start_index:]
                        #print(job)
                        D_A,m,b = Calc_Diffusivity(eq_time,eq_msd,'curve_fit')
                        Ts.append(quench_T)
                        Ds.append(D_A)
                    if use_first_trial:
                        break#just using the first data point in this quench_T instead of mean
    Ts=np.asarray(Ts)
    Ds=np.asarray(Ds)
    return Ts,Ds

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),transparent=transparent)
```

```
import numpy as np
import math
from scipy.optimize import curve_fit
import sys,traceback

def f_t(times,C,H,Ea,kT,a_start,a_inf,breakAt_a=None,model='FO'):
    alphas = []
    the_times = []
    alpha=a_start
    #print(H)
    #a_inf = 0.96
    try:
        for t in times:
            k = H*math.exp(-Ea/(kT))
            if model ≡ 'SAFO':
                dadt = k*(a_inf-alpha)*(1+C*alpha)
            elif model ≡ 'FO':
                dadt = k*(a_inf-alpha)
            elif model ≡ 'SO':
                dadt = k*(a_inf-alpha)**2
            elif model ≡ 'SASO':
                dadt = k*(1-alpha)*(a_inf-alpha)*(1+C*alpha)
            alpha += dadt
            alphas.append(alpha)
            the_times.append(t)

            if (breakAt_a is ¬ None) ∧ (alpha ≥ breakAt_a):
                t_minutes = t/60
                print('{} reached @ {} minutes according to the model'.format(alpha,t_minutes
))
                return alphas,minutes,t_minutes
                #    print('done at',t)
                #    break
    except Exception as e:
        print('math.exp(-Ea/(kT))',math.exp(-Ea/(kT)))
        print('H',H)
```

```
            plt.plot(the_times,alphas,marker='+')
            print(the_times,alphas)
            raise e
    return alphas

def fit_curing_profile_with_model(job,model,print_error=False):
    #print('fitting job',job)
    bond_percent_index = 9
    #C=8.12 #Temperature independent acceleration constant. Aldridge, M., Wineman, A., Waas, A. & Kieffer, J.  (2014).
    suggested_C = 8.12#1e-10#0.0
    C_tolerance=1e-5
    data = np.genfromtxt(job.fn('out.log'),names=True)
    bond_percents = data['bond_percentAB']
    time_steps = data['timestep']
    alpha_inf = job.sp.stop_after_percent

    truncated_cure_fractions = []
    truncated_time_steps = []

    # We cut off the cure profile after stop_after_percent because we want the cure profile to be realistic.
    # After we stop bonding the cure profile is just flat and not realistic.
    last_index = next((i for i, v in enumerate(bond_percents) if v ≥alpha_inf), -1)
    first_index = next((i for i, v in enumerate(bond_percents) if v >0), -1)
    if last_index≤0:#maybe the system did not cure till the desired cure percent. So just take the last cure of the profile
        last_index=len(bond_percents)
    if last_index > 0 ∧ first_index ≥ 0 ∧ (last_index-first_index)>1:
        truncated_time_steps.extend(time_steps[first_index:last_index]*job.sp.md_dt)#/0.01)
        truncated_cure_fractions.extend(bond_percents[first_index:last_index]/100.)
        Ea=job.sp.activation_energy#1.0
        a_inf = truncated_cure_fractions[-1]#0.96
        import warnings
        np.seterr(all='raise')
        plot_fit_fails=True
        label='Successfully fit the curing curve'
        success=False
        try:
            #print('math.exp(-Ea/(kT))',math.exp(-Ea/(kT)))
            popt, pcov = curve_fit(lambda times, H: f_t(times,
                                                         suggested_C,
                                                         H,
                                                         Ea,
                                                         job.sp.kT,
                                                         truncated_cure_fractio
ns[0],
                                                         a_inf,model=model),
                                   truncated_time_steps,truncated_cure_fractions,
                                       p0=[1e-4],
                                       maxfev=20000,
                                       bounds=([0.0],
                                               [np.infty]))
                                               #[np.infty,np.infty]))
                                   #p0=[1e-4,1e-10],
                                   #bounds=([0,1e-10],#[0,1e-10],#[suggested_C,1e-4]
,
                                    #  [np.infty,np.infty]))
            success=True

        except Exception as e:
            if print_error:
                print(e)
                traceback.print_exc(file=sys.stdout)
        #if print_error:
        #    print(error)
        if success:
            ydata = np.asarray(truncated_cure_fractions)
            #fit_ydata = f_t(truncated_time_steps,C,*popt,Ea,job.sp.kT,truncated
_cure_fractions[0],a_inf,model=model)
            fit_ydata = f_t(truncated_time_steps,suggested_C,*popt,Ea,job.sp.kT,
truncated_cure_fractions[0],a_inf,model=model)

            residuals = ydata - fit_ydata
            ss_res = np.sum(residuals**2)
            ss_tot = np.sum((ydata-np.mean(ydata))**2)
            #print('ss_res',ss_res,'ss_tot',ss_tot)
            if ss_tot ≡ 0:
                r_squared = 0
            else:
                r_squared = 1 - (ss_res / ss_tot)
            C = suggested_C
            H = popt[0]
        else:
            r_squared=0.0
            C=suggested_C
            H=0.0
            truncated_cure_fractions=None
            fit_ydata=None
    else:
        success=False
        r_squared=0.0
        C=suggested_C
        H=0.0
        truncated_cure_fractions=None
        fit_ydata=None
        #print('Did not try to fit curing curves. last_index:',last_index,'first
_index',first_index)
    return success,r_squared,C,H, truncated_time_steps,fit_ydata,first_index,las
t_index

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),transparent=transparent)
```

```python
## import os

import math

import gsd
import gsd.fl
import gsd.hoomd
from scipy.signal import argrelextrema as argex
from cme_utils.analyze import autocorr
import numpy as np
import os

def get_all_maximas(lx,q,intensities):
    half_box_length = lx*0.6
    q_half_length = 2*math.pi/(half_box_length)
    peaks_q = []
    peaks_I = []
 #   print(q)
    maxima_i = argex(intensities,np.greater)[0]
    for i in maxima_i:
        if q[i] > q_half_length:
  #          print(i)
            peaks_q.append(q[i])
            peaks_I.append(intensities[i])
    #if len(peaks_I)==0:
    #    print(job)
    #    print(q_half_length,maxima_i)
    return peaks_q,peaks_I

def get_highest_maxima(lx,q,intensities):
    peaks_q,peaks_I = get_all_maximas(lx,q,intensities)
    if len(peaks_I) > 0:
        largest_peak_I = np.max(peaks_I)
        index_largest_I = peaks_I.index(largest_peak_I)
        largest_peak_q = peaks_q[index_largest_I]
    else:
        largest_peak_q=None
        largest_peak_I=None
    return largest_peak_q,largest_peak_I


def get_nth_maxima(job,q,intensities,n=1):
    '''
  Use 'n'=1 for first maxima and 'n'=2 for second maxima etc..
  '''
    peaks_q,peaks_I = get_all_maximas(job,q,intensities)
    sorted_peaks_I = np.sort(peaks_I)
    #print(peaks_q,peaks_I,sorted_peaks_I)
    nth_largest_peak_I = sorted_peaks_I[n*-1]
    index_nth_largest_I = peaks_I.index(nth_largest_peak_I)
    nth_largest_peak_q = peaks_q[index_nth_largest_I]
    return nth_largest_peak_q,nth_largest_peak_I

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),bbox_inches='tight',transparent=tran
sparent)

from mpl_toolkits.mplot3d import axes3d

class MyAxes3D(axes3d.Axes3D):

    def __init__(self, baseObject, sides_to_draw):
        self.__class__ = type(baseObject.__class__.__name__,
                              (self.__class__, baseObject.__class__),
                              {})
        self.__dict__ = baseObject.__dict__
        self.sides_to_draw = list(sides_to_draw)
        self.mouse_init()

    def set_some_features_visibility(self, visible):
        for t in self.w_zaxis.get_ticklines() + self.w_zaxis.get_ticklabels():
            t.set_visible(visible)
        self.w_zaxis.line.set_visible(visible)
        self.w_zaxis.pane.set_visible(visible)
        self.w_zaxis.label.set_visible(visible)

    def draw(self, renderer):
        # set visibility of some features False
        self.set_some_features_visibility(False)
        # draw the axes
        super(MyAxes3D, self).draw(renderer)
        # set visibility of some features True.
        # This could be adapted to set your features to desired visibility,
        # e.g. storing the previous values and restoring the values
        self.set_some_features_visibility(True)

        zaxis = self.zaxis
        draw_grid_old = zaxis.axes._draw_grid
        # disable draw grid
        zaxis.axes._draw_grid = False

        tmp_planes = zaxis._PLANES

        if 'l' in self.sides_to_draw :
            # draw zaxis on the left side
            zaxis._PLANES = (tmp_planes[2], tmp_planes[3],
                             tmp_planes[0], tmp_planes[1],
                             tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)
        if 'r' in self.sides_to_draw :
            # draw zaxis on the right side
            zaxis._PLANES = (tmp_planes[3], tmp_planes[2],
                             tmp_planes[1], tmp_planes[0],
                             tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)
```

common.py Page 4/5

```python

        zaxis._PLANES = tmp_planes

        # disable draw grid
        zaxis.axes._draw_grid = draw_grid_old

import gsd
import gsd.fl
import gsd.hoomd
from scipy.signal import argrelextrema as argex
from cme_utils.analyze import autocorr
import numpy as np
import os
import math

def get_all_maximas(box_length,q,intensities):
    half_box_length = box_length*0.6
    q_half_length = 2*math.pi/(half_box_length)
    peaks_q = []
    peaks_I = []
 #   print(q)
    maxima_i = argex(intensities,np.greater)[0]
    for i in maxima_i:
        if q[i] > q_half_length:
  #          print(i)
            peaks_q.append(q[i])
            peaks_I.append(intensities[i])
    #if len(peaks_I)==0:
    #    print(job)
    #    print(q_half_length,maxima_i)
    return peaks_q,peaks_I

def get_highest_maxima(box_length,q,intensities):
    peaks_q,peaks_I = get_all_maximas(box_length,q,intensities)
    if len(peaks_I) > 0:
        largest_peak_I = np.max(peaks_I)
```

```
            index_largest_I = peaks_I.index(largest_peak_I)
            largest_peak_q = peaks_q[index_largest_I]
        else:
            largest_peak_q=None
            largest_peak_I=None
        return largest_peak_q,largest_peak_I
```

```
## import os

import math

import gsd
import gsd.fl
import gsd.hoomd
from scipy.signal import argrelextrema as argex
from cme_utils.analyze import autocorr
import numpy as np
import os

def get_all_maximas(lx,q,intensities):
    half_box_length = lx*0.6
    q_half_length = 2*math.pi/(half_box_length)
    peaks_q = []
    peaks_I = []
 #   print(q)
    maxima_i = argex(intensities,np.greater)[0]
    for i in maxima_i:
        if q[i] > q_half_length:
  #          print(i)
            peaks_q.append(q[i])
            peaks_I.append(intensities[i])
    #if len(peaks_I)==0:
    #    print(job)
    #    print(q_half_length,maxima_i)
    return peaks_q,peaks_I

def get_highest_maxima(lx,q,intensities):
    peaks_q,peaks_I = get_all_maximas(lx,q,intensities)
    if len(peaks_I) > 0:
        largest_peak_I = np.max(peaks_I)
        index_largest_I = peaks_I.index(largest_peak_I)
        largest_peak_q = peaks_q[index_largest_I]
    else:
        largest_peak_q=None
        largest_peak_I=None
    return largest_peak_q,largest_peak_I


def get_nth_maxima(job,q,intensities,n=1):
    '''
  Use 'n'=1 for first maxima and 'n'=2 for second maxima etc..
  '''
    peaks_q,peaks_I = get_all_maximas(job,q,intensities)
    sorted_peaks_I = np.sort(peaks_I)
    #print(peaks_q,peaks_I,sorted_peaks_I)
    nth_largest_peak_I = sorted_peaks_I[n*-1]
    index_nth_largest_I = peaks_I.index(nth_largest_peak_I)
    nth_largest_peak_q = peaks_q[index_nth_largest_I]
    return nth_largest_peak_q,nth_largest_peak_I

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),bbox_inches='tight',transparent=tran
sparent)

from mpl_toolkits.mplot3d import axes3d

class MyAxes3D(axes3d.Axes3D):

    def __init__(self, baseObject, sides_to_draw):
        self.__class__ = type(baseObject.__class__.__name__,
                              (self.__class__, baseObject.__class__),
                              {})
        self.__dict__ = baseObject.__dict__
        self.sides_to_draw = list(sides_to_draw)
        self.mouse_init()

    def set_some_features_visibility(self, visible):
        for t in self.w_zaxis.get_ticklines() + self.w_zaxis.get_ticklabels():
            t.set_visible(visible)
        self.w_zaxis.line.set_visible(visible)
        self.w_zaxis.pane.set_visible(visible)
        self.w_zaxis.label.set_visible(visible)

    def draw(self, renderer):
        # set visibility of some features False
        self.set_some_features_visibility(False)
        # draw the axes
        super(MyAxes3D, self).draw(renderer)
        # set visibility of some features True.
        # This could be adapted to set your features to desired visibility,
        # e.g. storing the previous values and restoring the values
        self.set_some_features_visibility(True)

        zaxis = self.zaxis
        draw_grid_old = zaxis.axes._draw_grid
        # disable draw grid
        zaxis.axes._draw_grid = False

        tmp_planes = zaxis._PLANES

        if 'l' in self.sides_to_draw :
            # draw zaxis on the left side
            zaxis._PLANES = (tmp_planes[2], tmp_planes[3],
                             tmp_planes[0], tmp_planes[1],
                             tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)
        if 'r' in self.sides_to_draw :
            # draw zaxis on the right side
            zaxis._PLANES = (tmp_planes[3], tmp_planes[2],
                             tmp_planes[1], tmp_planes[0],
                             tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)

        zaxis._PLANES = tmp_planes

        # disable draw grid
        zaxis.axes._draw_grid = draw_grid_old

import gsd
import gsd.fl
import gsd.hoomd
from scipy.signal import argrelextrema as argex
from cme_utils.analyze import autocorr
import numpy as np
import os
import math

def get_all_maximas(box_length,q,intensities):
    half_box_length = box_length*0.6
    q_half_length = 2*math.pi/(half_box_length)
    peaks_q = []
    peaks_I = []
 #   print(q)
    maxima_i = argex(intensities,np.greater)[0]
    for i in maxima_i:
        if q[i] > q_half_length:
  #          print(i)
            peaks_q.append(q[i])
            peaks_I.append(intensities[i])
    #if len(peaks_I)==0:
    #    print(job)
    #    print(q_half_length,maxima_i)
    return peaks_q,peaks_I

def get_highest_maxima(box_length,q,intensities):
    peaks_q,peaks_I = get_all_maximas(box_length,q,intensities)
    if len(peaks_I) > 0:
        largest_peak_I = np.max(peaks_I)
```

common.py Page 5/5

```
            index_largest_I = peaks_I.index(largest_peak_I)
            largest_peak_q = peaks_q[index_largest_I]
        else:
            largest_peak_q=None
            largest_peak_I=None
        return largest_peak_q,largest_peak_I
```

```
## import os

import math

import gsd
import gsd.fl
import gsd.hoomd
from scipy.signal import argrelextrema as argex
from cme_utils.analyze import autocorr
import numpy as np
import os

def get_all_maximas(lx,q,intensities):
    half_box_length = lx*0.6
    q_half_length = 2*math.pi/(half_box_length)
    peaks_q = []
    peaks_I = []
 #   print(q)
    maxima_i = argex(intensities,np.greater)[0]
    for i in maxima_i:
        if q[i] > q_half_length:
  #         print(i)
            peaks_q.append(q[i])
            peaks_I.append(intensities[i])
    #if len(peaks_I)==0:
    #    print(job)
    #    print(q_half_length,maxima_i)
    return peaks_q,peaks_I

def get_highest_maxima(lx,q,intensities):
    peaks_q,peaks_I = get_all_maximas(lx,q,intensities)
    if len(peaks_I) > 0:
        largest_peak_I = np.max(peaks_I)
        index_largest_I = peaks_I.index(largest_peak_I)
        largest_peak_q = peaks_q[index_largest_I]
    else:
        largest_peak_q=None
        largest_peak_I=None
    return largest_peak_q,largest_peak_I


def get_nth_maxima(job,q,intensities,n=1):
    '''
  Use 'n'=1 for first maxima and 'n'=2 for second maxima etc..
  '''
    peaks_q,peaks_I = get_all_maximas(job,q,intensities)
    sorted_peaks_I = np.sort(peaks_I)
    #print(peaks_q,peaks_I,sorted_peaks_I)
    nth_largest_peak_I = sorted_peaks_I[n*-1]
    index_nth_largest_I = peaks_I.index(nth_largest_peak_I)
    nth_largest_peak_q = peaks_q[index_nth_largest_I]
    return nth_largest_peak_q,nth_largest_peak_I

def save_frame(job,frame):
    with gsd.hoomd.open('Frame{}.gsd'.format(frame), 'wb') as t_new:
        f = gsd.fl.GSDFile(job.fn('data.gsd'), 'rb')
        t = gsd.hoomd.HOOMDTrajectory(f)
        snap = t[frame]
        t_new.append(snap)
    #hoomd.deprecated.dump.xml(group=hoomd.group.all(), filename=job.fn('Frame{}
.hoomdxml'.format(frame)), position=True)

from mpl_toolkits.mplot3d import axes3d

class MyAxes3D(axes3d.Axes3D):

    def __init__(self, baseObject, sides_to_draw):
        self.__class__ = type(baseObject.__class__.__name__,
                              (self.__class__, baseObject.__class__),
                              {})
        self.__dict__ = baseObject.__dict__
        self.sides_to_draw = list(sides_to_draw)
```

```python
            self.mouse_init()

    def set_some_features_visibility(self, visible):
        for t in self.w_zaxis.get_ticklines() + self.w_zaxis.get_ticklabels():
            t.set_visible(visible)
        self.w_zaxis.line.set_visible(visible)
        self.w_zaxis.pane.set_visible(visible)
        self.w_zaxis.label.set_visible(visible)

    def draw(self, renderer):
        # set visibility of some features False
        self.set_some_features_visibility(False)
        # draw the axes
        super(MyAxes3D, self).draw(renderer)
        # set visibility of some features True.
        # This could be adapted to set your features to desired visibility,
        # e.g. storing the previous values and restoring the values
        self.set_some_features_visibility(True)

        zaxis = self.zaxis
        draw_grid_old = zaxis.axes._draw_grid
        # disable draw grid
        zaxis.axes._draw_grid = False

        tmp_planes = zaxis._PLANES

        if 'l' in self.sides_to_draw :
            # draw zaxis on the left side
            zaxis._PLANES = (tmp_planes[2], tmp_planes[3],
                             tmp_planes[0], tmp_planes[1],
                             tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)
        if 'r' in self.sides_to_draw :
            # draw zaxis on the right side
            zaxis._PLANES = (tmp_planes[3], tmp_planes[2],
                             tmp_planes[1], tmp_planes[0],
                            tmp_planes[4], tmp_planes[5])
            zaxis.draw(renderer)

        zaxis._PLANES = tmp_planes

        # disable draw grid
        zaxis.axes._draw_grid = draw_grid_old

def _get_decorrelation_time(prop_values,
                            time_steps):
    t = time_steps - time_steps[0]
    dt = t[1] - t[0]
    acorr = autocorr.autocorr1D(prop_values)
    for acorr_i in range(len(acorr)):
        if acorr[acorr_i]<0:
            break
    lags = [i*dt for i in range(len(acorr))]

    decorrelation_time = int(lags[acorr_i])
    if decorrelation_time ≡ 0:
        decorrelation_time = 1
    decorrelation_stride = int(decorrelation_time/dt)
    nsamples = (int(t[-1])-t[0])/decorrelation_time
    temps = "There are %.5e steps, (" % t[-1]
    temps = temps + "%d" % int(t[-1])
    temps = temps + " frames)\n"
    temps = temps + "You can start sampling at t=%.5e" % t[0]
    temps = temps + " (frame %d)" % int(t[0] )
    temps = temps + " for %d samples\n" % nsamples
    temps = temps + "Because the autocorrelation time is %.5e" % lags[acorr_i]
    temps = temps + " (%d frames)\n" % int(lags[acorr_i])
    #print(temps)
    return decorrelation_time, decorrelation_stride

from scipy import stats

```

```
def get_mean_sf_from_independent_frames(job,equilibrated_percent=None):
    typeId=2
    n_views=40
    grid_size=512
    diffract_dir_pattern ='diffract_type_{}_n_views_{}_grid_size_{}_frame'.format(typeId,
                                                                                   n_views,
                                                                                   grid_size)
    directories = os.listdir(job.workspace())
    directories = [d for d in os.listdir(job.workspace()) if d.startswith(diffract_dir_pattern)]
    directories.sort(key = lambda x: int(x.split('_')[-1]))
    #print(len(directories))
    #print(directories)
    num_frames = len(directories)
    log_path = job.fn('out.log')
    data = np.genfromtxt(log_path, names=True)
    time_steps = data['timestep']
    #print('time steps',time_steps)
    if equilibrated_percent==None:
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, data['potential_energy'])
    else:
        start_i=int(len(time_steps)*equilibrated_percent/100)
        start_t=time_steps[start_i]
    decorrelation_time, decorrelation_stride = _get_decorrelation_time(data['potential_energy'][start_i:], time_steps[start_i:])
    #print('decorrelation_stride:',decorrelation_stride)
    #print('decorrelation_time:',decorrelation_time)
    #print('start_i:',start_i)
    #print('start_t:',start_t)
    if num_frames > 0:
        qs_for_all_times=[]
        Is_for_all_times=[]
        times_for_all_times=[]
        qs_list = []
        times_list = []
        Is_list = []
        Qs_list=[]

        for i,diffract_dir in enumerate(directories):
            #print("Progress {:2.1%}".format(i / num_frames), end="\r")

            #print(diffract_dir)

            if diffract_dir.startswith(diffract_dir_pattern):
                frame = int(diffract_dir.split('_')[-1])

                decorrelated_frame_stride = int(decorrelation_time/job.sp.dcd_wr
ite)
                decorrelated_frame_stride = max(decorrelated_frame_stride,1)
                time = round(frame*job.sp.dcd_write)
                #print('decorrelated frame stride is:',decorrelated_frame_stride
)
                #print('Equilibriated after time:',start_t)
                #print('time:{}, {}%{}={}'.format(time,frame,decorrelated_frame_
stride,frame%decorrelated_frame_stride))
                if time ≥ start_t:# and frame%decorrelated_frame_stride==0:# and
 frame <3e6/job.sp.dcd_write:#==119 or frame==123:#%100 == 0:#num_frames/30:
                    if job.isfile('{}/asq.txt'.format(diffract_dir)):
                        data=np.genfromtxt(job.fn('{}/asq.txt'.format(diffract_dir)
))

                        legend = '{} $\\Delta t(\Gamma:{})$'.format(time,job.sp.gamma)
                        qs = data[:,0]
                        Is = data[:,1]
                        qs_for_all_times.append(qs)
                        Is_for_all_times.append(Is)
                        times_for_all_times.append(time)

                        dq=qs[1]-qs[0]
                        Is_exp = np.exp(Is)
                        q_sq = qs**2
                        Q = np.sum(Is_exp*qs*dq)
                        Qs_list.append(Q)
                        #first_peak_q,first_peak_i = get_highest_maxima(job,qs,Is)
                        #if first_peak_q >0.8 and time > 2.0e5:
                        #    first_peak_q=q_half_length

                        #qs_list.append(first_peak_q)
                        #times_list.append(time)
                        #Is_list.append(first_peak_i)
                    else:
                        print(job,'did not contain diffraction data in ',diffract_dir)
                #else:
                #    print(job,'directory {} is not as expected:{}'.format(diffract_dir,diffract_dir_pattern))
    else:
        print(job,'did not contain diffraction data for time evolution')
    print('Number of independent frames for average:',len(qs_for_all_times))
    m_q = np.mean(qs_for_all_times,axis=0)
    std_q = stats.sem(qs_for_all_times,axis=0)
    m_I = np.mean(Is_for_all_times,axis=0)
    std_I = stats.sem(Is_for_all_times,axis=0)
    return m_q,std_q,m_I,std_I

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),bbox_inches='tight',transparent=transparent)
```

common.py Page 1/24

```
import cme_utils
from cme_utils.analyze import autocorr
import numpy as np
import matplotlib.pyplot as plt


def get_split_quench_job_msd(job,prop_name):
    times = []
    prop_vals = []
    qTs=[]
    if job.isfile('msd.log'):
        log_path = job.fn('msd.log')
        data = np.genfromtxt(log_path, names=True)
        PROP_NAME =prop_name
        prop_values = data[PROP_NAME]#'pair_lj_energy']
        time_steps = data['timestep']
        len_prof = len(job.sp.quench_temp_prof)
        for i in range(0,len_prof,2):
            current_point = job.sp.quench_temp_prof[i]
            next_point = job.sp.quench_temp_prof[i+1]
            start_time = current_point[0]
            end_time = next_point[0]
            if current_point[1]≠next_point[1]:
                print('WARNING! Detected a non isothermal step')
            target_T = current_point[1]
            #print(time_steps)
            #print(start_time,end_time)
            indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
            start_index = indices[0][0]
            end_index = indices[0][-1]
            sliced_ts = time_steps[start_index:end_index+1]
            sliced_prop_vals = prop_values[start_index:end_index+1]
            #sliced_pe = pe[start_index:end_index+1]
            #mean,std = get_mean_and_std(job,sliced_ts,sliced_prop_vals,sliced_p
e)
            #means.append(mean)
            #stds.append(std)
            times.append(sliced_ts)
            prop_vals.append(sliced_prop_vals)
            qTs.append(target_T)
    return times,prop_vals,qTs

def _get_decorrelation_time(prop_values,
                             time_steps):
    t = time_steps - time_steps[0]
    dt = t[1] - t[0]
    acorr = autocorr.autocorr1D(prop_values)
    for acorr_i in range(len(acorr)):
        if acorr[acorr_i]<0:
            break
    lags = [i*dt for i in range(len(acorr))]

    decorrelation_time = int(lags[acorr_i])
    if decorrelation_time ≡ 0:
        decorrelation_time = 1
    decorrelation_stride = int(decorrelation_time/dt)
    nsamples = (int(t[-1])-t[0])/decorrelation_time
    temps = "There are %.5e steps, (" % t[-1]
    temps = temps + "%d" % int(t[-1])
    temps = temps + " frames)\n"
    temps = temps + "You can start sampling at t=%.5e" % t[0]
    temps = temps + " (frame %d)" % int(t[0] )
    temps = temps + " for %d samples\n" % nsamples
    temps = temps + "Because the autocorrelation time is %.5e" % lags[acorr_i]
    temps = temps + " (%d frames)\n" % int(lags[acorr_i])
    #print(temps)
    return decorrelation_time, decorrelation_stride

def get_mean_and_std_from_time_step(job, time_steps, prop_values,start_t):
    start_i = np.where(time_steps ≥ start_t)[0]
    if len(start_i) >0:
        start_i=start_i[0]
    else:
        start_i = 0

    if start_i < len(time_steps):
        independent_vals_i = np.arange(start_i, len(prop_values)-1, 1)
        independent_vals = prop_values[independent_vals_i]
        #print(independent_vals)
        mean=np.mean(independent_vals)
        std=np.std(independent_vals)
    else:
        print('the {} values given have not reached equilibrium.'.format(prop))
        mean = None
        std = None
    return mean, std

def get_mean_and_std(job, time_steps, prop_values,pe,mean_from_second_half=False
):
    if mean_from_second_half:
        start_i = int(len(time_steps)*0.75)
        start_t = time_steps[start_i]
    else:
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, pe)

    if start_i < len(time_steps):
        decorrelation_time, decorrelation_stride = _get_decorrelation_time(prop_
values[start_i:], time_steps[start_i:])
        #print('decorrelation_time:',decorrelation_time)
        independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrelatio
n_stride)
        independent_vals = prop_values[independent_vals_i]
        #print(independent_vals)
        mean=np.mean(independent_vals)
        std=np.std(independent_vals)
    else:
        print('the {} values for {} given have not reached equilibrium.'.format(job,prop))
        mean = None
        std = None
    return mean, std



def get_mean_and_std_from_log(job, prop):
    if job.isfile('out.log'):
        log_path = job.fn('out.log')
        data = np.genfromtxt(log_path, names=True)
        prop_values = data[prop]
        time_steps = data['timestep']
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, prop_va
lues)
        if start_i < len(time_steps):
            decorrelation_time, decorrelation_stride = _get_decorrelation_time(p
rop_values[start_i:], time_steps[start_i:])
            independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrel
ation_stride)
            independent_vals = prop_values[independent_vals_i]
            #print(independent_vals)
            mean=np.mean(independent_vals)
            std=np.std(independent_vals)
        else:
            print('the {} values given have not reached equilibrium.'.format(prop))
            mean = None
            std = None
    else:
        print('could not find log file for {}'.format(job))
        mean=None
        std=None
    #print(mean)
    return mean, std


def plot_equilibriation(df_filtered,
                        project,
                           prop_name,
                           draw_decorrelated_samples=False,
                           draw_equilibrium_window=True,
                           mean_from_second_half=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])
    df_grouped = df_sorted.groupby('quench_T')


    quenchTs=[]
    mean_vols=[]
    vol_stds=[]
    colors = plt.cm.plasma(np.linspace(0,1,len(df_grouped)))
    i=0
    for name,group in df_grouped:
        time_steps_temp = []
        mean_vals_temp = []
        val_stds_temp = []
        for job_id in group.index:
    #for i,job_id in enumerate(df_sorted.index):
            job = project.open_job(id=job_id)
            #print(job)
            if job.isfile('out.log'):
                log_path = job.fn('out.log')
                data = np.genfromtxt(log_path, names=True)
                PROP_NAME =prop_name
                prop_values = data[PROP_NAME]#'pair_lj_energy']
                time_steps = data['timestep']
                if mean_from_second_half:
                    start_i = int(len(time_steps)*.75)
                    #print(job,'start_i',start_i,len(time_steps),time_steps)
                    start_t = time_steps[start_i]
                else:
                    start_i, start_t = autocorr.find_equilibrated_window(time_st
eps, data['potential_energy'])
                decorrelation_time, decorrelation_stride = _get_decorrelation_ti
me(data['potential_energy'][start_i:], time_steps[start_i:])
```

```
                #print('decorrelation_time:',decorrelation_time)
                independent_vals_i = np.arange(start_i, len(prop_values)-1, deco
rrelation_stride)
                independent_vals = time_steps[independent_vals_i]
                #starttime_steps.index(start_t)

                if 'quench_T' in job.sp:
                    label = 'q_T:{},cure:{}'.format(job.sp.quench_T,job.sp.stop_aft
er_percent)
                    #label = 'tau:{}, tauP:{}'.format(job.sp.tau,job.sp.tauP)
                else:
                    label = 'kT:{},cure:{}'.format(job.sp.kT,job.sp.stop_after_perc
ent)
                time_steps_temp.append(time_steps)
                mean_vals_temp.append(prop_values)
            else:
                print('did not find out.log for',job)
        mean_time_steps = np.mean(time_steps_temp,axis=0)
        mean_prop_values = np.mean(mean_vals_temp,axis=0)
        plt.plot(mean_time_steps,mean_prop_values,label=label,color=colors[i],li
newidth=1.0)
        i+=1
        if draw_decorrelated_samples:
            for xval in independent_vals:
                plt.axvline(x=xval,linestyle='--',linewidth=0.2)
        if draw_equilibrium_window:
            plt.plot(mean_time_steps[start_i],
                     mean_prop_values[start_i],
                     marker='*',
                     color='r',
                     markersize=10)
            #print(time_steps)
            #decorr_i = np.where(time_steps >= decor_time)[0][0]
            #print(decorr_



def get_values_for_quenchTs(df_filtered,project, prop,mean_from_second_half=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])
    df_grouped = df_sorted.groupby('quench_T')
    quenchTs=[]
    mean_vals=[]
    val_stds=[]
    for name,group in df_grouped:
        quench_Ts_temp = []
        mean_vals_temp = []
        val_stds_temp = []

        for job_id in group.index:
            #job_id = group.signac_id
            #print(name,job_id)
            job = project.open_job(id=job_id)
            #print(job)
            if job.isfile('out.log'):
                log_path = job.fn('out.log')
                data = np.genfromtxt(log_path, names=True)
                prop_value = data[prop]
                time_steps = data['timestep']
                pe = data['potential_energy']
                #print(job)
                mean,std = get_mean_and_std(job,time_steps,prop_value,pe,mean_from_second_half)
                if mean is ¬ None:
                    quench_Ts_temp.append(job.sp.quench_T)
                    mean_vals_temp.append(mean)
                    val_stds_temp.append(std)

        quenchTs.append(np.mean(quench_Ts_temp))
        mean_vals.append(np.mean(mean_vals_temp))
        val_stds.append(np.mean(val_stds_temp))
    return quenchTs,mean_vals,val_stds
```

```


def line_intersect(m1, b1, m2, b2):
    if m1 ≡ m2:
        print ("These lines are parallel!!!")
        return None
    # y = mx + b
    # Set both lines equal to find the intersection point in the x direction
    # m1 * x + b1 = m2 * x + b2
    # m1 * x - m2 * x = b2 - b1
    # x * (m1 - m2) = b2 - b1
    # x = (b2 - b1) / (m1 - m2)
    x = (b2 - b1) / (m1 - m2)
    # Now solve for y -- use either line, because they are equal here
    # y = mx + b
    y = m1 * x + b1
    return x,y

from scipy.optimize import curve_fit
from scipy.interpolate import InterpolatedUnivariateSpline
from piecewise.regressor import piecewise #https://www.datadoghq.com/blog/engine
ering/piecewise-regression/
from piecewise.plotter import plot_data_with_regression

def DiBenedetto(alphas,T1,T0,inter_param):
    Tgs = []
    for alpha in alphas:
        Tg = inter_param*alpha*(T1-T0)/(1-(alpha*(1-inter_param))) +T0
        Tgs.append(Tg)
    return Tgs

def fit_Tg_to_DiBenedetto(alphas,Tgs,T1,T0=None):
    import warnings
    np.seterr(all='raise')
    plot_fit_fails=True
    inter_parm=0.5
    try:
        if T1≡None ∧ T0≡None:
            smallestTg=Tgs[0]
            largestTg=Tgs[-1]
            popt, pcov = curve_fit(lambda Xs,T1,T0: DiBenedetto(Xs,T1,T0,inter_parm),
                                   alphas,Tgs,
                                   #p0=[0,0],
                                   p0=[largestTg,smallestTg],
                                   #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                   bounds=([0,0],[largestTg*1.5,smallestTg*1.2]))#,maxfev=200000)
        elif T1≡None ∧ T0≠None:
            popt, pcov = curve_fit(lambda Xs,T1: DiBenedetto(Xs,T1,T0,inter_parm),
                                   alphas,Tgs,
                                   #p0=[0,0],
                                   p0=[1],
                                   #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                   bounds=([0],[np.infty]))#,maxfev=200000)
        else:
            popt, pcov = curve_fit(lambda Xs,T0: DiBenedetto(Xs,T1,T0,inter_parm),
                                       alphas,Tgs,
                                       #p0=[0,0],
                                       p0=[0],
                                       #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                                       bounds=([-np.infty],[np.infty]))#,maxfev=200000)
        #print('found fit')
    except FloatingPointError:
        print('Curve fitting failed(FloatingPointError)')
    except RuntimeError:
            print('Curve fitting failed(RuntimeError)')
        except TypeError:
            print('Curve fitting failed(TypeError)')
        except ValueError:
            print('Curve fitting failed(ValueError)')

        ydata = np.asarray(Tgs)
        if T1≡None ∧ T0≡None:
            fit_ydata = DiBenedetto(alphas,*popt,inter_parm)
        elif T1≡None ∧ T0≢None:
            fit_ydata = DiBenedetto(alphas,*popt,T0,inter_parm)
        else:
            fit_ydata = DiBenedetto(alphas,T1,*popt,inter_parm)
        residuals = ydata - fit_ydata
        ss_res = np.sum(residuals**2)
        ss_tot = np.sum((ydata-np.mean(ydata))**2)
        #print('ss_res',ss_res,'ss_tot',ss_tot)
        if ss_tot ≡ 0:
            #print('found ss_tot: 0')
            r_squared = 0
        else:
            r_squared = 1 - (ss_res / ss_tot)
        if T1≡None ∧ T0≡None:
            return r_squared,fit_ydata,popt[0],inter_parm,popt[1]
        else:
            return r_squared,fit_ydata,popt[0],inter_parm#,popt[1]

    def find_Tg(quenchTs, mean_vals,sap):
        print(sap)
        if True:#sap<=50.:
            use_first_deviation = False
            if use_first_deviation:
                model = piecewise(quenchTs, mean_vals)
                if len(model.segments) ≡ 2:
                    lines = []
                    l1 = model.segments[0]
                    m1 = l1.coeffs[1]
                    b1 = l1.coeffs[0]
                    l2 = model.segments[1]
                    m2 = l2.coeffs[1]
                    b2 = l2.coeffs[0]
                f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
                dxdT = f.derivative(n=1)
                dx_dTs = dxdT(quenchTs)
                dev_index = np.where(np.abs(dx_dTs)>m1)[0][0]
                x=quenchTs[dev_index]
                y=mean_vals[dev_index]
            else:
                print('using derivatives')
                f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
                dxdT = f.derivative(n=1)
                d2xdT = f.derivative(n=2)
                dx_dTs = dxdT(quenchTs)
                d2x_dT2s = d2xdT(quenchTs)
                max_dx2 = np.max(d2x_dT2s)
                min_dx2 = np.min(d2x_dT2s)
                max_i = np.where(d2x_dT2s≡max_dx2)[0][0]
                min_i = np.where(d2x_dT2s≡min_dx2)[0][0]
                x = (quenchTs[min_i]+quenchTs[max_i])/2
                y = (mean_vals[min_i]+mean_vals[max_i])/2
        else:
            print('using line iftting')
            #plot_data_with_regression(quenchTs, mean_vals)
            model = piecewise(quenchTs, mean_vals)
            #print(model)
            if len(model.segments) ≡ 2:
                lines = []
                l1 = model.segments[0]
                m1 = l1.coeffs[1]
                b1 = l1.coeffs[0]
                l2 = model.segments[1]
                m2 = l2.coeffs[1]
```

```
                b2 = l2.coeffs[0]
                x,y = line_intersect(m1,b1,m2,b2)

        else:
            print('WARNING: found more or less than 2 line segments in regression!')
    return x,y

def plot_this(job,time_steps,prop_values,pe,color,label=None,normalize_by_mean=F
alse,mean_from_second_half=True):
    if mean_from_second_half:
        start_i = int(len(time_steps)*.75)
        start_t = time_steps[start_i]
    else:
        start_i, start_t = autocorr.find_equilibrated_window(time_steps, pe)
    decorrelation_time, decorrelation_stride = _get_decorrelation_time(prop_valu
es[start_i:], time_steps[start_i:])
    #print('decorrelation_time:',decorrelation_time)
    independent_vals_i = np.arange(start_i, len(prop_values)-1, decorrelation_st
ride)
    independent_vals = time_steps[independent_vals_i]
    #starttime_steps.index(start_t)
    #for xval in independent_vals_i:
    #    plt.axvline(x=xval,linestyle='--',linewidth=0.2)
    indices = list(range(0,len(prop_values)))
    if len(indices) ≠ len(prop_values):
        print('Check the length of arrays')
    #print(indices)
    #print(prop_values)
    if normalize_by_mean:
        mean,std = get_mean_and_std(job,time_steps,prop_values,pe)
        prop_values = prop_values/mean
        plt.axhline(y=1.0,linewidth=1.0,linestyle='--')
    plt.plot(indices,prop_values,label=label,linewidth=1,color=color)
    plt.plot(start_i,prop_values[start_i],marker='*',color='r', markersize=10)

def get_split_quench_job_property_mean_std(job,prop_name):
    means = []
    stds = []
    times = []
    temps = []
    if job.isfile('out.log'):
        log_path = job.fn('out.log')
        data = np.genfromtxt(log_path, names=True)
        PROP_NAME =prop_name
        prop_values = data[PROP_NAME]#'pair_lj_energy']
        time_steps = data['timestep']
        pe = data['potential_energy']
        print(job)
        len_prof = len(job.sp.quench_temp_prof)
        for i in range(0,len_prof,2):
            current_point = job.sp.quench_temp_prof[i]
            next_point = job.sp.quench_temp_prof[i+1]
            start_time = current_point[0]
            end_time = next_point[0]
            if current_point[1]≠next_point[1]:
                print('WARNING! Detected a non isothermal step')
            target_T = current_point[1]
            #print(time_steps)
            #print(start_time,end_time)
            indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
            start_index = indices[0][0]
            end_index = indices[0][-1]
            sliced_ts = time_steps[start_index:end_index+1]
            sliced_prop_vals = prop_values[start_index:end_index+1]
            sliced_pe = pe[start_index:end_index+1]
            mean,std = get_mean_and_std(job,sliced_ts,sliced_prop_vals,sliced_pe
)
            means.append(mean)
            stds.append(std)
            times.append((start_time,end_time))
            temps.append(target_T)
    return means,stds,times,temps
```

```

def split_log(df_filtered,project,prop_name,filter_temp,rtol=0.1,show_all=True,normalize_by_mean=False):
    df_sorted = df_filtered.sort_values(by=['quench_T'])


    for job_id in df_sorted.index:
        job = project.open_job(id=job_id)
        #print(job)
        if job.isfile('out.log'):
            log_path = job.fn('out.log')
            data = np.genfromtxt(log_path, names=True)
            PROP_NAME =prop_name
            prop_values = data[PROP_NAME]#'pair_lj_energy']
            time_steps = data['timestep']
            pe = data['potential_energy']
            print(job)
            len_prof = len(job.sp.quench_temp_prof)
            colors = plt.cm.plasma(np.linspace(1,0,len_prof/2))
            for i in range(0,len_prof,2):
                current_point = job.sp.quench_temp_prof[i]
                next_point = job.sp.quench_temp_prof[i+1]
                start_time = current_point[0]
                end_time = next_point[0]
                if current_point[1]≠next_point[1]:
                    print('WARNING! Detected a non isothermal step')
                target_T = current_point[1]
                #print(start_time,end_time)
                #print(time_steps)
                if np.isclose(target_T,filter_temp,rtol=rtol) ∨ show_all:
                    #print(time_steps)
                    #print(start_time,end_time)
                    indices = np.where((time_steps≥start_time)&(time_steps≤end_time))
                    #print(indices)
                    start_index = indices[0][0]
                    end_index = indices[0][-1]
                    #print('start_index',start_index,'end_index',end_index)
                    #print('start_index',start_index,'end_index',end_index)
                    sliced_ts = time_steps[start_index:end_index+1]
                    sliced_prop_vals = prop_values[start_index:end_index+1]
                    sliced_pe = pe[start_index:end_index+1]
                    #print(sliced_ts)
                    #print(sliced_prop_vals)
                    label = 'T:{}'.format(target_T)
                    #print(i/2)
                    plot_this(job,
                              sliced_ts,
                              sliced_prop_vals,
                              sliced_pe,
                              colors[int(i/2)],
                              label,
                              normalize_by_mean=normalize_by_mean)


def line_intersect(m1, b1, m2, b2):
    if m1 ≡ m2:
        print ("These lines are parallel!!!")
        return None
    # y = mx + b
    # Set both lines equal to find the intersection point in the x direction
    # m1 * x + b1 = m2 * x + b2
    # m1 * x - m2 * x = b2 - b1
    # x * (m1 - m2) = b2 - b1
    # x = (b2 - b1) / (m1 - m2)
    x = (b2 - b1) / (m1 - m2)
    # Now solve for y -- use either line, because they are equal here
    # y = mx + b
    y = m1 * x + b1
    return x,y

def find_Tg(quenchTs, mean_vals):
        if False:#sap<=50.:
            use_first_deviation = True
            if use_first_deviation:
                model = piecewise(quenchTs, mean_vals)
                if len(model.segments) ≡ 2:
                    lines = []
                    l1 = model.segments[0]
                    m1 = l1.coeffs[1]
                    b1 = l1.coeffs[0]
                    l2 = model.segments[1]
                    m2 = l2.coeffs[1]
                    b2 = l2.coeffs[0]
                f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
                dxdT = f.derivative(n=1)
                dx_dTs = dxdT(quenchTs)
                dev_index = np.where(np.abs(dx_dTs)>m1)[0][0]
                x=quenchTs[dev_index]
                y=mean_vals[dev_index]
            else:
                print('using derivatives')
                f = InterpolatedUnivariateSpline(quenchTs, mean_vals, k=2)
                dxdT = f.derivative(n=1)
                d2xdT = f.derivative(n=2)
                dx_dTs = dxdT(quenchTs)
                d2x_dT2s = d2xdT(quenchTs)
                max_dx2 = np.max(d2x_dT2s)
                min_dx2 = np.min(d2x_dT2s)
                max_i = np.where(d2x_dT2s≡max_dx2)[0][0]
                min_i = np.where(d2x_dT2s≡min_dx2)[0][0]
                x = (quenchTs[min_i]+quenchTs[max_i])/2
                y = (mean_vals[min_i]+mean_vals[max_i])/2
        else:
            print('using line iftting')
            #plot_data_with_regression(quenchTs, mean_vals)
            model = piecewise(quenchTs, mean_vals)
            #print(model)
            if len(model.segments) ≡ 2:
                lines = []
                l1 = model.segments[0]
                m1 = l1.coeffs[1]
                b1 = l1.coeffs[0]
                l2 = model.segments[1]
                m2 = l2.coeffs[1]
                b2 = l2.coeffs[0]
                x,y = line_intersect(m1,b1,m2,b2)

            else:
                print('WARNING: found {} line segments in regression!Expecting 2'.format(len(model.
    segments)))
        return x,y

    def Fit_Diffusivity1(Ts,
                        Ds,
                        method='use_viscous_region',
                        min_D=1e-8,
                        ver=1,
                        viscous_line_index=1,
                        l1_T_bounds=[0,1],
                        l2_T_bounds=[0,1]):
        indices = np.where(Ds>min_D)#0.00000095)
        print("in common, indices:",indices)
        print("00", indices[0][0])
        start_index = indices[0][0]
        D_As=Ds[start_index:]
        quenchTs=Ts[start_index:]
        #print('quenchTs',quenchTs)
        model = piecewise(quenchTs, D_As)
        #print(ver)
        if ver≡4:
            #print('ver 4')
            line_vals=[]
            Ts_low_i = np.where(Ts≥l1_T_bounds[0])[0]
            if len(Ts_low_i)≡0:
                raise ValueError('lower bound for T fitting of line 1 too low. Use a higher T')
            l1_low_i = Ts_low_i[0]
            Ts_low_i = np.where(Ts≥l2_T_bounds[0])[0]
            if len(Ts_low_i)≡0:
                raise ValueError('lower bound for T fitting of line 2 too low. Use a higher T')
            l2_low_i = Ts_low_i[0]

            Ts_high_i = np.where(Ts≤l1_T_bounds[1])[0]
            if len(Ts_high_i)≡0:
                raise ValueError('upper bound for T fitting of line 1 too high. Use a lower T')
            l1_high_i = Ts_high_i[-1]
            Ts_high_i = np.where(Ts≤l2_T_bounds[1])[0]
            if len(Ts_high_i)≡0:
                raise ValueError('upper bound for T fitting of line 2 too high. Use a lower T')
            l2_high_i = Ts_high_i[-1]
            #print('Ts_high_i',Ts_high_i)
            l1Ts=Ts[l1_low_i:l1_high_i+1]
            l1Ds=Ds[l1_low_i:l1_high_i+1]
            #print(l1_low_i,l1_high_i,l1Ts)
            l2Ts=Ts[l2_low_i:l2_high_i+1]
            l2Ds=Ds[l2_low_i:l2_high_i+1]
            #print(l2_low_i,l2_high_i,l2Ts,'Ts',Ts)
            par = np.polyfit(l1Ts, l1Ds, 1, full=True)
            m1 = par[0][0]#0-slope, 1-intercept
            b1 = par[0][1]
            xs = np.linspace(l1Ts[0],l1Ts[-1])
            ys = m1*xs+b1
            line_vals.append((xs,ys))

            par = np.polyfit(l2Ts, l2Ds, 1, full=True)
            m2 = par[0][0]#0-slope, 1-intercept
            b2 = par[0][1]
            xs = np.linspace(l2Ts[0],l2Ts[-1])
            ys = m2*xs+b2
            line_vals.append((xs,ys))

            x,y = line_intersect(m1,b1,m2,b2)
            Tg=x
            Tg_prop = y

            return Tg,Tg_prop,line_vals
        elif ver≡3:
            line_vals=[]
            Ts_low_i = np.where(Ts≥l1_T_bounds[0])[0]
            if len(Ts_low_i)≡0:
                raise ValueError('lower bound for T fitting of line 1 too low. Use a higher T')
            l1_low_i = Ts_low_i[0]
            Ts_low_i = np.where(Ts≥l2_T_bounds[0])[0]
            if len(Ts_low_i)≡0:
                raise ValueError('lower bound for T fitting of line 2 too low. Use a higher T')
            l2_low_i = Ts_low_i[0]

            Ts_high_i = np.where(Ts≤l1_T_bounds[1])[0]
            if len(Ts_high_i)≡0:
                raise ValueError('upper bound for T fitting of line 1 too high. Use a lower T')
            l1_high_i = Ts_high_i[-1]
            Ts_high_i = np.where(Ts≤l2_T_bounds[1])[0]
            if len(Ts_high_i)≡0:
                raise ValueError('upper bound for T fitting of line 2 too high. Use a lower T')
            print('Ts_high_i',Ts_high_i)
            l2_high_i = Ts_high_i[-1]

            l1Ts=Ts[l1_low_i:l1_high_i]
            l1Ds=Ds[l1_low_i:l1_high_i]
            print(l1_low_i,l1_high_i,l1Ts)
            l2Ts=Ts[l2_low_i:l2_high_i]
            l2Ds=Ds[l2_low_i:l2_high_i]
            print(l2_low_i,l2_high_i,l2Ts)
            par = np.polyfit(l1Ts, l1Ds, 1, full=True)
            m1 = par[0][0]#0-slope, 1-intercept
            b1 = par[0][1]
```

```
            xs = np.linspace(l1Ts[0],l1Ts[-1])
            ys = m1*xs+b1
            line_vals.append((xs,ys))

            par = np.polyfit(l2Ts, l2Ds, 1, full=True)
            m2 = par[0][0]#0-slope, 1-intercept
            b2 = par[0][1]
            xs = np.linspace(l2Ts[0],l2Ts[-1])
            ys = m2*xs+b2
            line_vals.append((xs,ys))
            if viscous_line_index≡0:
                Tg = -b1/m1
                Tg_prop = 0.
            elif viscous_line_index≡1:
                Tg = -b2/m2
                Tg_prop = 0.
            else:
                x,y = line_intersect(m1,b1,m2,b2)
                Tg=x
                Tg_prop = y

            return Tg,Tg_prop,line_vals
        elif ver≡2:
            n_lines=len(model.segments)
            if n_lines ≡ 0:
                raise ValueError('Found zero lines in piecewise fitting')
            lines=[]
            line_vals=[]
            for i in range(n_lines):
                line = model.segments[i]
                lines.append(line)
                xs = np.linspace(line.start_t,line.end_t)
                ys = line.coeffs[1]*xs+line.coeffs[0]
                line_vals.append((xs,ys))

            if method≡'use_viscous_region':
```

```
            if n_lines>1:
                l2=lines[viscous_line_index]
            else:
                l2=lines[0]
            m2 = l2.coeffs[1]
            b2 = l2.coeffs[0]
            Tg = -b2/m2
            Tg_prop = 0.
        else:
            Tg,Tg_prop=find_Tg(mean_vals=Ds,quenchTs=Ts)
        return Tg,Tg_prop,line_vals
    elif ver≡1:
        if len(model.segments) ≡ 2:
            l1 = model.segments[0]
            m1 = l1.coeffs[1]
            b1 = l1.coeffs[0]
            l2 = model.segments[1]
            m2 = l2.coeffs[1]
            b2 = l2.coeffs[0]
            x,y = line_intersect(m1,b1,m2,b2)
            xs1 = np.linspace(l1.start_t,l1.end_t)#np.linspace(l1.start_t,(x+(l1.end_t-l1.start_t)*0.2))
            ys1 = l1.coeffs[1]*xs1+l1.coeffs[0]
            xs2 = np.linspace(l2.start_t,l2.end_t)#np.linspace((x-(l2.end_t-l2.start_t)*0.2),l2.end_t)
            ys2 = l2.coeffs[1]*xs2+l2.coeffs[0]

            if method≡'use_viscous_region':
                Tg = -b2/m2
                Tg_prop = 0.
            elif method ≡ 'intersection':
                Tg=x
                Tg_prop=y
        else:
            print('WARNING: found {} line segments in regression!'.format(len(model.segments)))
        return Tg,Tg_prop,xs1,ys1,xs2,ys2


def Calc_Diffusivity(eq_time,
                     eq_msd,
                     fit_method='curve_fit'):
    #fit_method='curve_fit'#'power_law','poly_fit'
    if fit_method=='curve_fit':
        norm_eq_time = (eq_time-eq_time[0])
        #print(norm_eq_time,eq_msd)
        popt, pcov = curve_fit(lambda t,m,b: m*t+b ,
                                       eq_time,
                                       eq_msd,
                                       p0=[1.,0.0],
                                       bounds=([-1,0.0],[np.infty,np.infty]))
        drdt_A = popt[0]
        m=popt[0]
        b=popt[1]
    elif fit_method=='poly_fit':
        par = np.polyfit(time, msd, 1, full=True)
        drdt_A = par[0][0]#0-slope, 1-intercept
        m=par[0][0]
        b=par[0][1]
    elif fit_method=='power_law':
        popt, pcov = curve_fit(lambda t,w,x1: (w*t)**x1 ,
                               time,
                               msd,
                               p0=[0.2,1.0],
                               #p0=[1.0],
                               #bounds=([-np.infty,-np.infty],[np.infty,np.infty])
                               #bounds=([0],[4.0]))
                               maxfev=2000000,
                               bounds=([0.0,0.0],[1.0,4.0]))
        raise NotImplementedError('Diffusivity not determined')
    #calculate the diffusion coefficient
    dimensions=3
    D = drdt_A/(2*dimensions)
    return D,m,b

def getDiffusivities(project,df_curing,sortby='quench_T',name='bparticles',quench_time=1e7,use_first_trial=True):
    """
  returns diffusivity in units of D^2/tau where D and tau are distance and time units.
  Note that time is not in time steps.
  """
    Ts=[]
    Ds=[]
    for key,df_grp in df_curing.groupby('cooling_method'):
        if key≡'quench' ∧ quench_time is ¬ None:
            df_filt = df_grp[(df_grp.quench_time≡quench_time)]
        else:
            df_filt = df_grp
        df_sorted=df_filt.sort_values(sortby)
        for q_T,q_T_grp in df_sorted.groupby('quench_T'):
            for job_id in q_T_grp.index:
                job = project.open_job(id=job_id)
                if job.isfile('msd.log'):
                    log_path = job.fn('msd.log')
                    data = np.genfromtxt(log_path, names=True)
                    prop_values = data[name]#'pair_lj_energy']
                    equilibriated_ts_percentage = 0.5
                    if key≡'anneal':
                        times,msds,qTs = get_split_quench_job_msd(job,name)
                        for j,msd in enumerate(msds):
                            start_index = int(len(times[j])*equilibriated_ts_percentage)
                            time=times[j]*job.sp.md_dt
                            quench_T = qTs[j]
                            eq_msd = msd[start_index:]
                            eq_time = time[start_index:]
                            D_A,m,b = Calc_Diffusivity(eq_time,eq_msd,'curve_fit')
                            Ts.append(quench_T)
                            Ds.append(D_A)
                    else:
                        all_time_steps = data['timestep']
                        start_index = int(len(all_time_steps)*equilibriated_ts_percentage)
                        time=all_time_steps*job.sp.md_dt
                        quench_T = job.sp.quench_T
                        eq_msd = prop_values[start_index:]
                        eq_time = time[start_index:]
                        #print(job)
                        D_A,m,b = Calc_Diffusivity(eq_time,eq_msd,'curve_fit')
                        Ts.append(quench_T)
                        Ds.append(D_A)
                    if use_first_trial:
                        break#just using the first data point in this quench_T instead of mean
    Ts=np.asarray(Ts)
    Ds=np.asarray(Ds)
    return Ts,Ds

def savefig(plt,nbname,figname,transparent=True):
    import os
    if ¬ os.path.exists(nbname):
        os.makedirs(nbname)
    plt.savefig(os.path.join(nbname,figname),transparent=transparent)
```

tg_analysis.py Page 1/4

```
import signac
import numpy as np
import pandas as pd

import sys

from common import (
    getDiffusivities,
    fit_Tg_to_DiBenedetto,
    DiBenedetto,
    Fit_Diffusivity1,
)




def get_custom_ranges(cooling_method):
    if cooling_method ≡ "quench":
        custom_ranges_l1 = {
            00.0: [0.1, 0.8],
            30.0: [0.1, 0.8],
            50.0: [0.1, 0.8],
            70.0: [0.1, 0.8],
        }
        custom_ranges_l2 = {
            00.0: [0.7, 1.2],
            30.0: [0.85, 1.4],
            50.0: [1.0, 1.8],
            70.0: [1.15, 2.5],
        }
    elif cooling_method ≡ "anneal":
        custom_ranges_l1 = {
            00.0: [0.1, 0.8],
            30.0: [0.1, 0.8],
            50.0: [0.1, 0.8],
            70.0: [0.1, 0.8],
        }
        custom_ranges_l2 = {
            00.0: [0.7, 1.2],
            30.0: [0.85, 1.4],
            50.0: [1.0, 1.8],
            70.0: [1.15, 2.5],
        }
    else:
        raise ValueError(cooling_method + "is unknown")
    return custom_ranges_l1, custom_ranges_l2


def get_tg_data(data_path, df):
    project = signac.get_project(data_path)

    PROP_NAME = "bparticles"
    filter_saps = [0.0, 30.0, 50.0, 70.0]
    Tgs = []
    Tgs_tangent = []
    cure_percents = []
    Cure_Ts = []
    cooling_method = "quench"

    df_filtered = df[
        (df.quench_T ≤ 3.0)
        & (df.quench_T ≥ 0.1)
        & (df.CC_bond_angle ≠ 109.5)
        & (df.cooling_method ≡ cooling_method)
    ]
    for i, sap in enumerate(filter_saps):
        for j, (cooling_method, df_grp) in enumerate(df_filtered.groupby("cooling_
method")):
            df_curing = df_grp[
                (df_grp.bond ≡ False)
                & (df_grp.calibrationT ≡ 305)
                & (df_grp.cooling_method ≡ cooling_method)
```

```
                  & (df_grp.stop_after_percent ≡ sap)
              ]
              cure_percent = df_curing.cure_percent.mean()
              cure_percents.append(cure_percent)
              Ts, Ds = getDiffusivities(project, df_curing, name=PROP_NAME)
              Cure_Ts.append(Ts)
              # Pretty sure this helps with the fits
              mul_fact = 1000000
              Ds_scaled = Ds * mul_fact
              custom_ranges_l1, custom_ranges_l2 = get_custom_ranges(cooling_metho
d)
              Tg, Tg_prop, line_vals = Fit_Diffusivity1(
                  Ts,
                  Ds_scaled,
                  method="use_viscous_region",
                  min_D=0,
                  ver=4,
                  viscous_line_index=0,
                  l1_T_bounds=custom_ranges_l1[sap],
                  l2_T_bounds=custom_ranges_l2[sap],
              )
              Tgs.append(Tg)
      Tgs = np.asarray(Tgs)
      cure_percents = np.asarray(cure_percents)

      cure_percents = np.asarray(cure_percents)
      Tgs = np.asarray(Tgs)
      Tgs_tangent = np.asarray(Tgs_tangent)
      cure_percents_ss = cure_percents
      Tgs_ss = Tgs
      R2, fit_Tgs, T1, inter_parm, T0 = fit_Tg_to_DiBenedetto(
          cure_percents_ss / 100.0, Tgs_ss, T1=None, T0=None
      )
      alphas = np.linspace(0, 1)
      fit_ydata = DiBenedetto(alphas, T1, T0=T0, inter_param=inter_parm)
      cure_percents = np.asarray(cure_percents)
```

```python
    Tgs = np.asarray(Tgs)
    Tgs_tangent = np.asarray(Tgs_tangent)
    cure_percents_ss = cure_percents
    Tgs_ss = Tgs
    R2, fit_Tgs, T1, inter_parm, T0 = fit_Tg_to_DiBenedetto(
        cure_percents_ss / 100.0, Tgs_ss, T1=None, T0=None
    )
    alphas = np.linspace(0, 1)
    fit_ydata = DiBenedetto(alphas, T1, T0=T0, inter_param=inter_parm)

    return alphas, fit_ydata, R2, cure_percents, Tgs
```

ipython_nbconvert_config.py Page 1/1

```
c = get_config()

c.NbConvertApp.export_format = 'pdf'
c.Exporter.template_file = './better_article'
```